PARALLEL LEPP-BASED ALGORITHMS FOR THE GENERATION AND REFINEMENT OF TRIANGULATIONS

TESIS PARA OPTAR AL GRADO DE DOCTOR EN CIENCIAS, MENCION COMPUTACIÓN

PEDRO ANGEL RODRÍGUEZ MORENO

PROFESOR GUÍA:
MARÍA CECILIA RIVARA ZÚÑIGA

MIEMBROS DE LA COMISIÓN:
LUIS MATEU BRULÉ
BENJAMÍN BUSTOS CÁRDERNAS
NIKOS CHRISOCHOIDES

"Powered@NLHPC: Esta investigación fue parcialmente apoyada por la infraestructura de supercomputo del NLHPC (ECM-02)"

SANTIAGO DE CHILE
2015
Resumen

La generación y refinamiento de mallas son temas de gran interés en aplicaciones tales como simulación de fenómenos físicos mediante el uso de los métodos de elementos finitos, en aplicaciones CAD, modelado geométrico y mallas geométricas. Una malla es un conjunto de elementos geométricos (polígonos o poliedros) que no se superponen, los cuales están conectados por medio de vértices, aristas y caras, que se usan para aproximar dominios geométricos. Los algoritmos de refinamiento producen mallas cada vez más finas para discretizar dominios complejos, representar objetos con topologías arbitrarias y también superficies con formas complejas.

En esta tesis se estudió la paralelización de algoritmos de refinamiento basados en el concepto de Lepp para sistemas multicore (multinúcleo) y sistemas distribuidos. Se consideraron dos problemas: (1) refinamiento de mallas de buena calidad: aquí dada una malla de entrada de buena calidad, ésta es iterativa y localmente refinada (de acuerdo a un requerimiento externo a la aplicación) para producir una malla final de calidad análoga a la inicial; (2) refinamiento de triangulaciones Delaunay de mala calidad, donde dada una triangulación Delaunay de entrada de mala calidad (con una geometría dada), deseamos producir una triangulación Delaunay de buena calidad y de tamaño óptimo.

Algoritmos basados en el concepto de Lepp son algoritmos refinamiento por la arista más larga mejorados donde el refinamiento de cualquier triángulo $t$ tiene asociado un Lepp($t$).

En el contexto de los sistemas multicore se desarrollaron algoritmos Lepp-bisección multicore eficientes y escalables para el refinamiento de mallas de 2 y 3 dimensiones. También se desarrolló un algoritmo Lepp-Delaunay multicore para la generación de mallas Delaunay de buena calidad.

En el contexto de los sistemas de memoria distribuida se desarrolló un algoritmo Lepp-bisección distribuido para el refinamiento de mallas de 2 dimensiones donde la malla inicial es subdividida dentro de un conjunto de submallas (o subparticiones), las cuales son distribuidas entre los procesadores. También se desarrolló una estrategia eficiente para garantizar que se obtiene una malla final válida (conforme) en las interfaces de submallas vecinas.

Se realizaron evaluaciones empíricas de los algoritmos paralelos sobre arquitecturas multicore y sistemas de memoria distribuida que muestran que los algoritmos paralelos tienen buen desempeño.
Abstract

Mesh generation and mesh refinement are topics of great interest for different applications such as for the simulation of physical phenomena by using finite element methods, for CAD applications, for geometric modeling and computer graphics. A mesh is a set of non-overlapping geometrical elements (polygons or polyhedra) which are connected by vertices, edges and faces used to approximate a geometric domain. Refinement algorithms produce finer meshes that can be used to discretize complex domains, to represent objects of arbitrary topology and surfaces of complex shapes.

In this thesis we studied the parallelization of Lepp-based refinement algorithms both over multicore and distributed systems. Two different problems are considered: (1) Refinement of good quality meshes: here given a good quality input triangulation, the mesh is iteratively and locally refined (according to an external application requirement) to produce a final refined triangulation of analogous quality to the initial triangulation; (2) Refinement of bad-quality Delaunay triangulations where for a bad-quality Delaunay triangulation (of an input geometry), we need to produce a good-quality optimal size Delaunay triangulation.

Lepp-based algorithms are improved longest edge based algorithms where the refinement of any triangle \( t \) is confined over an associated Lepp(\( t \)) submesh.

In the multicore context we developed efficient and scalable Lepp-bisection multicore algorithms for the refinement of 2-dimensional and 3-dimensional meshes. We also developed a multicore Lepp-Delaunay algorithm for the generation of Delaunay quality meshes. The parallel algorithms prohibit the simultaneous refinement of triangles whose Lepp-submeshes overlap.

In the distributed memory system context we developed a distributed Lepp-bisection algorithm for the refinement of 2-dimensional meshes based on partitioning the mesh in a set of sub-meshes which are distributed between the processors. We also developed an efficient strategy to guarantee that a final valid mesh is obtained over the interfaces of neighboring sub-meshes.

We performed extensive empirical evaluations of the parallel algorithms over multicore architectures and distributed memory systems that showed that the parallel algorithms show good performance and scalability.
Dedico este trabajo a mi esposa Gisela,
a mis hijos Nicolás y Pedro,
y a mis padres.
Agradecimientos

Quiero agradecer en primer lugar a mi familia, a mi esposa Gisela, a mis hijos y a mis
padres, quienes siempre me apoyaron y creyeron en mi durante toda mi vida y durante esta
especial travesía.

A mi profesora guía, María Cecilia Rivara, por su infinita paciencia y sabios consejos, y
que también junto a mi infinita paciencia se logró todo este trabajo, el cual no termina aquí.

A Nancy Hitschfeld que me abrió las puertas de Beauchef para ser nuevamente parte de
este equipo.

A Angélica Aguirre, por sus buenos consejos y palabras de aliento.

A Renato Cerro por su valiosa ayuda en la revisión de este texto.

A mis compañeros de estudio, Cecilia, Eliana, Carlos, Alcides, Víctor y Rodrigo, con
quienes compartí muchos momentos agradables y también en aquellos duros y largos períodos
de estudio.

A mis amigos y colegas Manuel, Claudio, Pedro, Patricio y Gilberto, quienes en todo
momento siempre me alentaron y apoyaron para llegar al final de este trayecto.

A Mónica, a quien agradezco su apoyo y colaboración para poder terminar esta tesis. También a Elizabeth, por dedicar algunas horas de su valioso tiempo para revisar parte de este documento.

A mis amigos y músicos Raul, Marcos, Samuel, Felipe, Andrés, Karim, y tantos otros con
quienes pasamos y vivimos juntos muchas cosas de la amistad y de la carrera musical. A un
gran amigo y compañero de la música, Jaime Salgado, quien ya no está con nosotros, pero
que antes de partir siempre me apoyó y me alentó para seguir adelante.

Agradezco también al DCC y a la Universidad de Chile por haberme dado la oportunidad
de pasar una vez más por sus aulas y corredores de la ciencia.
# Contents

1 Introduction

2 Meshing problems, concepts and algorithms
   2.1 Mesh generation by using triangulations .................................... 7
   2.1.1 Delaunay triangulation .................................................. 8
   2.2 Mesh Quality ................................................................. 12
   2.3 Refinement and improvement of triangular meshes ....................... 15
      2.3.1 Algorithms based on edge/triangle partitions ....................... 16
         2.3.1.1 Longest edge refinement algorithms .......................... 16
         2.3.1.2 Other bisection refinement techniques ....................... 17
         2.3.1.3 Algorithms based on templates ............................... 18
      2.3.2 Delaunay refinement algorithms for triangulation improvement .... 18

3 Previous Serial and Parallel Longest Edge Algorithms ................................................................. 19
   3.1 Introduction ................................................................. 19
   3.2 Refinement algorithms based on the the longest edge bisection of triangles .... 19
      3.2.1 Longest-edge bisection algorithm .................................. 19
      3.2.2 The 4-triangles refinement algorithm ................................ 21
      3.2.3 Lepp-bisection Algorithm ............................................ 22
      3.2.4 Properties of the longest edge refinement algorithms .............. 23
   3.3 Lepp-Delaunay algorithms for triangulation improvement ............... 24
      3.3.1 Lepp-Delaunay midpoint method .................................... 25
      3.3.2 Lepp-Delaunay Centroid method .................................... 26
   3.4 Parallel Algorithms based on longest edge bisection ..................... 28
      3.4.1 Consistency issues for shared memory parallel refinement algorithms . 29
      3.4.2 Issues related to parallel refinement algorithm for distributed memory systems ........................................... 30
   3.5 Proof meshes used in the test problems .................................. 31

4 A review on parallel architectures and programming ................................................................. 33
   4.1 Parallel Computers .......................................................... 33
      4.1.1 Types of parallel computers: Flynn’s Taxonomy .................. 33
      4.1.2 Classification of MIMD systems .................................... 35
      4.1.3 Interconnection networks ............................................ 37
   4.2 Design of Parallel Algorithms ............................................. 38
      4.2.1 Foster’s design methodology ....................................... 39
4.2.2 Design objectives for better performance .......................... 40
4.2.3 Levels of granularity ................................................. 41
4.3 Parallelization with shared memory architectures .................. 41
4.4 Parallelization with distributed memory architectures .......... 43
  4.4.1 Workload assignment and load balancing ...................... 43
  4.4.2 Levels of coupling .................................................. 44
4.5 Parallelization on mixed distributed systems ...................... 44
4.6 Performance measures for parallel algorithms ..................... 45

5 Multithread Lepp-bisection algorithms in 2-dimensions .......... 46
  5.1 Introduction ........................................................... 46
  5.2 Data structure .......................................................... 46
  5.3 Serial Lepp-bisection algorithm ................................... 47
  5.4 Multicore architectures .............................................. 48
  5.5 Multicore algorithms .................................................. 50
  5.6 Multicore algorithm based on storing partial Lepp computation ... 51
  5.7 Multicore algorithm based on Lepp-Recomputation ............... 52
  5.8 Edge Monitor-based algorithm ...................................... 54
  5.9 Properties of the parallel algorithms .............................. 56
  5.10 Testing problems ...................................................... 57
  5.11 Testing Algorithms .................................................... 58
    5.11.1 Evaluation of algorithm behavior for problem TP1 .......... 58
    5.11.2 Evaluation of algorithm behavior for problem TP2 .......... 64
    5.11.3 Testing the parallel algorithms over a big input mesh for problem TP1 69

6 Multithread Relaxed Lepp-Delaunay algorithm in 2-dimensions ... 74
  6.1 Introduction ........................................................... 74
  6.2 A serial relaxed Lepp-Delaunay algorithm ........................ 75
  6.3 Multithread Lepp-Delaunay algorithm ................................ 76
  6.4 Study on the serial algorithm behavior ................................ 78
    6.4.1 Statistics on refined meshes for different values of $K$ .... 78
    6.4.2 Comparing Relaxed Lepp-Delaunay algorithm and Triangle .... 80
  6.5 Practical performance of the parallel Lepp-Delaunay algorithm ..... 86

7 Multithread Lepp-Bisection Algorithms for 3D meshes ........... 91
  7.1 Introduction ........................................................... 91
  7.2 Longest edge refinement algorithms for tetrahedral meshes .... 91
  7.3 Data structures ......................................................... 92
  7.4 Preliminary definitions ............................................... 93
    7.4.1 Refinement of a terminal star .................................. 95
  7.5 Serial 3D Lepp-Bisection algorithms ................................ 95
    7.5.1 Full Lepp-bisection algorithm .................................. 96
    7.5.2 A partial Lepp algorithm ......................................... 96
    7.5.3 Partial Lepp algorithm with partial Lepp-update ............ 97
    7.5.4 Properties of the 3D refinement algorithms based on longest edge bisection. ....................................... 97
  7.6 Multithread 3D Lepp-bisection algorithm .......................... 98
7.7 Empirical testing ................................................. 99
  7.7.1 Behavior of the serial algorithm .......................... 101
  7.7.2 Study on the speedup and efficiency .................... 105
  7.7.3 Empirical study on the quality of refined 3D triangulations .... 106
  7.7.4 Size of 3D Lepp ................................................. 108

8 Mesh partitioning methods ........................................ 111
  8.1 Introduction ....................................................... 111
  8.2 Previous mesh partitioning methods. ......................... 114
  8.3 Using METIS/ParMETIS libraries ............................. 114
  8.4 Strip-shaped partitioning ...................................... 118
  8.5 Recursive two-dimensional space subdivision based on quadtrees .... 120

9 Distributed Parallel Lepp-bisection algorithm for 2-dimensional mesh refinement ............................................. 123
  9.1 Introduction ....................................................... 123
  9.2 State of the art on the development of parallel and distributed meshing applications ........................................ 124
  9.3 Design and implementation of the parallel software .......... 126
  9.4 Parallel Distributed algorithm ................................ 126
  9.5 Conformity step along interfaces ............................ 127
  9.6 Empirical testing with Parallel Distributed Lepp-bisection algorithm .... 130

10 Conclusions ......................................................... 138

Bibliography ............................................................ 141
List of Tables

3.1 Different 2-dimensional geometries. ......................................................... 32
3.2 Different 3D geometries. ................................................................................. 32

5.1 Architectures used. ......................................................................................... 48
5.2 Statistics on Partial Lepp Storing algorithm, problem TP1, Intel Xeon E5550 (4 cores). ................................................................. 59
5.3 Execution time and efficiency measures, Partial Lepp Storing algorithm, problem TP1, Intel Xeon E5550 (8 cores). ............................. 59
5.4 Statistics on Lepp Recomputation Algorithm, problem TP1. Intel Core i7 (1 socket, 4 cores). .......................................................... 60
5.5 Execution time and efficiency measures, Lepp Recomputation Algorithm, problem TP1. Intel Core i7 (1 socket, 4 cores). ................. 60
5.6 Statistics on Lepp Recomputation Algorithm, problem TP1. Intel Xeon E5550 (2 sockets, 8 cores). .................................................. 61
5.7 Execution time and efficiency measures, Lepp Recomputation Algorithm, problem TP1. Intel Xeon E5550 (2 sockets, 8 cores). ............ 62
5.8 Statistics on Edge Monitor based algorithm, problem TP1. Intel Xeon E5550 (2 sockets, 8 cores). ......................................................... 63
5.9 Execution time and efficiency measures, Edge Monitor based algorithm, problem TP1. Intel Xeon E5550 (2 sockets, 8 cores). ............... 63
5.10 Statistics on Partial Lepp Storing Algorithm, problem TP2, L domain, circle refinement region, r=0.3, δ=0.001, Intel Xeon E5550, (two sockets, 8 cores). .................................................. 64
5.11 Execution time and efficiency measures, Partial Lepp Storing Algorithm, problem TP2, L domain, circle refinement region, r=0.3, δ=0.001, Intel Xeon E5550, (two sockets, 8 cores) .................................... 65
5.12 Statistics on Lepp Recomputation Algorithm, problem TP2 (L domain, circle refinement region, R_c = 0.3), δ=0.001. Intel Core i7 (1 socket, 4 cores). ............................................................. 65
5.13 Execution time and efficiency measures, Lepp Recomputation Algorithm, problem TP2 (L domain, circle refinement region, R_c = 0.3), δ=0.001. Intel Core i7 (1 socket, 4 cores). ............................................................. 66
5.14 Statistics on Lepp Recomputation Algorithm, problem TP2 (L domain, circle refinement region, R_c = 0.3), δ=0.001. Intel Xeon E5550 (2 sockets, 8 cores). ........................................ 67
5.15 Execution time and efficiency measures, Lepp Recomputation Algorithm, problem TP2 (L domain, circle refinement region, R_c = 0.3), δ=0.001. Intel Xeon E5550 (2 sockets, 8 cores). ................................. 67
5.16 Statistics on Edge Monitor Based Algorithm, problem TP2 (L domain, circle refinement region, \( R_c = 0.3 \)), \( \delta = 0.001 \). Intel Xeon E5550 (2 sockets, 8 cores). .................................................. 68
5.17 Execution time and efficiency measures, Edge Monitor Based Algorithm, problem TP2 (L domain, circle refinement region, \( R_c = 0.3 \)), \( \delta = 0.001 \). Intel Xeon E5550 (2 sockets, 8 cores). .................................................. 68
5.18 Statistics on iterative refinement, Partial Lepp storing algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores). .................................................. 70
5.19 Execution time and efficiency measures, Partial Lepp storing algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores). .................................................. 70
5.20 Statistics on iterative refinement, Lepp recomputation algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores). .................................................. 70
5.21 Execution time and efficiency measures, Lepp recomputation algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores). .................................................. 71
5.22 Statistics on iterative refinement, edge monitor based algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores). .................................................. 71
5.23 Execution time and efficiency measures, edge monitor based algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores). .................................................. 72

6.1 Problem P1. Final meshes to different values of \( K \) and percentage of non-Delaunay triangles. .................................................. 79
6.2 Problem P1. Distribution (in percentage) of smallest angles for different values of \( K \) and \( \theta_{\text{tol}} = 30^\circ \). .................................................. 79
6.3 Problem P1. Distribution (in percentage) of biggest angles for different values of \( K \) and \( \theta_{\text{tol}} = 30^\circ \). .................................................. 80
6.4 Lake Superior shape. \( K = 10 \). .................................................. 81
6.5 Key shape. \( K = 10 \). .................................................. 82
6.6 A shape. \( K = 10 \). .................................................. 83
6.7 Guitar shape. \( K = 10 \). .................................................. 83
6.8 “You suck” shape. \( K = 10 \). .................................................. 84
6.9 Face shape. \( K = 10 \). .................................................. 85
6.10 Doublex hexagon shape. \( K = 10 \). .................................................. 85
6.11 Problem P2. Final meshes for different values of \( K \) and percentage of Delaunay triangles, threshold angle 25°. Intel Xeon E5550. .................................................. 86
6.12 Problem P2. Final meshes for different values of \( K \) and percentage of Delaunay triangles, threshold angle 30°. Intel Xeon E5550. .................................................. 87
6.13 Problem P2. Execution time, threshold angle 25°. Intel Xeon E5550. .................................................. 87
6.15 Problem P2. Execution time, threshold angle 30°. Intel Xeon E5550. .................................................. 89

7.1 Test problems .................................................. 101
7.2 TP1 Convex Polyhedron. Random data (75,000 points randomly generated); refinement of 200,000 largest tetrahedra by iteration until longest edge \( \leq \delta \); \( \delta = 100 \). Intel Xeon E5550. .................................................. 102
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3</td>
<td><strong>TP2</strong> Convex Polyhedron. Random data (75,000 points randomly generated); refinement of 100,000 randomly selected tetrahedra by iteration. Intel Xeon E5550.</td>
</tr>
<tr>
<td>7.4</td>
<td><strong>TP3</strong> Mesh statistics. Convex polyhedron of randomly generated points; refinement of 50,000 randomly selected tetrahedra by iteration until longest edge ( \leq \delta; \delta = 11 ). Intel Xeon E5550.</td>
</tr>
<tr>
<td>7.5</td>
<td><strong>TP4</strong> 3D solid object. Shaft geometry; refinement of 50,000 largest tetrahedra by iteration (if possible) until longest edge ( \leq \delta; \delta = 3 ). Intel Xeon E5550.</td>
</tr>
<tr>
<td>7.6</td>
<td><strong>TP5</strong> Mesh statistics. Shaft. Refinement of 1,000 randomly selected tetrahedra by iteration. Intel Xeon E5550.</td>
</tr>
<tr>
<td>7.7</td>
<td><strong>TP6</strong> Mesh statistics. Elephant geometry. Refinement of 1,000 randomly selected tetrahedra by iteration until longest edge ( \leq \delta; \delta = 0.02 ). Intel Xeon E5550.</td>
</tr>
<tr>
<td>7.8</td>
<td>Execution Time (ms) of the test problems. Intel Xeon E5550 (2 sockets, 8 cores).</td>
</tr>
<tr>
<td>7.9</td>
<td>Performance measures of the test problems: speedup and efficiency. Intel Xeon E5550 (2 sockets, 8 cores).</td>
</tr>
<tr>
<td>7.10</td>
<td>Basic information of the input mesh.</td>
</tr>
<tr>
<td>7.11</td>
<td>Lepp Information.</td>
</tr>
<tr>
<td>7.12</td>
<td>Additional Lepp Information.</td>
</tr>
<tr>
<td>8.1</td>
<td>Input mesh to be partitioned using .m2d format.</td>
</tr>
<tr>
<td>8.2</td>
<td>Input mesh of Table 8.1 is transformed to .mesh format.</td>
</tr>
<tr>
<td>8.3</td>
<td>Partitioning information.</td>
</tr>
<tr>
<td>9.1</td>
<td>Performance measures: execution time, speedup and efficiency.</td>
</tr>
<tr>
<td>9.2</td>
<td>Other measures.</td>
</tr>
<tr>
<td>9.3</td>
<td>Performance measures: execution time, speedup and efficiency.</td>
</tr>
<tr>
<td>9.4</td>
<td>Other measures.</td>
</tr>
<tr>
<td>9.5</td>
<td>Performance measures: execution time, speedup and efficiency.</td>
</tr>
<tr>
<td>9.6</td>
<td>Other measures.</td>
</tr>
<tr>
<td>9.7</td>
<td>Results comparison.</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Two-dimensional mesh: (a) Conforming mesh, (b) Non-conforming mesh. ................. 7
2.2 Three-dimensional mesh: (a) Conforming mesh, (b) Non-conforming mesh. ............... 7
2.3 Triangle $t$ satisfies the Delaunay condition. ......................................................... 8
2.4 (a) $t$ and $t^*$ are not Delaunay triangles; (b) $t$ and $t^*$ are Delaunay triangles. They hold the Delaunay condition. ................................................................. 9
2.5 (a) Triangle $t$ contains the point $P$ to be incrementally inserted; (b) three new triangles $(t_1$, $t_2$ and $t_3)$ are created by inserting a new point $P$ into triangle $t$; (c) triangles $t_2$ and $t_3$ and their neighboring triangles $n_2$, $n_3$, respectively, do not meet the Delaunay condition; (d) A flip operation is carried out between the pair of triangles $t_2$, $n_2$ and $t_3$, $n_3$. .......................................................... 11
2.6 (a) Point P falls over an edge which is shared by triangles $t_1$ and $t_2$; (b) Triangles $t_1$ and $t_2$ are subdivided and four new triangles $(t_{11}$, $t_{12}$, $t_{21}$ and $t_{22}$) are created; (c) the new triangles $(t_{21}$ and $t_{12})$ and their neighboring triangles $n_2$ and $n_3$, respectively, do not meet the Delaunay condition; (d) An edge flipping operation is carried out between the pair of triangles $t_{21}$, $n_2$ and $t_{12}$, $n_3$. ........................................................................ 12
2.7 (a) $P$ is the point to be Delaunay inserted in the mesh. (b) The triangle $t$ and its neighboring triangles whose circumcircles contain $P$ are eliminated. (c) New triangles are created within the cavity by joining $P$ and the border edges. 13
2.8 PSLG CMU. Initial constrained Delaunay triangulation which contains a set of constrained edges. ............................................................................................................. 13
2.9 The constrained Delaunay triangulation of the PSLG with holes and additional points. ......................................................................................................................... 14
2.10 Good quality triangles: (a) Regular, very good element (b) good; Bad quality triangles: (c) cap, (d) needle ................................................................. 14
2.11 Good quality tetrahedra: (a) Regular, very good element (b) good; Bad quality tetrahedra: (c) needle, (d) cap, (e) sliver ......................................................... 15
2.12 Triangle Division Methods. (a) Bisection (b) Regular Refinement. ................. 17
2.13 Refinement using newest-node. ................................................................. 17

3.1 The bisection of a triangle by the midpoint of its longest edge. (a) AB longest edge of triangle ABC; (b) longest edge bisection of triangle. P is the midpoint of the longest edge. ......................................................... 20
3.2 The insertion of point $P$ produces an intermediate non-conforming mesh. .... 20
3.3 The 4 partition patterns used by the 4-triangles algorithm. (a) 4-triangles partition; (b), (c) and (d) correspond to longest-edge partition and 3-triangles partitions, respectively, to assure conformity of the mesh (refinement propagation). ............................................................. 21
3.4 Terminal edge of Lepp\(t_0\): (a) AB is an interior terminal edge; (b) AB is a boundary terminal edge. ................................................................. 22
3.5 Complete Lepp\(t_0\) procedure. ................................................................. 23
3.6 A representation of lake Superior. ............................................................... 24
3.7 After a refinement process using Lepp-Delaunay refinement/improvement algorithm. ................................................................. 25
3.8 Lepp-Delaunay Midpoint method. ............................................................... 25
3.9 Lepp Delaunay centroid method. ............................................................... 28
3.10 Synchronization problems in parallel refinement with shared memory. (a) Two processors \(P_1\) and \(P_2\) must not concurrently insert the same point \(V\) on the same edge; (b) The neighborhood information must be consistently established between the new obtained triangles by bisecting two neighboring triangles by two processors \(P_1\) and \(P_2\). ................................................................. 30
3.11 Multigrain parallel refinement. The mesh is divided into three different sub-meshes. Each sub-mesh has a set of Lepp to be processed. ................. 31
4.1 Conventional SISD architecture. One processor with a single instruction stream and a single data stream. ............................................................. 34
4.2 Conventional SIMD architecture. A set of processing units which receive the same instruction stream to process different data streams. ............................................................. 34
4.3 Conventional MISD architecture. A set of processing units which receive different instruction streams to process a single data stream. ................. 35
4.4 Conventional MIMD architecture. A set of autonomous processing units which receive different instruction streams to process different data streams. .. 35
4.5 Shared memory system. P: processor; M: memory ........................................ 36
4.6 Distributed memory system. P: processor; M: memory .................................. 37
4.7 Static networks. (a) Linear array; (b) Mesh; (c) Hypercube. ......................... 38
4.8 Dynamic networks. (a) Bus; (b) Crossbar; (c) Omega network. .................. 38
4.9 Foster’s parallel algorithm design methodology ......................................... 40
4.10 Cache memory levels L1, L2 and L3 .......................................................... 42
4.11 Cache memory system. (a) Multicore architecture with two cache memory levels (L1, L2); (b) Multicore architecture with three cache memory levels (L1, L2, L3) ............................................................. 43
5.1 Triangle data structure. .............................................................................. 47
5.2 (a) \(\text{Lepp}(t_0) = \{t_0, t_1, t_2, t_3\}\), with terminal triangles are \(t_2, t_3\); (b) Once the refinement of the terminal triangles \(\{t_2, t_3\}\) is performed, \(\text{Lepp}(t_0)\) is partially recomputed and updated from \(t_1\); (c) final refined mesh where \(t_0\) was refined. ............................................................. 48
5.3 Intel Core i7 processor. .............................................................................. 49
5.4 Intel Xeon E5550 processor. ...................................................................... 49
5.5 Dual Socket architecture based on Intel Xeon E5550 processors. ................. 49
5.6 Lepp Collision: \(\text{Lepp}(t_0) = \{t_0, t_1, t_2, t_3, t_4, t_5\}\) and \(\text{Lepp}(t_0^*) = \{t^*, t_1^*, t_2^*, t_3^*, t_2^-\), \(t_3, t_4, t_5\}\). ............................................................. 50
5.7 Lepp Collision at the edges level: thread $P_2$ is blocked in the edge $L$. Lepp($t_0$) $= \{t_0, t_1, t_2, t_3, t_4, t_5\}$, the longest edges of the triangles $\{t_0, t_1, t_2, t_3, t_4, t_5\}$ are locked, and Lepp($t_0'$) $= \{t_0', t_1', t_2', t_3, t_4, t_5\}$ where only the longest edges of the triangles $\{t_0', t_1', t_2', t_3\}$ are locked. 

5.8 Triangulation inside of a rectangular region. (a) Initial mesh; (b) Refined mesh

5.9 L-shaped domain with refinement region $R_c$. ........................................ 57

5.10 Intermediate triangulation obtained by refining triangulation of Figure 5.9 over $R_c = 0.3$, center B = (5,5). ...................................................... 58

5.11 Speedup Partial Lepp Storing Algorithm, problem TP1, Intel Xeon E5550 (8 cores), for the last six iterations (iter15 to iter20). ................... 59

5.12 Speedup Lepp Recomputation Algorithm, problem TP1, Intel Core i7 (4 cores), random selection of triangles. Speedups obtained for the last seven iterations (iter14 to iter20). ................................. 60

5.13 Speedup Lepp Recomputation Algorithm, problem TP1, Intel Xeon E5550. The speedup achieves a slight improvement when 8 cores are used to execute the parallel refinement algorithm. Speedups obtained for the last seven iterations (iter15 to iter21). ................................. 61

5.14 Speedup Edge Monitor based algorithm, problem TP1, Intel Xeon E5550. The speedup deteriorates when the number of cores increase. Speedups obtained for the last seven iterations (iter15 to iter21). ................................. 62

5.15 Speedup behavior comparison. Problem TP1. Parallel algorithms: (PA1) Partial Lepp storing algorithm; (PA2) Lepp recomputation algorithm; and (PA3) Edge Monitor based algorithm. Intel core i7 and Intel Xeon E5550 architectures. ................................................................. 63

5.16 Speedup Partial Lepp Storing Algorithm, problem TP2, Intel Xeon E5550 (8 cores), for the last five iterations (iter22 to iter26). .............. 64

5.17 Speedup Lepp Recomputation Algorithm, problem TP2. Intel Core i7 (1 socket, 4 cores), for the last five iterations (iter22 to iter26). ............... 65

5.18 Speedup Lepp Recomputation Algorithm, problem TP2, Intel Xeon E5550 (2 sockets, 8 cores), for the last five iterations (iter22 to iter26). ............... 66

5.19 Speedup Edge Monitor Based Algorithm, problem TP2, Intel Xeon E5550 (2 sockets, 8 cores), for the last five iterations (iter22 to iter26). ............... 67

5.20 Speedup behavior comparison. Problem TP2. Parallel algorithms: (PA1) Partial Lepp storing algorithm; (PA2) Lepp recomputation algorithm; and (PA3) Edge monitor based algorithm. Intel core i7 and Intel Xeon E5550 architectures, $R_c = 0.3$ and $\delta=0.001$. ......................................................... 68

5.21 Speedup Partial Lepp storing algorithm, problem TP1, Intel Xeon E5550 (2 sockets, 8 cores). .............................................................. 69

5.22 Speedup Lepp recomputation algorithm, problem TP1, Intel Xeon E5550 (2 sockets, 8 cores). .............................................................. 70

5.23 Speedup edge monitor based algorithm, problem TP1, Intel Xeon E5550 (2 sockets, 8 cores). .............................................................. 71

5.24 Speedup behavior comparison. Intel Xeon E5550, random selection. Problem TP1. ........................................................................ 72

xix
Green triangles show the $K - \text{Neighbor}(E)$. (a) $0 - \text{Neighbor}(E)$ includes the terminal triangles; (b) $1 - \text{Neighbor}(E)$ includes terminal triangles ($0 - \text{Neighbor}(E)$) and their immediate neighbors; (c) $2 - \text{Neighbor}(E)$ includes $1 - \text{Neighbor}(E)$ and their immediate neighbors.

(a) Four new triangles $t_{41}$, $t_{42}$, $t_{51}$ and $t_{52}$ are created from terminal triangles $t_4$ and $t_5$ of Figure 6.1 (c); (b) Delaunay insertion of centroid $M$ with $K=2$

6.3 Initial mesh with Lake Superior shape. .............................................. 81
6.4 Final mesh with Lake Superior shape, using $30^\circ$. .............................. 81
6.5 Final mesh with Lake Superior shape, using $35^\circ$. .............................. 82
6.6 Key geometric shape. (a) initial mesh; (b) final mesh obtained with angle of $30^\circ$; (c) final mesh obtained with an angle of $35^\circ$. ...................... 82
6.7 'A' geometric shape. (a) initial mesh with; (b) Final mesh obtained with an angle of $30^\circ$. ...................................................... 83
6.8 Guitar shape. (a) initial mesh; (b) Final mesh obtained with an angle of $30^\circ$. 84
6.9 You-suck shape. (a) initial mesh; (b) Final mesh obtained with an angle of $30^\circ$. 84
6.10 Face shape. (a) initial mesh; (b) final mesh with an angle of $30^\circ$. ........ 85
6.11 Doublex hexagon shape (a) initial mesh; (b) Final mesh with an angle of $30^\circ$. 86
6.12 Problem P2. Execution time for $K=0,1,2,4,10$, threshold angle $25^\circ$. Intel Xeon E5550, 2, 4 and 8 cores. ...................................................... 88
6.13 Problem P2. Speedup for $K=0,1,2,4,10$, threshold angle $25^\circ$. Intel Xeon E5550, 2, 4 and 8 cores. ...................................................... 88
6.14 Problem P2. Execution time for $K=0,1,2,4,10$, threshold angle $30^\circ$. Intel Xeon E5550, 2, 4 and 8 cores. ...................................................... 89
6.15 Problem P2. Speedup for $K=0,1,2,4,10$, threshold angle $30^\circ$. Intel Xeon E5550, 2, 4 and 8 cores. ...................................................... 90

7.1 Tetrahedron structure. .......................................................... 92
7.2 Tetrahedron with positive orientation. ........................................... 93
7.3 (a) Closed star, (b) Open star. ................................................... 93
7.4 The longest-edge bisection of tetrahedron ($v_0$, $v_1$, $v_2$, $v_3$). .......... 94
7.5 (a) The longest-edge bisection applied over a closed terminal star from Figure 7.3 (a), (b) The longest-edge bisection applied over an open terminal star from Figure 7.3 (b). ...................................................... 95
7.6 Input mesh of convex geometry for TP1, TP2 and TP3 testing problems. The tetrahedra marked with blue color correspond to the selected elements to be refined. ...................................................... 100
7.7 Shaft geometry. An object with irregular shape. ............................... 100
7.8 Shaft geometry with marked elements with blue color that correspond to the selected elements to be refined. ...................................................... 100
7.9 Elephant geometry. The tetrahedra shaded in blue color correspond to the selected elements to be refined. ...................................................... 101
7.10 Speedup for TP1 to TP6 testing problems. ...................................... 106
7.11 TP1: Three-dimensional convex hull. Refinement of largest tetrahedra. Quality distribution for initial and final meshes. ...................................................... 107
7.13 TP5: Shaft. Random selection. Percentage of quality by range: initial and refined mesh. .................................................. 107

8.1 Examples of mesh subdivision. (a) An input mesh subdivided into eight different homogeneous strip partitions; (b) An input irregular mesh (PSLG) subdivided into four partitions. ........................................... 112
8.2 Triangles $t_i$ and $t_{i+1}$ are located within partitions $S_i$ and $S_{i+1}$, respectively, and share a common interface edge $e$. ......................... 112
8.3 (a) Input mesh formed by a set of points randomly generated by using CGAL library; (b) Sixteen partitions obtained from the input mesh shown in (a) which was divided using METIS library. .......................... 116
8.4 (a) Lake superior; (b) Lake superior divided into four sub-meshes by using METIS library. ......................................................... 116
8.5 (a) 3D mesh of a set of random generated points using CGAL library. (b) Six sub-meshes obtained using METIS library. .................. 117
8.6 (a) A complex skull geometry; (b) Skull was subdivided in eight sub-meshes using METIS. ......................................................... 117
8.7 The eight sub-meshes obtained from the partitioned mesh of the Skull shown in Figure 8.6 (b) using METIS library. ......................... 117
8.8 Input mesh subdivide into four partitions. Region $R$ represents the zone where the candidate elements to be refined are uniformly distributed across the mesh. Sub-regions $S_1$, $S_2$, $S_3$ and $S_4$ have roughly the same workload. Note that $x_{min} = s x_{min}$ and $x_{max} = s x_{max}$. ................................. 118
8.9 Input mesh subdivide into four partitions. Region $R$ represents the zone where the candidate elements to be refined are concentrated into a bounded sub-region into the mesh achieving an equilibrate workload. Sub-regions $S_1$, $S_2$, $S_3$ and $S_4$ have roughly the same workload. Note that $x_{min} < s x_{min}$ and $x_{max} > s x_{max}$. .................................................. 119
8.10 Lake superior divided in four partitions by using strip-shaped partitions. 120
8.11 The input space and the cardinal directions of the quadrants. ............... 120
8.12 Input mesh subdivide into 22 partitions by using a partitioning algorithm based on quadtree. ......................................................... 121
8.13 Quadtree represented by a tree structure that corresponds to the Figure 8.12. 121
8.14 Mesh shown in Figure 8.3 (a) was partitioned into 10 partitions by using quadtree. The sub-meshes obtained are not homogeneous which means that the set of triangles to be refined are not uniformly distributed across the mesh. 122

9.1 General process of the parallel distributed mesh refinement. .................. 126
9.2 Procedure to obtain a conforming mesh on the interface. Triangles $t_i$ and $t_{i+1}$ belong to different sub-meshes and share the same interface edge $e$. ........... 128
9.3 (a) Initial two-dimensional mesh (1,000 vertices and 1,981 triangles) (b) The initial mesh is subdivided in 8 partitions by using METIS library) ....... 129
9.4 (a) Detail of the interface over an initial mesh; (b) Final non-conforming mesh after the parallel refinement along the interface; (c) Final conforming mesh along the interface (conformity step performed). ......................... 129
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.5 Speedup. Using partitions constructed by using ParMETIS library.</td>
<td>131</td>
</tr>
<tr>
<td>9.6 Speedup. Using strip partitions.</td>
<td>133</td>
</tr>
<tr>
<td>9.7 Speedup. Using recursive space subdivision based on quadtree.</td>
<td>135</td>
</tr>
<tr>
<td>9.8 Speedup comparison.</td>
<td>136</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Triangular mesh generation has been extensively used by engineers and numerical analysts for finite element applications since for the last five decades. Finite element methods are numerical techniques for finding approximate solutions of complex physical problems modeled by partial differential equations [65], which require an appropriate discretization of the domain, usually a triangulation or a mesh of quadrilaterals in 2-dimensions, or a mesh of tetrahedra or hexahedra in 3-dimensions. Because of their flexibility, 2 and 3-dimensional triangulations are preferred practical tools.

