24th International Meshing Roundtable (IMR24)

Multithread Lepp-bisection algorithms in 2-dimensions

Pedro A. Rodriguez\textsuperscript{a}, Maria-Cecilia Rivara\textsuperscript{b,}\textsuperscript{*}

\textsuperscript{a}Departamento de Sistemas de Información, Universidad del Bio-Bio, Av. Collao 1202, Concepción, 4051381, Chile
\textsuperscript{b}Departamento de Ciencias de la Computación, Universidad de Chile, Avenida Beauchef 851, Santiago, 8370456, Chile

Abstract

We discuss and compare three multithread Lepp-bisection algorithms for the refinement of triangulations over multicore architectures. We have obtained an efficient and robust serial implementation, and a partially scalable and efficient multithread method.

\textsuperscript{*}Corresponding author. Tel.: +56-2-2978-4365 ; fax: +56-2-2689-5531.
E-mail address: mcrivara@dcc.uchile.cl (Maria-Cecilia Rivara)

1. Introduction

Longest edge algorithms for local refinement of triangulations guarantee the construction of refined triangulations that maintain the quality of the input mesh [2,5,7]. Lepp-bisection algorithm is an efficient reformulation of the longest edge algorithm with the following advantages: (a) only local refinement operations are performed which always maintain a conforming mesh (the intersection of pairs of triangles is either a common vertex, or a common edge); (b) the use of the Lepp concept allows to easily design parallel algorithms.

Distributed longest edge based algorithms for the parallel refinement of triangulations have been discussed in the literature. In a review paper for fluid dynamics applications, Williams [10] recommends the use of parallel 4-triangles longest edge algorithm for the refinement of huge triangulations; Jones and Plassmann [4] discuss in detail a parallel distributed 4-triangles refinement algorithm; Castaños and Savage [3] proposed a distributed parallelization of the original longest edge algorithm in 3-dimensions; Rivara et al [6] proposed a simple algorithm for the global refinement of tetrahedral meshes. Balman in [1] proposed an algorithm that uses a 8-tetrahedra longest edge algorithm.

A previous simple multithread Lepp-bisection algorithm (PA2 algorithm in this paper) for two-dimensional triangulations over a simple architecture having 4 cores in one socket is studied in [8]. An algorithm for the refinement of tetrahedral meshes is presented in [9]. Here we study the behavior of three variants of multithread Lepp-bisection algorithm over an Intel E5-2660, 20 cores, 2 sockets, architecture. We present an efficient and robust serial implementation which is used to compute the speedup measures.
2. Serial Lepp-bisection algorithm

An edge $E$ is a terminal edge in a triangular mesh $\tau$ if $E$ is the longest edge of the triangles that share this edge.

In two dimensions, $\text{Lepp}(t_0)$, the longest edge propagating path of a triangle $t_0$ [5,7], is a sequence of $N$ increasing neighbor triangles such that $t_i$ is the neighbor triangle of $t_{i-1}$ by the longest edge of $t_{i-1}$ that allows to find a unique terminal edge either shared by one boundary triangle ($t_N$) or two terminal triangles ($t_{N-1}$, $t_N$). Thus $\text{Lepp}(t_0)$ is a 2D submesh that finishes when a terminal edge associated to triangle $t_0$ is found in the mesh (see Fig. 1 (a)).

Given a triangle $t_0$, the serial Lepp-bisection algorithm computes $\text{Lepp}(t_0)$ and refines the couple of terminal triangles by longest edge bisection of these triangles. The process is repeated until the triangle $t_0$ is refined (see Fig. 1). This algorithm is formulated as follows [5]:

1: Serial-Lepp-Bisection-Algorithm($\tau$, $S$)
2: Input: $\tau$, a quality triangulation, and $S \subset \tau$, set of triangles to be refined.
3: Output: $\tau_f$, a refined and conforming final triangulation.
4: for ( each triangle $t \in S$ ) do
5: while ( $t$ remains without being bisected ) do
6: Find $\text{Lepp}(t)$, terminal triangles $t_{N-1}$, $t_N$ (triangle $t_N$ can be null if $AB$ is a boundary edge)
7: Bisect the terminal triangles
8: end while
9: end for

Fig. 1. (a) $\text{Lepp}(t_0) = \{t_0, t_1, t_2, t_3, t_4\}$ and $\text{Lepp}(t'_0) = \{t'_0, t'_1, t'_2, t'_3, t_2, t_3, t_4\}$, $AB$ is a terminal edge; (b) First step of Lepp-bisection refinement of $t_0$.

