25th International Meshing Roundtable

A Parallel Local Reconnection Approach for Tetrahedral Mesh Improvement

Mengmeng Shanga,b, Chaoyan Zha,b,c, Jianjun Chena,b*, Zhoufang Xiaoa,b, Yao Zhenga,b

aCenter for Engineering and Scientific Computation, Zhejiang University, Hangzhou 310027, China
bSchool of Aeronautics and Astronautics, Zhejiang University, Hangzhou 310027, China
cNingbo Institute of Technology, Zhejiang University, Ningbo 315100, China

Abstract

A multi-threaded parallel local reconnection algorithm is proposed for tetrahedral meshes. It defines a feature point within the region involved in each operation, and sorts the features points along a Hilbert curve. The decomposition of this Hilbert curve results in a load-balanced distribution of local operations. Meanwhile, the regions of concurrently executed local operations are separated far away, such that the possibility of interference is reduced to a very low level. Finally, a parallel mesh improver is developed by combining the proposed algorithm with a parallel mesh smoothing algorithm, and its effectiveness and efficiency is verified in various numerical experiments.

Keywords: Mesh generation; Quality improvement; Unstructured mesh; Parallel algorithm; Multi-threaded

1. Introduction

The Delaunay triangulation (DT) [1-7] and the advancing front technique (AFT) [8, 9] are two of the most successful tetrahedral mesh generation approaches, although both approaches may generate low-quality elements. Firstly, they usually rely on surface inputs, and as a result the quality of a volume mesh is limited by the quality of its surface mesh. Secondly, both approaches are still far away from being perfect. The AFT mainly considers creating an element in each step of forwarding a front. After a number of front-forwarding steps, the fronts that
define the unmeshed region may contain undesired geometry features. Low-quality elements have to be introduced to ensure the termination of the mesh generation process. With respect to the DT based approach, quality guaranteed algorithms have been proposed [8, 9]. However, their 3D versions are still problematic due to the issues of sliver elements and boundary integrity. Therefore, an improvement procedure must be followed after calling an AFT or a DT mesher to ensure the mesh quality meets the requirement of downstream simulations.

Although various improvement approaches have been proposed for tetrahedral meshes, the prevailing ones involve at least two types of local operations. One is smoothing, which repositions mesh points to improve adjacent elements [10-12]. The others are local reconnections [12-20], which replace a local mesh with another mesh that fills up the similar region but has different point connections. A general purpose improver usually needs to combine both types of operations and execute them iteratively [12, 13]. This process is demonstrated to be very time-consuming, in particular when the simulation needs a quality mesh containing hundreds of millions of elements. In our experience, a sequential Delaunay mesh can now generate one hundred million elements in about ten minutes [22]. Owing to the rapid advance of parallel algorithms, the time cost for a parallel mesh can be further reduced to a very low level [24, 25]. However, the following improvement procedure may take several hours to improve this mesh. If a higher standard is set for mesh quality, the time cost for mesh improvement can even grow to be larger than the sequential meshing time by several orders [13]. In this sense, the real performance bottleneck of generating tetrahedral meshes for complicated aerodynamics models lies in the phase of quality improvement rather than mesh generation itself.

Parallelization is a feasible way to speed up the mesh improvement procedure and enable it to handle large-scale meshes. Although parallel meshing algorithms have been extensively investigated in the literature [24-28], much fewer algorithms have been reported on parallel mesh improvement. In general, existing approaches for parallel mesh improvement could be classified into two types: distributed parallel approaches [25-27] and multi-threaded parallel approaches [11, 29].

Presently, distributed parallel approaches are preferred in some studies for their ability to employ sequential algorithms as a black box [25-27]. In these studies, the meshes to be improved (in most cases, these meshes are the outputs of a parallel meshing.) are usually subdivided into the same number of submeshes as the number of parallel processes involved in the mesh improvement task. Then, the input meshes could be improved concurrently by employing the sequential mesh improvement algorithm on each submesh with the inter-domain boundary fixed. However, the main issue is that elements may not be in shape near the inter-domain boundary. A possible solution to this issue is to introduce inter-process local operations to improve the elements in the neighborhood of the inter-domain boundary, based on the same idea as that introduced in a parallel Delaunay meshing [26]. These inter-process operations could be time-consuming because they involve a huge amount of communication and synchronization costs, not to mention the complication of their implementation. As a compromise, Ito et al. suggested a two-stage strategy to deal with this issue [27]. Firstly, the submeshes are improved concurrently with the inter-domain boundary fixed. Secondly, a few layers of elements adjacent to the inter-domain boundary are collected into a single mesh, and then this single mesh is improved sequentially. Evidently, the second stage could become a performance bottleneck due to its sequential nature. Differently, Löhner suggested redistributing the submeshes after the first pass of mesh improvement and then performing a second pass of mesh improvement on the redistributed submeshes [28]. Basically, the second mesh improvement pass could remove most badly shaped elements that the first pass is not able to treat, although survivors might be there if they are near the inter-domain boundary after shifting. Besides, because many elements are sent from the processes with high rank values to neighbouring processes with small rank values in the step of redistributing elements near inter-domain interfaces, the processes with small rank values may have to treat many more elements in the second mesh improvement pass than the processes with high rank values.