Computational methods for generating and refining triangular and tetrahedral finite element meshes in 2 and 3-dimensions can be roughly classified as Delaunay based methods, [13, 53, 222, 211], methods based on the partition of triangles and tetrahedra [198, 188, 186, 138], and quadtree methods [134] [205] [206].

For finite element methods, two important requirements are necessary to guarantee convergence: 1) conforming meshes are needed, where the intersection of two adjacent triangles or tetrahedra is either a common vertex, or a common edge or a common face; 2) the mesh needs to have good geometric quality [113] [187].

Unstructured triangulations which include triangles with different sizes and shapes, are flexible tools for finite elements applications since they allow to mesh highly complex geometries and easily manage the insertion and removal of points for local adaptation of the mesh [161]. Mesh refinement for adaptive finite element methods requires adding more points on the mesh [110] in those areas of higher error in the partial differential equation solution [24].

Longest edge refinement algorithms for triangulations, based on the longest edge bisection of triangles (obtained by joining the midpoint of the longest edge with the opposite vertex) were especially designed by Rivara [187, 192, 198, 14] to deal with adaptive multi-grid finite element methods. They are efficient and simple techniques which guarantee that quality and conforming meshes are obtained.

Lepp-bisection algorithm is a reformulated and improved version of the previous longest-edge algorithm (the first longest-edge bisection algorithm published by Rivara in 1984 [187])
where the refinement of the Lepp($t$) of every triangle $t$ is confined over a small Lepp-submesh which guarantees the conformity of the mesh on every refinement iteration.

The parallelization of mesh refinement algorithms over distributed systems is a problem that presents important challenges [58] [70] [238] such as to achieve an adequate domain division, good load balancing and the minimization of the message transfer [14] [114] [192].

Previous parallel refinement algorithms based on bisection of triangles and tetrahedra have been proposed and studied: Castaños and Savage [38] and Jones and Plassmann [113] parallelized the previous longest-edge algorithm and 4-triangles refinement algorithm, respectively, while Balman [14] and Rivara et al. [192] have parallelized refinement algorithms that use the Lepp concept for 3D meshes. Parallel algorithms based on Delaunay refinement have been proposed by Antonopoulos [6] and Chernikov et al. [52], [223].

In this thesis we study and implement parallel Lepp-based algorithms [198] [193] [191] over multicore and distributed systems.

Problem Statement and Hypothesis

We consider the parallel refinement of triangulations by using Lepp-based algorithms in two and three-dimensions in the following two settings: (1) multicore (shared memory) architectures; and (2) parallel distributed memory architectures based on message passing. Two different mesh refinement problems are considered:

1. Refinement of good quality meshes. The problem is defined as follows: given a good quality input triangulation, we need to produce a final triangulation which is iteratively and locally refined (according to an external application requirement) to produce a final triangulation of analogous quality to the initial triangulation. To deal with the problem, we use Lepp-bisection algorithms based on longest edge bisection of triangles.

2. Refinement of bad-quality Delaunay triangulations. The problem is defined as follows: given a bad-quality Delaunay triangulation (of an input geometry), we need to produce a good-quality output Delaunay triangulation. To deal with the problem we use Lepp-Delaunay algorithms.

To deal with these problems we take advantage of the properties of Lepp-based algorithms which support the following hypothesis:

1. Taking advantage of the properties of Lepp and terminal edge concepts, efficient and scalable parallel Lepp-bisection and parallel Lepp-Delaunay algorithms can be developed for multicore architectures and distributed systems.

2. Parallel algorithms based on Lepp concept are simple, efficient and have good scalability for mesh refinement.

3. Lepp-based parallel Delaunay algorithms are well suited for generating an initial quality mesh of complex geometries.
Contributions of this thesis

The main results of this thesis are:

1. In the multicore context we have developed efficient and scalable Lepp-bisection multicores algorithms for the refinement of 2-dimensional and 3-dimensional meshes. We have also developed a multicore Lepp-Delaunay algorithm for the Delaunay refinement of 2-dimensional meshes.

   The multicore (shared memory) algorithms take advantage of the properties of Lepp-based algorithms. Here for any triangle $t$ to be refined Lepp($t$) identifies a small Lepp-submesh where the refinement is confined. Furthermore the algorithms performance is independent of the order of triangle processing. These properties allow to develop efficient and scalable multicore algorithms over any multicore architecture.

2. In the distributed memory system context we have developed a distributed Lepp-bisection algorithm for the refinement of 2-dimensional meshes based on partitioning the mesh in a set of sub-meshes which are distributed between the processors. We have also developed an efficient strategy to guarantee that a final valid mesh is obtained over the interfaces of neighboring sub-meshes. Three mesh partitioning methods were used to distribute the workload to processors: strip-shaped partitions, quadtrees and partitions methods supported by METIS and ParMETIS libraries.

The contributions of this thesis are supported by publications related to some topics illustrated in this document.

1. Publications in Journals:
   
   - “Multithread parallelization of Lepp-bisection algorithms”, the parallelization of Lepp-bisection algorithms in multicore architectures [196], published in Applied Numerical Mathematics (APNUM2012, associated with chapter 5).
   

2. Publications in Proceedings:
   
   
3. Conferences:


- “Multithread Lepp-Bisection Algorithm for Tetrahedral Meshes”, the parallelization of Lepp-bisection algorithms for 3-dimensional meshes in multicore architectures, 22nd International Meshing Roundtable (IMR) [201], October 2013, Orlando, Florida, USA. (IMR2013, associated with chapter 7).

- “Parallel Lepp-bisection algorithm over distributed memory systems”, the parallelization of Lepp-bisection algorithms for 2-dimensional meshes on distributed systems, XXXII International Conference of the Chilean Computer Science Society, Universidad Católica de Temuco, Universidad Católica de Temuco, Nov. 2013, Temuco, Chile. [200] (associated with chapter 9).


Dissertation Organization

Chapter 2 introduces concepts and algorithms related to triangular mesh generation, triangular mesh refinement and mesh quality measures for 2-dimensional and 3-dimensional meshes.

Chapter 3 reviews the properties of longest edge algorithms which support Lepp-Bisection and Lepp-Delaunay algorithms and that inspire the development of this thesis. In addition, previous parallel refinement algorithms based on longest-edge bisection are discussed.

Chapter 4 reviews concepts and principles for designing and developing parallel algorithms over shared and distributed memory systems.

In chapter 5 we discuss three multithread refinement algorithms. The first parallel algorithm shows a partially scalable speedup up to 4 cores, however, this deteriorates when 8 cores are used. The deterioration of the speedup for 8 cores is due to the fact that when a set of partial Lepps remain into a list L, the parallel algorithm postpones the processing of these Lepps and their associated triangles. The second parallel algorithm behaves well with 8 cores, because when a thread detects collisions (when it tries to catch a triangle which is locked by other thread) the thread picks up another triangle to be refined. This strategy
avoids the blocking of the threads into the queue of a mutex. The third parallel algorithm does not behave well when the number of cores increases, and the speedup does not scale well due to the high contention and the blocking times of the threads into the queue of the monitors.

The randomization strategy reduces algorithm dependence on data distribution over memory resources, minimizing the rate of collisions between different Lepps, which are concurrently computed.

Chapter 6 discusses a serial and multicore relaxed Lepp-Delaunay algorithm based on the centroid point insertion criterion [191] for the quality triangulation problem [193]. To test the algorithm we use input meshes associated with irregular polygons and meshes created from a set of randomly generated points.

Chapter 7 presents sequential and parallel Lepp-bisection algorithms to iteratively refine 3-dimensional tetrahedral meshes. The parallel algorithm runs over multicore architectures where the set of cores is distributed over different sockets. To test the algorithms we use input meshes associated with irregular 3-dimensional geometries and convex polyhedral meshes created from a set of randomly generated points.

Chapter 8 discusses mesh partitioning methods for distributed memory systems. The goal is to equilibrate the workload between the nodes of the distributed system. We use partitioning methods provided by METIS/ParMETIS libraries, and algorithms based on strip-shaped partitions and quadtree techniques.

Chapter 9 discusses the development of parallel distributed mesh refinement algorithms. Distributed systems are able to process huge meshes which are not possible to process over multicore and shared memory systems. The insertion of new points on the interface between two neighboring partitions forces the resolution of conformity problems between elements located in the interface. This problem is solved by using message passing mechanisms.
Chapter 2

Meshing problems, concepts and algorithms

In this chapter we introduce concepts, problems and algorithms related with triangular mesh generation and triangular mesh refinement. We review refinement and improvement algorithms based on different point selection criteria. We also review concepts on mesh quality (quality measures) for two-dimensional and three-dimensional meshes.

A mesh is set of non-overlapping elements (polygons or polyhedra) which are connected by vertices, edges and faces used to approximate a geometric domain. A formal mathematical definition for triangulations is given below:

Let $\Omega$ be a closed and bounded domain in $\mathbb{R}^2$ or $\mathbb{R}^3$ [93] [97]:

A triangular mesh $\tau$ of $\Omega$ is a set of elements $t_k \{k = 1,...,n\}$ (triangles in 2-dimensions, tetrahedra in 3-dimensions) which discretizes the domain $\Omega$, such that:

a $\Omega = \bigcup_{t_k \in \tau} t_k$. The union of the elements of $\tau$ that covers or approximates the domain $\Omega$.

b Each $t_k$ has an area/volume greater than zero,

c The intersection of the interior of pairs of elements $t_i$ and $t_j$ in $\tau$ is always empty.

d $\tau$ is a conforming mesh of $\Omega$, where for $t_i, t_j$ in $\tau$, then $t_i \cap t_j$ is either: a common vertex, or a common edge, or a common face, or the empty set [190].

We say that $\tau$ is a valid or conforming mesh if points c and d hold.

Figures 2.1 and 2.2 show conforming and non-conforming meshes in two-dimensional and three-dimensional spaces, respectively. The non-conformity in Figures 2.1 (b) and 2.2 (b) is due to that d does not hold. A triangle/tetrahedron has two neighboring elements across a same edge.
2.1 Mesh generation by using triangulations

Given a set of points over a domain $\Omega$, we need to construct a geometric mesh that fits well over this particular domain. The mesh needs then to be iteratively refined to satisfy specific quality requirements for the resolution of a practical problem.

For different applications, the meshes must satisfy the following requirements:

1. **Conformity condition.** The intersection between two adjacent elements (triangles or tetrahedra) is either a common vertex, or a common edge or a common face.

2. **Element quality.** There are several metrics and measures of quality which help to identify good or bad quality elements [113] [187]. In 2-dimensions the most used quality measures are the minimum angle, the shape of the elements, aspect ratio, etc. In two dimensions it has been proved that the quality metrics associated to the shape of the elements are equivalents [137].

3. **Mesh size.** The size of the output mesh is crucial. The number of vertices and elements must be small enough to approximate the geometry and physical phenomenas with accurate resolution.

4. **Efficient software.** In practice it is necessary to implement efficient and robust software for meshing tasks and applications.
For constructing a triangulation of a geometric domain, three classes of methods are used in general:

1. **Delaunay methods.** These methods generate a mesh of triangles or tetrahedra which satisfies the empty circle/sphere criterion in 3D [24] [61] [162] as discussed in the next section.

2. **Advancing Front methods.** Triangles or tetrahedra are gradually built from the border of the geometry towards the interior of the domain [58] [108] [141] [162].

3. **Octree methods.** The input domain is recursively and adaptively subdivided into eight cubes of equal size [24] [162] until it does not contain elements or it has the maximum number of selected elements. Quadtree is the equivalent in two-dimensional space where the domain is also recursively and adaptively subdivided into four quads. Both spatial structures form an n-ary tree.

### 2.1.1 Delaunay triangulation

The Delaunay triangulation was introduced and studied in 1934 [69].

**Definition 2.1** Given a set $V$ of points in the plane, a Delaunay triangulation $DT(V)$ (Delaunay Triangulation of $V$) is a triangulation where no point in $V$ is inside of the circumsphere of any triangle in $DT(V)$ (Delaunay condition) [69] (See Figure 2.3).

It has been proven that the Delaunay triangulation maximizes the minimum angle of a triangulation of a set of points [155].

For constructing the Delaunay triangulation of a set $V$ of $n$ points, algorithms based on divide and conquer strategy have optimal $O(n \log n)$ cost [20].

![Figure 2.3: Triangle $t$ satisfies the Delaunay condition.](image)
Figure 2.4: (a) \( t \) and \( t^* \) are not Delaunay triangles; (b) \( t \) and \( t^* \) are Delaunay triangles. They hold the Delaunay condition.

In practice incremental algorithms are in general used for constructing of a Delaunay triangulation. Here each point of the fixed input set \( V \) is incrementally inserted into the triangulation. There are two kinds of incremental algorithms to calculate a Delaunay triangulation: (1) algorithms based on edge flipping; (2) algorithms based on the cavity.

Edge flipping operation refers to the following property: if two triangles do not meet the Delaunay condition as shown in Figure 2.4 (a), then an edge flipping operation (diagonal interchange) produces a local Delaunay triangulation as shown in Figure 2.4 (b). We also say that \( v_0v_3 \) in Figure 2.4 (b) is a legal edge. Any 2-dimensional triangulation is a Delaunay triangulation if all its edges are legals [20].

Algorithm 1 illustrates the operation of legalization of a pair of triangles by edge flipping operation.

**Algorithm 1** LegalizeEdge\( (p_r, p_dp_j, T) \)

\( \text{(* The point being inserted is } p_r, \text{ and } p_dp_j \text{ is the edge of } T \text{ that may need to be flipped. *\}) \)

**if** \( p_dp_j \text{ is illegal} \text{ then} \)

Let \( p_dp_jp_k \) be the triangle adjacent to \( p_dp_jp \) along \( p_dp_j \).

Replace \( p_dp_j \) with \( p_dp_k \). (\( \text{(* Flip } p_dp_j: \) *)

LegalizeEdge\( (p_r, p_dp_k, T) \).

LegalizeEdge\( (p_r, p_dp_j, T) \).

**end if**

Algorithm 2 [20] illustrates the Delaunay triangulation algorithm based on edge flipping. Given a set of points \( P \), the algorithm computes an initial large triangle which contains the set \( P \). The algorithm sequentially selects a point \( p_r \) \( (r=0, 1, \ldots, n-1) \in P \) to be Delaunay inserted into the triangulation \( T \).

When a new point \( P \) needs to be inserted in the current Delaunay mesh (see Figure 2.5 (a)), the triangle \( t \) that contains \( P \) is located which is then subdivided into new three triangles \( t_1, t_2 \) and \( t_3 \) by joining \( P \) with the vertices of \( t \) (Figure 2.5 (b)). See example of Figure 2.5 where triangles \( t_2 \) and \( t_3 \), and their respective neighboring triangles \( n_2 \) and \( n_3 \), do not meet
Algorithm 2 Delaunay-Triangulation($P$)

Input: A set $P = p_i$, $i = 0, 1, \ldots, n-1$ of $n$ points in the plane.
Output: A Delaunay triangulation of $P$ DT($P$).
Construct an initial triangle $T$ that contains the set $P$. Let $p_0$, $p$ and $q$ the vertices of $T$. ($/ * p_0$, $p$, $q \notin P */$)

for $r=0$ to $n-1$ do

Locate the triangle $p_ip_jp_k \in T$ containing $p_r$.

if $p_r$ lies inside the triangle of $p_ip_jp_k$ then

Add edges from $p_r$ to the vertices of $p_ip_jp_k$ and subdivide $p_ip_jp_k$ into three new triangles.

LegalizeEdge($p_r$, $p_i$, $p_j$, $T$).

LegalizeEdge($p_r$, $p_j$, $p_k$, $T$).

LegalizeEdge($p_r$, $p_k$, $p_i$, $T$).

else

(* $p_r$ lies on edge $p_ip_j$ *)

Add edges from $p_r$ to $p_i$ and to $p_j$, the third vertex of the other sharing $p_ip_k$ and subdivide two triangles sharing $p_ip_k$ into four new triangles.

LegalizeEdge($p_r$, $p_i$, $p_k$, $T$).

LegalizeEdge($p_r$, $p_i$, $p_j$, $T$).

LegalizeEdge($p_r$, $p_j$, $p_k$, $T$).

LegalizeEdge($p_r$, $p_k$, $p_i$, $T$).

end if

end for

Remove all edges containing vertices $p$ and $q$.

return $T$.

the Delaunay condition. A flip operation is carried out between pairs of triangles $(t_2, n_2)$ and $(t_3, n_3)$. Figure 2.5 (d) illustrates the final mesh.

When a point $P$ is inserted over an edge shared by two triangles (see Figure 2.6 (a)), both triangles $(t_1$ and $t_2)$ are subdivided to produce four new triangles $\{t_{11}, t_{12}, t_{21}, t_{22}\}$ (see Figure 2.6 (b)). If the new triangles and their respective neighboring triangles do not meet the Delaunay condition then an edge flipping operation must be carried out to obtain a legal mesh (see Figure 2.6 (c) and (d)). Procedure 1 illustrates the operation of legalization of a pair of triangles by an edge flipping operation when the new point is located over an edge.

Another incremental algorithm can be formulated using the cavity concept. Given a point $P$ to be inserted in the mesh, the cavity $C(P)$ is defined as the polygon formed by the set of triangles whose circumcircles include the new point $P$ [30] [249]. See Figure 2.7. The triangles that form a cavity are eliminated and they are then replaced by new elements by joining the new point $P$ to be inserted with the cavity border edges. See Figure 2.7 (c).

Let $\tau$ be a triangulation defined over a set of points $P$ and a set of constrained edges in the plane. We say that $\tau$ is a constrained Delaunay triangulation (CDT) if for every triangle $t \in \tau$ the set of constrained edges is respected. The set of prespecified constrained edges and defined in the initial set, must remain in the final mesh. The constrained Delaunay triangulation is the triangulation of the points with the following properties [53]:
Figure 2.5: (a) Triangle $t$ contains the point $P$ to be incrementally inserted; (b) three new triangles ($t_1$, $t_2$ and $t_3$) are created by inserting a new point $P$ into triangle $t$; (c) triangles $t_2$ and $t_3$ and their neighboring triangles $n_2$, $n_3$, respectively, do not meet the Delaunay condition; (d) A flip operation is carried out between the pair of triangles $t_2$, $n_2$ and $t_3$, $n_3$.

1. The prespecified constrained edges are included and respected in the triangulation.
2. $\tau$ is as close to the Delaunay triangulation.

Given a set of points and a set of considered edges, a CDT must be computed as follows:

1. Compute the non-constrained Delaunay triangulation over the set of points.
2. Perform a post-process triangulation by applying a constrained Delaunay triangulation which respects the constrained edges.

The set of vertices of $\tau$ is augmented by inserting additional vertices (Steiner points) so that in the conforming resultant Delaunay triangulation each constrained edge is represented by a contiguous linear sequence of edges of the triangulation [220]. Every constrained edge is shared by two triangles of the resultant mesh. The resultant mesh is not always Delaunay, but it depends on the algorithm used to compute the CDT.

Figure 2.8 illustrates a PSLG domain with initial constrained Delaunay triangulations. The inside of the PSLG has a set of constrained edges that form empty holes. This initial triangulation respects the inner constrained edges. Figure 2.9 illustrates the final constrained Delaunay triangulation with additional points.
Figure 2.6: (a) Point P falls over an edge which is shared by triangles $t_1$ and $t_2$; (b) Triangles $t_1$ and $t_2$ are subdivided and four new triangles ($t_{11}$, $t_{12}$, $t_{21}$ and $t_{22}$) are created; (c) the new triangles ($t_{21}$ and $t_{12}$) and their neighboring triangles $n_2$ and $n_3$, respectively, do not meet the Delaunay condition; (d) An edge flipping operation is carried out between the pair of triangles $t_{21}$, $n_2$ and $t_{12}$, $n_3$.

2.2 Mesh Quality

Quality triangulations are needed for most of the practical applications, in particular for the finite element method [224]. Triangles and tetrahedra that are closed to regular shapes are desirable, while elements that are closed to degenerate shapes must be avoided.

Low quality elements influence the stability, convergence and precision of the resolution of the finite element method. Small and large angles can have serious effect on interpolation errors and the shape of the elements (triangles/tetrahedra) of the mesh used to approximate a partial differential equation [11].

Figures 2.10 and 2.11 show examples of good and bad quality elements in 2 and 3-dimensions, respectively.

The quality indicators in general are defined as equal to one for the regular triangle/tetrahedron while for a bad quality element approach zero. In 3-dimensional meshes, the geometric metrics are based on the volume of an element, the aspect ratio and the solid and dihedral angles. On the other hand, when the volume of a tetrahedron is negative, a measure of quality can establish that the tetrahedron is an inverted element.

Desirable properties of a quality measure are described by Shewchuk in [224, 223, 127, 137]. Some geometric quality measures used for two-dimensional meshes are [224, 127, 137]:

\[ \text{Quality} = \frac{\text{Volume}}{\text{Area} \times \text{Height}} \]
Figure 2.7: (a) $P$ is the point to be Delaunay inserted in the mesh. (b) The triangle $t$ and its neighboring triangles whose circumcircles contain $P$ are eliminated. (c) New triangles are created within the cavity by joining $P$ and the border edges.

Figure 2.8: PSLG CMU. Initial constrained Delaunay triangulation which contains a set of constrained edges.

- Smallest and largest angles. This is a general measure of triangle shape based on the interior angles size of a triangle.
- Aspect ratio. This is defined as the length of the longest edge ($l_e$) divided by the length of the shortest altitude ($h$) of the triangle.
- Radius ratio. This is defined as the ratio of inradius (radius of the incircle) to circum-radius (radius of the circumcircle) of a triangle.
- Circumradius-to-shortest edge ratio of a triangle. This is defined as the quotient of the circumradius $r$ of a triangle $t$ and the length $l$ of its shortest edge ($r/l$). It is desirable for this ratio to be as small as possible.
- Metric based on the area of a triangle. This is defined as follows:

\[ q(t) = C \frac{\text{area}(t)}{(\text{longest-edge})^2}, \text{where } C = \frac{4.0}{\sqrt{3.0}} \]

where $\text{area}(t)$ is the area of the triangle $t$ and "longest − edge" is the length of its longest edge [224, 223].

This is equal to 1 for the equilateral triangle and zero for any degenerate (zero area)
Figure 2.9: The constrained Delaunay triangulation of the PSLG with holes and additional points.

Figure 2.10: Good quality triangles: (a) Regular, very good element (b) good; Bad quality triangles: (c) cap, (d) needle

triangle.

Some quality measures used for three-dimensional meshes are:

- Minimum solid angle. This is a general measure of the shape based on the solid angles of a tetrahedron.

- Aspect ratio. This is defined as the length of the longest edge \( (l_e) \) divided by the length of the shortest altitude \( (h) \) of the tetrahedron.

- Radius ratio. This is defined as the ratio of inradius (radius of the insphere) to circumradius (radius of the circumsphere) of a tetrahedron [137].

- Circumradius-to-shortest edge ratio of a tetrahedron. Similar to the metric defined for two-dimensional meshes, the circumradius-to-shortest edge ratio corresponds to the quotient of the circumradius \( r \) of a tetrahedron \( t \) and the length \( l \) of its shortest edge \( (r/l) \).

- Volume based metric. It is a continuous function of the position of the nodes that evaluates the quality of a tetrahedron. It has to be a maximum for the regular tetrahedron and minimum for a degenerate tetrahedron. The values used to measure the quality of the elements must be scaled in the interval \([0,1]\) and must cost 1 for the regular tetrahedron. Examples of these measures are:
Figure 2.11: Good quality tetrahedra: (a) Regular, very good element (b) good; Bad quality tetrahedra: (c) needle, (d) cap, (e) sliver

- The Liu and Joe’s metric for the bisection of a tetrahedron is given by the next expression:

\[ \eta(t) = \frac{12(3v)^{2/3}}{\sum_{i=1}^{6} l_i^2} \]

where \( v \) is the volume and \( l_i \) (\( i=1,...,6 \)) is the length of every edge \( i \) of the tetrahedron \( t \).

- The tetrahedra shape measure used by this work is given by the next expression:

\[ q(t) = C \frac{\text{volume}(t)}{\text{longest-edge}^3}, \text{ where } C = \frac{12.0}{\sqrt{2.0}} \]

where \( \text{volume}(t) \) is the volume of the tetrahedron \( t \) and “longest – edge” is the length of its longest edge [224, 223].

2.3 Refinement and improvement of triangular meshes

For the purposes of this thesis we distinguish two important mesh refinement problems:

1. **Selective local refinement of an acceptable quality mesh.** Given an initial quality mesh, some selected elements and some neighbor elements are refined [187] [15] [223] producing a quality mesh analogous to the initial mesh. This is a useful strategy for finite element applications where the PDE solution varies greatly over the domain.

2. **Mesh improvement.** The elements of the mesh are refined in order to improve the mesh quality.
When an adaptive mesh refinement method is used, an external application iteratively defines a subset of elements to be refined in the mesh by using a triangle error indicator [223] [26]. The following procedure illustrates the use of the adaptive mesh refinement for the finite element method [113].

**Algorithm 3** Adaptive FEM($\tau_0$)

Input: $\tau_0$, input mesh of domain $\Omega$, and the partial differential equation (PDE).
Output: Refined mesh $\tau_f$.

Use the finite element method to obtain a numerical solution of the PDE problem over $\tau_0$.

Solve the linear system associated to the partial differential equations on $\tau_0$.

Estimate the error on each triangle.

$k=0$.

**while** The approximation triangle error is higher than the given tolerance **do**

Based on the errors calculated over each triangle in $\tau_k$, determine a set of triangles $S_k$ to be refined.

Refine mesh $\tau_k$ to form $\tau_{k+1}$.

Solve PDE in $\tau_{k+1}$.

Estimate the error on each triangle.

$k = k + 1$.

**end while**

Algorithms for mesh refinement can be classified as follows:

1. Algorithms based on edge/triangle partition [15] [148] [187]. Here, the new triangles are contained in the triangles of the previous mesh [24] [162] as illustrated in Figure 2.12.

2. Delaunay refinement algorithms. A new point associated with a poor quality triangle is selected and inserted into the mesh using the Delaunay strategy. See section 2.3.2 that describes circumcenter-based algorithms and Lepp-Delaunay algorithm [51] [53] [54] [59] [162] [222] [191].

3. Algorithms based on templates. Mesh elements are subdivided using the same specific subdivision pattern [109] [162].

### 2.3.1 Algorithms based on edge/triangle partitions

#### 2.3.1.1 Longest edge refinement algorithms

Due to the mathematical properties of the iterative partitioning of triangles by the longest edge, methods based on longest-edge bisection produce conforming and nested meshes of quality analogous to the initial mesh, which are useful for the adaptive finite element method. To bisect a triangle $t$, the longest edge midpoint of the triangle is connected to its opposite vertex to create two new triangles [24] [162] [187] [198] [194] as shown in figure 2.12 (a). A detailed description on these algorithms is given in chapter 3.
2.3.1.2 Other bisection refinement techniques

**Regular refinement algorithm.** The regular refinement algorithm of Bank [15] [148] [24] divides triangles to be refined into four triangles. Temporary intermediate bisections are made in neighboring triangles to build a conforming mesh without small angles. Figure 2.12 (b) shows the refinement pattern of triangles using regular refinement.

**Newest-Node Algorithm.** The Newest-node algorithm of Sewell is also based on the bisection of triangles, but not necessarily by the longest edge [148]. This algorithm bisects triangles using the inserted newest node.

The triangle is divided by connecting the peak node to the midpoint of the base. The newest node becomes the peak of the two new triangles. Figure 2.13 illustrates the refinement process using this algorithm. For example, according to Figure 2.13, the triangle 1 is divided, connecting its node peak to the midpoint of its base, obtaining two new triangles (triangle 2 and triangle 3). The newest node (the new peak node of triangles 2 and 3) is connected to the midpoint of the bases of these two triangles obtaining four triangles.

Figure 2.12: Triangle Division Methods. (a) Bisection (b) Regular Refinement.

Figure 2.13: Refinement using newest-node.
2.3.1.3 Algorithms based on templates

A pattern corresponds to a template or a precalculated refinement configuration which describes a specific decomposition of a triangle/tetrahedron [157] [162] [210]. Each selected triangle of the input mesh is then replaced by using a pattern consisting of subtriangles/subtetrahedra. For example, a triangle can be subdivided into four similar triangles by inserting a new node on midpoint of each edge [68]. Figure 2.12 shows examples of two division patterns which are also considered as templates of refinement. Examples of mesh refinement based on templates are given by the 4-Triangle algorithm of Rivara [187], [258] and the regular refinement algorithm of Bank [15].

2.3.2 Delaunay refinement algorithms for triangulation improvement

Given a bad quality initial Delaunay mesh $\tau$, we need to obtain a final good quality Delaunay mesh whose inner angles are bounded by a $\theta_{\text{min}}$ angle. [240]. The main objective is to eliminate poor quality triangles [52].

Methods that produce a sequence of improved constrained Delaunay triangulations (CDT) have been developed to deal with the quality triangulations of a planar straight line graph (PSLG). If the input Delaunay mesh presents inner and border constrained edges then the refinement algorithm must respect these constrained edges.

There are roughly two types of Delaunay refinement algorithms: circumcenter methods and Lepp-Delaunay methods. The circumcenter methods have been studied by Chew, [54], Ruppert [203], and Shewchuk [222]. They use the insertion of a new point which is either the circumcenter of a poor quality triangle [94] or the midpoint of an encroached segment [59] [51]. These two operations are carried out to eliminate bad quality triangles. Algorithms based on off-center insertions have been recently presented by Ungor [240] as an additional point insertion criterion.

Rivara proposed and studied refinement and improvement algorithms based on the Lepp concept over Delaunay triangulations [198, 193, 191]. The new point is roughly selected as the midpoint of a local largest edge in the mesh [191]. These algorithms are reviewed in detail in chapters 3 and 6.

To measure the quality of a triangle, either the minimum angle or the circumradius-to-shortest edge ratio are used for the 2-dimensional algorithms.
Chapter 3

Previous Serial and Parallel Longest Edge Algorithms

3.1 Introduction

Here we discuss serial and parallel longest edge algorithms for the quality mesh refinement problem defined as follows: given a good quality (bounded smallest angle) and conforming mesh $\tau_0$, we need to construct a locally refined mesh $\tau_f$ of quality analogous to $\tau_0$.

Longest edge bisection refinement algorithms for triangulations were designed to support the development of adaptive finite element software [188, 187]. These algorithms perform iterative local refinement by maintaining the quality of the input mesh. The properties of these algorithms [1, 101] are summarized in sections 3.2.1 and 3.2.4.

The longest edge algorithms were also generalized to derefine the mesh and for developing 3-dimensional mesh refinement algorithms [189, 186, 156].

3.2 Refinement algorithms based on the the longest edge bisection of triangles

3.2.1 Longest-edge bisection algorithm

The longest edge bisection of a triangle $t$ is the partition of $t$ by joining the midpoint of its longest edge with the opposite vertex [187]. A new edge is created which separates the two new triangles created. Figure 3.1 illustrates the longest edge bisection of a triangle. The properties of the longest edge bisection of a triangle can be summarized as follows:

1. The iterative longest-edge bisection of a triangle $t$ produces new nested triangles whose minimum angle is bounded by $\alpha_t \geq \alpha_0/2$, where $\alpha_0$ is the minimum angle of triangle $t$
and $\alpha_t$ is the minimum angle of the new nested triangles [190].

2. A finite number of similar and different triangles are generated [190].

![Figure 3.1: The bisection of a triangle by the midpoint of its longest edge.](image)

(a) $AB$ longest edge of triangle ABC; (b) longest edge bisection of triangle. $P$ is the midpoint of the longest edge.

Longest-edge bisection algorithms guarantee nested and unstructured good quality triangulations [187] [233] [236] with interior angles of analogous quality to the initial mesh [24].

The original longest edge bisection algorithm introduced by Rivara [187] refines a set $S$ of target triangles and a set of neighboring triangles to produce a conforming triangulation. This algorithm produces intermediate triangulations which includes some non-conforming triangles [187] (see Figure 3.2).

![Figure 3.2: The insertion of point $P$ produces an intermediate non-conforming mesh.](image)

The temporal complexity of the algorithm is $O(k|S_k|) + O(kn)$, where $k$ is the number of iterations to process sets of non-conforming triangles $S_k$. $|S_k|$ is the cardinality of the $k$-th set of non-conforming triangles. In the worst of case $|S_k| = n$, and we see that for $|S_k| < n$ and $|S_k| \leq \tau_n^k$, the complexity is $O(kn)$, where $k << n$. Algorithm 4 illustrates the procedure to refine a set of triangles $S$ of a mesh $\tau$. 
Algorithm 4 Longest Edge Bisection Algorithm ($\tau_0$, $S$)

1: Input: $\tau_0$, any initial conforming triangulation defined over a polygonal region $\Omega$, $S$ set of triangles to be refined.
2: Output: $\tau_1$, a new refined conforming triangulation.
3: Bisect triangle $t$ by its longest edge, for each $t_i \in S$. Let $\tau_0^1$ be the non-conforming triangulation generated in this way.
4: $k \leftarrow 1$.
5: while $\tau_0^k$ is non-conforming do
6: Find set $S_k \subset \tau_0^k$ of triangles $t_i \in \tau_0^k$ such that the midpoint of one of its sides is a non-conforming node.
7: Bisect $t$ by its the longest edge, for each $t_i \in S_k$. Let $\tau_0^{k+1}$ be the triangulation generated in this way.
8: $k \leftarrow k + 1$.
9: end while
10: Let $\tau_1 = \tau_0^k$

3.2.2 The 4-triangles refinement algorithm

This is a special and simple longest edge algorithm that uses the 4-triangles (4T) partition (see Figure 3.3) over the target triangles to produce a conforming triangulation. The 4-T partition of a triangle is obtained by joining the longest-edge midpoint with its opposite vertex and with the midpoints of the remaining two sides [194] as illustrated in Figure 3.3 (a). The rest of the refinement patterns illustrated in the Figure 3.3 (b), (c) and (d) are used to assure the conformity of the mesh. The 3D variant is known as the 8-tetrahedral longest-edge algorithm [174]. This algorithm inherits the same properties of the longest edge algorithm.