3. Multithread Lepp-bisection algorithms

We consider a shared memory multicore computer having $p$ physical cores. To perform the refinement task, each core $P_i$ ($i=1,\ldots,p$) is in charge of the parallel processing of an individual triangle $t$ in $S$ and its associated changing Lepp sequence until the triangle $t$ is refined in the mesh. Once the refinement of $t$ is performed, $P_i$ will pick up another triangle of $S$ to continue the refinement task.

To perform the parallel work, we need to deal with the following synchronization issues [8]: (a) To avoid collisions associated to the parallel processing of triangles whose Lepp sub-meshes overlap; (b) To avoid data structure inconsistencies due to the parallel refinement of adjacent terminal triangles that belong to adjacent (non-overlapping) Lepp sub-meshes. Mutexes are used as synchronization methods to control the access to shared data structure and to avoid inconsistencies. Fig. 1 (a) illustrates the case of overlapping Lepp sub-meshes for triangles $t_0$ and $t_0'$ in 2-dimensions, where $\text{Lepp}(t_0) \cap \text{Lepp}(t'_0) = \{t_2, t_3, t_4\}$.

We present three versions of multicore Lepp-bisection algorithms:

1. PA1. Marking Lepp Algorithm. If not collision is found, this algorithm locks the triangles of the full Lepp computed and performs refinement of terminal triangles. If collision is detected, the partial Lepp computed (until triangle $t'_3$ in $\text{Lepp}(t'_0)$ in Fig. 1 (a)) is discarded and the thread proceeds to pick up another triangle.
2. PA2. Partial Lepp Storing Algorithm. This multicore algorithm additionally to the work performed by the PA1 algorithm, also stores the partial Lepp computed for being later processed.

3. PA3. Lepp Recomputation Algorithm. This multicore algorithm recomputes Lepp\((t)\) when triangle \(t\) is again processed (neither the full Lepp is locked, neither the partial Lepps are stored). Only the terminal triangles and their immediate neighbors are locked.

In the case of the Partial Lepp Storing algorithm (PA2), if the Lepp\((t)\) is successfully computed, the triangles that belong to Lepp\((t)\) are marked as occupied and the terminal triangles are refined. On the contrary if a collision is detected, the partial Lepp\((t)\) is saved into the list \(L\) to be later processed (function \text{RefineListOfPartialLepp}(L)\) is invoked at final of the algorithm shown below, to process pending partial Lepps).

```plaintext
1: PartialLeppStoringAlgorithm\((\tau, S)\)
2: Input: a quality triangulation \(\tau\); \(S \subset \tau\) set of triangles to be refined
3: Output: a refined and conforming triangulation \(\tau_f\)
4: Initialize a list \(L\) of pending partial Lepps
5: while \(S \neq \emptyset\) do
6: A free thread selects a triangle \(t\) from \(S\); mark \(t\) as occupied.
7: while \(t\) remains in mesh do
8: Compute Lepp\((t)\) while non-occupied triangle is found
9: if collision is detected then
10: Insert partial Lepp\((t)\) into \(L\)
11: else
12: Mark all the triangles in Lepp\((t)\) as occupied
13: if neighbor triangles of terminal triangles are unmarked then
14: Refine terminal triangles and update \(S\).
15: end if
16: end if
17: end while
18: end while
19: \text{RefineListOfPartialLepp}(L)
```