The preference of this study is a multi-threaded parallel approach, which attempts to utilize the local properties of mesh improvement operations. The pioneering work of this type was conducted by Freitag et al. [11]. Their parallel smoothing algorithm considers the region covered by elements adjacent to one single mesh point as an individual submesh. In order to avoid the synchronization costs required by the operations of repositioning adjacent mesh points, the mesh points are classified into many independent sets. The points belonging to the same set must not be adjacent to each other, where mesh points of different independent sets are differently colored. Evidently, the smoothing of points in the same point set is parallelizable.
The above parallel smoothing algorithm can improve the mesh quality to a similar level as its sequential counterpart. However, this algorithm cannot reuse the sequential algorithm as a black box because no general schemes can apply the concept of independent sets for all local operations [29]. In contrast, these schemes have to be revised case by case. A main reason is that the parallel algorithm needs to define different types of mesh dual graphs, depending on the types of local operations to be parallelized, to subdivide work loads into different independent sets. For instance, the dual graph used in parallel mesh smoothing considers each mesh point to be smoothed as a graph node, and graph arcs exits between adjacent mesh points. However, for a 2D edge flip operation, to avoid the quad regions involved in concurrently executed flip operations (each region is composed of two triangles sharing an edge) overlap each other, the dual graph considers each mesh edge as a graph node, and graph arcs exist between mesh edges meeting at one ending node [29]. Apparently, this may complicate the parallel implementation of a mesh improver greatly, in particular when this improver may incorporate quite a few types of local operations.

Apart from the high complexity of implementation, another drawback of extending the approach based on independent sets for parallel local reconnection is due to the fact that local reconnection operations change mesh topology while smoothing operations do not. As a result, if we attempt to enhance the mesh improvement effect by executing several passes of local reconnection operations consecutively, we need to renew the mesh dual graph at the end of each pass, while this renew step is unnecessary for mesh smoothing.

In this study, a different parallel approach is developed for parallel local reconnections, which is an extension of the approach proposed for Delaunay triangulation in [30]. Our approach is based on the following observation: if a few local reconnection operations are geometrically separated enough, the possibility of overlaps between the mesh regions involved in different operations should be very rare. In the case of no overlapping, we execute these operations in parallel; otherwise, we simply give up the execution of some local operations such that the remaining operations do not interfere each other. If the possibility of overlapping is low enough, the sacrificed performance costs due to the simple technique resolving the overlapping issue could be reduced to an acceptable level.

We demonstrate the efficiency and effectiveness of the new approach by parallelizing the local reconnection scheme based on the edge removal operation. This operation is considered to be one of the most powerful local reconnection operations for mesh quality improvement in previous studies. Meanwhile, we re-implement a graph partitioning based parallel mesh smoothing algorithm. Combining the parallelized local reconnection scheme and mesh smoothing algorithm, we finally develop a multi-threaded parallel tetrahedral mesh improver. Experiments show that the current version of this improver could achieve a speedup of about 8 on a 16 core computer. Meanwhile, the mesh quality achieved by the parallel improver is comparable to that achieved by its sequential counterpart.

2. The parallel local reconnection approach

2.1. Local reconnection operations for mesh improvement

In the early stage of mesh improvement studies, the most frequently used local reconnection technique for tetrahedral meshes is based on elementary flips [14], including 2-3, 3-2 and 4-4 flips (note that the numbers in these names denote the number of tetrahedra removed and created by the flips, respectively, see Figures 1a and 1b). Because the elementary flips simply make a selection from several possible configurations within a relatively small region, their effectiveness in mesh quality improvement is usually confined. To overcome this limit, three advanced flips that involve more elements were later suggested, i.e., edge removal [15], multi-face removal [16] and multi-face retriangulation [17] (see Figure 1c). They enrich the possible configurations within relatively larger regions and therefore behave more effectively in mesh quality improvement than the elementary flips.

As an initial step to demonstrate the proposed parallel approach, the edge removal operation is selected for parallelization in this study. This is because this operation is vastly applied in many state-of-the-art mesh improvers, e.g., Grummp and Stellar. It is worthy of noting that the proposed parallel approach for edge removal could be easily extended for other local reconnection operations. We will complete these extensions in the near future. However, we will focus on the parallelization of edge removal only in this study.
2.2. Edge removal based local reconnection scheme: sequential implementation

If one edge or face of a bad element is removed, the element will be removed accordingly. Based on this fact, Algorithm 1 presents a local reconnection scheme that attempts to remove bad elements by removing the edges or faces of these elements.

All of bad elements (referring to elements having angles below 30° or above 150° here) are stored in a heap in an ascending order of the element quality. The edge removal routine is then called on an edge of the first element of the heap. If the element is removed, the edge removal routine succeeds; otherwise, the routine is repeated on another edge of the element until all edges of the element are attempted. To protect the mesh boundary, the edges attempted for removal must be interior edges of the mesh.