Figure 3.3: The 4 partition patterns used by the 4-triangles algorithm. (a) 4-triangles partition; (b), (c) and (d) correspond to longest-edge partition and 3-triangles partitions, respectively, to assure conformity of the mesh (refinement propagation).
3.2.3 Lepp-bisection Algorithm

This is an improved reformulation of the longest edge bisection algorithm (see section 3.2.1) based on the Lepp and terminal edge concepts [190] [193] [198] [192].

Definition 3.1 For any triangle $t_0$ of any conforming triangulation $\tau$, the Lepp of $t_0$, denoted by $\text{Lepp}(t_0)$, is the ordered list of all triangles $t_0$, $t_1$, $t_2$, ..., $t_{n-1}$, $t_n$, such as $t_i$ is the neighbor triangle of $t_{i-1}$ by the longest edge of $t_{i-1}$, for $i = 1, 2, ..., n$.

Definition 3.2 An edge $AB$ is called a terminal edge [198] in a triangulation $\tau$, if $AB$ is the common longest edge of the triangles that share $AB$, which are called terminal triangles [198]. Note that in 2-dimensions either $AB$ is shared by two terminal triangles $t_{n-1}$, $t_n$ if $AB$ is an interior edge, or $AB$ is a boundary terminal edge of a single terminal triangle $t_n$ as shown in Figure 3.4.

Properties. For any triangulation $\tau$ and any triangle $t_0$ of $\tau$, the following properties of the Lepp($t_0$) hold:

1. For any triangle $t_0$, $\text{Lepp}(t_0)$ is always finite.
2. The ordered triangles $t_0$, $t_1$, $t_2$, ..., $t_{n-1}$, $t_n$ of $\text{Lepp}(t_0)$, have strictly increasing longest edge sizes.

![Figure 3.4: Terminal edge of Lepp($t_0$): (a) AB is an interior terminal edge; (b) AB is a boundary terminal edge.](image)

Given a triangle $t_0$, the Lepp-bisection algorithm repeatedly computes the Lepp($t_0$) and refines couples of terminal triangles until the triangle $t_0$ is refined. Figure 3.5 illustrates the refinement of triangle $t_0$ showing the order in which the new points are inserted. This algorithm is formulated as follows [198]:

[190] [193] [198] [192]
Algorithm 5 Lepp-Bisection-Algorithm-A(τ, S)

Input: τ, a quality triangulation, and S ⊂ τ, set of triangles to be refined.  
Output: τ_f, a refined and conforming final triangulation.  

for each triangle t ∈ S do  
  while (t remains without being bisected) do  
    Find Lepp(t), terminal triangles t_1, t_2 and terminal edge AB.  
    Triangle t_2 can be null if AB is a boundary edge.  
    Bisect the terminal triangles.  
  end while  
end for

Figure 3.5: Complete Lepp(t_0) procedure.

The Lepp-bisection algorithm produces a final triangulation without using intermediate non-conforming points in the mesh. In the general case, the temporal complexity of the algorithm is O(|K|^2), where |K| is the number of triangles refined by the refinement process over the set of Lepp of S and K = ∪ Lepp(t_i), where t_i ∈ S. In a pessimistic case the temporal complexity is O(k^2|S|), where k is the length of the longest Lepp of S and |S| is the cardinality of set S.

3.2.4 Properties of the longest edge refinement algorithms

The mathematical properties of the longest edge algorithms can be summarized as follows [18, 190, 196]:

1. The algorithms always produce triangles whose smallest interior angles are greater than or equal to α_0/2 (α_0 is the smallest interior angle of the initial triangulation τ_0).

2. Longest-edge refinement algorithms always terminate in a finite number of steps.
3. The area of a triangulation $\tau$ tends to be covered by quasi-equilateral triangles with smallest angles $\geq 30^\circ$, when a global refinement is iteratively applied over a triangulation $\tau$.

### 3.3 Lepp-Delaunay algorithms for triangulation improvement

Here we deal with the quality triangulation problem. Given a bad quality conforming triangulation $\tau$ and a threshold $\theta_{\text{min}}$, we need to construct a good quality mesh $\tau_f$ such that every triangle in $\tau_f$ has a minimum angle greater than or equal to $\theta_{\text{min}}$.

These methods use the longest edge propagation path (Lepp) associated with a poor quality triangle $t_0$ to find a terminal edge in the current mesh. Two versions of the Lepp-Delaunay algorithms have been proposed and studied:

1. Algorithm based on selecting the terminal edge midpoint [193].
2. Algorithm based on selecting the centroid of the terminal quadrilateral formed by pairs of terminal triangles [191].

It has been proven that the Lepp-Delaunay algorithms guarantee that for $\theta_{\text{min}} = 30^\circ$ the triangles in the output mesh have angles between $30^\circ$ and $120^\circ$ [191, 193]. Figures 3.6 and 3.7 illustrate the improvement of a mesh with irregular shape region by applying the Lepp-Delaunay algorithm. Figure 3.6 shows the initial mesh and Figure 3.7 shows the refined mesh.

![Figure 3.6: A representation of lake Superior.](image)
3.3.1 Lepp-Delaunay midpoint method.

This algorithm selects the midpoint $P$ of the terminal edge associated with the Lepp($t_0$) and proceeds to carry out the Delaunay insertion of $P$ into the mesh. Figure 3.8 illustrates a non-Delaunay initial mesh (Figure 3.8 (a)) and the application of Lepp-Delaunay midpoint method: (a) Select the midpoint $P$ of the terminal edge; (b) Terminal triangles $t_3$ and $t_4$ are bisected when point $P$ is inserted over the terminal edge; (c) New triangle $t'_3$ and triangle $t_2$ do not meet the Delaunay condition.

Edge flipping operations are then executed to obtain a legal mesh; (d) Lepp($t_0$) is recalculated and triangles $t_0$ and $t'_1$ are the terminal triangles. A new point $P$ is inserted over the terminal edge; (e) Triangles $t_0$ and $t'_1$ are bisected, and new triangle $t''_0$ and $t''_1$ do not meet the Delaunay condition; (f) Finally, edge flipping operations are carried out to obtain a legal mesh. It is worth noting that the algorithm can receive as input either a Delaunay mesh or a non-Delaunay mesh.

Figure 3.7: After a refinement process using Lepp-Delaunay refinement/improvement algorithm.

Figure 3.8: Lepp-Delaunay Midpoint method.
Algorithms 6 and 7 illustrate the Lepp-Delaunay Midpoint algorithm.

\textbf{Algorithm 6} Lepp-Midpoint-Algorithm-\(A(\tau,\theta_{tol})\)

Input: an input triangulation \(\tau\), and angle tolerance \(\theta_{tol}\).
Output: a refined and conforming final triangulation \(\tau_f\).
Find \(S\), the set of bad quality triangles regarding to \(\theta_{tol}\).
\textbf{for} (each triangle \(t \in S\)) \textbf{do}
  \textbf{while} (\(t\) remains int \(\tau\)) \textbf{do}
    Find the Lepp(\(t\)), terminal triangles \(t_1, t_2\) and terminal edge \(l\).
    Triangle \(t_2\) can be null for boundary \(l\).
    SelectPoint(\(P, t_1, t_2, l\)).
    Perform constrained Delaunay insertion of \(P\) into \(\tau\).
  \textbf{end while}
\textbf{end for}

\textbf{Algorithm 7} SelectPoint(\(P, t_1, t_2, l\))

Input: terminal triangles \(t_1, t_2\); terminal edge \(l\).
Output: new point to insert \(P\).
\textbf{if} (second longest edge of \(t_1\) is not constrained and second longest edge of \(t_2\) is not constrained) or (\(l\) is constrained) \textbf{then}
  Select \(P\) = midpoint of \(l\).
\textbf{return}.
\textbf{else}
  \textbf{for} (\(j=1,2\)) \textbf{do}
    \textbf{if} (\(t_j\) is not null and has constrained second longest edge \(l^*\)) \textbf{then}
      Select \(P\) = midpoint of \(l^*\).
    \textbf{end if}
  \textbf{end for}
\textbf{end if}

\subsection{3.3.2 Lepp-Delaunay Centroid method.}

Rivara and Calderon introduced in [191] the Lepp-centroid algorithm for quality triangulations both to avoid a rare loop situation of Lepp-Delaunay midpoint algorithm discussed in [193, 191] and to improve the algorithm performance in the number of points inserted. Instead of selecting the midpoint \(P\) of the terminal edge, the algorithm selects the centroid of the terminal quadrilateral (terminal-quad) defined as the quadrilateral formed by a couple of terminal triangles. Figure 3.9 illustrates the method. Algorithms 8 and 9 illustrate the Lepp-Delaunay Centroid algorithm.
Algorithm 8 Lepp-Delaunay-Centroid-Algorithm-A($\tau, \theta_{tol}$)

Input: an input triangulation $\tau$, and angle tolerance $\theta_{tol}$.
Output: a refined and conforming final triangulation $\tau_f$.
Find $S$ = the set of bad triangles regarding to $\theta_{tol}$.
for each triangle $t \in S$ do
  while $t$ remains int $\tau$ do
    Find the Lepp($t$), terminal triangles $t_1$, $t_2$ and terminal edge $l$.
    Triangle $t_2$ can be null for boundary $l$.
    NewSelectPoint($P$, $t_1$, $t_2$, $l$).
    Perform constrained Delaunay insertion of $P$ into $\tau$.
    Update $S$.
  end while
end for

Rivara and Calderon show in [191] that the Lepp-Delaunay centroid algorithm produces triangulations both with average smallest angles greater than those obtained with the midpoint version, and it also avoids the rare looping case associated to the midpoint method. In practice the centroid method: (1) computes significantly smaller triangulation than the terminal edge midpoint, (2) produces globally better triangulations, (3) terminates for higher threshold angle $\theta$ (angles bigger than $25^o$) and (4) experimental results show that the convergence of Lepp-Delaunay centroid is faster than Lepp-Delaunay midpoint. Lepp-Delaunay Centroid algorithm can also receive as input both Delaunay and non-Delaunay meshes.

Algorithm 9 NewSelectPoint($P$, $t_1$, $t_2$, $l$)

Input: terminal triangles $t_1$, $t_2$; terminal edge $l$.
Output: new point to insert $P$.
if ($l$ is constrained) then
  Select $P$ = midpoint of $l$.
  return.
end if
if (second longest edge of triangle $t_1$ is not constrained and second longest edge of triangle $t_2$ is not constrained) then
  Select $P$ = centroid of quad($t_1$, $t_2$).
  return.
else
  if (second longest edge of triangle $t_1$ is constrained and second longest edge of triangle $t_2$ is constrained) then
    Select $P$ = midpoint of a constrained second longest edge.
else
    for $t^*$ in {$t_1$, $t_2$} with constrained second longest edge do
      Select ConstrainedQuad($P$, $t_1$, $t_2$, $l$, $t^*$).
    end for
  end if
end if
Figure 3.9 illustrates a non-Delaunay initial mesh (Figure 3.9 (a)) and the application of Lepp-Delaunay centroid method: (a) Select centroid $P$ of the terminal quadrilateral formed by terminal triangles $t_3$ and $t_4$; (b) Four new triangles are created by joining centroid $P$ with the vertices of the terminal quadrilateral. New triangle $t'_3$ and $t_2$ do not meet the Delaunay condition; (c) Edge flipping operations are executed to obtain a legal mesh; (d) $\text{Lepp}(t_0)$ is recalculated and triangles $t_0$ and $t'_1$ are the terminal triangles. The centroid $P$ of the terminal quadrilateral formed by terminal triangles $t_0$ and $t'_1$ is Delaunay inserted; (e) Four new triangles are created and new triangle $t''_0$ and $t''_1$ do not meet the Delaunay condition; (f) Finally, edge flipping operations are carried out to obtain a legal mesh.

Figure 3.9: Lepp Delaunay centroid method.

Chapter 6 presents a study and experimental results on a particular case of modified Lepp-Delaunay centroid applied on PSLG meshes.

3.4 Parallel Algorithms based on longest edge bisection

Parallel algorithms on distributed environments based on Rivara’s longest-edge bisection algorithms, have been proposed by Williams [252], Castaños and Savage [38] [36] and Jones and Plassmann [114] [113]. For these methods the authors report good results with these parallel applications.

Jones and Plassmann proposed a parallel distributed algorithm based on 4-triangles refinement algorithm in 2-dimensions [194]. They use a dual graph concept [114] [113] [41] [91] to calculate successive independent sets for controlling the synchronization issues [114] illustrated in Figure 3.10. The algorithm bisects the triangles within each independent set. The remaining non-conforming triangles are selected for refinement and non-conformities are solved. This solution was theoretically defined for a shared memory parallel system, and then extended and implemented on distributed memory systems.

Castaños and Savage [38] discussed a distributed memory refinement algorithm based on the longest edge bisection algorithm [187] without using the Lepp concept. A binary tree is used to store the information of the whole refinement process. When a triangle is refined,
the information of children triangles is stored in the nodes of the binary tree [38]. In order to distribute the mesh, a partition method called PNR (Parallel Nested Repartitioning) provided in a library called PARED is used [36].

Rivara et al. [192] proposed a decoupled distributed algorithm to implement a global refinement algorithm for tetrahedral meshes based on terminal edge refinement. The algorithm repeatedly refines the terminal tetrahedra until obtaining the maximum length of the longest edge allowed (i.e., longest edge \( \leq \delta \)).

Balman [14] proposed a distributed refinement algorithm for Adaptive Mesh Refinement (AMR) of tetrahedral meshes. This solution calculates each Lepp and refines the tetrahedra using the 8-T LE algorithm [174], previously calculating an acyclic directed graph which contains information of each Lepp within the sub-meshes. ParMETIS is used here for partitioning the input mesh and achieving a level of partial coupling.

Foteinos and Chrisochoides [84] present a robust and dynamic Parallel Delaunay Triangulator for computing 3-dimensional triangulations. They implement a method that parallelizes a complex operation that dynamically inserts and removes points from a tetrahedral mesh.

Foteinos and Chrisochoides [85] present FEM simulations on four dimensional meshes which are generated from binary medical images. The simulations are executed over larger distributed memory systems with faster shared-memory layers.

Foteinos, Chernikov, and Chrisochoides [89] present a tetrahedral Delaunay refinement algorithm for meshing medical images. They prove that the tetrahedral output mesh have good radius-edge ratio and all the boundary facets have planar angles larger than 30 degrees.

Foteinos and Chrisochoides [90] present a tetrahedral Delaunay parallel Image-to-Mesh Conversion algorithm to manage biological objects using high quality and good shape tetrahedra. The parallel program is very efficient and is able to generate large tetrahedral meshes. They also compare their application with other software such as CGAL and TetGen.

### 3.4.1 Consistency issues for shared memory parallel refinement algorithms

When we face the designing of a parallel algorithm for mesh refinement in shared memory systems, we observe two problems of consistency between two adjacent elements (triangles) which require synchronization of the processors [113].

- Two processors \( P_1 \) and \( P_2 \) must not concurrently bisect the same edge [113]. Figure 3.10 (a) shows this case.
- When two processors \( P_1 \) and \( P_2 \) concurrently bisect two neighboring triangles, then four new neighboring triangles are produced. The neighborhood information between the new triangles must be consistently established. Figure 3.10 (b) shows the refinement of triangle \( W \) in subcomponents \( W_1 \) and \( W_2 \). The problem is that if the triangle \( U \) is refined in sub-components \( U_1 \) and \( U_2 \), \( W \) is even considered by \( U_1 \) as its neighbor.
triangle and not $W_1$.

![Diagram](image)

**Figure 3.10:** Synchronization problems in parallel refinement with shared memory. (a) Two processors $P_1$ and $P_2$ must not concurrently insert the same point $V$ on the same edge; (b) The neighborhood information must be consistently established between the new obtained triangles by bisecting two neighboring triangles by two processors $P_1$ and $P_2$.

### 3.4.2 Issues related to parallel refinement algorithm for distributed memory systems

The design of parallel distributed algorithms for mesh refinement needs to take into consideration the following questions:

1. **Domain division.** The input mesh must be divided into a set of smaller sub-domains (sub-meshes) to be distributed among the different nodes of the system. This also implies: (a) workload assignment, (b) load balancing, and (c) minimizing communication between nodes [14] [114] [192].

2. **Solving interface inconsistencies between neighbor sub-domains.** This refers to ensuring the consistency between two neighbor triangles sharing one edge which is located on the interface of two neighbor sub-domains. On the other hand, refinement propagation between two adjacent sub-domains must be treated with care.

3. **Dealing with coupling level.** The coupling level measures the intensity of the flow of messages between neighbor nodes of a distributed system.

Algorithms for domain division have been studied by several authors over the last 20 years. Techniques based on non-directed graphs [114] [113] [41] manage the mesh as a partition problem. The mesh is translated into a non-directed graph (called a dual graph) where one vertex represents one element of the mesh (e.g., triangle) and one edge of the graph represents one edge of the mesh shared by two triangles [91]. Libraries as METIS and ParMETIS use graphs to make subpartitions from an input mesh [14] [192]. Other partitioning methods are based on the use of spatial data structures such as quadtrees (2D) and octrees (3D) [64] [239].
Figure 3.11 shows a triangular mesh subdivided using strip partitions. Strip partitions can be obtained by geometric partitioning algorithms. In a distributed system, each sub-mesh can be processed by different processors.

![Submeshes](image)

Figure 3.11: Multigrain parallel refinement. The mesh is divided into three different sub-meshes. Each sub-mesh has a set of Lepp to be processed.

The mesh subdivision task can be performed in two ways:

1. **Static subdivision.** One of the nodes of the system makes the partitions and distributes them to the rest of the nodes of the distributed system, in addition to carrying out the load balancing and the control of the coupling level among the nodes [14] [239]. This is a complex strategy since it needs to have a very powerful node capable of sustaining a big set of input data. The disadvantage is that some processes may not be kept busy during the partitioning process.

2. **Dynamic subdivision.** The dynamic partition methods are applied during execution of all the refinement process. Partitioning tasks and refinement tasks are intercalated during the execution of the parallel refinement application to adjust the load of the nodes (or processors). In some cases, each node of the system builds the input elements in-situ defining partitions at runtime and the limits of partitions [239]. This is the most suitable type of partition for adaptive refinement. The complexity is due to the fact that processors need to carry out the load setting at each iteration among themselves [37] [161]. The advantage of the dynamic subdivision is that the processes are working and operating practically without idle times.

### 3.5 Proof meshes used in the test problems

The meshes used in the test problems of this thesis were selected from a set of meshes that are commonly used by different authors and libraries for generating and refining of meshes.

Table 3.1 illustrates a set of some of the meshes with regular and irregular shapes that were used in the experiments of this thesis and others were created and used by Jonathan Shewchuk [219], [222], [218] who developed Triangle, a software to create and refine 2-dimensional
meshes. Shewchuk also developed Stellar, a software to create and refine 3D meshes based on Delaunay refinement algorithms. Stellar also uses and processes the kind of 3-dimensional meshes illustrated in Table 3.2.

Ruppert and Chew used these kinds of meshes in their experiments, which have been published in their papers [203], [54], [56]. Other authors, such as Ramsay Dyer [73] used some 3-dimensional meshes based on tetrahedra and shown in Table 3.2.

Table 3.1: Different 2-dimensional geometries.

Authors such as Shewchuk, Labelle, Shen and Klingner used some 3-dimensional meshes as shown in Table 3.2, [221], [131], [215], [126]. These kinds of meshes have irregular shapes which are difficult to be created and refined. Others authors, such as, Lecant, Li, Mullen, etc. also have used irregular 3-dimensional meshes in their experiments [133], [135], [154], [39], [146], [57].

Table 3.2: Different 3D geometries.

Some libraries such as CGAL allow for generation and refining some of the 3-dimensional tetrahedral meshes shown in Table 3.2 [27], [28], [43], [77].
Chapter 4

A review on parallel architectures and programming

At present high-performance computing is required in a variety of applications based on complex mathematical models such as simulation of weather forecasts and crash simulations for the car industry, as well as for computer graphics applications related with film and advertising industry.

In this chapter we review general issues on parallel architectures and parallel programming over shared and distributed memory systems. We review the main characteristics of the parallel computers, classification of parallel computers and interconnection networks.

4.1 Parallel Computers

A parallel computer is a system which consists of several physical processors which are able to autonomously process in parallel different programs with different sets of data [251] [177]. The set of physical processors can either share the memory system, or not.

4.1.1 Types of parallel computers: Flynn’s Taxonomy

The Flynn’s’ parallel architecture classification [82] considers the following data and control flows classes:

1. SISD (Single Instruction Single Data). In this model we identify a single instruction stream which is executed by one processing unit to process a single data stream (see Figure 4.1). SISD is the conventional sequential computer according to the von Neumann model.

2. SIMD (Single Instruction Multiple Data). There is a control unit and multiple homogeneous execution units (see Figure 4.2). The control unit fetches and dispatches the
same instruction stream to the execution units to process different data streams.

3. MISD (Multiple Instruction Simple Data). Different processing units execute different instruction streams to process a single data stream (see Figure 4.3).

4. MIMD (Multiple Instruction Multiple Data). Different autonomous processing units receive different instruction streams to process different data streams (see Figure 4.4).

Figure 4.1: Conventional SISD architecture. One processor with a single instruction stream and a single data stream.

Figure 4.2: Conventional SIMD architecture. A set of processing units which receive the same instruction stream to process different data streams.
4.1.2 Classification of MIMD systems

Most of the general purpose parallel systems are based on the MIMD architecture. Regarding to physical memory organization, MIMD parallel computers are classified as follows:

1. MultiProcessor systems with shared memory (tightly-coupled systems). All the processors share the same global physical memory. Examples include SMP (Symmetric MultiProcessor) systems and the modern multicore microprocessors [100] [130]. Figure 4.5 illustrates a general scheme of a shared memory system.

2. Multicomputer systems with distributed memory (weakly coupled system). Here a number of independent processing units or computers (called nodes) are connected by
using an interconnection network which supports the transfer of data between nodes. Thus the physical memory of the system is distributed on all the computers or nodes of the parallel system. Examples of such systems include clusters of computers. Figure 4.6 illustrates a general scheme of a distributed memory system.

3. Parallel systems with virtually shared memory. Correspond to a hybrid organization which provides a virtually shared memory over a physically distributed memory.

Shared-memory multiprocessor systems use a programming model based on the concurrent execution of multiple threads within the same address space. A thread is an execution context with limited resources running over one core or processor, where variables and other resources are usually shared with other threads [100] [130].

Multithreading programming allows to improve the performance and concurrency of the system. Among the disadvantages are the complexity of programming (since the threads must be synchronized to access shared variables) [7] [100], the low scalability, difficulty of debugging, the limited amount of data to process and the size of memory.

Multicomputer systems are a powerful alternative to process massive amount of data into a distributed environment. Autonomous processing units with local memory are connected with each other by a very fast interconnection network to execute parallel programs that use the message passing paradigm. Among the advantages of the distributed memory systems are the high availability of memory, reliability, highly scalable and the sharing data problem is solved by the message passing. Among the disadvantages are the high complexity of programming, the efficiency of communication highly depends of the interconnection network, load balancing problem and cost.

![Diagram of a shared memory system](image)

Figure 4.5: Shared memory system. P: processor; M: memory
4.1.3 Interconnection networks

The communication among processors can be performed by using either a shared memory system or by using a message passing communication in distributed systems. In the first kind of system the processors use the interconnection network to access a shared resource (main memory, cache memory). For example, modern multicore processors have some on-chip interconnection network [2] to facilitate the communication among cores and cache memories. The cores are either connected via a crossbar, by a 2-D mesh, by a multiple bus system, etc [143]. In the distributed systems the processors use the interconnection network to send messages to remote processors. For example, multicomputer systems such as clusters of computers use the interconnection network to establish communication between nodes by using message passing.

The topology of interconnection network can be either static (direct) or dynamic (indirect) [177] [163]. In static networks, processors are connected by using fixed point-to-point links which do not change when the parallel program is running [115]. See examples in Figure 4.7. Static networks are more appropriate for problems with predictable communication patterns.

In dynamic networks the processors are connected with non-fixed links by using switches that can be reconfigured to satisfy the required communication pattern. The switches connect processors to either processors or memory modules [115]. The topology of the dynamic network can change during the execution of the parallel program. See some examples in Figure 4.8. Dynamic networks allow for major flexibility in the interconnection among processors because the routing in the network is managed by hardware instead of software.
4.2 Design of Parallel Algorithms

A sequential algorithm is a sequence of steps used to solve a given problem by using a high-performance single-core processor. On the contrary, a parallel algorithm is a set of steps distributed on different processing units to solve a problem by using either high-performance multi-core processors or multicomputer systems based on a message passing paradigm.

Parallel algorithms must be carefully designed to run over multicore computers and multicomputer systems [177]. The performance of the parallel algorithm must improve as the number of processors and the size of data increase.

To design parallel algorithms several strategies are used [100]:

- Identifying smaller (independent) units of a task that can be executed in parallel. The goal is to improve the algorithm performance over a parallel computer.
• Mapping smaller parallel units of the initial task onto multiple parallel processes.

• Identifying parts of the parallel program that require synchronization when shared data is used.

• Identifying the communication levels between the different parallel units if the message passing paradigm is used.

4.2.1 Foster’s design methodology

Ian Foster proposed a design methodology of parallel algorithm that considers the four-step process [83] [177] [163] described below.

Partitioning. Partitioning technique is the process of dividing the problem (computation and data) into smaller disjoint pieces [83] [177] to avoid the replication of data and computation (see Figure 4.9 (a)). The partitioning process can be performed by using either domain decomposition (data-centric approach) or functional decomposition (computation-centric approach).

Domain decomposition divides the input data into smaller data sets and then computation is then associated with each data set on which they operate. The result obtained is a fixed number of tasks, each having data and a set of operations on the data. A good partition method adequately associates both computations and data.

Functional decomposition is a complementary strategy where the computation is divided into smaller computation pieces or disjoint tasks. The data is then associated with individual tasks.

Communication. Once the primitives tasks have been identified, the next step is to identify the patterns and the levels of communication among the tasks (see Figure 4.9 (b)). The tasks that concurrently run can require data from another task. In this case the tasks can not be executed independently and the information must be transferred between neighboring tasks in order to perform a computation. The information that flows between neighboring tasks are specified in this step of the design [83] [177].

The level of coupling (tightly or weakly coupled) and the quantity of data transferred among the tasks are related to the levels of communication. Tightly coupled communication refers to a high flow of data between two tasks, whereas weakly coupled communication refers to a low flow of data.

Agglomeration. Agglomeration is the process of grouping tasks into larger tasks [83] for the parallel program performance improvement, reducing the number of messages among tasks, minimizing the communication overhead and increasing the locality. We can also reduce the number of tasks to exactly the same number of available physical processors.

Mapping. The main objective of the mapping step is to assign tasks to physical processors. In a shared memory system the operating system automatically maps processes to physical
processors by using scheduling algorithms [177] without the intervention of the programmer. In distributed systems the mapping task is carried out by the parallel program where the objectives are to both maximize the processor utilization (maximizing efficiency) and minimize the interprocessor communication (minimizing communication overhead). See Figure 4.9 (d). The mapping problem is known to be NP-hard [177].

Figure 4.9 illustrates the Foster’s parallel algorithm design methodology. The input problem is (a) subdivided in small pieces or tasks; (b) the communication patterns are defined to indicate which tasks communicate; (c) combine primitive tasks into larger tasks to map them onto physical processors; (d) assign larger tasks to physical processors to achieve a maximum use of the processors and minimize communication overhead. When there are tasks with a high level of communication, they must be assigned to a same processor.

![Diagram of Foster’s parallel algorithm design methodology](image)

Figure 4.9: Foster’s parallel algorithm design methodology

### 4.2.2 Design objectives for better performance

When designing shared memory parallel algorithm some design objectives must be reached:

- **Maximizing parallelism.** Using many fine-grained tasks rather than only a few coarse-grained tasks helps keep more CPUs busy, improves load balancing, locality and scalability, decreases the percentage of time that CPUs must idly wait for one another, and leads to greater throughput. This form of maximization of the parallelism is feasible when the level of coupling between the fine-grained tasks is low.

- **Minimizing overhead.** Overhead produced by computation, data communication and synchronization must be minimized [214].


• Maximizing locality. Good locality of reference is crucial because of the latency of a non-local memory access. Achieving this objective may significantly improve the parallel algorithm performance on modern cached processors.

4.2.3 Levels of granularity

The level of granularity can be determined by either the size of the pieces (subtasks) in a parallel application, or by the amount of data processed on each piece [100]. In addition, two levels of granularity are considered [6] [130]:

1. Fine grain. The problem is divided into a large number of small tasks [100] [130] whose number can be bigger than the number of processors. In distributed systems a large task is subdivided into smaller subtasks which can have a high level of communication. This high level of communication can lead to a high level of coupling.

2. Coarse grain. The problem is decomposed into N big subtasks [100] [130], where N can correspond to the quantity of processors in the system.

4.3 Parallelization with shared memory architectures

Multicore architectures integrate all the cores (or CPUs) on a single chip or die (Chip MultiProcessor, CMP). The main advantage of this model is that the latency of communication between all the cores in the chip is reduced, compared with old multiprocessor architectures (the processors communicate via a common bus). A single chip that contains all the cores, consumes less energy than a multiprocessor system.

However, from the point of view of the computer architecture [129] [164], the multicore processors differ greatly from one another, which affects the design and performance of the parallel algorithms. In particular, the varied multicore architectures use different types of interconnection networks to communicate cores with cache memory (L1, L2, L3) which can be either private or shared [143].

Cache memory is a small and fast memory which allows the transfer of data between the processor and main memory and alleviates the pressure on the memory bandwidth [143]. The data stored in cache is always a copy and a subset of the most frequently used main memory locations. When a data in cache is modified, the data must be updated in main memory.

Most of the modern multicore processors include a multi-level cache hierarchy with three cache levels: L1, L2 and L3 [143] to reduce the average memory access time. Figure 4.10 illustrates the different levels of memories including the main memory.
Accessing data and fetching instructions from cache memories are ever faster than accessing and fetching from RAM memory. Cache design is different on various processors because case level numbers change. On the other hand, the level of data sharing is different, in some architectures, L1 and L2 levels are private for one core and L3 cache tends to be shared by all the cores located on the processor.

Cache is used to maintain copies of recently used data and instructions by the processor. Data is fetched from main memory and stored in cache as an anticipation that they will be used by the processor in a very close future.

Regarding to the locality, we can distinguish:

- **Temporal locality.** When a recently used set of data or instructions tend to be reused by the processor.
- **Spatial locality.** When the processor accesses a set of data or instructions which are close to elements that were recently used.

One of the primary functions of the cache memory system is to take advantage of this temporal and spatial locality characteristics of a program. In general, three levels of cache memory are found in the modern architectures:

- **Level L1 cache.** It is a small but fast memory. Today the size of a L1 cache is sometimes as small as 32KB. L1 cache is usually located inside the chip of the processor and its main function is maintain the most recently used bytes of data or instructions. In most of the modern architectures, L1 cache memory is considered as a private memory of the cores.

- **Level L2 cache.** It is bigger in size and slower than L1 cache. It is also located inside the chip of the processor and its size is sometimes as small as 256KB. The main function of L2 cache is to maintain the most recently used data or instructions that are the closer vicinity of data maintained by L1 cache.

- **Level L3 cache.** It is much bigger in size (close to 8MB) and slower than L1 and L2 caches. L3 cache is located outside the chip of the processor.

Figure 4.11 (a) and (b) shows examples of two different architectures with two and three cache memories levels, respectively.
Figure 4.11: Cache memory system. (a) Multicore architecture with two cache memory levels (L1, L2); (b) Multicore architecture with three cache memory levels (L1, L2, L3)

4.4 Parallelization with distributed memory architectures

A parallel system with distributed memory is composed of two or more nodes where each node has a private memory with local address space [100] [130]. Communication between nodes of the system is performed by the message passing paradigm [59] [100].

In general, there are two well-known programming models on parallel distributed systems that can be considered: SPMD (Single Program, Multiple Data) where each node executes the same program (a binary program), but with different data sets; and MPMD (Multiple Program, Multiple Data) where each node runs different programs with different data sets.

When we develop a parallel distributed application, two important issues must be considered: workload assignment and levels of coupling between the nodes (processors).

4.4.1 Workload assignment and load balancing

In the literature, several strategies for load balancing and workload assignment are described and studied. The main criteria used is that the processors must be ever occupied. Here we emphasize three types of strategies for workload assignment:

1. Static techniques. A master processor divides the input problem (preprocessing step) into subproblems and distributes them to the nodes. The workload assignment is maintained during the entire of refinement process. A master-slave scheme exists where a master processor controls the workload assignment to slave processors [244] [251].

2. Dynamic techniques. The number of initial tasks can be either variable in the time or bigger than the number of processors. The workload is adjusted and assigned to processors as the parallel program proceeds [244] [251]. We can distinguish two types of dynamic workload: centralized and decentralized.

3. Adaptive techniques for fully distributed workload assignment. Data migration can be used to improve the efficiency of the system and the workload is dispatched according
to the performance of the network [160]. The processors can distribute tasks to other processors during the execution of the parallel program to avoid idle processors and load imbalance. Each processor can also request workload from the other processors [251].

4.4.2 Levels of coupling

The coupling level measures the intensity of the flow of messages between nodes of the system and depends on the problem and the algorithm. It can be tightly coupled, partially coupled or even decoupled [167] [192]:

- **Tightly coupled.** Interface problems are solved as they are detected. This results on a great flow of messages between processors [192].
- **Partially coupled.** The problem is treated in stages. For example, in a first phase each node processes a set of data, postponing treatment of boundary problems to second phase until the data processing is completed in every node [192].
- **Weakly coupled or decoupled.** There is practically no communication between processors [192] [167] or this is minimum. The inherent problems of data processing are locally treated by each node.

4.5 Parallelization on mixed distributed systems

An integrated and mixed distributed system takes advantage of both core and node parallel levels that is capable of processing huge data sets [59]. This type of system has several advantages: low cost, good scalability, and capacity to deal with large amounts of data almost without memory restrictions [130]. Among the disadvantages are: complex programming [164], and that the network could become a bottleneck if the system is saturated with messages.

Mixed parallel distributed systems allow for exploiting different levels of parallelism or nested parallelism by mixing different parallel programming paradigms [213] such as multithreading and message passing. This model of parallelism allows improvement of parallel application performance on nodes with multicore processors in multicomputer systems [178].

Multicomputer systems such as clusters of computers for high performance computing (HPC) [180] offer high bandwidth, low latency of data transfers [143] and several levels of parallelism (multicore processing and distributed processing). Multicore processing is performed by autonomous nodes of the distributed system. Every node performs local shared-memory parallelism to solve a part of a global problem. The global problem is solved by the distributed parallelism by using message passing.