The Marking Lepp Algorithm is the same than the previous algorithm without the instructions 4, 9, 10 and 19. For the Lepp Recomputation Algorithm, if two threads \(P_0\) and \(P_1\) refine in parallel triangles \(t_0, t'_0\), with overlapping Lepps (see Fig. 1 (a)), the thread that access a locked triangle is freed and allowed to refine another triangle. The triangles in the propagation path are not marked as occupied, and the terminal triangles and their immediate neighbors are locked. Thus the refinement of a pair of terminal triangles is not performed if at least one of its neighboring triangles is locked.

```plaintext
1: LeppRecomputationAlgorithm\((\tau, S)\)
2: Input: a quality triangulation \(\tau\); \(S \subset \tau\) set of triangles to be refined
3: Output: a refined and conforming triangulation \(\tau_f\)
4: while \(S \neq \emptyset\) do
5: A free thread \(P\) selects a non-locked triangle \(t\) from \(S\)
6: Compute full Lepp\((t)\) (find the terminal triangles)
7: if Collision is not detected while computing Lepp\((t)\) then
8: if Neighbor triangle of terminal triangles is not locked then
9: Lock terminal triangles and neighbor triangles
10: Refine terminal triangles
11: Unlock neighbor triangles and update \(S\).
12: end if
13: end if
14: end while
```
A randomization technique was also used in the three algorithms. The random assignment of the triangles of \( S \) to the processors contributes to minimize the number of collisions produced by parallel processing of overlapping Lepps.

4. Empirical testing

Table 1 shows the behavior of the serial algorithm for an input mesh of approximately three millions of randomly generated input points. The triangles to be refined are randomly selected at each refinement iteration.

Table 1. Behavior of the serial algorithm. Random data (set of points randomly generated)

<table>
<thead>
<tr>
<th>Iter</th>
<th>Mesh Size #</th>
<th>Triangles to be refined</th>
<th>Final mesh size</th>
<th># Added Triangles</th>
<th>Total time (ms)</th>
<th>Average time by Triangle (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5,999,945</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5,999,945</td>
<td>1,199,495</td>
<td>10,789,679</td>
<td>4,789,734</td>
<td>27,955</td>
<td>0.0058</td>
</tr>
<tr>
<td>2</td>
<td>10,789,679</td>
<td>2,157,626</td>
<td>18,192,544</td>
<td>7,402,865</td>
<td>14,191</td>
<td>0.0019</td>
</tr>
<tr>
<td>3</td>
<td>18,192,544</td>
<td>3,637,167</td>
<td>29,741,177</td>
<td>11,548,633</td>
<td>23,906</td>
<td>0.0021</td>
</tr>
<tr>
<td>4</td>
<td>29,741,177</td>
<td>5,950,849</td>
<td>47,694,435</td>
<td>17,953,258</td>
<td>38,014</td>
<td>0.0021</td>
</tr>
</tbody>
</table>

Tables 2, 3 and 4 summarize statistics obtained for the four refinement steps of Table 1 by respectively using Marking Lepp algorithm (PA1), Partial Lepp Storing algorithm (PA2) and Lepp recomputation algorithm (PA3), over an Intel E5-2660 architecture. Figure 2 summarizes the speedup obtained for the three algorithms.

