Parallel Lepp-bisection algorithm over distributed memory systems

Pedro A. Rodriguez
Department of Information Systems
Universidad del Bio-Bio
Concepcion, Chile
Email: prodrigu@{ubiobio.cl, dcc.uchile.cl}

Maria-Cecilia Rivara
Department of Computer Science
Universidad de Chile
Santiago, Chile
Email: mcrivara@dcc.uchile.cl

Abstract—We discuss a parallel Lepp-bisection algorithm for two-dimensional mesh refinement over distributed memory systems. We discuss the subdivision of the mesh, the assignment of the workload between processors, and the management of interface refinement and communication. We also consider the use of heterogeneous systems having both multi-core and distributed parallel processing.

Keywords—distributed memory; Lepp; Lepp-bisection; longest edge; mesh partition; parallel system; refinement

I. INTRODUCTION

Longest edge bisection algorithms were designed to deal with the iterative and local refinement of triangulations for finite element applications. They guarantee the construction of refined triangulations that maintain the quality of the input mesh [2], [5], [6]. 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; (b) the use of the Lepp concept allows to easily design parallel algorithms.

Parallel longest-edge algorithms that do not use the Lepp-concepts have previously developed: Jones and Plassmann [4] proposed a parallel distributed algorithm based on 4T algorithm to refine 2D triangular meshes, while Castaños and Savage [3] proposed a longest edge bisection algorithm for 2D and 3D meshes. Rivara, Pizarro and Chrisochoides proposed a global refinement algorithm for tetrahedral meshes based on terminal edge refinement. Balman [1] presents a parallel distributed memory algorithm for tetrahedral meshes, which calculates the Lepp of a tetrahedron t₀ and then refines the tetrahedra using the 8-T LE algorithm.

More recently we have discussed an efficient parallel Lepp-bisection refinement algorithm over multicore and shared memory architectures to refine two-dimensional triangulations [7], [8]. Randomization and prefetching techniques were used to develop scalable and efficient multicore algorithm non-dependant of the architecture.

II. A SEQUENTIAL LEPP-BISECTION ALGORITHM

Given any conforming triangulation T, (where the intersection of two adjacent triangles is either a common vertex or a common edge) with bounded smallest angle. The goal is to create a good quality conforming triangulation T_f, where every triangle t in a set S_c is refined.

For any triangle t₀ in T, the Longest-Edge Propagation Path of t₀, or Lepp(t₀), is the ordered list of the triangles t₀, t₁, t₂, ..., t_n−1, t_n, such that t_i is the neighbor triangle of t_{i−1} by the longest edge of t_{i−1} [5]. A Lepp finishes by finding a terminal edge E (a common longest edge for the triangles that share E). The terminal edge is either on the border or shared by two terminal triangles. Then we say that two triangles are terminal triangles if they share the same terminal edge.

The Lepp-bisection algorithm is as follows: every triangle t is refined by repeatedly finding Lepp(t) and the associated terminal triangles which are bisected by midpoint of its longest edge. See Algorithm 1.

Algorithm 1 SerialLeppBisectionEdgeFlip2D(τ, S)

Input: initial mesh τ, set S of triangles to be refined.
Output: Final mesh τ_f.
Find S, the set of marked triangles to be refined.
while S ≠ ∅ do
  For each triangle t₀ ∈ S.
    while t₀ remains in the mesh do
      Compute Lepp(t₀)
      Find terminal edge L and associated terminal triangles.
      bisect terminal triangles by longest edge L.
      Update S.
    end while
  end while

III. PARALLEL DISTRIBUTED ALGORITHM

To design the algorithm we need to deal with:

1 Dividing the input mesh into several homogeneous subpartitions (or submeshes).
2 Assigning roughly a well-balanced workload between processors.
3 Solving conformity problems in those triangles located on the frontier which join two submeshes.
4 Achieving a successful termination. The parallel program must finish in acceptable time, with good performance, and providing a conforming final mesh.

A geometric non-parallel subdivision algorithm was used to provide a set of subpartitions or submeshes to be distributed between the available processors. To deal with the conformity problem on the submesh interfaces, a global list of (interface) frontier edges is initially created. In turn, each processor stores its own local list of frontier edges referencing the global list of frontier edges. Thus each local frontier edge \( E \) (repeated in both processor that share \( E \)) respectively store the list of new points locally created during the submesh refinement. When all the processors finish the refinement process, they communicate to their neighbors the refined frontier edges with the current list of frontier points. Each processor verifies if these points were already inserted. The new points are inserted by applying the Lepp-bisection algorithm. See Algorithm 2.

The process is repeated until every interface edge has an associated empty set of new points. This guarantees that a conforming mesh was obtained.

**Algorithm 2**

```
ParallelMeshRefinementAlgorithm()
1: Input: \( \tau \) the input triangular mesh, \( k \) processors \( P_i \).
2: Output: \( \tau_f \) an output conforming triangular mesh.
3: Create \( k \) subpartitions from \( \tau \). Each subpartition \( i \) corresponds to a submesh \( \tau_i \).
4: Send submesh \( \tau_i \) to processor \( P_i \) (\( i=0,...,n-1 \)).
5: Send lists of frontier edges \( lfe \) to each processor \( P_i \).
6: Each processor \( P_i \) refines submesh \( \tau_i \).
7: All the processors repeatedly send and receive list of frontier edges to carry out the insertion of new points until the mesh is conforming.
8: Each process \( P_i \) sends submesh \( \tau_i \) to master process.
9: Master process receives submeshes \( \tau_f \) from the slave processes.
10: Create final mesh \( \tau_f \) from each \( \tau_f \).
```

**IV. EMPIRICAL STUDY**

We have used triangulations of sets of randomly generated points. We computed the speedup, which is the time of the serial algorithm divided by the \( k \)-processors algorithm time. Table I shows the parallel execution time (which excludes the initial mesh partitioning step) and speedup. Figure 1 shows the curve of speedup. Note that even when the speedup obtained is low, this scales well until 30 processors (cores). Note that, the partitioning algorithm can be parallelized to obtain a full parallel program. On the other hand, according to the obtained parallel time, additional overhead is obtained by the communications between neighboring processes (initial workload assignment and the transmission of new frontier points) and by the local insertions of new frontier points.

At present we are studying new strategies of mesh partitioning and work load assignment to improve the performance of the parallel program. We plan to compare our mesh partition methods with methods provided by PARMETIS library.

<table>
<thead>
<tr>
<th>Parallel execution time (seconds)</th>
<th>1P</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>12P</th>
<th>16P</th>
<th>20P</th>
<th>26P</th>
<th>28P</th>
<th>30P</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>184.7</td>
<td>193.9</td>
<td>102.6</td>
<td>51.9</td>
<td>37</td>
<td>28.4</td>
<td>23.8</td>
<td>22.7</td>
<td>17.9</td>
<td>15.4</td>
</tr>
</tbody>
</table>

**Speedup**

<table>
<thead>
<tr>
<th>Speedup</th>
<th>1</th>
<th>1.0</th>
<th>1.8</th>
<th>3.6</th>
<th>4.9</th>
<th>6.5</th>
<th>7.8</th>
<th>8.1</th>
<th>10.3</th>
<th>12.0</th>
</tr>
</thead>
</table>

**REFERENCES**