Some libraries used to create distributed applications are MPI (Message Passing Interface) and PVM (Parallel Virtual Machine) [178].
4.6 Performance measures for parallel algorithms

The performance of a parallel algorithm is usually measured by using the speedup and efficiency measures [116] [130] [177].

1. Speedup. It is defined as the ratio between the execution time of the sequential algorithm ($T_s$) and the time of the parallel algorithm ($T_p$) by using $p$ processors:

$$S = T_s / T_p$$

2. Efficiency. It corresponds to the ratio between speedup ($S$) and number of processors ($p$), and it measures the processor utilization, that is:

$$E = S/p$$

3. Serial Fraction. It reports the degree of gain or loss of parallelism of a parallel algorithm. If the value of this factor increases as the number of processors increases (in this case the number of cores) then it is stated that the parallel algorithm has poor performance. This factor is calculated as:

$$F = (1/S - 1/p)/(1 - 1/p)$$

The ideal speedup is equal to $p$, while the ideal efficiency is equal to 1. Note however that in practice it is common that a parallel implementation does not achieve linear speedup ($S = p$) since the parallel implementation usually requires additional overhead for the management of parallelism [182, 100].

Note also that the scalability of the parallel code can be observed by studying how the speedup changes as more cores are available. When an application scales well, the speedup should increase at the same rate (or close to it) as the amount of cores increases. That is, if we double the number of cores, the speedup should also double [31].

Thus, for an ideal and scalable parallel algorithm, the graph of the speedup versus the number of processors corresponds to a 45° straight line (this behavior is called linear), while a good and scalable behavior corresponds to an approximate straight line with angle less than 45°.

These performance measures can be affected by the amount of critical sections of an algorithm and the lock operations of the threads [130] [164] in a shared memory system. The speed-up may also depend on other factors such as the number of threads, the strategy of parallelization and the compiler [164]. On the other hand, the tasks of synchronization of threads in a concurrent system may have the following defects: competitive conditions (race conditions) to access critical sections; deadlock, where two or more threads may remain forever blocked; and livelock, where two or more threads could remain running without being able to complete the task [164] [130].
Chapter 5

Multithread Lepp-bisection algorithms in 2-dimensions

5.1 Introduction

In this chapter we discuss three parallel Lepp-bisection algorithms for iterative local refinement of triangular meshes. Thus given a conforming and quality triangular mesh $\tau$, the algorithm is iteratively and locally applied over the current triangulation to produce a conforming final mesh $\tau_f$.

For empirical testing we used two kinds of meshes: (TP1) meshes formed from sets of randomly generated points over a rectangular domain, and (TP2) an L-shaped domain over which iterative refinement is performed around a reentrant vertex (the set of triangles $S$ to be refined is located in a circular region with its center in the reentrant vertex).

Two different criteria are used for selecting elements to be refined: (1) Largest triangles, where a $\delta$ threshold length value is used to impose a bound on maximum length of the longest edge; (2) Random selection of triangles to be refined.

The parallel software was implemented by using C++ programming language and Qt library 4.7. To synchronize the threads and to control the access to shared resources, software versions based on mutexes and monitors were developed. Since, monitors are not supported by Qt library, we emulate them by using mutexes and condition variables defined by Qt library.

5.2 Data structure

A simple data structure, analogous to that used by CGAL library [27][237], was designed to model a triangle and to implement the serial and parallel algorithms since we need to manage the neighborhood relationships between the elements (triangles or tetrahedra) to
traverse the mesh. Each triangle is represented by pointers to its three vertices (in counterclockwise order) \((v_0, v_1, v_2)\), to its three ordered neighboring triangles \((t_0, t_1, t_2)\) and to its three ordered edges \((e_0, e_1, e_2)\) (see Figure 5.1). The \(i\)th vertex is opposite the \(i\)th neighboring triangle. The pointers to the neighboring triangles facilitate the navigation and the rapid access of these elements throughout the mesh.

![Triangle data structure](image)

**Figure 5.1:** Triangle data structure.

### 5.3 Serial Lepp-bisection algorithm

As discussed in section 3.2.3, given a target triangle to be refined, the serial (sequential) Lepp-bisection algorithm refines \(t_0\) by repeatedly finding \(\text{Lepp}(t_0)\), the terminal edge and the associated pair of terminal triangles \(t_{n-1}, t_n\) and then bisecting the associated terminal edge \(L\) and terminal triangles. The process finishes when \(t_0\) is destroyed.

Figure 5.2 illustrates the refinement of triangle \(t_0\) by applying the serial Lepp-bisection algorithm. Initially \(\text{Lepp}(t_0) = \{t_0, t_1, t_2, t_3\}\), \(AB\) is the terminal edge and triangles \(t_2, t_3\) are the terminal triangles (Figure 5.2 (a)). A new vertex (vertex 1) is added by the longest edge bisection of the terminal triangles (Figure 5.2 (b)). Then \(\text{Lepp}(t_0) = \{t_0, t_1, \tilde{t}_2\}\) and triangles \(\{t_1, \tilde{t}_2\}\) are the new terminal triangles. Figure 5.2 (c) shows the final mesh where the vertices 2 and 3 were sequentially inserted.
Figure 5.2: (a) \( \text{Lepp}(t_0) = \{t_0, t_1, t_2, t_3\} \), with terminal triangles are \( t_2, t_3 \); (b) Once the refinement of the terminal triangles \( \{t_2, t_3\} \) is performed, \( \text{Lepp}(t_0) \) is partially recomputed and updated from \( t_1 \); (c) final refined mesh where \( t_0 \) was refined.

5.4 Multicore architectures

The parallel algorithms were tested using two parallel shared memory computers with a different number of physical processors (sockets). The first computer has a Quad Core Intel Core i7 processor with four cores in one chip (4 GB of memory, see Figure 5.3). The second computer has two Quad Core Intel Xeon E5550 processors (dual socket architecture, 23.5 GB of memory) where the set of eight cores is distributed between two sockets (see Figure 5.5). The architecture of a Quad Core Intel Xeon E5550 processor is illustrated by Figure 5.4.

The main characteristics of the architecture of Intel Core i7 and Intel Xeon E5550 processors are described by Table 5.1.

<table>
<thead>
<tr>
<th>Processor</th>
<th>NC (^1)</th>
<th>OCIN (^1)</th>
<th>L1 (^1)</th>
<th>L2 (^1)</th>
<th>L3 (^1)</th>
<th>Speed (GHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i7 920</td>
<td>4</td>
<td>Crossbar</td>
<td>256KB</td>
<td>1MB (Pr (^1))</td>
<td>8MB (Shc (^1))</td>
<td>2.67</td>
</tr>
<tr>
<td>Xeon E5550</td>
<td>8</td>
<td>Crossbar</td>
<td>256KB</td>
<td>1MB (Pr)</td>
<td>8MB (Shc (^1))</td>
<td>2.67</td>
</tr>
</tbody>
</table>

\(^1\) Processors are Intel\(^\text{®}\). NC: Number of cores; OCIN: On-chip interconnection network; L1: cache Instruction/Data 8×32KB, (total size 256KB); L2: cache size 4×256KB (total size 1MB); L3: cache size (8MB); Pr: Private cache; Shc: Shared cache

The processor Intel Core i7 has four cores on-chip (see Figure 5.3). Each core has two private L1 caches (data and instructions) and a private L2 cache. All cores share an L3 cache (see Table 5.1). The Intel Xeon E5550 processor has similar physical characteristics as the Intel Core i7 (see Figure 5.4). Both processors, Intel Core i7 and Intel Xeon E5550, belong to the Nehalem family processors.
The dual socket architecture (see Figure 5.5) has two Quad Core Intel Xeon E5550 processors. The inter-node (socket) communication and connectivity are provided by Quick Path Interconnect (QPI) interface (see Figure 5.5). The QuickPath architecture consists of a serial point-to-point interconnection network which allows communication among multiple CPU (sockets).
5.5 Multicore algorithms

Let us consider a shared memory multicore computer with $p$ physical cores, where $p << N$ and $N$ is the number of triangles. Each individual triangle $t$ in $S$ will be assigned to an individual core $P_i$ ($i = 1,..,p$). Once the refinement of $t$ is performed, the associated core will select another triangle from $S$ to continue the refinement task.

To obtain a robust implementation of a parallel algorithm over shared memory systems we need to deal with the following synchronization issues:

$SP_1$ To avoid the simultaneous processing of triangles whose Lepp collide. Collision is produced when two overlapping Lepps are simultaneously processed by two threads running on two different cores. Figure 5.6 illustrates two overlapping Lepps where $\text{Lepp}(t_0) \cap \text{Lepp}(t_0^*) = \{t_2, t_3, t_4, t_5\}$. In this case we can have a problem of shared data structure manipulation. This is called a data race in concurrent programming.

$SP_2$ To avoid refinement inconsistencies. When a couple of terminal triangles is refined, the neighborhood information consistency between the new triangles generated by the refinement and their neighboring triangles must be hold.

![Figure 5.6: Lepp Collision: $\text{Lepp}(t_0) = \{t_0, t_1, t_2, t_3, t_4, t_5\}$ and $\text{Lepp}(t_0^*) = \{t^*, t_1', t_2', t_3', t_2, t_3, t_4, t_5\}$](image)

In order to develop an efficient multicore algorithm, a randomization technique is also used. The random assignment of the triangles of $S$ to the processors improves the performance of the parallel refinement algorithm minimizing the collisions produced by parallel processing of overlapping Lepps.

Mutexes and monitors are used by different parallel algorithms as synchronization methods to control the access to shared resources between multiple threads running inside a shared memory space. In both cases, mutexes and monitors ensure that the shared data will only be processed by a single thread at a time.

A mutex provides mutual exclusion to control the access to the shared data by using an associated queue that blocks the threads that need to wait for accessing shared data. A
monitor provides mutual exclusion to control the access to the members (data and methods) of an object. This uses an entry queue, and one or more condition variables. A condition variable is used for signaling purposes and for managing the exclusive access to shared resources into the members of the object.

A mutex allows a single thread to execute a critical section (code that manages the shared resource) while the rest of threads that want to execute the same code must wait in the mutex queue. Thus a mutex synchronizes and serializes the execution of the critical section.

A monitor also controls the access to a shared resource in shared memory by synchronizing the execution of the code that manages the shared resource. This means that a single thread must execute that code (critical section) and the rest of threads that want to execute the same code must wait in the entry queue (generally the queue of a mutex). Once a thread enters to execute the critical section and verifies that shared resource is busy by another thread, the thread is locked in the queue of the condition variable of the monitor.

5.6 Multicore algorithm based on storing partial Lepps computation

Here when a thread detects a collision while computing a Lepp(t), the Lepp computation of Lepp(t) is suspended for being processed later for the same thread until no collision is detected. The set of partial Lepps remains in a local list L for later processing.

Some comments about Lepp collision issues are in order:

1. The triangles that belongs to a Lepp(t₀) must be marked as Lepp-occupied, and consequently no other thread can take any of these marked triangles until t₀ is refined.

2. If two threads P₀ and P₁ are processing in parallel triangles t₀, t₁ whose Lepp collide (see Figure 5.6), one of the threads must be freed and the computed partial Lepp is stored.

The next algorithm shows the complete procedure:

1: **PartialLeppStoringAlgorithm**(τ, S)
2: Input: τ a quality triangulation; S set of triangles to be refined.
3: Output: a refined and conforming triangulation τᵣ.
4: Initialize a list L of pending partial Lepps.
5: while (S ≠ φ) do
6: Select a non-locked triangle t from S.
7: Mark t as occupied.
8: Construct a structure Lepp and insert t into Lepp(t).
9: Take the first and the last triangle, t₀ and t of Lepp(t), respectively.
10: while (t₀ remains in τ) do
11: Update Lepp(t) from triangle t (marking the elements located in the propagation
path and inserting them into Lepp($t$)), until finding the terminal triangles \{t_e, n_e\} associated with the terminal edge $e$.

12:  Try to lock the terminal triangles \{t_e, n_e\} and their neighboring triangles.
13:  if ( Lepp($t$) collides with another one ) then
14:  Unlock the terminal triangles \{t_e, n_e\} and their neighboring triangles.
15:  Mark Lepp($t$) as not refined.
16:  Store Lepp($t$) into $L$.
17:  else
18:  Take the previous marked triangle $t_{prev}$ of the terminal triangle $t_e$.
19:  if ( $n_e == \text{null}$ ) then
20:    Refine border triangle $t_e$.
21:  else
22:    Refine triangles $t_e$ and $n_e$.
23:  end if
24:  Update neighborhood information.
25:  Unlock refined triangles and their old neighboring triangles.
26:  $t = t_{prev}$.
27:  Update $S$.
28:  end if
29: end while
30: Mark Lepp($t$) as refined.
31: end while
32: RefineListOfLepp($L$).
33: end

Note that a Lepp($t$) is immediately computed and refined if no collision is detected. On the contrary if a thread detects collision when computing a Lepp($t$), the partial Lepp computed (before collision) is saved in the Lepp list $L$. The partial Lepps stored in the list $L$ are processed as the last refinement step of the algorithm.

Mutexes are used to lock the terminal triangles to be refined and their neighbors, to this way assuring mutual exclusion on shared data structure. Once the terminal triangles have been refined, all the involved triangles are unlocked.

5.7 Multicore algorithm based on Lepp-Recomputation

Here the terminal triangles of the full Lepp computed and their immediate neighboring triangles are carried to the closer cache memory. Once the terminal triangles and their neighbors are located in cache memory (none of them was found locked by other thread) the terminal triangles are refined and the data structure is actualized.

To manage synchronization issues we use two simple strategies:

1. If two threads $P_0$ and $P_1$ refine in parallel triangles $t_0$, $t_0^*$, with overlapping Lepps (see
Figure 5.6), one of the threads is freed and allowed to refine another triangle, whereas the other thread continues with the refinement task with the same triangle. None triangle in the propagation path is marked as occupied, only the terminal triangles are locked for being refined.

2. The refinement of a pair of locked terminal triangles is not performed if some of its neighboring triangles are locked by other threads.

Let $S$ be the set of triangles to be refined. The next parallel algorithm, is executed in parallel by the threads to refine every triangle $t$ in $S$.

1: **LeppRecomputationAlgorithm**($\tau$, $S$)
2: Input: a quality triangulation $\tau$; $S$ set of triangles to be refined.
3: Output: a refined and conforming triangulation $\tau_f$.
4: while ($S \neq \emptyset$) do
5: Select a new non-locked triangle $t$ from $S$.
6: Lock $t$.
7: repeat
8: $r$ = RefineTriangle($t$);
9: until $t$ is refined ($r$=t) or collision is detected ($r$=0)
10: end while
11: end

1: **RefineTriangle**($t$)
2: Input: $t$ a triangle to be refined.
3: Output: a refined and conforming triangulation $\tau_f$.
4: Let $l$ be the longest edge of $t$.
5: Obtain neighboring triangle $n$ of $t$ by its longest edge $l$.
6: if ($n$ is locked) then
7: Unlock $t$. (*Collision is detected*)
8: return 0.
9: end if
10: Lock $n$.
11: if ($t$ is a terminal triangle) then
12: Obtain neighboring triangles of terminal triangles \{t,n\}.
13: if (some neighboring triangle is locked) then
14: Unlock triangles $t$ and $n$. (*Collision is detected*)
15: return 0.
16: end if
17: Lock neighboring triangles.
18: if ($t$ is a border triangle) then
19: Refine terminal triangle $t$.
20: else
21: Refine terminal triangles $t$ and $n$.
22: end if
23: Update neighborhood information.
24: Unlock $t$, $n$ and neighboring triangles.
25: Update $S$.
26: return $t$.
27: else
28: Unlock $t$. ($t$ is not a terminal triangle)
29: return RefineTriangle($n$). (Recursive call to advance inside the Lepp($t$))
30: end if
31: end

Note that this parallel algorithm only locks the terminal triangles and their immediate neighbors, then the rest of the triangles of the Lepp($t$) neither are locked and nor marked as occupied. In the case when a collision is found, the thread picks up another triangle $t$ to be refined from $S$ and proceeds to compute a new Lepp($t$).

Note that, there are two cases when the algorithm detects collisions: (1) when the neighboring triangle $n$ by the longest edge of $t$ is locked; (2) when one of the neighboring triangles of the terminal triangles $\{t,n\}$ is locked.

5.8 Edge Monitor-based algorithm

The access to a pair of terminal triangles to be refined is synchronized by using a monitor associated with the common terminal edge. This parallel algorithm locks every edge in the propagation path by using the monitor associated with each edge of the Lepp($t$) [145]. The thread remains blocked into the queue of the condition variable of the monitor while this can not catch this edge.

In this case the collisions are controlled at the edge level. Figure 5.7 illustrates a case when two threads $P_1$ and $P_2$ are concurrently computing the Lepp($t_0$) and Lepp($t_0^*$), respectively. Thread $P_1$ computes a complete Lepp($t_0$) and all the longest edges of every triangle are locked, whereas the Lepp of the thread $P_2$ (Lepp($t_0^*$)) collides with the longest edge $L$ of the triangle $t_2$ that belongs to Lepp($t_0$). In this last case, thread $P_2$ is blocked into the monitor associated with the edge $L$.

To manage synchronization issues the algorithm uses three simple strategies:

1. If two threads $P_1$ and $P_2$ refine in parallel triangles $t_0$, $t_0^*$, with overlapping Lepps (see Figure 5.7), one of the threads is blocked into the monitor associated with an edge of the Lepp, whereas the other thread continues with the refinement task with the same triangle.

2. All the longest edges of every triangle in the Lepp are locked. When a collision is detected, one of the threads is blocked into a monitor associated with one of the edges of the Lepp.

3. If a thread takes an edge that was already bisected, this abandons that edge and returns to the previous triangle $t_p$ of the Lepp to update the Lepp($t$) from $t_p$. According to Figure 5.7 the thread $P_2$ is blocked in the edge $L$, and once $L$ is bisected and freed, $P_2$...
wakes and moves back inside the Lepp($t'_0$) to update it from the triangle $t'_3$.

Figure 5.7: Lepp Collision at the edges level: thread $P_2$ is blocked in the edge $L$. Lepp($t_0$) = \{ $t_0, t_1, t_2, t_3, t_4, t_5$ \}, the longest edges of the triangles \{ $t_0, t_1, t_2, t_3, t_4, t_5$ \} are locked, and Lepp($t'_0$) = \{ $t'_0, t'_1, t'_2, t'_3, t_2, t_3, t_4, t_5$ \} where only the longest edges of the triangles \{ $t'_0, t'_1, t'_2, t'_3$ \} are locked.

Let $S$ be the set of triangles to be refined. The next parallel algorithm [145], is executed in parallel by the threads to refine every triangle $t$ in $S$.

1: **EdgeMonitorBasedAlgorithm**($\tau, S$)  
2: Input: $t$ a triangle to be refined; $S$ set of triangles to be refined.  
3: Output: a refined and conforming triangulation $\tau_f$  
4: while ( $S \neq \phi$ ) do  
5: Select a new triangle $t$ from $S$.  
6: RefinePar($t$).  
7: end while  
8: end

1: **RefinePar**($t$)  
2: Input: $t$ a triangle to be refined.  
3: Output: a refined and conforming triangulation $\tau_f$  
4: Find longest edge $l$ of $t$.  
5: Let $ml$ be the monitor of $l$.  
6: repeat  
7: if ( $l$ is bisected ) then  
8: return  
9: end if  
10: Find neighbor triangle $n$ by the longest edge of $t$.  
11: if ( $n$ == null ) then  
12: Refine $t$. ($t$ is a border triangle.)  
13: Mark $l$ as bisected.  
14: Update neighborhood information.  
15: Unlock $ml$. (Wake up blocked threads and one of them is randomly selected to be executed)
16: return
17: end if
18: if (t and n are terminal triangles) then
19: Refine triangles t, n.
20: Mark l as bisected.
21: Update neighborhood information.
22: Unlock ml. (Wake up blocked threads and one of them is randomly selected to be executed)
23: return
24: end if
25: RefinePar(n). (Recursive call to advance inside the Lepp because t is not a terminal triangle.)
26: until
27: end

If a thread takes an edge that was already bisected, the thread abandons that edge and either returns to the previous triangle of the Lepp(t) or takes a new triangle from S (see lines 7 and 8, RefinePar function). If the triangle t is a border triangle (n = null), then the algorithm performs a simple refinement (lines 12 to 16, RefinePar function). Otherwise, if the neighboring triangle n is not null, then the algorithm refines the terminal triangles t and n (lines 18 to 22, RefinePar function). If the triangle t is not a terminal triangle, then the algorithm recursively continues computing the Lepp(n) (line 24, RefinePar function).

5.9 Properties of the parallel algorithms

For the parallel algorithms, the following properties hold:

Lemma 5.1 The parallel algorithms produce the same triangulations as the serial algorithm if the refinement of triangles having more than one longest edge is consistently performed by selecting the same longest edge.

Proof. The proof is direct due to the fact that every triangle $t_0$ in the mesh has a Lepp($t_0$) = $\{t_0, t_1, \ldots, t_{n-1}\}$ where every triangle in the Lepp($t_0$) has strictly increasing longest edges (if n > 1) [198] and on the other hand both the serial and the parallel algorithms use the same strategy to compute and to obtain the longest edge of any triangle.

Lemma 5.2 The parallel algorithms behavior does not depend on the processing order of the triangles of S [17, 18, 19].
5.10 Testing problems

We consider two types of problems:

**Testing Problem TP1.** We consider Delaunay triangulations (obtained with CGAL) of sets of randomly generated points inside a convex region. Examples are shown in Figure 5.8

![Figure 5.8: Triangulation inside of a rectangular region. (a) Initial mesh; (b) Refined mesh](image)

**Testing Problem TP2.** This problem considers an L-shaped domain which contains a reentrant vertex $B$ of coordinates $(5,5)$ (see Figure 5.9).

![Figure 5.9: L-shaped domain with refinement region $R_e$.](image)

The refinement process is performed over the set of selected triangles that intersect a circular region of center $B$ and radius $r = R_e$. The triangles to be refined are selected according to a stopping tolerance $\delta$ which imposes a bound on the maximum length of longest edge allowed (i.e., longest edge $\leq \delta$). For an illustration, see Figure 5.10 where we
performed iterative refinement of all the triangles that intersect a circular region with radius $R_c = 0.3$ until obtaining triangles of size $\delta = 0.001$. We started with the initial mesh of 6 triangles of Figure 5.9.

A small refined mesh of the $L$ shaped domain (width=10, height=10) is shown in Figure 5.10. Note that the refinement concentrates in the interior of the circular region with radius $r=R_c$ where the number of triangles refined by propagation (located out of the circular region) remains low across the refinement iterations.

![Intermediate triangulation obtained by refining triangulation of Figure 5.9 over $R_c = 0.3$, center B = (5,5).]

5.11 Testing Algorithms

In this section we present empirical results related with problems TP1 and TP2 and the execution of the three parallel algorithms presented in the previous sections: (PA1) Partial Lepp storing algorithm; (PA2) Lepp recomputation algorithm; and (PA3) Edge monitor based algorithm.

5.11.1 Evaluation of algorithm behavior for problem TP1

**Partial Lepp Storing algorithm, problem TP1, Intel Xeon E5550 (two sockets, 8 cores)**

An input mesh of 100 vertices and 181 triangles was iteratively refined until obtaining 6 million triangles. The set of triangles to be refined were randomly selected. Table 5.2 summarizes statistics for 4 cores cores and Table 5.3 illustrates the performance measures for 8 cores. Note that for this architecture the parallel algorithm shows a partially scalable speedup for 4 cores, however, the algorithm deteriorates for 8 cores. Figure 5.11 illustrates the graphic of the speedup.
Table 5.2: Statistics on Partial Lepp Storing algorithm, problem TP1, Intel Xeon E5550 (4 cores).

<table>
<thead>
<tr>
<th>Number of Iteration</th>
<th>Mesh size</th>
<th>Triangles to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum Time</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>282235</td>
<td>84464</td>
<td>480663</td>
<td>632</td>
<td>1584</td>
<td>2.60</td>
<td>12</td>
</tr>
<tr>
<td>15</td>
<td>480663</td>
<td>144094</td>
<td>818241</td>
<td>1096</td>
<td>2680</td>
<td>2.59</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>818241</td>
<td>245248</td>
<td>1390903</td>
<td>1812</td>
<td>4492</td>
<td>2.59</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>1390903</td>
<td>415925</td>
<td>2360786</td>
<td>3170</td>
<td>7662</td>
<td>2.59</td>
<td>13</td>
</tr>
<tr>
<td>18</td>
<td>2360786</td>
<td>706683</td>
<td>4003109</td>
<td>5593</td>
<td>13255</td>
<td>2.59</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>4003109</td>
<td>1202290</td>
<td>6793614</td>
<td>9189</td>
<td>22444</td>
<td>2.47</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.3: Execution time and efficiency measures, Partial Lepp Storing algorithm, problem TP1, Intel Xeon E5550 (8 cores).

<table>
<thead>
<tr>
<th>% Iteration</th>
<th>Execution Time (ms)</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>2P</td>
<td>4P</td>
</tr>
<tr>
<td>15</td>
<td>2789</td>
<td>2472</td>
<td>1584</td>
</tr>
<tr>
<td>16</td>
<td>4856</td>
<td>4195</td>
<td>2680</td>
</tr>
<tr>
<td>17</td>
<td>8441</td>
<td>7010</td>
<td>4492</td>
</tr>
<tr>
<td>18</td>
<td>14664</td>
<td>11766</td>
<td>7662</td>
</tr>
<tr>
<td>19</td>
<td>25396</td>
<td>20128</td>
<td>13255</td>
</tr>
<tr>
<td>20</td>
<td>43909</td>
<td>35465</td>
<td>22444</td>
</tr>
</tbody>
</table>

Figure 5.11: Speedup Partial Lepp Storing algorithm, problem TP1, Intel Xeon E5550 (8 cores), for the last six iterations (iter15 to iter20).

The deterioration of the speedup for 8 cores is due to the fact that when a set of partial Lepps remain into the list $L$, the parallel algorithm postpones the processing of these Lepps and their associated triangles, which produces an increase in the number of collisions. This problem can be similar when the threads remain blocked. On the other hand, threads running on different cores can require by other triangles blocked into partial Lepps.
Lepp Recomputation Algorithm, problem TP1, Intel Core i7 (one socket, 4 cores)

An input mesh of 100 vertices and 185 triangles was iteratively refined until 5,304,903 triangles were obtained. Tables 5.4 and 5.5 summarize statistics on these results using 4 cores. Note that for this architecture the parallel algorithm shows efficient and scalable behavior.

Table 5.4: Statistics on Lepp Recomputation Algorithm, problem TP1. Intel Core i7 (1 socket, 4 cores).

<table>
<thead>
<tr>
<th>Number of Iteration</th>
<th>Mesh size</th>
<th>Triangles to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum. Time</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>179251</td>
<td>35729</td>
<td>291533</td>
<td>535</td>
<td>1265</td>
<td>2.81</td>
<td>12</td>
</tr>
<tr>
<td>15</td>
<td>291533</td>
<td>57738</td>
<td>473182</td>
<td>636</td>
<td>1901</td>
<td>2.81</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>473182</td>
<td>93431</td>
<td>767774</td>
<td>1454</td>
<td>3355</td>
<td>2.81</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td>767774</td>
<td>152116</td>
<td>1245812</td>
<td>1661</td>
<td>5016</td>
<td>2.81</td>
<td>13</td>
</tr>
<tr>
<td>18</td>
<td>1245812</td>
<td>247221</td>
<td>2022205</td>
<td>2915</td>
<td>7931</td>
<td>2.81</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>2022205</td>
<td>399801</td>
<td>3275606</td>
<td>4385</td>
<td>12316</td>
<td>2.80</td>
<td>12</td>
</tr>
<tr>
<td>20</td>
<td>3275606</td>
<td>647600</td>
<td>5304903</td>
<td>7121</td>
<td>19437</td>
<td>2.80</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.5: Execution time and efficiency measures, Lepp Recomputation Algorithm, problem TP1. Intel Core i7 (1 socket, 4 cores).

<table>
<thead>
<tr>
<th>‡ Iteration</th>
<th>Execution Time (ms)</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>2P</td>
<td>4P</td>
</tr>
<tr>
<td>14</td>
<td>3218</td>
<td>1942</td>
<td>1265</td>
</tr>
<tr>
<td>15</td>
<td>5240</td>
<td>3120</td>
<td>1901</td>
</tr>
<tr>
<td>16</td>
<td>8507</td>
<td>5040</td>
<td>3355</td>
</tr>
<tr>
<td>17</td>
<td>13788</td>
<td>8153</td>
<td>5016</td>
</tr>
<tr>
<td>18</td>
<td>22339</td>
<td>13239</td>
<td>7931</td>
</tr>
<tr>
<td>19</td>
<td>36184</td>
<td>21456</td>
<td>12316</td>
</tr>
<tr>
<td>20</td>
<td>58596</td>
<td>34719</td>
<td>19437</td>
</tr>
</tbody>
</table>
Figure 5.12: Speedup Lepp Recomputation Algorithm, problem TP1, Intel Core i7 (4 cores), random selection of triangles. Speedups obtained for the last seven iterations (iter14 to iter20).

The Lepp recomputation algorithm achieves a good speedup over a quadcore architecture due to the fact that the Lepp data is stored into the local cache memories of the same socket. The set of terminal triangles and their neighboring triangles are taken to local cache memories before carrying out the refinement process. This operation also locks the elements in cache memory and avoids that other threads catch them.

**Lepp Recomputation Algorithm, problem TP1, Intel Xeon E5550 (two sockets, 8 cores)**

Here we ran TP1 over a machine with two Intel Xeon E5550 processors. Tables 5.6 and 5.7 summarize the statistics on these results using 8 cores. Figure 5.13 illustrates the speedup obtained by the parallel algorithm. This parallel algorithm achieves a best speedup than the partial Lepp storing algorithm.

**Table 5.6: Statistics on Lepp Recomputation Algorithm, problem TP1. Intel Xeon E5550 (2 sockets, 8 cores).**

<table>
<thead>
<tr>
<th>Number of Iteration</th>
<th>Mesh size</th>
<th>Triangles to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum. Time</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>225557</td>
<td>74287</td>
<td>393272</td>
<td>546</td>
<td>1265</td>
<td>2.57</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>393272</td>
<td>130182</td>
<td>685933</td>
<td>821</td>
<td>2086</td>
<td>2.57</td>
<td>11</td>
</tr>
<tr>
<td>17</td>
<td>685933</td>
<td>225961</td>
<td>1193735</td>
<td>1403</td>
<td>3489</td>
<td>2.56</td>
<td>12</td>
</tr>
<tr>
<td>18</td>
<td>1193735</td>
<td>393827</td>
<td>2077391</td>
<td>2421</td>
<td>5910</td>
<td>2.56</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>2077391</td>
<td>685733</td>
<td>3610977</td>
<td>4094</td>
<td>10004</td>
<td>2.55</td>
<td>12</td>
</tr>
<tr>
<td>20</td>
<td>3610977</td>
<td>1192238</td>
<td>6270131</td>
<td>7259</td>
<td>17263</td>
<td>2.55</td>
<td>12</td>
</tr>
<tr>
<td>21</td>
<td>6270131</td>
<td>2069108</td>
<td>10876398</td>
<td>12709</td>
<td>29972</td>
<td>1.54</td>
<td>13</td>
</tr>
</tbody>
</table>
Table 5.7: Execution time and efficiency measures, Lepp Recomputation Algorithm, problem TP1. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th># Iteration</th>
<th>Execution Time (ms)</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>2P</td>
<td>4P</td>
</tr>
<tr>
<td>15</td>
<td>2789</td>
<td>2024</td>
<td>1182</td>
</tr>
<tr>
<td>16</td>
<td>4856</td>
<td>3488</td>
<td>2038</td>
</tr>
<tr>
<td>17</td>
<td>8441</td>
<td>6044</td>
<td>3491</td>
</tr>
<tr>
<td>18</td>
<td>14664</td>
<td>10231</td>
<td>6043</td>
</tr>
<tr>
<td>19</td>
<td>25396</td>
<td>17248</td>
<td>10481</td>
</tr>
<tr>
<td>20</td>
<td>43909</td>
<td>29988</td>
<td>18245</td>
</tr>
<tr>
<td>21</td>
<td>76039</td>
<td>52322</td>
<td>31632</td>
</tr>
</tbody>
</table>

![Speedup PA2, Problem TP1, Intel Xeon E5550](image)

Figure 5.13: Speedup Lepp Recomputation Algorithm, problem TP1, Intel Xeon E5550. The speedup achieves a slight improvement when 8 cores are used to execute the parallel refinement algorithm. Speedups obtained for the last seven iterations (iter15 to iter21).

Edge Monitor based algorithm, problem TP1, Intel Xeon E5550 (two sockets, 8 cores)

An input mesh of 100 vertices and 185 triangles was iteratively refined. Tables 5.8 and 5.9 summarize statistics on these results using 8 cores. Note that for this architecture this parallel algorithm shows a non-efficient behavior. Figure 5.14 illustrates the speedup obtained by parallel algorithm.
Table 5.8: Statistics on Edge Monitor based algorithm, problem TP1. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Number of Iteration</th>
<th>Mesh size</th>
<th>Mesh to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum. Time [ms]</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>262736</td>
<td>78973</td>
<td>447417</td>
<td>2946</td>
<td>7126</td>
<td>2.57</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>447417</td>
<td>134135</td>
<td>759980</td>
<td>4992</td>
<td>12118</td>
<td>2.57</td>
<td>11</td>
</tr>
<tr>
<td>17</td>
<td>759980</td>
<td>228494</td>
<td>1290991</td>
<td>8583</td>
<td>20701</td>
<td>2.56</td>
<td>12</td>
</tr>
<tr>
<td>18</td>
<td>1290991</td>
<td>387746</td>
<td>2191102</td>
<td>14298</td>
<td>34999</td>
<td>2.56</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>2191102</td>
<td>657160</td>
<td>3715077</td>
<td>24278</td>
<td>59277</td>
<td>2.55</td>
<td>12</td>
</tr>
<tr>
<td>20</td>
<td>3715077</td>
<td>1113773</td>
<td>6291772</td>
<td>40801</td>
<td>100078</td>
<td>2.55</td>
<td>12</td>
</tr>
<tr>
<td>21</td>
<td>6291772</td>
<td>1887918</td>
<td>10655540</td>
<td>69471</td>
<td>169549</td>
<td>1.54</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 5.9: Execution time and efficiency measures, Edge Monitor based algorithm, problem TP1. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Execution Time (ms)</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial 2P 4P 8P</td>
<td>2P 4P 8P</td>
<td>2P 4P 8P</td>
</tr>
<tr>
<td>15</td>
<td>2789 977 2891 7126</td>
<td>2.85 0.96 0.39</td>
<td>1.43 0.24 0.05</td>
</tr>
<tr>
<td>16</td>
<td>4856 1672 4941 12118</td>
<td>2.90 0.98 0.40</td>
<td>1.45 0.25 0.05</td>
</tr>
<tr>
<td>17</td>
<td>8441 2840 8367 20701</td>
<td>2.97 1.01 0.41</td>
<td>1.49 0.25 0.05</td>
</tr>
<tr>
<td>18</td>
<td>14664 4811 14268 34999</td>
<td>3.05 1.03 0.42</td>
<td>1.52 0.26 0.05</td>
</tr>
<tr>
<td>19</td>
<td>25396 8156 24626 59277</td>
<td>3.11 1.03 0.43</td>
<td>1.56 0.26 0.05</td>
</tr>
<tr>
<td>20</td>
<td>43909 13541 42034 100078</td>
<td>3.24 1.04 0.44</td>
<td>1.62 0.26 0.05</td>
</tr>
<tr>
<td>21</td>
<td>76039 23230 69344 169549</td>
<td>3.27 1.10 0.45</td>
<td>1.64 0.27 0.06</td>
</tr>
</tbody>
</table>

Figure 5.14: Speedup Edge Monitor based algorithm, problem TP1, Intel Xeon E5550. The speedup deteriorates when the number of cores increase. Speedups obtained for the last seven iterations (iter15 to iter21).