Table 2. Statistics on Marking Lepp algorithm (PA1), random selection. Intel Xeon E5-2660 (2 sockets, 20 cores).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27955</td>
<td>18681</td>
<td>10424</td>
<td>6721</td>
<td>6403</td>
<td>5993</td>
<td>5925</td>
<td>6077</td>
<td>1.5</td>
<td>2.7</td>
<td>4.2</td>
<td>4.4</td>
<td>4.7</td>
<td>4.7</td>
<td>4.6</td>
</tr>
<tr>
<td>2</td>
<td>70102</td>
<td>45561</td>
<td>28463</td>
<td>16945</td>
<td>17324</td>
<td>15409</td>
<td>15585</td>
<td>17032</td>
<td>1.5</td>
<td>2.5</td>
<td>4.1</td>
<td>4.0</td>
<td>4.5</td>
<td>4.5</td>
<td>4.1</td>
</tr>
<tr>
<td>3</td>
<td>136154</td>
<td>93503</td>
<td>54613</td>
<td>36627</td>
<td>32594</td>
<td>32496</td>
<td>32606</td>
<td>37655</td>
<td>1.5</td>
<td>2.5</td>
<td>3.7</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
<td>3.6</td>
</tr>
<tr>
<td>4</td>
<td>240220</td>
<td>165247</td>
<td>95734</td>
<td>62095</td>
<td>64368</td>
<td>61135</td>
<td>62622</td>
<td>72920</td>
<td>1.5</td>
<td>2.5</td>
<td>3.9</td>
<td>3.7</td>
<td>3.9</td>
<td>3.8</td>
<td>3.3</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27955</td>
<td>19512</td>
<td>11295</td>
<td>7651</td>
<td>7015</td>
<td>6532</td>
<td>6421</td>
<td>7317</td>
<td>1.4</td>
<td>2.8</td>
<td>3.7</td>
<td>3.9</td>
<td>4.3</td>
<td>4.4</td>
<td>3.8</td>
</tr>
<tr>
<td>2</td>
<td>70102</td>
<td>50834</td>
<td>30019</td>
<td>20525</td>
<td>19091</td>
<td>17330</td>
<td>17133</td>
<td>17846</td>
<td>1.4</td>
<td>2.3</td>
<td>3.4</td>
<td>3.7</td>
<td>4.1</td>
<td>4.1</td>
<td>3.9</td>
</tr>
<tr>
<td>3</td>
<td>136154</td>
<td>101037</td>
<td>60213</td>
<td>38725</td>
<td>34060</td>
<td>33457</td>
<td>35982</td>
<td>38046</td>
<td>1.4</td>
<td>2.3</td>
<td>3.5</td>
<td>4.0</td>
<td>4.1</td>
<td>3.8</td>
<td>3.6</td>
</tr>
<tr>
<td>4</td>
<td>240220</td>
<td>186306</td>
<td>104086</td>
<td>61763</td>
<td>58282</td>
<td>58567</td>
<td>64411</td>
<td>69762</td>
<td>1.3</td>
<td>2.3</td>
<td>3.9</td>
<td>4.1</td>
<td>4.1</td>
<td>3.7</td>
<td>3.4</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27955</td>
<td>18286</td>
<td>10622</td>
<td>6907</td>
<td>6671</td>
<td>6253</td>
<td>6310</td>
<td>6685</td>
<td>1.5</td>
<td>2.6</td>
<td>4.1</td>
<td>4.2</td>
<td>4.5</td>
<td>4.4</td>
<td>4.2</td>
</tr>
<tr>
<td>2</td>
<td>70102</td>
<td>49107</td>
<td>28083</td>
<td>18178</td>
<td>18036</td>
<td>16546</td>
<td>15000</td>
<td>18858</td>
<td>1.4</td>
<td>2.5</td>
<td>3.9</td>
<td>3.9</td>
<td>4.2</td>
<td>4.7</td>
<td>3.7</td>
</tr>
<tr>
<td>3</td>
<td>136154</td>
<td>100154</td>
<td>57974</td>
<td>32402</td>
<td>32471</td>
<td>32475</td>
<td>31240</td>
<td>37156</td>
<td>1.4</td>
<td>2.4</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
<td>4.4</td>
<td>3.7</td>
</tr>
<tr>
<td>4</td>
<td>240220</td>
<td>181162</td>
<td>101169</td>
<td>58420</td>
<td>54109</td>
<td>54047</td>
<td>57430</td>
<td>65744</td>
<td>1.3</td>
<td>2.4</td>
<td>4.1</td>
<td>4.4</td>
<td>4.4</td>
<td>4.2</td>
<td>3.7</td>
</tr>
</tbody>
</table>
5. Conclusions

An efficient and linear serial implementation was obtained (see Table 1). This requires 0.002 ms for each new triangle introduced in the refined mesh, independently of the mesh size. For each multithread algorithm, the speedup is computed as $T_s/T_p$, where $T_s$ and $T_p$ are the times of the serial and multithread algorithms, respectively.

The analysis of the speedup behavior shows that Lepp recomputation algorithm (PA3) shows a better partially scalable behavior until 12 cores, since this minimizes the number of blocked triangles. The PA1 and PA2 algorithms achieved similar behavior but by below of the algorithm PA3 for 8 to 12 cores.

Note that at each refinement step (Table 1) we have refined approximately 20% of the triangles in the current mesh. Thus an almost global refinement is performed at each iteration which tends to maximize the number of Lepp collisions. A next step in our research will consider dividing the mesh into two parts to be assigned to each socket to reduce inter socket communication.

Acknowledgements

Work partially supported by Departamento de Ciencias de la Computación, Univ. de Chile, Departamento de Sistemas de Información and Research Group GI150115/EF, Univ. del Bio-Bio. We used the supercomputing infrastructure of the NLHPC (ECM-02).

References