Figure 5.15 allows to compare the speedup achieved for the last refined meshes with the three algorithms (PA1, PA2, PA3). Clearly the speedup of the edge monitor based
algorithm deteriorates for 4 and 8 cores whereas the Lepp recomputation algorithm has a better performance for both Intel core i7 and Intel Xeon E5550 processors. The partial Lepp storing algorithm (PA1) has good behavior for 4 cores, but deteriorates for 8 cores.

![Graph of Speedup comparison, Problem TP1](image)

Figure 5.15: Speedup behavior comparison. Problem TP1. Parallel algorithms: (PA1) Partial Lepp storing algorithm; (PA2) Lepp recomputation algorithm; and (PA3) Edge Monitor based algorithm. Intel core i7 and Intel Xeon E5550 architectures.

### 5.11.2 Evaluation of algorithm behavior for problem TP2

**Partial Lepp Storing Algorithm, problem TP2, Intel Xeon E5550, (two sockets, 8 cores)**

An input mesh of 8 vertices and 6 triangles was iteratively refined until obtaining 1,213,840 triangles. Tables 5.10 and 5.11 summarize statistics for the case \( R_c = 0.3 \) and center \( B=(5,5) \) using 8 cores. Note that for this architecture the parallel algorithm shows a partially scalable speedup for 4 cores, however, the algorithm deteriorates for 8 cores. Figure 5.16 illustrates the graphic of the speedup.

<table>
<thead>
<tr>
<th>Number of Iteration</th>
<th>Mesh size to be refined</th>
<th>Triangles</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum Time</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>39744</td>
<td>38016</td>
<td>78246</td>
<td>122</td>
<td>281</td>
<td>2.07</td>
<td>14</td>
</tr>
<tr>
<td>23</td>
<td>78246</td>
<td>75780</td>
<td>154586</td>
<td>232</td>
<td>513</td>
<td>2.05</td>
<td>14</td>
</tr>
<tr>
<td>24</td>
<td>154586</td>
<td>151174</td>
<td>306534</td>
<td>473</td>
<td>986</td>
<td>2.04</td>
<td>14</td>
</tr>
<tr>
<td>25</td>
<td>306534</td>
<td>301815</td>
<td>609538</td>
<td>914</td>
<td>1900</td>
<td>2.02</td>
<td>14</td>
</tr>
<tr>
<td>26</td>
<td>609538</td>
<td>602868</td>
<td>1213840</td>
<td>1891</td>
<td>3791</td>
<td>2.02</td>
<td>14</td>
</tr>
</tbody>
</table>
Table 5.11: Execution time and efficiency measures, Partial Lepp Storing Algorithm, problem TP2, L domain, circle refinement region, $r=0.3$, $\delta=0.001$, Intel Xeon E5550, (two sockets, 8 cores)

<table>
<thead>
<tr>
<th># Iteration</th>
<th>Execution Time (ms)</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>2P</td>
<td>4P</td>
</tr>
<tr>
<td>22</td>
<td>526</td>
<td>414</td>
<td>281</td>
</tr>
<tr>
<td>23</td>
<td>1040</td>
<td>801</td>
<td>513</td>
</tr>
<tr>
<td>24</td>
<td>2065</td>
<td>1554</td>
<td>986</td>
</tr>
<tr>
<td>25</td>
<td>4108</td>
<td>3045</td>
<td>1900</td>
</tr>
<tr>
<td>26</td>
<td>8176</td>
<td>6020</td>
<td>3791</td>
</tr>
</tbody>
</table>

Figure 5.16: Speedup Partial Lepp Storing Algorithm, problem TP2, Intel Xeon E5550 (8 cores), for the last five iterations (iter22 to iter26).

Lepp Recomputation Algorithm, problem TP2, Intel Core i7, (1 socket, 4 cores)

Tables 5.12 and 5.13 summarize statistics for the case $R_c = 0.3$ and center B=(5,5) using 4 cores. The algorithm shows a good behavior with efficiency close to 0.8 until the last iteration. The speedup grows as the size of the mesh grows. Figure 5.17 illustrates the speedup obtained by parallel algorithm.

Table 5.12: Statistics on Lepp Recomputation Algorithm, problem TP2 (L domain, circle refinement region, $R_c = 0.3$), $\delta=0.001$. Intel Core i7 (1 socket, 4 cores).

<table>
<thead>
<tr>
<th>Number of Iteration</th>
<th>Mesh size</th>
<th>Triangles to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum Time [ms]</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>39744</td>
<td>38016</td>
<td>78246</td>
<td>167</td>
<td>379</td>
<td>2.07</td>
<td>14</td>
</tr>
<tr>
<td>23</td>
<td>78246</td>
<td>75780</td>
<td>154586</td>
<td>320</td>
<td>699</td>
<td>2.05</td>
<td>14</td>
</tr>
<tr>
<td>24</td>
<td>154586</td>
<td>151174</td>
<td>306534</td>
<td>506</td>
<td>1205</td>
<td>2.04</td>
<td>14</td>
</tr>
<tr>
<td>25</td>
<td>306534</td>
<td>301815</td>
<td>609538</td>
<td>1002</td>
<td>2207</td>
<td>2.02</td>
<td>14</td>
</tr>
<tr>
<td>26</td>
<td>609538</td>
<td>602868</td>
<td>1213840</td>
<td>2013</td>
<td>4220</td>
<td>2.02</td>
<td>14</td>
</tr>
</tbody>
</table>
Table 5.13: Execution time and efficiency measures, Lepp Recomputation Algorithm, problem TP2 (L domain, circle refinement region, $R_c = 0.3$), $\delta = 0.001$. Intel Core i7 (1 socket, 4 cores).

<table>
<thead>
<tr>
<th># Iteration</th>
<th>Execution Time (ms)</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>2P</td>
<td>4P</td>
</tr>
<tr>
<td>22</td>
<td>831</td>
<td>531</td>
<td>379</td>
</tr>
<tr>
<td>23</td>
<td>1652</td>
<td>1011</td>
<td>699</td>
</tr>
<tr>
<td>24</td>
<td>3281</td>
<td>1970</td>
<td>1205</td>
</tr>
<tr>
<td>25</td>
<td>6576</td>
<td>3873</td>
<td>2207</td>
</tr>
<tr>
<td>26</td>
<td>13042</td>
<td>7664</td>
<td>4220</td>
</tr>
</tbody>
</table>

Figure 5.17: Speedup Lepp Recomputation Algorithm, problem TP2. Intel Core i7 (1 socket, 4 cores), for the last five iterations (iter22 to iter26).

Lepp Recomputation Algorithm, problem TP2, Intel Xeon E5550, (two sockets, 8 cores)

Tables 5.14 and 5.15 summarize statistics obtained for the last five refinement steps with $R_c = 0.3$ using 8 cores. Note that the speedup grows as the size of the mesh grows in each iteration. Figure 5.18 illustrates the speedups when $R_C = 0.3$ for the last five iterations.

The speedup grows as the size of the mesh grows in every new iteration up to four cores and slightly grows up to 8 cores.
Table 5.14: Statistics on Lepp Recomputation Algorithm, problem TP2 (L domain, circle refinement region, $R_c = 0.3$), $\delta=0.001$. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Number of Iteration</th>
<th>Mesh size</th>
<th>Triangles to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum Time</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>39744</td>
<td>38016</td>
<td>78246</td>
<td>112</td>
<td>270</td>
<td>2.07</td>
<td>14</td>
</tr>
<tr>
<td>23</td>
<td>78246</td>
<td>75780</td>
<td>154586</td>
<td>256</td>
<td>526</td>
<td>2.05</td>
<td>14</td>
</tr>
<tr>
<td>24</td>
<td>154586</td>
<td>151174</td>
<td>306534</td>
<td>420</td>
<td>946</td>
<td>2.04</td>
<td>14</td>
</tr>
<tr>
<td>25</td>
<td>306534</td>
<td>301815</td>
<td>609538</td>
<td>809</td>
<td>1755</td>
<td>2.02</td>
<td>14</td>
</tr>
<tr>
<td>26</td>
<td>609538</td>
<td>602868</td>
<td>1213840</td>
<td>1599</td>
<td>3354</td>
<td>2.02</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 5.15: Execution time and efficiency measures, Lepp Recomputation Algorithm, problem TP2 (L domain, circle refinement region, $R_c = 0.3$), $\delta=0.001$. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Serial</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>526</td>
<td>396</td>
<td>308</td>
<td>270</td>
<td>1.33</td>
<td>1.71</td>
<td>1.95</td>
<td>0.66</td>
<td>0.43</td>
<td>0.24</td>
</tr>
<tr>
<td>23</td>
<td>1040</td>
<td>764</td>
<td>523</td>
<td>526</td>
<td>1.36</td>
<td>1.99</td>
<td>1.98</td>
<td>0.68</td>
<td>0.50</td>
<td>0.25</td>
</tr>
<tr>
<td>24</td>
<td>2065</td>
<td>1465</td>
<td>1178</td>
<td>946</td>
<td>1.41</td>
<td>1.75</td>
<td>2.18</td>
<td>0.70</td>
<td>0.44</td>
<td>0.27</td>
</tr>
<tr>
<td>25</td>
<td>4108</td>
<td>2918</td>
<td>2016</td>
<td>1755</td>
<td>1.41</td>
<td>2.04</td>
<td>2.34</td>
<td>0.70</td>
<td>0.51</td>
<td>0.29</td>
</tr>
<tr>
<td>26</td>
<td>8176</td>
<td>5816</td>
<td>3657</td>
<td>3354</td>
<td>1.41</td>
<td>2.24</td>
<td>2.44</td>
<td>0.70</td>
<td>0.56</td>
<td>0.30</td>
</tr>
</tbody>
</table>

Figure 5.18: Speedup Lepp Recomputation Algorithm, problem TP2, Intel Xeon E5550 (2 sockets, 8 cores), for the last five iterations (iter22 to iter26).

**Edge Monitor Based Algorithm, problem TP2, Intel Xeon E5550, (two sockets, 8 cores)**

Tables 5.16 and 5.17 summarize statistics obtained for the last five refinement steps with $R_c = 0.3$ using 8 cores. Note that the speedup and efficiency deteriorate as the number of
cores grows in each iteration. Figure 5.19 illustrates the speedups when \( R_C = 0.3 \) for the last five iterations.

Table 5.16: Statistics on Edge Monitor Based Algorithm, problem TP2 (L domain, circle refinement region, \( R_c = 0.3 \), \( \delta=0.001 \). Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Number of Iteration</th>
<th>Mesh size to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum Time</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>39744</td>
<td>78246</td>
<td>637</td>
<td>1264</td>
<td>2.07</td>
<td>14</td>
</tr>
<tr>
<td>23</td>
<td>78246</td>
<td>154586</td>
<td>1292</td>
<td>2556</td>
<td>2.05</td>
<td>14</td>
</tr>
<tr>
<td>24</td>
<td>154586</td>
<td>306534</td>
<td>2521</td>
<td>5077</td>
<td>2.05</td>
<td>14</td>
</tr>
<tr>
<td>25</td>
<td>306534</td>
<td>609538</td>
<td>5036</td>
<td>10113</td>
<td>2.05</td>
<td>14</td>
</tr>
<tr>
<td>26</td>
<td>609538</td>
<td>1213840</td>
<td>9962</td>
<td>20075</td>
<td>2.05</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 5.17: Execution time and efficiency measures, Edge Monitor Based Algorithm, problem TP2 (L domain, circle refinement region, \( R_c = 0.3 \), \( \delta=0.001 \). Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Execution Time (ms)</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>2P</td>
<td>4P</td>
</tr>
<tr>
<td>22</td>
<td>526</td>
<td>199</td>
<td>607</td>
</tr>
<tr>
<td>23</td>
<td>1040</td>
<td>318</td>
<td>1202</td>
</tr>
<tr>
<td>24</td>
<td>2065</td>
<td>623</td>
<td>2375</td>
</tr>
<tr>
<td>25</td>
<td>4108</td>
<td>1282</td>
<td>4826</td>
</tr>
<tr>
<td>26</td>
<td>8176</td>
<td>2625</td>
<td>9723</td>
</tr>
</tbody>
</table>

Figure 5.19: Speedup Edge Monitor Based Algorithm, problem TP2, Intel Xeon E5550 (2 sockets, 8 cores), for the last five iterations (iter22 to iter26)

For both problems (TP1 and TP2), the speedup deterioration is due to the high level of contention produced when multiple threads compete for accessing to shared data by using mutual exclusion primitives (e.g., locks) to have exclusive access to this data.
Figure 5.20 allows to compare the speedup achieved for the last refined meshes with the three algorithms (PA1, PA2, PA3). The partial Lepp storing algorithm (PA1) has good behavior for 4 cores, whereas for 8 cores the speedup deteriorates. Clearly the speedup of the edge monitor based algorithm (PA3) deteriorates for 4 and 8 cores, whereas the Lepp recomputation algorithm (PA2) has a better performance for both Intel core i7 and Intel Xeon E5550 processors. Nevertheless the Lepp recomputation algorithm (PA2) achieves a slight improvement of performance for 8 cores - very close to the performance obtained for 4 cores.

Figure 5.20: Speedup behavior comparison. Problem TP2. Parallel algorithms: (PA1) Partial Lepp storing algorithm; (PA2) Lepp recomputation algorithm; and (PA3) Edge monitor based algorithm. Intel core i7 and Intel Xeon E5550 architectures, $R_e = 0.3$ and $\delta = 0.001$.

### 5.11.3 Testing the parallel algorithms over a big input mesh for problem TP1

In order to execute this experiment we consider an input mesh with 2,999,998 vertices and 5,999,953 triangles. The mesh was created from a set of randomly generated points by using CGAL library. The experiments were executed over a double quadcore architecture based on Intel Xeon E5550 (8 cores).

**Partial Lepp storing algorithm, problem TP1, Intel Xeon E5550, (two sockets, 8 cores)**

Tables 5.18 and 5.19 summarize statistics obtained for the last two refinement steps with random selection by using the partial Lepp storing algorithm for 8 cores. Note that the speedup grows for 4 cores, but deteriorates for 8 cores. Figure 5.21 illustrates the speedup.
Table 5.18: Statistics on iterative refinement, Partial Lepp storing algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Number of</th>
<th>Mesh</th>
<th>Triangles to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum Time</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5,999,945</td>
<td>1,199,905</td>
<td>10,788,829</td>
<td>39,961</td>
<td>39,961</td>
<td>1.56</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>10,788,829</td>
<td>2,157,711</td>
<td>18,187,866</td>
<td>60,048</td>
<td>100,009</td>
<td>0.92</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 5.19: Execution time and efficiency measures, Partial Lepp storing algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th># Iteration</th>
<th>Serial</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37,375</td>
<td>25,024</td>
<td>15,593</td>
<td>39,961</td>
<td>1.49</td>
<td>2.40</td>
<td>0.94</td>
<td>0.75</td>
<td>0.60</td>
<td>0.12</td>
</tr>
<tr>
<td>2</td>
<td>92,332</td>
<td>64,918</td>
<td>52,952</td>
<td>100,009</td>
<td>1.42</td>
<td>1.74</td>
<td>0.92</td>
<td>0.71</td>
<td>0.44</td>
<td>0.12</td>
</tr>
</tbody>
</table>

Figure 5.21: Speedup Partial Lepp storing algorithm, problem TP1, Intel Xeon E5550 (2 sockets, 8 cores).

Lepp recomputation algorithm, problem TP1, Intel Xeon E5550, (two sockets, 8 cores)

Tables 5.20 and 5.21 summarize statistics obtained for the last two refinement steps with random selection by using the Lepp recomputation algorithm and 8 cores. Note that the speedup grows as the number of cores grows. Figure 5.22 illustrates the speedup. The speedup grows as the size of the mesh grows in every new iteration up to 8 cores.

Table 5.20: Statistics on iterative refinement, Lepp recomputation algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Number of</th>
<th>Mesh size</th>
<th>Triangles to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum Time</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5,999,953</td>
<td>1,199,411</td>
<td>10,795,697</td>
<td>12,462</td>
<td>12,462</td>
<td>2.75</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>10,795,697</td>
<td>2,158,734</td>
<td>18,202,638</td>
<td>20,874</td>
<td>33,336</td>
<td>1.50</td>
<td>18</td>
</tr>
</tbody>
</table>
Table 5.21: Execution time and efficiency measures, Lepp recomputation algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th># Iteration</th>
<th>Execution Time (ms)</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial 2P 4P 8P</td>
<td>2P 4P 8P</td>
<td>2P 4P 8P</td>
</tr>
<tr>
<td>1</td>
<td>37,375 24,612 13,396 12,462</td>
<td>1.52 2.79 3.00</td>
<td>0.76 0.70 0.37</td>
</tr>
<tr>
<td>2</td>
<td>92,332 59,846 39,991 33,336</td>
<td>1.54 2.31 2.77</td>
<td>0.77 0.58 0.35</td>
</tr>
</tbody>
</table>

Figure 5.22: Speedup Lepp recomputation algorithm), problem TP1, Intel Xeon E5550 (2 sockets, 8 cores).

**Edge Monitor Based Algorithm, problem TP1, Intel Xeon E5550, (two sockets, 8 cores)**

Tables 5.22 and 5.23 summarize statistics obtained for the last two refinement steps with random selection by using the edge monitor based algorithm and 8 cores. The speedup and efficiency deteriorate when four and eight cores are used due to the contentions and collisions of Lepp processed in parallel and locked elements by other threads. Figure 5.23 illustrates the speedup.

Table 5.22: Statistics on iterative refinement, edge monitor based algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Number of Iteration</th>
<th>Mesh size</th>
<th>Triangles to be refined</th>
<th>Final Mesh Size</th>
<th>Execution Time [ms]</th>
<th>Accum Time</th>
<th>Avg Lepp</th>
<th>Longest Lepp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5,999,953</td>
<td>1,199,078</td>
<td>10,799,798</td>
<td>80,629</td>
<td>80,629</td>
<td>2.57</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>10,799,798</td>
<td>2,161,638</td>
<td>18,223,380</td>
<td>123,434</td>
<td>204,063</td>
<td>1.50</td>
<td>18</td>
</tr>
</tbody>
</table>
Table 5.23: Execution time and efficiency measures, edge monitor based algorithm, problem TP1, random selection. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Execution Time (ms)</th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial 2P 4P 8P</td>
<td>2P 4P 8P</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>37,375 10,991 37,467 80,629</td>
<td>3.40 1.00 0.46</td>
<td>1.70 0.25 0.06</td>
</tr>
<tr>
<td>2</td>
<td>92,332 28,159 92,233 204,063</td>
<td>3.28 1.00 0.45</td>
<td>1.64 0.25 0.06</td>
</tr>
</tbody>
</table>

Figure 5.23: Speedup edge monitor based algorithm, problem TP1, Intel Xeon E5550 (2 sockets, 8 cores).

Figure 5.24 allows to compare the speedup achieved for the last refined mesh with the three algorithms (PA1, PA2, PA3) executed on Intel Xeon E5550 processors. The results obtained are similar to the results obtained in the previous experiments. The partial Lepp storing algorithm shows a partially scalable speedup for 4 cores, however, this deteriorates when 8 cores are used. The speedup of the edge monitor based algorithm deteriorates for 4 and 8 cores whereas the Lepp recomputation algorithm achieves a better performance for both Intel core i7 and Intel Xeon E5550 processors. Nevertheless the Lepp recomputation algorithm shows a slight improvement of the speedup for 8 cores, very close to the speedup obtained for 4 cores, when this is executed on Intel Xeon E5550 processors.
Figure 5.24: Speedup behavior comparison. Intel Xeon E5550, random selection. Problem TP1.

The partial Lepp storing algorithm (PA1) shows a partially scalable speedup for 4 cores, but this deteriorates for 8 cores. The deterioration of the speedup for 8 cores is due to the fact that when a set of partial Lepps remains in the list $L$, the algorithm postpones the processing of these Lepps and their associated triangles, which increases the number of collisions. On the other hand, threads running on different cores try to catch triangles blocked into partial Lepps that belong to other cores.

The Lepp recomputation algorithm (PA2) behaves well for 8 cores due to the fact that when a thread detects collisions (when it tries to catch a triangle which is locked by another thread) the thread does not remain blocked and proceeds to pick up another triangle to be refined from set $S$. This strategy avoids the blocking of the threads into the mutex queue. When the terminal triangles and their neighbors are carried to cache memory, the threads refine the terminal triangles. The experimental results show that, when the set of cores are located over different quadcore sockets, this parallel algorithm achieves a slight improvement of the performance for 8 cores, very close to the performance obtained for 4 cores. This means that the parallel algorithm achieves a partially scalable speedup.

The edge monitor based algorithm (PA3) does not behave well when the number of cores increases due to the high contention and the blocking times of the threads into the monitor associated with a shared edge. The high contention is produced when multiple threads compete for access to shared data by using mutual exclusion primitives (e.g., monitors).

The number of collisions between Lepps processed in parallel can grow as the number of cores grows, however, this problem can be addressed by using randomization where the triangles to be refined are randomly selected without requiring a previous order. In this way, the multicore algorithms take advantage of the properties of the Lepp-bisection algorithm.

The consistency of the data structures and the neighborhood information between the triangles was achieved by synchronizing the threads and by the use of local operations over the data structures.
Chapter 6

Multithread Relaxed Lepp-Delaunay algorithm in 2-dimensions

6.1 Introduction

In this chapter we develop a multicore algorithm for solving the quality triangulation problem. This is stated as follows: given a conforming Delaunay triangulation $\tau$ and a threshold angle parameter $\theta_{tol}$, we need to obtain a good quality conforming (Delaunay or quasi Delaunay) mesh $\tau_f$, where for every triangle $t$ in $\tau_f$ the smallest angle $\geq \theta_{tol}$.

To solve this problem we will process the set $S$ of bad quality triangles with the smallest angle $< \theta_{tol}$. The goal is to improve each triangle $t$ in $S$ by selecting adequate points to be inserted in the mesh.

We propose a Relaxed Lepp-Delaunay algorithm, where the delaunization step is relaxed by using a neighborhood parameter $K$ that constrains the edge flipping propagation around the terminal edge. To this end we introduce the following definition:

**Definition 6.1** For a given terminal edge $E$, the $0 - \text{Neighbor}(E)$ set includes the terminal triangles associated with $E$, while the $K - \text{Neighbor}(E)$ set includes the triangles of $(K-1) - \text{Neighbor}(E)$ union the set of adjacent triangles to $(K-1) - \text{Neighbor}(E)$ by the exterior edges of $(K-1) - \text{Neighbor}(E)$.

Figure 6.1 illustrates the previous definition.

To study the practical behavior of the algorithm, we consider two kinds of meshes:

1. Meshes created from sets of randomly generated points inside of a convex region.
2. Triangulations of Planar Straight Line Graphs (PSLG) that contain constrained edges.
Figure 6.1: Green triangles show the $K - \text{Neighbor}(E)$. (a) $0 - \text{Neighbor}(E)$ includes the terminal triangles; (b) $1 - \text{Neighbor}(E)$ includes terminal triangles ($0 - \text{Neighbor}(E)$) and their immediate neighbors; (c) $2 - \text{Neighbor}(E)$ includes $1 - \text{Neighbor}(E)$ and their immediate neighbors.

### 6.2 A serial relaxed Lepp-Delaunay algorithm

We use the $K - \text{Neighbor}(E)$ sets of the definition 6.1. The serial relaxed Lepp-Delaunay centroid algorithm (Algorithm 10) proceeds as follows: every bad quality triangle $t$ is refined by finding the Lepp($t$), terminal edge $E$, the centroid $M$ of the terminal triangles and the set $K - \text{Neighbor}(E)$. Then, if the terminal edge is a non-constrained edge, the centroid $M$ is Delaunay inserted into the mesh. Otherwise, the midpoint of the terminal edge is Delaunay inserted into the mesh.

The parameter $K$ on the relaxed Lepp-Delaunay centroid algorithm constrains the propagation of the edge-flipping operations. This means that in general the final mesh $\tau_f$ is a quasi-Delaunay mesh.
Algorithm 10 Relaxed-Lepp-Delaunay Algorithm($\tau_0$, $\theta_{tol}$, K)

Input: $\tau_0$ initial conforming mesh, threshold $\theta_{tol}$ angle tolerance and parameter K that indicates the level of neighborhood around the terminal triangles.
Output: A conforming triangulation $\tau_f$.
Find $S \subset \tau$ the set of triangles with smallest angle $< \theta_{tol}$.
while $S \neq \emptyset$ do
    Select a triangle $t$ from $S$.
    while $t$ remains in $\tau$ do
        Calculate Lepp of $t$.
        Calculate the centroid $M$ of the terminal quadrilateral.
        Calculate the $K-Neighbor(E)$ of Lepp($t$) associated with the terminal edge $E$.
        Insert the centroid $M$ into the mesh by using the relaxed Lepp-Delaunay insertion.
        Update $S$.
    end while
end while

6.3 Multithread Lepp-Delaunay algorithm.

In this thesis we used a similar strategy to that used in the chapter 5 to parallelize the relaxed Lepp-Delaunay algorithm which tries to catch the terminal triangles and their neighboring triangles to be loaded into cache memory.

To design a robust multicore algorithm we prohibit simultaneous parallel processing of pairs of triangles whose $K-Neighbor(E)$ submeshes overlap. Thus, the parallel relaxed algorithm proceeds as follows:

1. For each bad quality triangle $t$, the algorithm finds Lepp($t$), and computes the centroid $M$ of the terminal quadrilateral.
2. The $K-Neighbor(E)$ set is found.
3. If $K-Neighbor(E)$ is computed without detecting collisions, then the centroid $M$ is Delaunay inserted into the mesh. Else, the computation is stopped and the core proceeds to pick up a new triangle from $S$.

Figure 6.1 (c) shows the relaxed Delaunay insertion of the centroid $M$ into the terminal quadrilateral compounded by terminal triangles $t_4$ and $t_5$ for $K=2$. These terminal triangles have immediate neighboring triangles $t_3$, $n_0$, $n_1$ and $n_2$, and $2-Neighbor(E) = \{t_2, t_3, t_4, t_5, n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8\}$. Figure 6.2 (a) shows the creation of four new triangles $t_{41}$, $t_{42}$, $t_{51}$ and $t_{52}$, from terminal triangles $t_4$ and $t_5$. Data structure consistency between neighboring triangles is assured by updating the neighborhood information between the triangles $t_3$, $n_0$, $n_1$, $n_2$, and their new neighboring triangles $t_{41}$, $t_{42}$, $t_{51}$ and $t_{52}$. Figure 6.2 (b) illustrates the final mesh after of the Delaunay insertion of $M$. 

76
Figure 6.2: (a) Four new triangles $t_{41}$, $t_{42}$, $t_{51}$ and $t_{52}$ are created from terminal triangles $t_4$ and $t_5$ of Figure 6.1 (c); (b) Delaunay insertion of centroid $M$ with $K=2$

Algorithm 11 summarizes the parallel relaxed Lepp-Delaunay centroid algorithm:

**Algorithm 11** Multicore Lepp-Delaunay algorithm

| Input: $\tau_0$ an initial mesh, threshold angle tolerance $\theta_{tol}$, parameter $K$. |
| Output: A conforming triangulation $\tau_f$. |
| Find $S$ the set of bad quality triangles. |
| **while** $S \neq \emptyset$ **do** |
| **while** $t$ remains in $\tau$ **do** |
| Calculate Lepp of $t$. |
| **if** Lepp($t$) collides **then** |
| Destroy Lepp($t$). |
| Take a new triangle $t$ from $S$. |
| **else** |
| Calculate centroid $M$ of the terminal quadrilateral. |
| Calculate $K - \text{Neighbor}(E)$ around terminal triangles of Lepp($t$) associated to terminal edge $E$. |
| **if** $K - \text{Neighbor}(E)$ collides **then** |
| Destroy Lepp($t$). |
| Take a new triangle $t$ from $S$. |
| **else** |
| Insert the centroid $M$ into the mesh by using the relaxed Lepp-Delaunay insertion. |
| Update $S$. |
| **end if** |
| **end if** |
| **end while** |
| **end while** |
6.4 Study on the serial algorithm behavior

In this section we study the behavior of the relaxed serial Lepp-Delaunay algorithm by two practical experiments: (a) the evolution of the angle distribution and the number of the non-Delaunay triangles obtained in the final mesh for different values of K; (b) a practical comparison the relaxed serial Lepp-Delaunay algorithm and the Delaunay circumcenter-based refinement algorithms.

6.4.1 Statistics on refined meshes for different values of $K$

To study the evolution of the angle distribution and the number of the non-Delaunay triangle, we consider problem P1 on an input quasi-Delaunay mesh (obtained by using Triangle software) with 1,000,000 randomly generated points and 1,999,963 triangles. The experiments were executed over a computer with two quadcore Intel Xeon E5550 processors (8 physical cores). Final meshes were obtained for a minimum angle $\theta_{\text{tol}} = 30^\circ$ and for different values of $K$.

The results on the mesh size and percentage of non-Delaunay triangles obtained with the serial algorithm, are presented in Table 6.1. This includes the size of the final meshes, number and percentage of non-Delaunay elements and execution time for different values of $K$. Note that the final meshes have approximately the same number of elements (vertices and triangles), but the number and percentage of non-Delaunay triangles in the final meshes are different. Note that when $K=0$ the algorithm only inserts the centroid into a couple of terminal triangles without carrying out flipping edge operations with the neighboring triangles.

Table 6.1 illustrates the results obtained. In this example the percentage of non-Delaunay elements obtained decreases when the value of $K$ increases. The percentage of non-Delaunay elements remains constant when $K \geq 3$. This means that when $k \to \infty$ the algorithm assures the maintaining of a good quality mesh, a very low percentage of non-Delaunay elements and the 100% of the elements meet with the threshold of minimum angle. This means that it is sufficient to apply the algorithm for $K=3$. 
Table 6.1: Problem P1. Final meshes to different values of K and percentage of non-Delaunay triangles.

<table>
<thead>
<tr>
<th>Lepp-Delaunay</th>
<th>Vertices</th>
<th>Triangles</th>
<th>Non-Delaunay Triangles (NDT)</th>
<th>Percentage of NDT</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init Mesh → K</td>
<td>1,000,000 1,999,963 2,441</td>
<td>0.122</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3,652,384 7,295,189 849,416</td>
<td>11.64</td>
<td>96,863</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3,637,471 7,265,328 12,358</td>
<td>0.17</td>
<td>101,430</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3,631,898 7,254,185 34</td>
<td>0.00047</td>
<td>108,406</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3,631,292 7,252,961 26</td>
<td>0.00036</td>
<td>112,228</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3,631,086 7,252,589 26</td>
<td>0.00036</td>
<td>112,970</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3,631,050 7,252,520 26</td>
<td>0.00036</td>
<td>114,489</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2 summarizes the distribution of the smallest angles (between 0 and 60 degrees) in percentage for the initial and final meshes, for \( K = 0, 1, 2, 3, \ldots \) Note that all the final meshes satisfy the angle tolerance \( \theta_{\text{tol}} \).

Table 6.2: Problem P1. Distribution (in percentage) of smallest angles for different values of K and \( \theta_{\text{tol}} = 30^\circ \).

<table>
<thead>
<tr>
<th>Degrees</th>
<th>Initial Mesh</th>
<th>Final Mesh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degrees</td>
<td>Percentage of triangles</td>
<td>Percentage of triangles</td>
</tr>
<tr>
<td>0° – 10°</td>
<td>6.11</td>
<td>0</td>
</tr>
<tr>
<td>10° – 20°</td>
<td>16.99</td>
<td>0</td>
</tr>
<tr>
<td>20° – 30°</td>
<td>24.37</td>
<td>0</td>
</tr>
<tr>
<td>30° – 40°</td>
<td>25.63</td>
<td>0</td>
</tr>
<tr>
<td>40° – 50°</td>
<td>9.59</td>
<td>0</td>
</tr>
<tr>
<td>50° – 60°</td>
<td>7.32</td>
<td>0</td>
</tr>
<tr>
<td>K</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.3 summarizes the distribution of the larger angles (between 60 and 180 degrees) in percentage for the initial and final meshes, for \( K = 0, 1, 2, 3, \ldots \) Note that, as expected, the percentage of the angles larger than or equal to 120° is zero for any value of \( K \). However, when \( K > 2 \) the percentage of angles between 60 to 120 degrees remain constant.
Table 6.3: Problem P1. Distribution (in percentage) of biggest angles for different values of K and $\theta_{tot} = 30^\circ$.

<table>
<thead>
<tr>
<th>Degrees</th>
<th>60° – 80°</th>
<th>80° – 100°</th>
<th>100° – 120°</th>
<th>120° – 140°</th>
<th>140° – 160°</th>
<th>160° – 180°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Mesh</td>
<td>Percentage of triangles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60° – 80°</td>
<td>26.53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80° – 100°</td>
<td>42.59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100° – 120°</td>
<td>22.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120° – 140°</td>
<td>7.42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140° – 160°</td>
<td>1.28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>160° – 180°</td>
<td>0.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final Mesh</td>
<td>Percentage of triangles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>60° – 80°</td>
<td>80° – 100°</td>
<td>100° – 120°</td>
<td>120° – 140°</td>
<td>140° – 160°</td>
<td>160° – 180°</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>37.02</td>
<td>49.92</td>
<td>13.06</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>40.85</td>
<td>50.25</td>
<td>8.90</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>40.80</td>
<td>50.30</td>
<td>8.90</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>40.79</td>
<td>50.31</td>
<td>8.90</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>40.79</td>
<td>50.31</td>
<td>8.90</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>40.79</td>
<td>50.31</td>
<td>8.90</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This experiment shows that we can obtain good quality meshes formed by triangles with good internal angles (threshold of $\theta_{tot}$) without demanding that the final mesh be fully Delaunay. Note that for K=0 all the bad quality triangles (needle, cap, etc) are eliminated from the mesh.

6.4.2 Comparing Relaxed Lepp-Delaunay algorithm and Triangle

Here we compare the practical behavior of the relaxed serial Lepp-Delaunay algorithm with circumcenter refinement algorithms implemented in Triangle \footnote{Software developed by Jonathan Shewchuk (http://www.cs.cmu.edu/~quake/triangle.html)} [222]. The parameter K with value $\gg 2$ (K=10) was used to obtain a Delaunay mesh by using the relaxed sequential Lepp-Delaunay algorithm. The last version of Triangle implements the Ruppert’s Delaunay refinement algorithm [203, 202] with improvement techniques proposed by Ungor [240]. To compare both softwares we used input meshes associated with polygonal PSLG (Planar Straight Line Graph) geometries. The figures shown in this section were obtained with the software developed by within thesis.

Table 6.4 shows the number of vertices, number of triangles and the execution time (using the serial algorithm) obtained by refining the polygonal shape of Lake Superior using both softwares. Figure 6.3 illustrates the initial mesh, and Figures 6.4 and 6.5 show examples of the refined mesh using threshold angles of 30° and 35°, respectively. The entries of the table labeled with $\infty$ symbol means that not convergence for that angle was achieved.
Table 6.4: Lake Superior shape. K=10.

<table>
<thead>
<tr>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>931</td>
<td>972</td>
<td>1014</td>
<td>1047</td>
<td>1086</td>
<td>1152</td>
<td>1230</td>
<td>1347</td>
<td>1412</td>
<td>1589</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Triangles</td>
<td>1344</td>
<td>1425</td>
<td>1506</td>
<td>1572</td>
<td>1649</td>
<td>1780</td>
<td>1936</td>
<td>2169</td>
<td>2296</td>
<td>2647</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>19</td>
<td>18</td>
<td>18</td>
<td>20</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>1036</td>
<td>1054</td>
<td>1076</td>
<td>1100</td>
<td>1139</td>
<td>1179</td>
<td>1220</td>
<td>1322</td>
<td>1407</td>
<td>1528</td>
<td>1822</td>
<td>2283</td>
<td>4159</td>
</tr>
<tr>
<td>Triangles</td>
<td>1555</td>
<td>1591</td>
<td>1632</td>
<td>1679</td>
<td>1737</td>
<td>1837</td>
<td>2020</td>
<td>2121</td>
<td>2285</td>
<td>2523</td>
<td>3105</td>
<td>4000</td>
<td>7640</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>20</td>
<td>22</td>
<td>23</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>28</td>
<td>31</td>
<td>34</td>
<td>37</td>
<td>48</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 6.4 shows the results obtained by using Triangle software [219] [222] and the relaxed Lepp-Delaunay algorithm. Note that both algorithms produce good quality triangulations of roughly the same size for 25° to 34° $\theta_{tol}$ angle tolerance. Triangle converges up to 34° while Lepp-Delaunay converges up to 37°.

It is worth noting that the Lepp-Delaunay algorithm does not use any special strategy to reduce the number of inserted points, while that Triangle includes the complex off-center strategy to reduce the number of inserted points [240].

![Figure 6.3: Initial mesh with Lake Superior shape.](image)

![Figure 6.4: Final mesh with Lake Superior shape, using 30°.](image)
Tables 6.5 to 6.10 summarize results on the geometries illustrated in Figures 6.6 to 6.11.

<table>
<thead>
<tr>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>80</td>
<td>81</td>
<td>81</td>
<td>89</td>
<td>91</td>
<td>93</td>
<td>116</td>
<td>142</td>
<td>136</td>
<td>183</td>
<td>208</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Triangles</td>
<td>101</td>
<td>103</td>
<td>103</td>
<td>116</td>
<td>120</td>
<td>131</td>
<td>166</td>
<td>214</td>
<td>214</td>
<td>285</td>
<td>330</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>106</td>
<td>110</td>
<td>116</td>
<td>117</td>
<td>117</td>
<td>119</td>
<td>129</td>
<td>138</td>
<td>150</td>
<td>170</td>
<td>191</td>
<td>210</td>
<td>349</td>
</tr>
<tr>
<td>Triangles</td>
<td>152</td>
<td>160</td>
<td>170</td>
<td>172</td>
<td>172</td>
<td>176</td>
<td>195</td>
<td>211</td>
<td>232</td>
<td>270</td>
<td>307</td>
<td>343</td>
<td>599</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 6.5: Final mesh with Lake Superior shape, using 35°.

Figure 6.6: Key geometric shape. (a) initial mesh; (b) final mesh obtained with angle of 30°; (c) final mesh obtained with an angle of 35°
Table 6.6: A shape. K=10.

<table>
<thead>
<tr>
<th>Triangle</th>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vertices</td>
<td>64</td>
<td>69</td>
<td>70</td>
<td>74</td>
<td>74</td>
<td>76</td>
<td>89</td>
<td>92</td>
<td>103</td>
<td>110</td>
<td>141</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td></td>
<td>Triangles</td>
<td>70</td>
<td>77</td>
<td>78</td>
<td>85</td>
<td>85</td>
<td>87</td>
<td>107</td>
<td>110</td>
<td>128</td>
<td>139</td>
<td>184</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td></td>
<td>Time (ms)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

| Lepp-Delaunay | Angles | 25° | 26° | 27° | 28° | 29° | 30° | 31° | 32° | 33° | 34° | 35° | 36° | 37° |
|               | Vertices | 71  | 71  | 72  | 72  | 79  | 83  | 86  | 104 | 113 | 109 | 115 | 148 | 229 |
|               | Triangles | 82  | 82  | 83  | 83  | 95  | 101 | 104 | 131 | 143 | 135 | 144 | 197 | 335 |
|               | Time (ms) | 1   | 1   | 1   | 1   | 2   | 2   | 2   | 3   | 3   | 3   | 4   | 5   | 7   |

Figure 6.7: ‘A’ geometric shape. (a) initial mesh with; (b) Final mesh obtained with an angle of 30°.

Table 6.7: Guitar shape. K=10.

<table>
<thead>
<tr>
<th>Triangle</th>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vertices</td>
<td>203</td>
<td>206</td>
<td>215</td>
<td>221</td>
<td>240</td>
<td>259</td>
<td>262</td>
<td>288</td>
<td>285</td>
<td>327</td>
<td>493</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td></td>
<td>Triangles</td>
<td>253</td>
<td>258</td>
<td>272</td>
<td>282</td>
<td>320</td>
<td>355</td>
<td>360</td>
<td>405</td>
<td>399</td>
<td>477</td>
<td>786</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td></td>
<td>Time (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

| Lepp-Delaunay | Angles | 25° | 26° | 27° | 28° | 29° | 30° | 31° | 32° | 33° | 34° | 35° | 36° | 37° |
|               | Vertices | 232 | 240 | 251 | 255 | 273 | 286 | 294 | 309 | 346 | 370 | 405 | 520 | 737 |
|               | Triangles | 308 | 323 | 342 | 350 | 385 | 409 | 424 | 451 | 518 | 564 | 629 | 842 | 1253 |
|               | Time (ms) | 2   | 2   | 3   | 3   | 4   | 5   | 5   | 6   | 7   | 7   | 8   | 11  | 15  |
Figure 6.8: Guitar shape. (a) initial mesh; (b) Final mesh obtained with an angle of 30°.

Table 6.8: “You suck” shape. K=10.

<table>
<thead>
<tr>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>78</td>
<td>78</td>
<td>100</td>
<td>102</td>
<td>102</td>
<td>103</td>
<td>121</td>
<td>121</td>
<td>121</td>
<td>130</td>
<td>130</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Triangles</td>
<td>66</td>
<td>76</td>
<td>88</td>
<td>90</td>
<td>91</td>
<td>109</td>
<td>109</td>
<td>109</td>
<td>123</td>
<td>123</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

Lepp-Delaunay

<table>
<thead>
<tr>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>82</td>
<td>82</td>
<td>94</td>
<td>95</td>
<td>95</td>
<td>95</td>
<td>98</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>100</td>
<td>109</td>
<td>173</td>
</tr>
<tr>
<td>Triangles</td>
<td>73</td>
<td>73</td>
<td>85</td>
<td>86</td>
<td>86</td>
<td>86</td>
<td>90</td>
<td>92</td>
<td>92</td>
<td>93</td>
<td>93</td>
<td>108</td>
<td>218</td>
</tr>
<tr>
<td>Time (ms)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 6.9: You-suck shape. (a) initial mesh; (b) Final mesh obtained with an angle of 30°.
Table 6.9: Face shape. K=10.

<table>
<thead>
<tr>
<th>Triangle</th>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>58</td>
<td>61</td>
<td>83</td>
<td>85</td>
<td>91</td>
<td>95</td>
<td>100</td>
<td>99</td>
<td>101</td>
<td>114</td>
<td>120</td>
<td>294</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangles</td>
<td>72</td>
<td>76</td>
<td>106</td>
<td>108</td>
<td>118</td>
<td>122</td>
<td>131</td>
<td>128</td>
<td>130</td>
<td>154</td>
<td>164</td>
<td>480</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time (ms)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lepp-Delaunay</th>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>48</td>
<td>50</td>
<td>66</td>
<td>71</td>
<td>82</td>
<td>98</td>
<td>102</td>
<td>110</td>
<td>124</td>
<td>130</td>
<td>156</td>
<td>208</td>
<td>345</td>
<td></td>
</tr>
<tr>
<td>Triangles</td>
<td>58</td>
<td>61</td>
<td>82</td>
<td>87</td>
<td>103</td>
<td>128</td>
<td>136</td>
<td>149</td>
<td>170</td>
<td>181</td>
<td>229</td>
<td>319</td>
<td>572</td>
<td></td>
</tr>
<tr>
<td>Time (ms)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.10: Face shape. (a) initial mesh; (b) final mesh with an angle of 30°.

Table 6.10: Doublex hexagon shape. K=10.

<table>
<thead>
<tr>
<th>Triangle</th>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>170</td>
<td>174</td>
<td>182</td>
<td>182</td>
<td>189</td>
<td>217</td>
<td>224</td>
<td>239</td>
<td>248</td>
<td>276</td>
<td>506</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>Triangles</td>
<td>246</td>
<td>254</td>
<td>270</td>
<td>270</td>
<td>284</td>
<td>340</td>
<td>354</td>
<td>384</td>
<td>402</td>
<td>458</td>
<td>915</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>Time (ms)</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lepp-Delaunay</th>
<th>Angles</th>
<th>25°</th>
<th>26°</th>
<th>27°</th>
<th>28°</th>
<th>29°</th>
<th>30°</th>
<th>31°</th>
<th>32°</th>
<th>33°</th>
<th>34°</th>
<th>35°</th>
<th>36°</th>
<th>37°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>166</td>
<td>177</td>
<td>181</td>
<td>191</td>
<td>197</td>
<td>201</td>
<td>207</td>
<td>214</td>
<td>235</td>
<td>262</td>
<td>191</td>
<td>379</td>
<td>492</td>
<td></td>
</tr>
<tr>
<td>Triangles</td>
<td>238</td>
<td>260</td>
<td>268</td>
<td>288</td>
<td>300</td>
<td>308</td>
<td>320</td>
<td>334</td>
<td>376</td>
<td>430</td>
<td>488</td>
<td>664</td>
<td>887</td>
<td></td>
</tr>
<tr>
<td>Time (ms)</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
Figure 6.11: Doublex hexagon shape (a) initial mesh; (b) Final mesh with an angle of 30°.

6.5 Practical performance of the parallel Lepp-Delaunay algorithm

We consider problem P2 that consists in input Delaunay mesh with 2,999,998 randomly
generated vertices and 5,999,953 triangles. We tested the algorithm for threshold angle \( \theta_{tol} \)
equal to 25° and 30°. We refined the \( K – Neighbor(E) \) submesh for different values of K.
The parallel algorithm processes the \( K – Neighbor(E) \) set into a local cache memory to take
advantage of the memory hierarchy in multicore systems.

Table 6.11 summarizes these results (size of the final meshes, number and percentage of
non-Delaunay elements and execution time) for different values of K and \( \theta_{tol} = 25° \). Table
6.12 shows the same statistics for \( \theta_{tol} = 30° \).

Table 6.11: Problem P2. Final meshes for different values of K and percentage of Delaunay
triangles, threshold angle 25°. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>Init Mesh ( \to )</th>
<th>Vertices</th>
<th>Triangles</th>
<th>Non-Delaunay Triangles</th>
<th>Percentage</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>3,000,000</td>
<td>5,999,953</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>8,815,150</td>
<td>17,617,163</td>
<td>2,702,669</td>
<td>15.34</td>
<td>214,788</td>
</tr>
<tr>
<td>1</td>
<td>8,700,262</td>
<td>17,387,493</td>
<td>107,268</td>
<td>0.62</td>
<td>220,039</td>
</tr>
<tr>
<td>2</td>
<td>8,665,151</td>
<td>17,316,846</td>
<td>488</td>
<td>0.0028</td>
<td>224,845</td>
</tr>
<tr>
<td>4</td>
<td>8,662,200</td>
<td>17,310,966</td>
<td>48</td>
<td>0.00028</td>
<td>224,980</td>
</tr>
<tr>
<td>10</td>
<td>8,662,302</td>
<td>17,311,182</td>
<td>14</td>
<td>0.000081</td>
<td>225,015</td>
</tr>
</tbody>
</table>
Table 6.12: Problem P2. Final meshes for different values of $K$ and percentage of Delaunay triangles, threshold angle $30^\circ$. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>$K$</th>
<th>Vertices</th>
<th>Triangles</th>
<th>Non-Delaunay Triangles (NDT)</th>
<th>Percentage of NDT</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10,914,181</td>
<td>21,813,087</td>
<td>2,551,395</td>
<td>11.69</td>
<td>284,014</td>
</tr>
<tr>
<td>1</td>
<td>10,878,820</td>
<td>21,740,996</td>
<td>37,064</td>
<td>0.1704</td>
<td>301,247</td>
</tr>
<tr>
<td>2</td>
<td>10,861,628</td>
<td>21,706,610</td>
<td>46</td>
<td>0.000212</td>
<td>316,655</td>
</tr>
<tr>
<td>4</td>
<td>10,858,665</td>
<td>21,700,733</td>
<td>14</td>
<td>0.000065</td>
<td>339,814</td>
</tr>
<tr>
<td>10</td>
<td>10,858,708</td>
<td>21,700,824</td>
<td>4</td>
<td>0.000018</td>
<td>340,328</td>
</tr>
</tbody>
</table>

Tables 6.13 and 6.14 illustrate the execution time and performance measures for threshold angle $\theta_{tol} = 25^\circ$, respectively. Figures 6.12 and 6.13 illustrate the graphics for execution time and speedup, respectively.

Table 6.13: Problem P2. Execution time, threshold angle $25^\circ$. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>$K$</th>
<th>1P</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>214,788</td>
<td>143,867</td>
<td>82,133</td>
<td>67,035</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>220,039</td>
<td>171,685</td>
<td>90,001</td>
<td>66,288</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>224,845</td>
<td>184,998</td>
<td>98,114</td>
<td>67,622</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>231,879</td>
<td>230,451</td>
<td>124,258</td>
<td>70,973</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>245,764</td>
<td>393,916</td>
<td>206,416</td>
<td>108,288</td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1P</td>
</tr>
<tr>
<td>$K$</td>
<td>1P</td>
</tr>
<tr>
<td>0</td>
<td>1.49</td>
</tr>
<tr>
<td>1</td>
<td>1.28</td>
</tr>
<tr>
<td>2</td>
<td>1.22</td>
</tr>
<tr>
<td>4</td>
<td>1.01</td>
</tr>
<tr>
<td>10</td>
<td>0.62</td>
</tr>
</tbody>
</table>
Figure 6.12: Problem P2. Execution time for $K=0,1,2,4,10$, threshold angle $25^\circ$. Intel Xeon E5550, 2, 4 and 8 cores.

Figure 6.13: Problem P2. Speedup for $K=0,1,2,4,10$, threshold angle $25^\circ$. Intel Xeon E5550, 2, 4 and 8 cores.

Tables 6.15 and 6.16 illustrate the execution time and performance measures for threshold angle $\theta_{tol} = 30^\circ$. Figures 6.14 and 6.15 illustrate the graphics for execution time and speedup, respectively.
Table 6.15: Problem P2. Execution time, threshold angle 30°. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>Execution Time (ms)</th>
<th>1P</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>284,014</td>
<td>208,745</td>
<td>119,189</td>
<td>97,504</td>
</tr>
<tr>
<td>1</td>
<td>301,247</td>
<td>232,329</td>
<td>135,173</td>
<td>98,266</td>
</tr>
<tr>
<td>2</td>
<td>316,655</td>
<td>256,163</td>
<td>150,098</td>
<td>101,825</td>
</tr>
<tr>
<td>4</td>
<td>339,814</td>
<td>321,908</td>
<td>168,796</td>
<td>106,910</td>
</tr>
<tr>
<td>10</td>
<td>340,328</td>
<td>556,182</td>
<td>286,096</td>
<td>151,360</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>1P</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6.14: Problem P2. Execution time for $K=0,1,2,4,10$, threshold angle 30°. Intel Xeon E5550, 2, 4 and 8 cores.
Figure 6.15: Problem P2. Speedup for \( K=0,1,2,4,10 \), threshold angle 30°. Intel Xeon E5550, 2, 4 and 8 cores.

Note that the algorithm produces good quality quasi-Delaunay meshes (with good internal angles) by using few edge flipping operations. We obtained a parallel algorithm with a partially scalable speedup. However, the efficiency values shown in Tables 6.14 and 6.16 suggest that a high overhead is present for \( K \geq 3 \). Note that for \( K=2 \) a good performance is achieved and the percentage of non-Delaunay triangles is low (see Tables 6.11 and 6.12).

This low percentage of non-Delaunay elements produced by the Relaxed Lepp-Delaunay algorithm for \( K \geq 3 \) is due to the fact that the modified algorithm is able to produce good quality meshes where the 100% of the elements meet with the threshold of the minimum angle. As the finite element method does not need fully refined Delaunay meshes, the modified algorithm produces refined meshes which are useful for this method.

The results obtained with the modified algorithm can be extrapolated to any kind of input mesh, in general for both regular and irregular meshes.
Chapter 7

Multithread Lepp-Bisection Algorithms for 3D meshes

7.1 Introduction

In this chapter we discuss multicore Lepp-bisection algorithms for iterative local refinement of tetrahedral meshes. Three different serial algorithms for the refinement of tetrahedral meshes are discussed. A parallel algorithm for multicore architectures is introduced to refine tetrahedral meshes.

To test the algorithms we consider two classes of meshes: (a) meshes defined by the convex hull of a set of randomly generated points, and (b) meshes of irregular polyhedral objects. The initial meshes were generated by using CGAL Delaunay software [27] [237].

To select the elements to be refined, two different criteria are used: (a) selection of largest triangles, where a \( \delta \) threshold is used to impose a bound of maximum length of longest edge; (b) random selection, where the elements are randomly selected from the mesh. We show that the parallel algorithm has a good behavior in architectures where the set of cores are located on different sockets.

7.2 Longest edge refinement algorithms of tetrahedral meshes

The problem is as follows: let \( \tau \) be a conforming and quality-acceptable tetrahedral mesh, this is iteratively and locally refined, over a set \( S \in \tau \) (\( S_k \) is the set of tetrahedra to be refined in iteration \( k \)), to create a conforming mesh \( \tau_f \) by using a longest-edge bisection algorithm. For each target tetrahedron \( t \in S \), Lepp(\( t \)) and its associated terminal edges are computed. Lepp(\( t \)) corresponds to a multi-directional searching process that allows for finding a set of terminal edges and associated terminal stars [201].
For two-dimensional meshes, the Lepp-bisection refinement algorithm guarantee that the smallest angles remain bounded. This property has not yet been fully tested for three-dimensional triangulations.

7.3 Data structures

A tetrahedral mesh involves four topological elements: vertices, edges, faces and tetrahedra. A tetrahedral mesh $\tau$ is a conforming mesh, where for every couple of tetrahedra $t_i$, then $t_j$ in $\tau$, $t_i \cap t_j$ is either a vertex, a common edge, a common face, or an empty set.

Each tetrahedron (see Figure 7.1) is represented by pointers to its four vertices ($v_0$, $v_1$, $v_2$, $v_3$), to its four neighboring tetrahedra ($n_0$, $n_1$, $n_2$, $n_3$) and to its six edges ($e_0$, $e_1$, $e_2$, $e_3$, $e_4$, $e_5$). The $i$-th vertex is opposite to the $i$-th face and to the $i$-th neighboring tetrahedron. The $i$-th neighboring tetrahedron is accessed by the $i$-th face. This representation facilitates the rapid access of the elements and the navigation over the mesh and is analogous to the representation used by CGAL library [27], [237].

Every tetrahedron in the mesh has a positive orientation. The orientation of a tetrahedron is defined by the orientation of its four vertices. This means that it is positive when $v_3$ is on the positive side of the plane defined by the vertices $v_0$, $v_1$ and $v_2$ (see Figure 7.2). Note that the vertices $v_0$, $v_1$ and $v_2$ are in counter-clockwise order.

A tetrahedron $t$ has a positive orientation if the vertex $v_3$ of $t$ is located on the positive side of the plane (face) compounded by other three vertices $v_j$, $v_k$ and $v_l$. Note that the side of the plane from which the rotation appears to be counter-clockwise corresponds to the positive side of the plane (see Figure 7.2).

---

Figure 7.1: Tetrahedron structure.
7.4 Preliminary definitions

To formulate the Lepp-bisection algorithm we need some definitions:

**Definition 7.1** Given an edge $L$, we define the star of $L$ ($S(L)$) as the set of tetrahedra that shares the edge $L$. In general, a star forms a non-convex polyhedron.

We say that $S(L)$ is a closed star when the edge $L$ is an interior edge ($L$ is not visible from the exterior). $S(L)$ is an open star if $L$ is a boundary edge. Figures 7.3 illustrates a closed and an open star, respectively, where $L$ is defined by vertices $v_0$ and $v_1$ in both cases.

**Definition 7.2** An edge $L$ is said terminal edge if $L$ is the common longest edge of a every tetrahedron that shares $L$. Every tetrahedron $t$ that shares a terminal edge $L$ (is said to be a
terminal tetrahedron). In addition we call terminal star of $L$ $TS(L)$ to the set of terminal tetrahedra which share $L$.

**Lemma 7.1** For any tetrahedron $t_0$ in $\tau$, Lepp($t_0$) has a finite number of terminal edges ($L_0, L_1, \ldots, L_{n-1}$) and associated terminal stars ($TS_0, TS_1, \ldots, TS_{n-1}$) [192, 195].

Note that for two-dimensional meshes, the Lepp($t_0$) is an ordered list of increasing triangles which includes one terminal edge. However, in three-dimensional, Lepp($t_0$) is not an ordered set of tetrahedra and includes several terminal edges.

**Definition 7.3** For any tetrahedron $t_0$ in $\tau$, Lepp($t_0$) is defined as follows [192]:

(a) Lepp($t_0$) contains every immediate tetrahedron $t$ that shares the longest edge of $t_0$ with $t$, and such that longest edge of $t >$ longest edge of $t_0$.

(b) For any tetrahedron $t_i$ ($i > 0$) in Lepp($t_0$), the 3D Lepp($t_0$) also contains every tetrahedron $\tilde{t}$ not contained yet in Lepp($t_0$), such that $\tilde{t}$ shares the longest edge of $t_i$, and where longest edge of $\tilde{t} >$ longest edge of $t_i$.

**Definition 7.4** For any tetrahedron $t$, the longest-edge bisection of $t$ corresponds to the partition of $t$ by joining the midpoint of its longest edge $L$ and the two vertices opposite to the edge $L$.

The partition of tetrahedron $t$ creates a new face defined by the plane formed by the midpoint $M$ of $L$ and its two opposite vertices. The new face face$_0$ is shared by two new tetrahedra nested in tetrahedron $t$. Figure 7.4 shows the longest-edge bisection of the tetrahedron defined by vertices $(v_0, v_1, v_2, v_3)$.

![Figure 7.4: The longest-edge bisection of tetrahedron $t$, showing new face face$_0$.](image)

Figure 7.4: The longest-edge bisection of tetrahedron $(v_0, v_1, v_2, v_3)$. 

94
7.4.1 Refinement of a terminal star

The refinement of a terminal star $TS(L)$ associated with a terminal edge $L$ corresponds to the longest edge bisection of every tetrahedron $t$ of $TS(L)$ [201].

**Lemma 7.2** The refinement of a terminal star $TS$ is a very local refinement operation that produces a conforming mesh.

**Proof.** Following the definitions of terminal star and terminal edge, every terminal star in the mesh has only one terminal edge $L$ which is shared by the set of terminal tetrahedra that compound the terminal star, therefore when the terminal edge and the terminal tetrahedra are bisected this refinement process does not produce any propagation to the neighboring tetrahedra of a terminal star. This local operation produces a conforming mesh. □

Figure 7.5 (a) illustrates the refinement procedure applied over a closed terminal star. Figure 7.5 (b) shows the refinement process over an open terminal star.

![Figure 7.5](image)

Figure 7.5: (a) The longest-edge bisection applied over a closed terminal star from Figure 7.3 (a), (b) The longest-edge bisection applied over an open terminal star from Figure 7.3 (b).

7.5 Serial 3D Lepp-Bisection algorithms.

For each tetrahedron $t \in S$ to be refined, the serial Lepp-bisection algorithms finds Lepp($t$) and an associated set $W$ of terminal edges. For each terminal edge $L \in W$ and associated terminal star $TS(L)$, the longest edge bisection of every tetrahedron of the terminal star $TS(L)$ is performed to produce a conforming mesh. This process is repeated until the target tetrahedron $t$ is refined.

In this section we present three new serial 3-dimensional refinement algorithms suitable for its parallelization.
7.5.1 Full Lepp-bisection algorithm

Given a target tetrahedron $t$, the algorithm (see Algorithm 12) computes the full Lepp($t$) which allows all the full sets of terminal edges and terminal stars to be found. The terminal stars are then refined. The process is repeated until $t$ is refined. Algorithm 12 describes one refinement step.

Algorithm 12 SerialFullLeppBisectionStrategy3D($\tau$, $S$)

Input: A quality 3-dimensional triangulation $\tau$, $S$ set of tetrahedra to be refined.
Output: Final conforming triangulation $\tau_f$.

while $S \neq \emptyset$ do
  For each tetrahedron $t \in S$.
  while $t$ remains in the mesh do
    Compute Lepp($t$).
    Find all terminal edges $L_i$ of Lepp($t$).
    for each terminal edge $L_i$ do
      Refine each tetrahedron that shares $L_i$ (refine a terminal star).
    end for
    Update $S$.
  end while
end while

7.5.2 A partial Lepp algorithm

Given a target tetrahedron $t$, the algorithm computes a partial Lepp until finding one terminal edge $L$ of Lepp($t$) whose associated terminal star $TS(L)$ is immediately refined. The algorithm then proceeds to find a new terminal edge starting from $t$. Note that tetrahedron $t$ is visited several times before being refined. See Algorithm 13.

Algorithm 13 SerialLeppBisectionStrategy3D($\tau$, $S$)

Input: A quality 3-dimensional triangulation $\tau$, $S$ set of tetrahedra to be refined.
Output: Final conforming triangulation $\tau_f$.

while $S \neq \emptyset$ do
  For each tetrahedron $t \in S$.
  while $t$ remains in the mesh do
    Compute Partial Lepp($t$) to find a terminal edge $L$.
    Refine terminal edge $L$.
    Update $S$.
  end while
end while
7.5.3 Partial Lepp algorithm with partial Lepp-update

This algorithm differs from the algorithm 13 by the fact that after refining the terminal star, the partial Lepp is updated from the previous element of the last terminal tetrahedron bisected during the refinement of the terminal star. See Algorithm 14.

Algorithm 14 SerialLeppBisectionStrategy3D(τ, S)

Input: A quality 3-dimensional triangulation τ, S set of tetrahedra to be refined.
Output: Final conforming triangulation τ_f.
while S ≠ ∅ do
    Take a tetrahedron t from S.
    Compute partial Lepp(t) until a terminal edge L and associated terminal star TS(L) are found.
    Refine terminal star TS(L).
    while t remains in the mesh do
        Take the not modified element t_prev which is the previous element of the refined terminal star.
        Update Lepp(t) from tetrahedron t_prev until a terminal edge L and associated terminal star TS(L) are found.
        Refine terminal star TS(L).
    end while
    Update S.
end while

7.5.4 Properties of the 3D refinement algorithms based on longest edge bisection.

The following properties hold:

Lemma 7.3 Longest-edge refinement algorithms for 3D meshes always terminate in a finite number of steps to construct a conforming tetrahedral mesh [192, 195].

Lemma 7.4 The 3D Lepp-Bisection refinement algorithm always produces conforming meshes [192, 195].

Remarks. Lepp-bisection refinement algorithms are robust methods that maintain conforming meshes through all the refinement process. They are suitable tools for finite element applications.
7.6 Multithread 3D Lepp-bisection algorithm

Given a set $S \subset \tau$ of tetrahedra to be refined, we want to perform the parallel refinement of the tetrahedra of $S$. To this end each core $P_i (i = 1,..,p)$ will process in parallel an individual tetrahedron $t$ in $S$ and its associated Lepp until the tetrahedron $t$ is bisected.

To achieve a good performance in the parallel algorithm, we follow the next procedure: before starting the refinement process, each thread computes a complete terminal star, which together with its immediate neighbors, are loaded into the closest cache memory.

Randomization is used to select elements from the set $S$, in order to minimize the probability of collisions between two Lepp processed in parallel. Algorithm 15 shows the parallel algorithm for multicore architectures. To deal with overlapping Lepp, the algorithm takes advantage of the following results:

**Proposition 7.1** (a) Each tetrahedron $t$ has an associated Lepp$(t)$ and an associated set of terminal edges. (b) For any pair of tetrahedra $t_i, t_j$ to be refined in parallel such that Lepp$(t_i) \cap$ Lepp$(t_j) \neq \emptyset$, there exists at least one common terminal edge.

**Proof.** Part (a) follows directly from the definitions of Lepp and terminal edge. Part (b) follows from the fact that once a common tetrahedra $t$ is found, the same set of tetrahedra is identified. □

Similarly to the sequential algorithm introduced in subsection 7.5.2, the parallel algorithm finds one terminal edge $L$ and immediately refines its associated terminal star. The parallel algorithm then recomputes a partial Lepp$(t)$ to find a new terminal edge. Empirical work shows that the number of collisions produced by parallel refinement of two or more tetrahedra is low, regarding to the number of Lepp computed and the number of terminal stars.

We distinguish two kinds of collisions: (1) two Lepps collide when threads reach common non-terminal elements; (2) two Lepps collide when the threads reach a common terminal edge. Algorithm 15 describes the parallel algorithm for multicore architectures.
Algorithm 15 Multithread3DLeppBisectionStrategy(τ)

Input: A quality 3-dimensional triangulation τ, a set S of tetrahedra to be refined.
Output: Final conforming triangulation τ_f

Find S the set of tetrahedra to be refined.

while S ≠ φ do
    Take a tetrahedron t ∈ S.
    while t remains in τ do
        Compute a partial Lepp(t), until find a terminal edge L
        Compute the terminal star TS(L) associated to the terminal edge L and capture the immediate neighboring tetrahedra of TS(L).
        if Terminal star TS(L) collides with another one then
            Destroy TS.
            Take another tetrahedron t ∈ S.
        else
            Refine TS.
            Update S.
        end if
    end while
end while

7.7 Empirical testing

The empirical tests were executed using a computer with two Quad Core processors Intel Xeon E5550 of 64 bits, and 23.5GB of memory. For testing, we have considered the following problems:

T1. Initial 3D Delaunay triangulations which corresponds to the convex hull of a set of randomly generated points. See Figure 7.6.

T2. Three-dimensional solid objects with irregular shapes shown in Figures 7.7, 7.8 and 7.9.

The initial meshes were obtained by using CGAL Delaunay software [27] [237]. For performing iterative refinement we have used two strategies for selecting the tetrahedra to be refined:

(1) Iterative refinement of sets of the largest tetrahedra (length of longest edge > δ) of the initial mesh τ; and,

(2) Iterative refinement of sets of randomly selected tetrahedra. The algorithm randomly selects the tetrahedra from S to be assigned to each thread as follows: a value r between 0 to 1 is randomly computed and assigned to every tetrahedron in the mesh to perform the random selection. Given a threshold δ between 0 to 1, we select those tetrahedra that satisfy the condition r ≥ δ.
Figure 7.6: Input mesh of convex geometry for TP1, TP2 and TP3 testing problems. The tetrahedra marked with blue color correspond to the selected elements to be refined.

Figure 7.7: Shaft geometry. An object with irregular shape.

Figure 7.8: Shaft geometry with marked elements with blue color that correspond to the selected elements to be refined.
7.7.1 Behavior of the serial algorithm

To study the practical behavior of the 3D serial Lepp-bisection algorithm we consider the six testing cases described in Table 7.1. The tests were executed over an Intel Xeon E5550 processor.

Table 7.1: Test problems

\begin{tabular}{ |l| }
\hline
\textbf{TP1}: Convex polyhedron (0.5 millions), refinement of largest tetrahedra (length of longest edge > $\delta$) \\
\textbf{TP2}: Convex polyhedron (0.5 millions), refinement of random tetrahedra \\
\textbf{TP3}: Big convex polyhedron (13.3 millions), refinement of largest tetrahedra \\
\textbf{TP4}: Shaft geometry (2654), refinement of largest tetrahedra \\
\textbf{TP5}: Shaft geometry (2654), refinement of random tetrahedra \\
\textbf{TP6}: Elephant geometry (1859); refinement of largest tetrahedra \\
\hline
\end{tabular}

For the refined meshes we compute: mesh size (number of tetrahedra), number of tetrahedra added, total time of execution (in milliseconds) and the average time used to add a new tetrahedron into the mesh. The data were obtained by performing 5 to 16 iterations of refinement. We observe that 12,000 tetrahedra are added per second. Tables 7.2 to 7.7 summarize the results for the refined meshes.

Table 7.2 and Table 7.3 present refinement statistics for the same initial mesh (around 500,000 tetrahedra) of a convex polyhedron for the refinement of the largest tetrahedra (TP1) and random selection (TP2), respectively. The final meshes obtained include 11,282,761 and 10,203,417 of tetrahedra, respectively. For the TP1 test problem (Table 7.2), 200,000 largest
tetrahedra were selected by iteration with largest edge greater than a δ value. For the TP2 test problem (Table 7.3) 100,000 randomly selected tetrahedra were refined by iteration.

Tables 7.2 and Table 7.3 summarize 16 and 5 iterations of refinement, respectively. Both Tables show that the processing time for adding a new tetrahedron into the mesh remains constant (0.070 - 0.085 ms).

Table 7.2: **TP1** Convex Polyhedron. Random data (75,000 points randomly generated); refinement of 200,000 largest tetrahedra by iteration until longest edge ≤ δ; δ = 100. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>It.</th>
<th>Mesh Size</th>
<th># Added Tetrahedra</th>
<th>Total time (ms)</th>
<th>Average Time (ms) by Tetrahedron</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>503,972</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1,360,438</td>
<td>856,466</td>
<td>73,591</td>
<td>0.08</td>
</tr>
<tr>
<td>2</td>
<td>2,405,694</td>
<td>1,045,256</td>
<td>89,550</td>
<td>0.08</td>
</tr>
<tr>
<td>3</td>
<td>3,506,501</td>
<td>1,100,807</td>
<td>94,374</td>
<td>0.08</td>
</tr>
<tr>
<td>4</td>
<td>4,557,270</td>
<td>1,050,769</td>
<td>90,083</td>
<td>0.08</td>
</tr>
<tr>
<td>5</td>
<td>5,561,370</td>
<td>1,004,100</td>
<td>86,031</td>
<td>0.08</td>
</tr>
<tr>
<td>6</td>
<td>6,539,275</td>
<td>977,905</td>
<td>80,739</td>
<td>0.08</td>
</tr>
<tr>
<td>7</td>
<td>7,473,237</td>
<td>933,962</td>
<td>71,411</td>
<td>0.08</td>
</tr>
<tr>
<td>8</td>
<td>8,291,995</td>
<td>818,758</td>
<td>62,491</td>
<td>0.08</td>
</tr>
<tr>
<td>9</td>
<td>9,057,137</td>
<td>765,142</td>
<td>54,372</td>
<td>0.08</td>
</tr>
<tr>
<td>10</td>
<td>9,721,170</td>
<td>664,033</td>
<td>50,638</td>
<td>0.08</td>
</tr>
<tr>
<td>11</td>
<td>10,267,847</td>
<td>546,677</td>
<td>41,542</td>
<td>0.08</td>
</tr>
<tr>
<td>12</td>
<td>10,826,347</td>
<td>558,500</td>
<td>42,080</td>
<td>0.08</td>
</tr>
<tr>
<td>13</td>
<td>11,194,281</td>
<td>367,934</td>
<td>27,580</td>
<td>0.08</td>
</tr>
<tr>
<td>14</td>
<td>11,278,476</td>
<td>84,195</td>
<td>6,229</td>
<td>0.08</td>
</tr>
<tr>
<td>15</td>
<td>11,282,736</td>
<td>4,260</td>
<td>302</td>
<td>0.08</td>
</tr>
<tr>
<td>16</td>
<td>11,282,761</td>
<td>25</td>
<td>2</td>
<td>0.080</td>
</tr>
</tbody>
</table>

Table 7.3: **TP2** Convex Polyhedron. Random data (75,000 points randomly generated); refinement of 100,000 randomly selected tetrahedra by iteration. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>It.</th>
<th>Mesh Size</th>
<th># Added Tetrahedra</th>
<th>Total time (ms)</th>
<th>Average Time (ms) by Tetrahedron</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>503,972</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1,670,808</td>
<td>1,166,836</td>
<td>80,831</td>
<td>0.069</td>
</tr>
<tr>
<td>2</td>
<td>3,161,759</td>
<td>1,490,951</td>
<td>102,685</td>
<td>0.068</td>
</tr>
<tr>
<td>3</td>
<td>5,121,789</td>
<td>1,960,030</td>
<td>135,210</td>
<td>0.069</td>
</tr>
<tr>
<td>4</td>
<td>7,443,137</td>
<td>2,321,348</td>
<td>160,592</td>
<td>0.069</td>
</tr>
<tr>
<td>5</td>
<td>10,203,417</td>
<td>2,760,280</td>
<td>191,696</td>
<td>0.069</td>
</tr>
</tbody>
</table>

Table 7.4 summarizes refinement statistics for the behavior of the test problem TP3. A huge initial mesh of 13,504,909 tetrahedra was processed until a final mesh of 17 million tetrahedra was obtained (16 iterations of refinement).
Table 7.4: TP3 Mesh statistics. Convex polyhedron of randomly generated points; refinement of 50,000 randomly selected tetrahedra by iteration until longest edge $\leq \delta$; $\delta = 11$. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>It.</th>
<th>Mesh Size</th>
<th># Added Tetrahedra</th>
<th>Total time (ms)</th>
<th>Average Time (ms) by Tetrahedron</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13,504,909</td>
<td>538,131</td>
<td>45,662</td>
<td>0.085</td>
</tr>
<tr>
<td>1</td>
<td>14,043,040</td>
<td>298,896</td>
<td>24,418</td>
<td>0.082</td>
</tr>
<tr>
<td>2</td>
<td>14,644,899</td>
<td>302,963</td>
<td>24,588</td>
<td>0.081</td>
</tr>
<tr>
<td>3</td>
<td>14,966,833</td>
<td>321,934</td>
<td>25,943</td>
<td>0.081</td>
</tr>
<tr>
<td>4</td>
<td>15,288,352</td>
<td>321,519</td>
<td>25,849</td>
<td>0.080</td>
</tr>
<tr>
<td>5</td>
<td>15,592,634</td>
<td>304,282</td>
<td>24,474</td>
<td>0.080</td>
</tr>
<tr>
<td>6</td>
<td>15,847,491</td>
<td>254,857</td>
<td>20,464</td>
<td>0.080</td>
</tr>
<tr>
<td>7</td>
<td>16,056,472</td>
<td>208,981</td>
<td>16,755</td>
<td>0.080</td>
</tr>
<tr>
<td>8</td>
<td>16,247,115</td>
<td>190,643</td>
<td>15,280</td>
<td>0.080</td>
</tr>
<tr>
<td>9</td>
<td>16,427,632</td>
<td>180,517</td>
<td>14,448</td>
<td>0.08</td>
</tr>
<tr>
<td>10</td>
<td>16,617,437</td>
<td>189,805</td>
<td>15,165</td>
<td>0.08</td>
</tr>
<tr>
<td>11</td>
<td>16,777,497</td>
<td>160,060</td>
<td>12,844</td>
<td>0.08</td>
</tr>
<tr>
<td>12</td>
<td>16,902,314</td>
<td>124,817</td>
<td>9,985</td>
<td>0.08</td>
</tr>
<tr>
<td>13</td>
<td>16,982,671</td>
<td>80,357</td>
<td>6,437</td>
<td>0.08</td>
</tr>
<tr>
<td>14</td>
<td>17,002,240</td>
<td>19,569</td>
<td>1,525</td>
<td>0.08</td>
</tr>
<tr>
<td>15</td>
<td>17,002,295</td>
<td>55</td>
<td>4</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Tables 7.5 and 7.6 summarize statistics for the shaft geometry of Figure 7.7. Table 7.7 shows the statistics for the elephant geometry. Small initial meshes were processed until obtaining larger final meshes after executing 9 to 10 iterations of refinement.

Table 7.5 shows that 50,000 largest tetrahedra were selected by iteration with the largest edge greater than a $\delta = 3$. Table 7.6 shows that 1,000 randomly selected tetrahedra were refined by iteration. Table 7.7 shows that 1,000 largest tetrahedra were selected by iteration with the largest edge greater than a $\delta = 0.02$. 
Table 7.5: **TP4** 3D solid object. Shaft geometry; refinement of 50,000 largest tetrahedra by iteration (if possible) until longest edge $\leq \delta$; $\delta = 3$. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>It.</th>
<th>Mesh Size # Tetrahedra</th>
<th># Added Tetrahedra</th>
<th>Total time (ms)</th>
<th>Average Time (ms) by Tetrahedron</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2,654</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9,412</td>
<td>6,758</td>
<td>465</td>
<td>0.07</td>
</tr>
<tr>
<td>2</td>
<td>29,749</td>
<td>20,337</td>
<td>1,376</td>
<td>0.07</td>
</tr>
<tr>
<td>3</td>
<td>89,760</td>
<td>60,011</td>
<td>4,129</td>
<td>0.07</td>
</tr>
<tr>
<td>4</td>
<td>224,786</td>
<td>135,026</td>
<td>9,370</td>
<td>0.07</td>
</tr>
<tr>
<td>5</td>
<td>369,415</td>
<td>144,629</td>
<td>10,083</td>
<td>0.07</td>
</tr>
<tr>
<td>6</td>
<td>428,445</td>
<td>59,030</td>
<td>4,165</td>
<td>0.07</td>
</tr>
<tr>
<td>7</td>
<td>436,536</td>
<td>8,091</td>
<td>566</td>
<td>0.07</td>
</tr>
<tr>
<td>8</td>
<td>436,920</td>
<td>384</td>
<td>26</td>
<td>0.07</td>
</tr>
<tr>
<td>9</td>
<td>436,926</td>
<td>6</td>
<td>1</td>
<td>0.17</td>
</tr>
</tbody>
</table>

Table 7.6: **TP5** Mesh statistics. Shaft. Refinement of 1,000 randomly selected tetrahedra by iteration. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>It.</th>
<th>Mesh Size # Tetrahedra</th>
<th># Added Tetrahedra</th>
<th>Total time (ms)</th>
<th>Average Time (ms) by Tetrahedron</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2,654</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9,640</td>
<td>6,986</td>
<td>589</td>
<td>0.08</td>
</tr>
<tr>
<td>2</td>
<td>21,781</td>
<td>12,141</td>
<td>1,010</td>
<td>0.08</td>
</tr>
<tr>
<td>3</td>
<td>37,299</td>
<td>15,518</td>
<td>1,293</td>
<td>0.08</td>
</tr>
<tr>
<td>4</td>
<td>54,681</td>
<td>17,382</td>
<td>1,451</td>
<td>0.08</td>
</tr>
<tr>
<td>5</td>
<td>74,756</td>
<td>20,075</td>
<td>1,679</td>
<td>0.08</td>
</tr>
<tr>
<td>6</td>
<td>95,896</td>
<td>21,140</td>
<td>1,771</td>
<td>0.08</td>
</tr>
<tr>
<td>7</td>
<td>117,930</td>
<td>22,034</td>
<td>1,851</td>
<td>0.08</td>
</tr>
<tr>
<td>8</td>
<td>139,845</td>
<td>21,915</td>
<td>1,836</td>
<td>0.08</td>
</tr>
<tr>
<td>9</td>
<td>161,989</td>
<td>22,144</td>
<td>1,861</td>
<td>0.08</td>
</tr>
<tr>
<td>10</td>
<td>183,883</td>
<td>21,894</td>
<td>1,841</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 7.7: **TP6** Mesh statistics. Elephant geometry. Refinement of 1,000 randomly selected tetrahedra by iteration until longest edge $\leq \delta$; $\delta = 0.02$. Intel Xeon E5550.

<table>
<thead>
<tr>
<th>It.</th>
<th>Mesh Size # Tetrahedra</th>
<th># Added Tetrahedra</th>
<th>Total time (ms)</th>
<th>Average Time (ms) by Tetrahedron</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9,053</td>
<td>7,194</td>
<td>560</td>
<td>0.08</td>
</tr>
<tr>
<td>2</td>
<td>31,486</td>
<td>22,433</td>
<td>1,719</td>
<td>0.08</td>
</tr>
<tr>
<td>3</td>
<td>91,553</td>
<td>60,067</td>
<td>4,674</td>
<td>0.08</td>
</tr>
<tr>
<td>4</td>
<td>190,037</td>
<td>98,484</td>
<td>7,715</td>
<td>0.08</td>
</tr>
<tr>
<td>5</td>
<td>283,051</td>
<td>93,014</td>
<td>7,321</td>
<td>0.08</td>
</tr>
<tr>
<td>6</td>
<td>321,824</td>
<td>38,773</td>
<td>3,047</td>
<td>0.08</td>
</tr>
<tr>
<td>7</td>
<td>323,844</td>
<td>2,020</td>
<td>156</td>
<td>0.08</td>
</tr>
</tbody>
</table>

104
### 7.7.2 Study on the speedup and efficiency

Tables 7.8 and 7.9 include statistics on the serial time, speedup and efficiency measures. Figure 7.10 illustrates the graphic of the speedup of the multithread algorithm for the six problems (TP1 to TP6) from the Table 7.1. It is worth noting that an efficient and scalable parallel implementation was obtained. The six problems show a similar behavior. The speedups obtained are close to ideal (see Figure 7.10).

Table 7.8: Execution Time (ms) of the test problems. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th></th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1P</td>
</tr>
<tr>
<td><strong>TP1:</strong> Convex polyhedron (LE ≥ δ)</td>
<td>748,312</td>
</tr>
<tr>
<td><strong>TP2:</strong> Convex polyhedron (Random)</td>
<td>671,014</td>
</tr>
<tr>
<td><strong>TP3:</strong> Big convex polyhedron (LE ≥ δ)</td>
<td>283,841</td>
</tr>
<tr>
<td><strong>TP4:</strong> Shaft (LE ≥ δ)</td>
<td>34,925</td>
</tr>
<tr>
<td><strong>TP5:</strong> Shaft (Random)</td>
<td>34,117</td>
</tr>
<tr>
<td><strong>TP6:</strong> Elephant</td>
<td>25,192</td>
</tr>
</tbody>
</table>

Table 7.9: Performance measures of the test problems: speedup and efficiency. Intel Xeon E5550 (2 sockets, 8 cores).

<table>
<thead>
<tr>
<th></th>
<th>Speed-Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1P</td>
<td>2P</td>
</tr>
<tr>
<td><strong>TP1:</strong> 3-dim convex hull (LE ≥ δ)</td>
<td>1.82</td>
<td>3.71</td>
</tr>
<tr>
<td><strong>TP2:</strong> 3-dim convex hull (Random)</td>
<td>1.91</td>
<td>3.71</td>
</tr>
<tr>
<td><strong>TP3:</strong> 3-dim convex hull (LE ≥ δ)</td>
<td>1.81</td>
<td>3.41</td>
</tr>
<tr>
<td><strong>TP4:</strong> Shaft (LE ≥ δ)</td>
<td>1.88</td>
<td>3.69</td>
</tr>
<tr>
<td><strong>TP5:</strong> Shaft (Random)</td>
<td>1.97</td>
<td>3.83</td>
</tr>
<tr>
<td><strong>TP6:</strong> Elephant</td>
<td>1.97</td>
<td>3.83</td>
</tr>
</tbody>
</table>
7.7.3 Empirical study on the quality of refined 3D triangulations

Figures 7.11 to 7.14 illustrate results on the quality of the refined meshes for the testing problems which were considered for this study.

To measure the quality of a tetrahedron \( t \), we used the next quality measure illustrated by the following expression:

\[
Q(t) = C \frac{\text{volume}(t)}{\text{longest-edge}^3},
\]

where \( C \) is such that \( Q(t') = 1 \) for the equilateral tetrahedron \( t' \). In our case, \( C = \frac{12.0}{\sqrt{2.0}} \), then we have,

\[
Q(t) = \frac{12.0}{\sqrt{2.0}} \frac{\text{volume}(t)}{l_e^3}
\]

\( l_e \) is the length of the longest edge of the tetrahedron \( t \).

Note that for all these test problems (shown in Figures 7.11 to 7.14) the 3D Lepp-bisection algorithm tends to improve the quality distribution of the tetrahedra. The blue line represents the quality distribution of the initial mesh, whereas the red line represents the quality distribution of the final mesh. The percentage of the worst elements ranges from 0.0 to 0.1, considerably diminishes.
Figure 7.11: TP1: Three-dimensional convex hull. Refinement of largest tetrahedra. Quality distribution for initial and final meshes.

Figure 7.12: TP2: Three-dimensional convex hull. Random selection. Quality distribution for initial and refined mesh.

Figure 7.13: TP5: Shaft. Random selection. Percentage of quality by range: initial and refined mesh.
Figure 7.14: TP6: Elephant. Refinement of largest tetrahedra. Quality distribution for initial and final meshes.

The results shown in Figures 7.11, 7.12 and 7.14 indicate that when the quality of the input mesh is low the 3D Lepp-bisection algorithm improves the quality of the tetrahedra obtained in the final mesh. For example, Figures 7.11 and 7.12 show that in the initial mesh, 30% of the bad quality tetrahedra range from 0.0 to 0.1 (see Figure 2.11 on bad quality tetrahedra). The elements in this range correspond to the worst quality elements of the input mesh. The Lepp-bisection algorithm diminishes the percentage of bad quality tetrahedra close to 10% and 20%, respectively. Similar results are obtained in the test shown in Figure 7.14 where the worst quality elements are located close to 12% and after applying the Lepp-bisection algorithm the bad elements are reduced under 10%. The main difference regarding to the results obtained in Figures 7.11 and 7.12 is due to the different selection criteria of tetrahedra to be refined and the number of refinement iterations.

Figure 7.13 illustrates a different situation. The initial mesh in general presents good quality elements. When the Lepp-bisection algorithm is applied on the initial mesh the quality of the final mesh does not present important variations compared to the quality of the initial mesh.

These results show that the Lepp-bisection algorithm for refinement of tetrahedral meshes presents important improvements in the quality of the final elements. This demonstrates that the refined tetrahedral meshes produced by Lepp-bisection algorithm are suited for being used in the finite element method.

7.7.4 Size of 3D Lepp

In this section we show basic statistical information on the size of the Lepps of every tetrahedron of the mesh. Table 7.10 shows the size of the initial mesh and the quantity of memory (bytes) used by the mesh. Table 7.11 illustrates the number and percentage of tetrahedra that possess a certain Lepp size. For example, the first row of Table 7.11 indicates that there are 72 tetrahedra (0.0011%) with a Lepp size of 1 tetrahedron. The second row indicates that there are 101 tetrahedra (0.0015%) with a Lepp size of 2 tetrahedra. The third row indicates that 507,680 tetrahedra (7.52%) with a Lepp size of 3 tetrahedra, etc. The last
row indicates that there is 1 tetrahedron (1.48165e-05%) whose Lepp size is 236 tetrahedra.

Table 7.10: Basic information of the input mesh

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Tetrahedra</th>
<th>Memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>6,749,243</td>
<td>2,831,685,288</td>
</tr>
</tbody>
</table>

Table 7.11: Lepp Information

<table>
<thead>
<tr>
<th>Size of Lepp</th>
<th>Number of tetrahedra</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>72</td>
<td>0.0011</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>0.0015</td>
</tr>
<tr>
<td>3</td>
<td>507,680</td>
<td>7.52</td>
</tr>
<tr>
<td>4</td>
<td>615,107</td>
<td>9.11</td>
</tr>
<tr>
<td>5</td>
<td>464,879</td>
<td>6.89</td>
</tr>
<tr>
<td>6</td>
<td>386,726</td>
<td>5.73</td>
</tr>
<tr>
<td>7</td>
<td>401,163</td>
<td>5.94</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>11</td>
<td>294,690</td>
<td>4.37</td>
</tr>
<tr>
<td>12</td>
<td>270,737</td>
<td>4.01</td>
</tr>
<tr>
<td>13</td>
<td>247,958</td>
<td>3.67</td>
</tr>
<tr>
<td>14</td>
<td>226,213</td>
<td>3.35</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>26</td>
<td>64,071</td>
<td>0.95</td>
</tr>
<tr>
<td>27</td>
<td>58,178</td>
<td>0.86</td>
</tr>
<tr>
<td>28</td>
<td>52,438</td>
<td>0.77</td>
</tr>
<tr>
<td>29</td>
<td>47,680</td>
<td>0.71</td>
</tr>
<tr>
<td>30</td>
<td>43,291</td>
<td>0.64</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>201</td>
<td>1</td>
<td>1.48165e-05</td>
</tr>
<tr>
<td>204</td>
<td>1</td>
<td>1.48165e-05</td>
</tr>
<tr>
<td>205</td>
<td>1</td>
<td>1.48165e-05</td>
</tr>
<tr>
<td>206</td>
<td>1</td>
<td>1.48165e-05</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>226</td>
<td>1</td>
<td>1.48165e-05</td>
</tr>
<tr>
<td>236</td>
<td>1</td>
<td>1.48165e-05</td>
</tr>
<tr>
<td>Tot tet</td>
<td>6,749,243</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 7.12 shows additional basic and important information about the tetrahedral mesh such as smaller and bigger terminal stars, number of open and closed terminal stars, average Lepp, etc.
Table 7.12: Additional Lepp Information

<table>
<thead>
<tr>
<th>Item</th>
<th>size / number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest Lepp:</td>
<td>1</td>
</tr>
<tr>
<td>Biggest Lepp:</td>
<td>236</td>
</tr>
<tr>
<td>Smallest terminal star:</td>
<td>1</td>
</tr>
<tr>
<td>Biggest terminal star:</td>
<td>9</td>
</tr>
<tr>
<td>Number of closed terminal stars:</td>
<td>4,131,607</td>
</tr>
<tr>
<td>Number of open terminal stars:</td>
<td>203,875</td>
</tr>
<tr>
<td>Total quantity of terminal stars:</td>
<td>4,335,482</td>
</tr>
<tr>
<td>Maximum number of terminal edges by Lepp:</td>
<td>11</td>
</tr>
<tr>
<td>Average Lepp:</td>
<td>14.55</td>
</tr>
</tbody>
</table>

Three new serial algorithms and one parallel algorithm for refinement of tetrahedral meshes were introduced in this chapter. According to the experimental results, good quality and conforming tetrahedral meshes are obtained by applying these algorithms.

We conclude that the performance obtained from the 3D parallel refinement algorithm is better than the performance obtained for the 2-dimensional parallel algorithm. The speedup of the 3-dimensional parallel refinement algorithm is closer to the ideal speedup (see Figure 7.10) since the sequential refinement algorithm for 2-dimensional meshes is more efficient (or faster) than the 3-dimensional sequential algorithm. When we compute the speedup of a parallel algorithm, we must use the best sequential algorithm.

Note that by using randomization the number of Lepp collisions remains rather low (6 to 10% of the computed Lepp) for all the test problems.
Chapter 8

Mesh partitioning methods

8.1 Introduction

Mesh partitioning methods are used to create a set of sub-meshes to be distributed as the workload to the processors in a distributed system without changing the global mesh topology (number of elements and neighborhood relations) [34]. Partitioning is a technique used to reduce a problem into simpler subproblems. This is a critical step which can affect the performance of the parallel solution [98] [114]. The partitioning algorithm must help to achieve a balanced workload among the processors as well as to provide partitions with a good aspect ratio [92] [114] in order to minimize the interprocessor communication. The aspect ratio of a partition is related to the shape of the partition [176] and it is defined as the ratio between the longest length to the shortest length of the domain [81] [230].

Minimizing the length of interface edges (dividing lines which connect two neighboring sub-meshes) is also a desirable goal in order to minimize the inter-node communication among neighboring processors. The length of interface edges refers to the total number of interface edges located between two neighboring sub-meshes obtained by applying of a mesh partitioning algorithm.

Figure 8.1 illustrates examples of mesh subdivision: figure (a) shows the subdivision of an input mesh into eight homogeneous strip-shaped partitions. Red lines represent the set of interface edges which divide and connect the neighboring sub-meshes. Figure (b) shows an irregular geometry (PSLG) subdivided into four non-homogeneous sub-meshes.

In this thesis we use three 2-dimensional methods:

1. Automatic partitions produced by METIS and ParMETIS libraries [122, 120, 118] that use graph-based strategies.

2. A mesh partitioning algorithm which constructs strip partitions.

3. An algorithm that constructs hierarchical and recursive quadtree-based partitions [205] [206].
Each mesh partitioning algorithm provides as output a text file which contains information about the location of each triangle in the mesh. This idea was taken from METIS and ParMETIS libraries which provides a list of integer numbers, where each number indicates the partition to which each triangle belongs.

![Figure 8.1: Examples of mesh subdivision.](image)

(a)

(b)

Figure 8.1: Examples of mesh subdivision. (a) An input mesh subdivided into eight different homogeneous strip partitions; (b) An input irregular mesh (PSLG) subdivided into four partitions.

A triangle $t$ is an interface triangle if it has at least one interface edge. Figure 8.2 shows the interface triangles $t_i$ and $t_{i+1}$ which share a common interface edge $e$.

![Figure 8.2: Triangles $t_i$ and $t_{i+1}$ are located within partitions $S_i$ and $S_{i+1}$, respectively, and share a common interface edge $e$.](image)
Two types of partitioning methods are mostly used: (1) graph-based partitioning, and (2) geometric partitioning.

**Graph partitioning methods.** Given a graph $G(V,E)$, where $V$ is the set of the nodes (e.g., a node is a triangle or a tetrahedron) to be distributed to the processors and $E$ the set of edges that describes relationships between nodes in $V$. An edge $e_{ij}$ (that connects two nodes $n_i$ and $n_j$ in the graph) exists in $E$ if both nodes $n_i$ and $n_j$ exist in the graph and can be located in the same processor or distributed in different processors [176]. If the nodes $n_i$ and $n_j$ share information and they are located in different processors then this information must be transferred between the neighboring processors.

The goal is to partition the set $V$ in such a way that each processor has roughly an equal number of nodes while the length of interface edges is minimized [104] [8]. Unfortunately, mesh partitioning algorithms based on graph partitioning in general are NP-Complete problems [247] and optimal partitioning algorithms over graphs are NP-Hard problems [98], which means that efficient algorithms can be hardly found. However, in this case we can find some heuristics which can provide acceptable solutions [119, 120, 234], such as those implemented in some libraries as METIS and ParMETIS [122, 120, 118].

In the context of mesh partitioning, the input mesh is transformed or translated to a graph structure where the elements of the mesh (triangles or tetrahedra) correspond to the nodes of the graph and the edges of the mesh correspond to the edges of the graph which connect the nodes.

**Geometric partitioning methods.** These methods are based on the geometric characteristics of the mesh [107] and use the geometric information of the elements (position or location of each element inside of the mesh and on the border, connectivity information, etc) to create the sub-partitions or sub-meshes to be assigned to the processors. The elements of the mesh are located within a particular partition according to their geometric coordinates [104]. Geometric partitioning methods in general generate a set of provably good quality partitions but can produce higher costs of communicating and the quality of the straight-line cuts used as separators are not good if they are compared with other solutions such as graph partitioning. However they allow a faster execution and are easy to implement [176].

Examples of geometric partitioning methods are: Grid, BSP (binary space partition), quadrees, strip-shaped partitions, Kd-Trees, etc. [124]. Geometric partitions can also be efficiently computed in parallel environments [107]. In this chapter we use two partitioning methods based on geometric partitions: the strip-shaped partition and a quadtree method. Both of them use the centroid of the triangles to identify the partition or submesh to which every triangle belongs.
8.2 Previous mesh partitioning methods.

Jones and Plassmann [114, 113] developed a parallel mesh partitioning heuristic called unbalanced recursive bisection (URB) which generates sub-domains with good aspect ratios (good shapes) with low communication costs. The mesh partitioning process is carried out by dividing the geometry in half [247] by orthogonal cuts. This partitioning method minimizes the internode communication during the refinement process.

Castaños and Savage developed PARED [38, 36] which is an adaptive framework for solving partial differential equations (PDE) and a method of graph repartitioning by using a technique called Parallel Nested Repartitioning (PNR).

Chen and Taylor developed PART [42], an automatic mesh partitioning method which was designed to run over distributed systems. PART considers heterogeneities in the distributed system including processors and network performance for partitioning regular and irregular finite element meshes. The partitioning program uses simulated annealing which is a generic probabilistic metaheuristic for the global optimization problem based on local search [241].

De Cougny and Shephard [68] propose a Parallel Octree AFT (Advancing Front Technique) to generate a distributed coarse-grain octree using a divide-and-conquer strategy. The leaf nodes of the octree define the sub-domains (sub-meshes). Lohner [140] uses an octree in order to decompose the continuous geometry. The octree is refined and re-partitioned several times until the whole domain is meshed.

Balman [14] developed a parallel application for tetrahedral mesh refinement which uses the ParMETIS library to carry out the partition of the input tetrahedral mesh. Ito et. al [108] developed a parallel framework to generate unstructured parallel tetrahedral meshes by using an advancing front technique starting with a coarse tetrahedral mesh. A dynamic workload balancing is carried out to distribute the load among the processors. An initial partition is achieved by using METIS library.

Other mesh partitioning methods have been also discussed in previous works: Dawes et. al [67] discuss a technique to generate meshes by using a parallel bottom-up octree mesh generation that ensures a good workload balance; Zhang [256] presents a parallel distributed algorithm for adaptive refinement of tetrahedral that use a bisection method based on a newest vertex approach [212] and a mesh partitioning method based on using the METIS library; Burstedde et. al [34] propose a parallel adaptive mesh refinement and use an octree based partitioning method.

8.3 Using METIS/ParMETIS libraries

METIS [120, 119] is a set of programs for partitioning unstructured graphs, hypergraphs and geometric meshes. ParMETIS [122, 121, 118] is a parallel extension of METIS which implements partitioning routines designed to run over parallel architectures by passing message
based on MPI (Message Passing Interface) library. Both METIS and ParMETIS \(^1\) use two partitioning schemes to compute a k-way partitioning: recursive bisectioning and multilevel direct k-way partitioning \([117, 120, 119]\), to obtain good quality partitions with good aspect ratio.

METIS and ParMETIS libraries provide as output a text file (see Table 8.3) which contains a list of integer numbers where each number indicates the partition number in which each element \(t\) is located, both for two-dimensional meshes or three-dimensional meshes. Table 8.1 illustrates the input file that contains the information of the input mesh. Each element of the mesh (vertices and triangles) has a unique numerical code. Table 8.3 illustrates an example with partitioning information for every triangle of the mesh showed in Table 8.3. For example, for four partitions (numbered from 0 to 3), according to the information provided in Table 8.3, triangles 1, 2, 3 and 4 belongs to partitions 2, 0, 1 and 3, respectively.

METIS needs an input file of the mesh by using the .mesh format. Table 8.2 shows a mesh represented by the use of the .mesh format, which is equivalent to the mesh of Table 8.1. The first column indicates the number of triangles of the input mesh.

<table>
<thead>
<tr>
<th># Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>v</td>
</tr>
<tr>
<td>v</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>t</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>n</td>
</tr>
<tr>
<td>n</td>
</tr>
</tbody>
</table>

---

\(^1\)METIS and ParMETIS were developed at the Department of Computer Science and Engineering at the University of Minnesota
Table 8.3: Partitioning information

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

The input mesh and the text file that contain the partitioning information are used to generate the sub-meshes by the parallel program before sending the workload to the processors (see Figure 9.1 in chapter 9).

Figure 8.3 (b) shows the 16 partitions obtained for the mesh of Figure 8.3 (a) by using METIS library. Note that every sub-mesh has a variable number of neighboring sub-meshes.

Figure 8.3: (a) Input mesh formed by a set of points randomly generated by using CGAL library; (b) Sixteen partitions obtained from the input mesh shown in (a) which was divided using METIS library.

Figure 8.4 (b) shows four partitions of the Lake Superior mesh of figure (a) by using METIS library.

Figure 8.4: (a) Lake superior; (b) Lake superior divided into four sub-meshes by using METIS library.

Figures 8.5 and 8.6 show examples of tetrahedral meshes partitioned by using METIS software.
Figure 8.5: (a) 3D mesh of a set of random generated points using CGAL library. (b) Six sub-meshes obtained using METIS library.

Figure 8.6: (a) A complex skull geometry; (b) Skull was subdivided in eight sub-meshes using METIS.

Figure 8.7: The eight sub-meshes obtained from the partitioned mesh of the Skull shown in Figure 8.6 (b) using METIS library.
Note that in general METIS and ParMETIS libraries provide sub-meshes (partitions) with good aspect ratio (well-formed partitions) and every sub-mesh has a variable number of neighboring sub-meshes.

8.4 Strip-shaped partitioning

To create \( n \) strip-shaped sub-meshes from an input mesh where each sub-mesh has roughly the same number of triangles to be refined, we must consider two situations: (a) the set of triangles to be refined is uniformly distributed over the mesh (see Figure 8.8), (b) the set of triangles to be refined is concentrated in a particular region of the mesh (region \( R \), see Figure 8.9).

Given an input triangular conforming mesh \( \tau \) and \( S \subset \tau \), the set of candidate triangles to be refined, the algorithm produces a set of strip-shaped sub-meshes by assigning to each processor a sub-mesh with roughly the same number of target triangles in order to equilibrate the workload.

Let \( x_{\text{min}} \) and \( x_{\text{max}} \) be the minimum and maximum limits of the \( x \)-coordinate of the mesh, respectively, and let \( sx_{\text{min}} \) and \( sx_{\text{max}} \) be the minimum and maximum limits of the \( x \)-coordinate of the region \( R \) where the set \( S \) of triangles to be refined is located. Note that \( x_{\text{min}} \leq sx_{\text{min}} \) and \( sx_{\text{max}} \leq x_{\text{max}} \) (see Figure 8.9). The region \( R \) is divided into \( n \) local partitions with the same width \( h \) except for the first and last partitions that can be bigger.

![Figure 8.8: Input mesh subdivide into four partitions. Region R represents the zone where the candidate elements to be refined are uniformly distributed across the mesh. Sub-regions \( S_1, S_2, S_3 \) and \( S_4 \) have roughly the same workload. Note that \( x_{\text{min}} = sx_{\text{min}} \) and \( x_{\text{max}} = sx_{\text{max}} \).](image)

Once the limits of the partitions (sub-meshes) have been established, the algorithm proceeds to distribute the triangles of the input mesh into each partition. The centroid of each triangle is used to locate it in each sub-mesh.
Figure 8.9: Input mesh subdivided into four partitions. Region R represents the zone where the candidate elements to be refined are concentrated into a bounded sub-region into the mesh achieving an equilibrate workload. Sub-regions $S_1$, $S_2$, $S_3$ and $S_4$ have roughly the same workload. Note that $x_{\text{min}} < sx_{\text{min}}$ and $x_{\text{max}} > sx_{\text{max}}$.

Let $\tau$ be the input mesh to be divided, and $\tau_i$ the partition $i$ $(i=1,...,n)$, Algorithm 16 illustrates the division of the input mesh to obtain a set of strip-shaped sub-meshes.

Algorithm 16 createStripPartitions($\tau$)

```
Input: A conforming triangulation $\tau$.
Output: $n$ partitions $\text{sub}_\tau_i$.
Let $x_{\text{min}}$ the minimum x coordinate and $x_{\text{max}}$ the maximum x coordinate of $\tau$.
Compute the limits $(sx_{\text{min}}, sx_{\text{max}})$ of the region $R$ where the set $S$ (the set of triangles to be refined) is located.
h = $(sx_{\text{max}} - sx_{\text{min}}) / n$ (h is the width of each partition along the x coordinate).
```

for $t \in \tau$ do
  Compute the centroid $c$ of triangle $t$.
  Determine which slice contains $c$.
  Mark $t$ with the partition $S_k$, $k=1,...,n$.
end for

Print the output file with partitioning information.

Figure 8.1 (a) shows a mesh subdivided into eight strip-shaped partitions. Figure 8.10 illustrates four partitions that were obtained by applying the strip partitioning algorithm from the mesh shown in Figure 8.4 (a). The algorithm also provides as output the list of integer numbers with the partitioning information as shown in Table 8.3.
Figure 8.10: Lake superior divided in four partitions by using strip-shaped partitions.

In general strip-shaped partitions do not provide sub-meshes with good aspect ratio because when the value of n (the number sub-meshes) grows then the sub-meshes become increasingly thin. In practice every sub-mesh can have two or more neighboring sub-meshes.

8.5 Recursive two-dimensional space subdivision based on quadtrees

Let \( R \) be a bounded two-dimensional space whose origin is \((x, y)\) and its size is specified by \textbf{width} and \textbf{height}, then we can define an initial partition of \( R \) into four initial quadrants whose cardinal directions are represented by \textbf{ne} (north-east), \textbf{nw} (noth-west), \textbf{sw} (south-west) and \textbf{se} (south-east) (see Figure 8.11).

A quadtree is a recursive and hierarchical data structure which is used to recursively partition the 2-dimensional space into new four quadrants [134] [205] [206]. A tree structure is used to represent a quadtree [205] [206] where each node of the tree has up four children (see Figure 8.13).

![Diagram of quadtree structure]

Figure 8.11: The input space and the cardinal directions of the quadrants.

Recursively each quadrant is subdivided into four cardinal directions whose limits are represented by the following expressions:
- \( \text{NE} = (x + \text{width}/2.0, y + \text{height}/2.0, \text{width}/2.0, \text{height}/2.0) \)
- \( \text{NW} = (x, y + \text{height}/2.0, \text{width}/2.0, \text{height}/2.0) \)
- \( \text{SW} = (x, y, \text{width}/2.0, \text{height}/2.0) \)
- \( \text{SE} = (x + \text{width}/2.0, y, \text{width}/2.0, \text{height}/2.0) \)

Figure 8.12 illustrates a case where the set of triangles to be refined are located into a bounded region \( R \). The space (the complete mesh) was subdivided by using quadtrees. We can see that the tree structure shown in Figure 8.13 is not a balanced tree. This is due to the fact that the partitioning algorithm subdivides only those sub-regions where more candidate elements to be refined are located. The leaf nodes of the tree correspond to the partitions to be used for the refinement process.

![Figure 8.12: Input mesh subdivide into 22 partitions by using a partitioning algorithm based on quadtrees.](image)

Figure 8.13: Quadtree represented by a tree structure that corresponds to the Figure 8.12.

Figure 8.14 illustrates the 10 quadrants that correspond to the subdivision of the mesh shown in Figure 8.3. When the sub-meshes (quadrants) do not have the same size or are not homogeneous this means that the set of triangles to be refined are not uniformly distributed across the mesh (see examples shown in Figures 8.12 and 8.14).

Algorithms 17 and 18 illustrate the general algorithm for recursive partitioning of the 2-dimensional space based on quadtrees. Algorithm 17 shows the creation of the four children nodes of a quadrant. Algorithm 18 shows a general scheme of the recursive algorithm to create a quadtree structure. The algorithm receives as input the reference of a quadrant
(rootquadtree) and the average number of triangles to be refined that each quadrant approximately must have. This is a heuristic method which tries to assign approximately the same number of candidate triangles to be refined to every quadrant and uses the centroid of the triangles to determine along which partition every triangle is located.

Let $C$ be a set of centroids in the plane and $R$ its bounding region, then the data structure quadtree is a rooted tree of quadrants, where: (a) the root of the tree is the bounding region $R$, (b) $R$ and all other sub-regions $R_i$ are recursively divided into the four quadrants, (c) every quadrant has approximately the same number of centroids.

Algorithm 17 createChildren(rootquadtree)

1: Create the four children (ne, nw, sw, se) of the quadrant rootquadtree.

Line 3 of Algorithm 18 determines if a quadrant must be partitioned into four new quadrants. In the Line 5, the algorithm distributes the mesh to the four children quadrants (represented by ne, nw, sw and se, respectively).

Algorithm 18 createQuadTree(rootquadtree, average)

1: Input: rootquadtree, a quadrant of the quadtree structure; average maximum number of triangles to be refined by quadrant.
2: Output: a quadtree structure.
3: if (number of triangles to be refined of rootquadtree > average) then
4: Create the four children nodes of rootquadtree (call createChildren(rootquadtree)).
5: Distribute elements of the mesh into the four descendants of the quadrant rootquadtree.
6: for each child (i=ne, nw, sw, se) do
7: createQuadTree(i, average).
8: end for
9: end if

Figure 8.14: Mesh shown in Figure 8.3 (a) was partitioned into 10 partitions by using quadtrees. The sub-meshes obtained are not homogeneous which means that the set of triangles to be refined are not uniformly distributed across the mesh.
Chapter 9

Distributed Parallel Lepp-bisection algorithm for 2-dimensional mesh refinement

9.1 Introduction

In this chapter, we discuss a distributed parallel Lepp-bisection algorithm for the refinement of 2-dimensional triangulations where the mesh is partitioned and distributed between the nodes of the system. We consider homogeneous distributed systems (all the nodes have the same architecture) with a fast interconnection network.

The performance of a parallel algorithm in general depends on the number and shape of the partitions [114], load distribution and the adequate exploitation of the heterogeneity of the available architecture.

An efficient and scalable distributed Lepp-bisection algorithm must adequately deal with the following issues:

1. Constructing a set of partitions (sub-meshes) that minimize the flow of messages between neighboring partitions [114].

2. Distributing a well-balanced workload between the processors of the system. To this end, the partitioning algorithm must provide balanced partitions with roughly the same number of elements to be refined.

3. Efficiently producing a fully conforming final mesh along the partitions interfaces.

4. Achieving successful termination in acceptable time.

To study the practical behavior of the distributed Lepp-bisection algorithm, we will use the standard measurement tool: execution time, speedup, efficiency and communication time. Minimizing the number of interface edges (called the edge-cut) is also a goal of the mesh
partition algorithm, in order to minimize the inter-nodes communication among neighboring processors.

We use three mesh partitioning strategies, discussed in chapter 8, to prepare the workload of the processors:

1. Automatic partitions produced by METIS and ParMETIS libraries.
2. Mesh partitioning algorithms which construct strip partitions.
3. Algorithms that construct hierarchical and recursive structures such as quadtrees.

Firstly a master processor receives the input mesh and creates the set of partitions and a global list of interface edges to be communicated to the system nodes. The master processor then sends the partitions to the slave processors. The nodes of the system receive a submesh, information on interface edges and can use either a serial Lepp-bisection algorithm or a multicore Lepp-bisection algorithm to carry out the refinement process. In this thesis we use a static technique to assign the workload to processors and the SPMD model [130] (both reviewed in chapter 4) as the programming model to design and implement the parallel application on distributed systems.

This means that the mesh partitioning algorithm is applied once, before of the workload assignment and the refinement process. As the triangle selection method is based on random selection, all the partitions will always have approximately the same number of triangles to be refined on every iteration. This strategy does not allow partitions to exceed the maximum capacity of the assigned memory during the distributed refinement process.

9.2 State of the art on the development of parallel and distributed meshing applications

In the literature several parallel algorithms based on Lepp, longest-edge and terminal edge concepts have been studied.

Jones and Plassmann [114, 113] studied a parallel 4-triangles refinement algorithm [194] in 2-dimensions for distributed systems. The performance of the parallel refinement algorithm critically depends on the quality of the partitions used [114].

Castaños and Savage on 1999 [38] proposed parallel algorithms based on Rivara’s longest edge bisection algorithm [187] to refine triangulations and tetrahedral meshes. The parallel algorithm was integrated in PARED [38, 36] to provide refinement and coarsening of huge meshes. A binary tree is used to store the information of the whole refinement process. They also proposed an un-refinement algorithm which uses this refinement tree where the leaves of the tree contain the elements obtained during the last refinement iteration [38].

Lepp-based parallel algorithms have been discussed in references [14], [192] and [195]. Rivara, Pizarro and Chrisochoides proposed [195] a parallel tetrahedral refinement algorithm based on the local terminal edge refinement and associated terminal starts. This simple
algorithm makes implicit use of the concepts used in the Lepp-bisection refinement algorithm. The parallel algorithm was executed over a cluster of workstations with 64 nodes achieving good performance and an almost decoupled solution [195]. This solution used a central processor to synchronize the subproblems in order to obtain a distributed conforming mesh.

Rivara et. al [192] discuss an efficient and stable algorithm for the global parallel refinement of tetrahedral meshes based on the refinement of terminal-edges. The algorithm is inherently a decoupled method that requires zero communication and synchronization between the subproblems to perform a global refinement.

Balman [14] discusses a distributed parallel algorithm for the refinement of tetrahedral meshes that calculates the Lepp of each target tetrahedron and then uses an 8-tetrahedra parallel algorithm [174]. The implementation uses an acyclic directed graph of the Lepp of the target tetrahedron which is previously calculated. The implementation requires a large amount of memory to store the Lepp graph structures.

Foteinos and Chrisochoides [85] present FEM simulations on four dimensional meshes which are generated from binary medical images. The simulations area executed over larger distributed memory systems with faster shared-memory layers.

Foteinos, Chernikov, and Chrisochoides [89] present a tetrahedral Delaunay refinement algorithm for meshing medical images. They prove that the tetrahedral output mesh have good radius-edge ratio and all the boundary facets have planar angles larger than 30 degrees.

Foteinos and Chrisochoides [90] present a tetrahedral Delaunay parallel Image-to-Mesh Conversion algorithm to manage biological objects using high quality and good shape tetrahedra. The parallel program is very efficient and is able to generate large tetrahedral meshes. They also compare their application with other software such as CGAL and TetGen.

For a review on parallel Delaunay algorithms and advancing front techniques, see [58].

It is worth noting that few authors show the real performance measures of their parallel algorithms. Kohout [128] shows the performance of a parallel Delaunay triangulation based on circum-circle where the efficiency of the parallel algorithm decreases when the number of processors increases. Chrisochoides and Nave [62] show the parallel execution time for a parallel Delaunay mesh generation algorithm for P processors where the speedup achieved is $O(\log P)$. Oliker et. al [161] show the speedup for different experiments by using up to 64 processors. The speedup decreases when the number of processors increases. Castaños and Savage [38], [36] only describe the parallel execution time for global refinement over irregular 2D and 3D meshes. Jones and Plassmann [113] present a scalable parallel solution but they only show the parallel time execution, and the average number and the maximum number of triangles refined per processor per second versus the number of processors.

9.3 Design and implementation of the parallel software

The parallel software we designed considers the following steps (see Figure 9.1):

1. Mesh partitioning step. This is a preprocess step which applies a mesh partitioning method to construct the sub-meshes.

2. Workload assignment step. A master processor generates a global list of interface edges and distributes the sub-meshes between the nodes of the system.

3. Parallel refinement step. Each processor refines the assigned sub-mesh.

4. Assuring conforming interface step. Non-conforming issues in the interfaces of neighbor sub-meshes are solved.

5. Reconstruction of the final mesh step. All the processors send the final sub-meshes to the master processor which constructs the final mesh.

Figure 9.1 illustrates a general scheme of the parallel mesh refinement process.

![Diagram](image)

Figure 9.1: General process of the parallel distributed mesh refinement.

The parallel software was implemented by using C++ programming language and the MPI (Message Passing Interface) library [95, 143, 177, 178]. MPI is a standardized and portable system that provides a large set of message passing routines for different programming languages. MPI was designed as a tool to develop complex parallel applications for high performance computing on both massively parallel computers and on workstation clusters. There is a large number of MPI implementations of which we use the Open MPI implementation [95].

9.4 Parallel Distributed algorithm

A parallel distributed mesh refinement algorithm for triangular meshes based on a Lepp-bisection algorithm is presented in Algorithm 19. We use the partitioning algorithms described in chapter 8 and a static partitioning technique to assign the workload.
Algorithm 19 ParallelDistributed-MeshRefinementAlgorithm (τ, k)
1: Input: A conforming triangulation τ; k number of processors.
2: Output: A final conforming triangulation τ_f.
3: Master processor constructs k partitions and global interface information from τ.
4: Send partition τ_i to each processor P_i.
5: For each processor P_i (i=0,...,k-1) perform in parallel the refinement of the partition τ_i applying a Lepp-bisection algorithm.
6: An iterative conformity process is performed to produce a full conforming mesh along the interface. This is the last step of the parallel distributed algorithm.
7: Process P_i sends a refined partition τ_f_i (i=1,...,k-1) to the master processor P_0.
8: Master processor constructs the final mesh τ_f from each τ_f_i.

According to Algorithm 19 master processor (P_0) also carries out refinement tasks over a partition τ_i. The conformity problems in the interface are solved in the last step of the global refinement process, such as shown in step 6 of the Algorithm 19, once all the processes or nodes have finished the local refinement process over the submeshes.

9.5 Conformity step along interfaces

This is the last step of the parallel distributed refinement algorithm which is carried out concurrently by all the processors of the system. Here neighboring processors send (message passing) the new interface vertices that were inserted on both sides of the interface during the previous parallel refinement step. Several interface iterations are usually performed to obtain a final conforming mesh.

Algorithm 20 illustrates the general procedure to obtain a final conforming mesh by iteratively inserting the interface vertices. Each processor P_i (i=0,1,...,k-1) executes this procedure to achieve a final conforming mesh after finishing all the refinement process (step 6 of the Algorithm 19).

Algorithm 20 Conformity-Step(τ_i)
1: Input: A refined triangular submesh τ_i.
2: Output: A conforming triangulation τ_f.
3: while non-conforming interface edges remains on the interfaces of the submesh of processor P_i do
4: Process P_i sends by message passing the new vertices inserted on the interface to their neighboring processors.
5: Process P_i inserts the new interface vertices received from the neighboring processors.
6: end while

The interface process finishes when all the processors verify that the last interface vertices received from neighbor partitions were previously inserted in the current partition.
Figure 9.2 shows a complete procedure to obtain a conforming mesh. Figure (a) shows the initial situation where triangles \( t_i \) and \( t_{i+1} \) are located in neighboring sub-meshes \( Submesh_i \) and \( Submesh_{i+1} \), respectively, and share a common interface edge \( e \).

![Diagram](image)

Figure 9.2: Procedure to obtain a conforming mesh on the interface. Triangles \( t_i \) and \( t_{i+1} \) belong to different sub-meshes and share the same interface edge \( e \).

Figure 9.2 (b) shows the non-conforming resultant mesh obtained when processors \( P_i \) and \( P_{i+1} \) perform independent refinement of the triangles \( t_i \) and \( t_{i+1} \) (in figure (a)). Both processors insert the same vertex \( v_0 \) on both sides of the interface edge \( e \), while processor \( P_i \) inserts a new vertex \( v_1 \) on the interface edge \( e \), and processor \( P_{i+1} \) inserts a new vertex \( v_2 \). Processor \( P_i \) then sends the vertex \( v_1 \) to processor \( P_{i+1} \) and processor \( P_{i+1} \) sends the vertex \( v_2 \) to processor \( P_i \) to be inserted in order to achieve a conforming mesh in the interface. Figures (c), (d) and (e) show the rest of the iterative interface process that inserts 4 vertices to obtain the final conforming mesh.

For the parallel distributed algorithm the following properties hold:

1. The parallel distributed mesh refinement algorithm based on Lepp-bisection algorithm produces a conforming mesh by a finite number of steps of refinement.

2. The parallel distributed mesh refinement algorithm based on Lepp-bisection algorithm produces triangulations with the same quality as the serial algorithm.

Figures 9.3 to 9.4 illustrate examples of mesh partitioning and mesh refinement, respectively. Figure 9.3 (a) illustrates the initial mesh and figure (b) the partition of the initial
mesh in 8 partitions by using METIS library.

Figure 9.3: (a) Initial two-dimensional mesh (1,000 vertices and 1,981 triangles)) (b) The initial mesh is subdivided in 8 partitions by using METIS library).

Figure 9.4 illustrates an example of refinement of a mesh: figure (a) illustrates the detail of the interface between two sub-meshes; figure (b) shows an example of refinement when the final step of the interface vertices insertion is not applied, then a final non-conforming is obtained; and figure (c) shows a situation when the last step of the interface vertices insertion is applied, then a final conforming mesh is obtained.

Figure 9.4: (a) Detail of the interface over an initial mesh; (b) Final non-conforming mesh after the parallel refinement along the interface; (c) Final conforming mesh along the interface (conformity step performed).
9.6 Empirical testing with Parallel Distributed Lepp-bisection algorithm

The experiments were executed over a cluster of computers (Levque, IBM iDataplex cluster 1350) composed of 66 nodes and 528 cores dedicated to run user jobs. Each node has two quad-core Intel Xeon E5550 processors (2.67GHz, 8 cores, with 23.5 GB RAM). Levque uses two interconnection networks: an Infiniband high speed network (at 40Gbit/s, 100% non-blocking) to communicate the nodes and I/O purposes, and an Ethernet network used for administration and user interaction. The storage area is composed of 8 TB of available space. Every user has up to 64 cores available and every user process (MPI process) has up 3 GB RAM for parallel distributed processing. It is worth noting that when an Infiniband network is used, a couple of nodes communicate at 40Gbit/s and 100% non blocking.

Three different partitioning methods were used to generate the workload of the processors: METIS/ParMETIS libraries, strip partitions, and a recursive partitioning method based on quadtrees. In general, the results show that there is no significant difference between the execution times and the performance of the parallel distributed algorithm when the three partitioning methods are used. Note that an important difference regarding to the number of interface edges was identified between the partitioning methods. METIS/ParMETIS libraries and the recursive partitioning method based on quadtrees provided the smaller number of interface edges. To execute the experiments we consider an input mesh with 2,999,998 vertices and 5,999,953 triangles. The mesh was created from a set of randomly generated points.

Using ParMETIS

Up to 64 partitions were created where every processor processes only one partition. Table 9.1 shows the performance (execution time, speedup and efficiency) obtained using up to 64 cores. The speedup obtained is partially scalable (see Figure 9.5), however, the efficiency decreases when the number of cores grows.

Table 9.2 shows communication time, computation time, partitioning time, read time of input mesh and partition file, sub-meshes creation time, sub-meshes send time to nodes, parallel refinement time, points insertion time on interface, sub-meshes send time to master process, disk write time of final mesh, number of interface edges and the size of the final mesh. Note that in general the computation/communication ratio is big, but decreases between 32 to 64 cores.
Table 9.1: Performance measures: execution time, speedup and efficiency.

<table>
<thead>
<tr>
<th>Execution time (seconds)</th>
<th>1P</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>16P</th>
<th>32P</th>
<th>64P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>92.33</td>
<td>76.22</td>
<td>40.71</td>
<td>23.35</td>
<td>13.87</td>
<td>10.07</td>
<td>7.72</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Speedup</th>
<th>1</th>
<th>1.21</th>
<th>2.27</th>
<th>3.95</th>
<th>6.66</th>
<th>9.17</th>
<th>11.96</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Efficiency</th>
<th>1</th>
<th>0.61</th>
<th>0.57</th>
<th>0.49</th>
<th>0.42</th>
<th>0.29</th>
<th>0.19</th>
</tr>
</thead>
</table>

Figure 9.5: Speedup. Using partitions constructed by using ParMETIS library.
Table 9.2: Other measures.

<table>
<thead>
<tr>
<th>Computation/communication ratio (Number of processors)</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>16P</th>
<th>32P</th>
<th>64P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication time (sec)</td>
<td>0.0245</td>
<td>0.0041</td>
<td>0.0046</td>
<td>0.0304</td>
<td>1.433</td>
<td>1.016</td>
</tr>
<tr>
<td>Computation time (sec)</td>
<td>98.06</td>
<td>52.57</td>
<td>27.28</td>
<td>15.92</td>
<td>11.47</td>
<td>8.26</td>
</tr>
<tr>
<td>Computation/communication ratio</td>
<td>4003.14</td>
<td>12821.463</td>
<td>18723.40</td>
<td>523.65</td>
<td>8.01</td>
<td>8.16</td>
</tr>
</tbody>
</table>

 Profiling information (time given in seconds)

<table>
<thead>
<tr>
<th>Disk read time of input mesh</th>
<th>161.37</th>
<th>165.46</th>
<th>155.14</th>
<th>168.18</th>
<th>164.00</th>
<th>166.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk read time of partition file</td>
<td>9.71</td>
<td>10.06</td>
<td>9.52</td>
<td>10.31</td>
<td>10.16</td>
<td>10.16</td>
</tr>
<tr>
<td>Sub-meshes creation time</td>
<td>15.95</td>
<td>16.41</td>
<td>15.80</td>
<td>16.75</td>
<td>15.74</td>
<td>15.81</td>
</tr>
<tr>
<td>Sub-meshes send time to slave nodes</td>
<td>0.11</td>
<td>0.15</td>
<td>0.24</td>
<td>0.38</td>
<td>0.27</td>
<td>0.30</td>
</tr>
<tr>
<td>Parallel refinement time</td>
<td>45.18</td>
<td>24.07</td>
<td>12.32</td>
<td>6.1</td>
<td>3.1</td>
<td>1.51</td>
</tr>
<tr>
<td>Conformity time (point insertion on interface)</td>
<td>0.016</td>
<td>0.019</td>
<td>0.035</td>
<td>0.036</td>
<td>1.04</td>
<td>1.08</td>
</tr>
<tr>
<td>Final sub-meshes send time to master process</td>
<td>4.88</td>
<td>8.84</td>
<td>11.20</td>
<td>13.71</td>
<td>14.17</td>
<td>14.30</td>
</tr>
<tr>
<td>Disk write time of final mesh</td>
<td>552.79</td>
<td>636.74</td>
<td>563.56</td>
<td>534.64</td>
<td>615.70</td>
<td>515.47</td>
</tr>
<tr>
<td>Sequential Partitioning time</td>
<td>6.82</td>
<td>6.80</td>
<td>6.86</td>
<td>6.87</td>
<td>6.91</td>
<td>6.96</td>
</tr>
<tr>
<td>Parallel Partitioning time</td>
<td>1.23</td>
<td>1.28</td>
<td>1.29</td>
<td>1.23</td>
<td>1.26</td>
<td>1.27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total number of interface edge and final mesh size</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of interface (edges)</td>
<td>2,090</td>
<td>4,193</td>
<td>7,801</td>
<td>12,356</td>
<td>19,239</td>
<td>28,573</td>
</tr>
<tr>
<td>Mesh Size (triangles)</td>
<td>18,208,784</td>
<td>18,194,008</td>
<td>18,208,355</td>
<td>18,212,587</td>
<td>18,244,673</td>
<td>18,213,142</td>
</tr>
</tbody>
</table>

Note that the parallel refinement time only includes the mesh refinement time, whereas the computation time refers to the parallel mesh processing time which includes the parallel refinement time, communication time between neighboring processors and workload preparing time. The workload preparing time refers to randomly selecting the triangles to be refined in every refinement iteration which is carried out by every processor.

Sequential partitioning time and parallel partitioning time were measured to compare the processing times of the sequential and parallel partition methods. Note that the parallel partition method is faster than the sequential method.

Using strip partitions

Up to 64 partitions were created where every processor processes only one partition. Table 9.3 shows the performance (execution time, speedup and efficiency) obtained using up to 64 cores. The speedup obtained is scalable (see Figure 9.6), however, as with the partitions of ParMETIS, the efficiency also decreases.

This partition method provides the biggest number of interface edges (see Table 9.4). Table 9.4 shows communication time, computation time, partitioning time, number of interface edges and the size of the final mesh. Note that in general the computation/communication
ratio is big, but decreases between 32 to 64 cores.

Table 9.3: Performance measures: execution time, speedup and efficiency.

<table>
<thead>
<tr>
<th></th>
<th>Execution time (seconds)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1P</td>
<td>92.33</td>
<td>1</td>
<td>1.000</td>
</tr>
<tr>
<td>2P</td>
<td>77.40</td>
<td>1.19</td>
<td>0.600</td>
</tr>
<tr>
<td>4P</td>
<td>40.75</td>
<td>2.27</td>
<td>0.570</td>
</tr>
<tr>
<td>8P</td>
<td>22.99</td>
<td>4.02</td>
<td>0.500</td>
</tr>
<tr>
<td>16P</td>
<td>13.83</td>
<td>6.68</td>
<td>0.420</td>
</tr>
<tr>
<td>32P</td>
<td>10.21</td>
<td>9.04</td>
<td>0.280</td>
</tr>
<tr>
<td>64P</td>
<td>7.95</td>
<td>11.61</td>
<td>0.180</td>
</tr>
</tbody>
</table>

Figure 9.6: Speedup. Using strip partitions.
Table 9.4: Other measures.

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>16P</th>
<th>32P</th>
<th>64P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication time (sec)</td>
<td>0.0112</td>
<td>0.014</td>
<td>0.0089</td>
<td>0.0066</td>
<td>1.018</td>
<td>0.033</td>
</tr>
<tr>
<td>Computation time (sec)</td>
<td>96.50</td>
<td>50.11</td>
<td>27.42</td>
<td>16.01</td>
<td>11.31</td>
<td>7.81</td>
</tr>
<tr>
<td>Computation/communication ratio</td>
<td>8638.02</td>
<td>3580.86</td>
<td>3090.87</td>
<td>1665.52</td>
<td>11.15</td>
<td>148.66</td>
</tr>
</tbody>
</table>

Profiling information (time given in seconds)

| Disk read time of input mesh | 163.13 | 158.21 | 162.42 | 166.23 | 168.89 | 163.45 |
| Disk read time of partition file | 10.49 | 10.08 | 11.84 | 10.04 | 10.13 | 9.87 |
| Sub-meshes creation time | 16.90 | 16.44 | 19.14 | 16.56 | 15.85 | 15.77 |
| Sub-meshes send time to slave nodes | 0.10 | 0.16 | 0.26 | 1.22 | 0.37 | 0.32 |
| Parallel refinement time | 45.87 | 23.7 | 12.18 | 6.10 | 3.13 | 1.62 |
| Conformity time (point insertion on interface) | 0.016 | 0.039 | 0.066 | 1.07 | 1.13 | 1.38 |
| Final sub-meshes send time to master process | 4.83 | 9.03 | 12.48 | 12.72 | 13.68 | 14.47 |
| Disk write time of final mesh | 511.55 | 544.02 | 597.92 | 517.62 | 503.05 | 511.45 |
| Sequential Partitioning time | 2.63 | 2.84 | 3.12 | 4.09 | 4.81 | 6.87 |
| Parallel partition time | 0.008 | 0.012 | 0.022 | 0.034 | 0.036 | 0.038 |

Total number of interface edge and final mesh size

| Length of interface (edges) | 2,007 | 6,015 | 14,169 | 30,306 | 62,656 | 126,730 |
| Mesh Size (triangles) | 18,207,592 | 18,193,967 | 18,211,365 | 18,193,431 | 18,209,303 | 18,148,458 |

Using recursive quadtree space subdivision

Table 9.5 shows the performance (execution time, speedup and efficiency) obtained using up to 64 cores. The speedup obtained is scalable (see Figure 9.7), as in ParMETIS partitions, and strip partitions, however, the efficiency also decreases.

Table 9.6 shows communication time, computation time, partitioning time, number of interface edges and the size of the final mesh. Note that in general the computation/communication ratio is big, but decreases when 64 cores are used.

Table 9.5: Performance measures: execution time, speedup and efficiency.

<table>
<thead>
<tr>
<th>Execution time (seconds)</th>
<th>1P</th>
<th>4P</th>
<th>16P</th>
<th>64P</th>
</tr>
</thead>
<tbody>
<tr>
<td>92.33</td>
<td>41.14</td>
<td>13.54</td>
<td>7.66</td>
<td></td>
</tr>
<tr>
<td>Speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.24</td>
<td>6.82</td>
<td>12.05</td>
<td></td>
</tr>
<tr>
<td>Efficiency</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.56</td>
<td>0.43</td>
<td>0.19</td>
<td></td>
</tr>
</tbody>
</table>
Figure 9.7: Speedup. Using recursive space subdivision based on quadtree.

<table>
<thead>
<tr>
<th>Table 9.6: Other measures.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation/communication ratio (Number of processors)</td>
</tr>
<tr>
<td>Communication time (sec)</td>
</tr>
<tr>
<td>Computation time (sec)</td>
</tr>
<tr>
<td>Computation/communication ratio</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Profiling information (time given in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk read time of input mesh</td>
</tr>
<tr>
<td>Disk read time of partition file</td>
</tr>
<tr>
<td>Sub-meshes creation time</td>
</tr>
<tr>
<td>Sub-meshes send time to nodes</td>
</tr>
<tr>
<td>Parallel refinement time</td>
</tr>
<tr>
<td>Conformity time (point insertion on interface)</td>
</tr>
<tr>
<td>Final sub-meshes send time to master process</td>
</tr>
<tr>
<td>Sending time to master process</td>
</tr>
<tr>
<td>Disk write time of final mesh</td>
</tr>
<tr>
<td>Sequential Partitioning time</td>
</tr>
<tr>
<td>Parallel partition time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total number of interface edge and final mesh size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of interface (edges)</td>
</tr>
<tr>
<td>Mesh Size (triangles)</td>
</tr>
</tbody>
</table>

As the triangles to be refined are randomly selected, the number of triangles of the final meshes slightly varies in the different mesh partitioning methods and also for different numbers of processors.

Table 9.7 illustrates a comparison of the speedups obtained during the execution of the parallel algorithm and the number of interface edges, according to the applied partitioning
method. Note that there are no important differences between the speedups (see also Figure 9.8) because the triangles to be refined are uniformly distributed over the domain and over the partitions. Table 9.7 also illustrates that the strip-shaped partitions produces the higher number of interface edges than the other partitioning methods.

Table 9.7: Results comparison.

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2P</td>
<td>4P</td>
<td>8P</td>
<td>16P</td>
<td>32P</td>
</tr>
<tr>
<td>ParMETIS</td>
<td>1.21</td>
<td>2.27</td>
<td>3.95</td>
<td>6.66</td>
<td>9.17</td>
</tr>
<tr>
<td>Strip Partitions</td>
<td>1.19</td>
<td>2.27</td>
<td>4.02</td>
<td>6.68</td>
<td>9.04</td>
</tr>
<tr>
<td>QuadTree</td>
<td>2.24</td>
<td></td>
<td>6.82</td>
<td></td>
<td>12.05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Number of Interface Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ParMETIS</td>
</tr>
<tr>
<td></td>
<td>2,090</td>
</tr>
<tr>
<td></td>
<td>4,193</td>
</tr>
<tr>
<td></td>
<td>7,801</td>
</tr>
<tr>
<td></td>
<td>12,356</td>
</tr>
</tbody>
</table>

Figure 9.8 illustrates a comparison of the three speedups obtained from the execution of the parallel program using the three partitioning methods.

![Speedup Comparison](image.png)

Figure 9.8: Speedup comparison.

We can conclude that the parallel distributed refinement algorithm behaves well with the partitions produced by the three partitioning methods when the triangles of $S_{ref}$ are randomly selected. The speedup obtained is good because it scales as the number of cores grows, however the efficiency decreases. Therefore we can say that the parallel distributed algorithm has a partially scalable speedup.

The communication times among processors were low during the parallel refinement because the interconnection network of the parallel system is very fast. This fact helped in
obtaining good speedups with the different mesh partitioning methods. The interconnection network of the cluster corresponds to an Infiniband high speed network which has a complete network topology.

The computation/communication ratio in general is big, but it greatly decreases between 32 to 64 cores. This means that for a big number of cores the parallel execution time decreases and the communication times increases.

Sequential partitioning time and parallel partitioning time were measured to compare the processing times of the sequential and parallel partition methods. We must note that good parallel partition times were obtained.
Chapter 10

Conclusions

In this thesis we studied and developed Lepp-based parallel mesh refinement algorithms for 2-dimensional and 3-dimensional triangulations over both shared memory and distributed systems. We performed an extensive empirical study that showed that the parallel mesh algorithms that run over multicore architectures and distributed systems achieved good scalability.

To test the algorithms we consider two classes of meshes: (a) meshes limited by the convex hull of sets of randomly generated points, and (b) meshes of irregular polygonal/polyhedral objects. The initial meshes were generated by using CGAL Delaunay software and the parallel software was developed by using the object oriented programming model, C++ programming language and Qt library.

We developed parallel mesh refinement algorithms over multicore architectures (shared memory systems) for 2-dimensional and 3-dimensional Lepp-based algorithms. To obtain a robust implementation of a multicore algorithm on shared memory systems we had to deal with the following synchronization problems: (1) To avoid simultaneous processing of triangles (tetrahedra) whose Lepps collide; (2) To avoid data structure and neighborhood inconsistencies with refinement operations.

For all the multicore algorithms developed in this thesis, the consistency of the data structures and the neighborhood information between the triangles/tetrahedra was achieved by synchronizing the threads and using local operations over the data structures. The problems of synchronization were treated by using the next strategies: (1) If two threads intend to refine in parallel two triangles (tetrahedra) with overlapping Lepps, then one of the threads is freed and allowed to refine another element; (2) If two threads intend to refine in parallel two triangles (tetrahedra) with overlapping Lepps, then one of the threads is blocked, whereas the other one continues computing the Lepp and refining the triangles; (3) The data structure consistency is protected during the refinement process by avoiding that two threads concurrently refine the same or neighboring triangle/tetrahedron. In this case we use mutual exclusion primitives (e.g., mutexes, monitors) to synchronize the access to shared data.
Three parallel algorithms based on the Lepp-bisection method were implemented and tested for iterative refinement of 2-dimensional meshes based on triangles. The partial Lepp storing algorithm (PA1), has good behavior for 4 cores, whereas for 8 cores the speedup deteriorates. The Lepp recomputation algorithm (PA2) has a good performance for both Intel core i7 and Intel Xeon E5550 processors. Nevertheless, this parallel algorithm achieves a slight improvement of the performance for 8 cores - very close to the performance obtained for 4 cores. On the other hand, the edge monitor based algorithm (PA3) does not behave well when the number of cores increases due to the high contention and the blocking times of the threads in the monitor associated with a shared edge. The high contention is produced when multiple threads compete for access to shared data by using mutual exclusion primitives (e.g. monitors).

Strategies such as randomization and the early load of the terminal elements and their immediate neighbors were used to produce efficient parallel algorithms that run on a set of cores distributed in different sockets by taking advantage of the properties of the Lepp-bisection algorithm where the set of elements to be refined do not need a previous and particular order. Randomization helped to minimize the collisions among Lepps processed in parallel. According to the experimental results, when early load of the terminal elements and their neighbors is not used and the set of cores are located over different quadcore sockets, the performance of the multicore algorithm deteriorates due to the high level of contention produced when multiple threads compete for access to shared data and the use of mutual exclusion primitives.

It is worth noting that the multicore algorithms are robust and guarantee the construction of quality and conforming meshes by achieving good performance. The multicore algorithms were validated by using small and large triangular meshes.

We also developed serial and parallel relaxed Lepp-Delaunay algorithms (based on Lepp-Delaunay centroid algorithm), to generate good quality meshes. We relaxed the delaunization process by using a neighborhood parameter $K$ that constrains the edge flipping propagation around the terminal edges which produces meshes that are not fully Delaunay. However, the threshold angle $\theta$ is satisfied and the bad quality elements are eliminated from the mesh. Note that for $K=2$ the multicore algorithm achieves good performance and the percentage of non-Delaunay triangles is low.

We also developed three serial algorithms and a parallel algorithm for multicore architectures based on the Lepp-bisection method for the iterative refinement of tetrahedral meshes. In spite of its complexity, parallel refinement algorithms for 3-dimensional meshes showed a better performance than the parallel refinement algorithms for 2-dimensional meshes over multicore architectures. In tetrahedral meshes we applied an early load of the terminal starts and its neighboring tetrahedra to achieve good speedup and efficiency. This strategy is similar to the strategy used by the 2-dimensional parallel algorithm based on re-computing Lepp($t$). There is evidence that the number of Lepp collisions in 2 and 3-dimensions is lower when randomization is used. We must note that for all the test problems, the 3-dimensional Lepp-bisection algorithm tends to improve the quality distribution of the tetrahedra.

For developing parallel refinement algorithms over distributed systems (to facilitate the workload assignment of the processors), three mesh partitioning methods were used: strip-
shaped partitions, quadtree based algorithms and partition methods supported by METIS/ParMETIS libraries. The results show that there is not significant difference between the execution times and the performance of the parallel distributed algorithm when the three partitioning methods were used. This is due to the fact that the set of target triangles to be refined are uniformly distributed over the mesh and a good workload assignment was achieved by using the three partitioning methods used in this thesis. The communication times among processors were low during the parallel refinement because the interconnection network of the parallel system is very fast. This fact helped to obtain good speedups. The interconnection network of the cluster corresponds to an Infiniband high speed network which has a complete network topology. Additional research is needed to test the algorithm behavior in more general practical contexts.

The Lepp-bisection algorithm in 2-dimensions was parallelized over a parallel distributed system achieving good performance and scalability up to 64 cores. Good quality and conforming meshes are obtained analogous to the sequential algorithm. The conformity of the global mesh on the interface between two neighboring partitions were satisfactorily achieved by using a robust conforming process.

Finally the research of this thesis can be extended in several directions:

- To study a 3-dimensional Lepp-bisection algorithm on distributed systems. Mesh partitioning methods such as METIS/ParMETIS library and a recursive 3-dimensional space partitioning method based on octrees can be used to distribute the workload to the processors.

- To study new strategies for the efficient workload assignment over distributed systems. In this thesis up until now we distributed the workload by using static assignment techniques. We suggest using dynamic assignment techniques to improve the workload assignment on runtime for more general (practical) problems.

- To develop parallel 3-dimensional Lepp-Delaunay algorithms for the refinement of tetrahedral meshes both over multicore and distributed systems.

- To develop more general parallel mesh refinement software able to refine and derefine the mesh.
Bibliography


[10] I. Babuska. *Accuracy estimates and adaptive refinements in finite element computa-


[77] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of cgal, the computational geometry algorithms library. 1998.

[78] Andriy Fedorov and Nikos Chrisochoides. Adaptive mesh refinement for non-rigid
registration of brain mri. In 8th World Congress on Computational Mechanics, 2008.


[119] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for


[146] Sia Meshkat and Dafna Talmor. Generating a mixed mesh of hexahedra, pentahedra and tetrahedra from an underlying tetrahedral mesh. 2000.


[154] Patrick Mullen, Fernando de Goes, Mathieu Desbrun, David Cohen-Steiner, and Pierre Alliez. Signing the unsigned: Robust surface reconstruction from raw pointsets.


[228] Bruce Simpson. How efficient are delaunay refined meshes? an empirical study. In *In


[244] Rafael Vandriessche and Dirk Roose. Load balancing computational fluid dynamics calculations on unstructured grids. In In AGARD, Special Course on Parallel Computing in CFD 26 p (SEE N96-16247 04-60), volume 1, 1995.