To avoid an infinite execution of the loop defined in Lines 2-11 of Algorithm 1, no matter the bad element for removal is removed or not, this element must be removed from the heap before the next iteration.

2.3. Edge removal based local reconnection scheme: parallel implementation

2.3.1 The basic idea. Each edge removal operation changes the topology of a local mesh only. This mesh is composed of elements meeting at one edge, referred to as the shell of the edge hereafter. If we want to execute multiple edge removal operations concurrently, the involved shells must not overlap each other. In other words, if a single element is included by one of these shells, this element must not be included by other shells. Evidently, if the involved shells are geometrically separated enough, the possibility of overlaps between them should be very rare.

Inspired by the work for parallel Delaunay point insertion [30], the idea of separating a sequence of edge removal
operations takes the following steps:

- **Step 1.** For a sequence of edge removal operations to be executed, we define a *feature point* for each operation. For instance, the feature point could be located at the geometrical center of the shell within which the operation is performed. As a result, we get a sequence of feature points, which is dual to the sequence of edge removal operations.
- **Step 2.** We sort the sequence of feature points along a Hilbert curve [6, 30].
- **Step 3.** Given the number of threads for the parallel execution, we separate the sorted feature points into the same number of parts. Each part contains a subset of consecutively numbered feature points, and the sizes of these subsets are approximately equal.
- **Step 4.** The edge removal operations dual to each subset of feature points are executed in each thread in order.

Algorithm 1. Sequential implementation of the edge removal based local reconnection scheme

```plaintext
localReconnection(M)

Inputs:
the mesh to be improved, denoted M

Variables:
the heap that stores all of bad elements, T_{bad}
1. Insert all of bad elements into T_{bad} in the ascending order of the element quality
2. while T_{bad} is not empty
3. t: the first element of T_{bad}
4. If t has been removed from M
5. goto Line 11
6. E = {e_1, e_2, …, e_n}: the set of edges of t qualified for removal (n <= 6)
7. for j = 1 to n
8. edgeRemoval(e_j)
9. if t is removed
10. goto Line 11
11. Remove t from T_{bad}
```

![Fig.2. A sequence of randomly distributed points and the result after Hilbert sorting: (a) Before sorting; (b) after sorting](image-url)
A simple analysis could reveal that the above strategy could ensure the edge removal operations concurrently executed in different threads are geometrically separated enough. Assuming that 10,000 edge removal operations will be executed in 4 threads. The above strategy could ensure any two feature points dual to two edge removal operations executed on neighboring threads are separated by about 2500 other points in the resorted sequence. Since the Hilbert sorting technique could result in a new sequence where the points with neighboring indices are usually geometrically close (see Figure 2 for an example), the distance of two feature points separated by 2500 points are most likely to be very large. In other words, the possibility that the shells corresponding to these feature points overlap each other should be very rare.

2.3.2 Parallel implementation. Based on the idea introduced in Section 2.3.1, Algorithm 2 presents a parallel implementation of Algorithm 1. The parallel programming tool OpenMP is adopted for this multi-threaded parallel implementation. Since the edge removal operations are defined on a sequence of bad elements included in the current mesh, a feature point is defined for each bad element, referring to the centroid of the element. After sorting the sequence of feature points, the sequence of bad elements is resorted as well. Meanwhile, the resorted sequence of bad elements is subdivided into $M_1$ subsets. Here, $M_1$ is the number of available threads. Note that all of non-boundary edges of a bad element might be selected for removal when the proposed algorithm attempts to remove a bad element. Therefore, the cavity, i.e., the local mesh that might be changed by one single successful element removal operations, should be the union of the shells of all of non-boundary edges of the element (see Line 7 of Algorithm 2). Rarely, the cavities of concurrently executed operations may overlap each other. In such cases, the threads involved in these overlaps and ranked with bigger values will give up their execution of edge removal operations, while other threads will execute edge removal operations as normal.

Algorithm 2. The first parallel implementation of Algorithm 1.

1. Let $T_{bad}$ be the set of elements to be improved
2. hilbertSort($T_{bad}$)
3. #pragma omp parallel num_threads($M_1$)
   {
   4.   $i = \text{omp}_\text{get}_\text{thread}_\text{num}()$;
   5.   for ($k=0$; $k < \text{size}(T_{bad})/M_1$; $k++$) {
   6.       index = $k + i\ast\text{size}(T_{bad})/M_1$;
   7.       cavities[$i$] = the union of shells of non-boundary edges of $T_{bad}$ [index]
   8. #pragma omp barrier
   9.   if (noOverlap(cavities[$i$]))
   10.      rmvAnElembyER($T_{bad}$ [index], cavities[$i$]); //remove an element by edge removal
   11. #pragma omp barrier
   }
}

In Algorithm 2, a thread needs to synchronize its execution twice with other threads when managing a single element (Line 8 and Line 11 of Algorithm 2). The first synchronization ensures all of threads step into the overlap check simultaneously, and the second synchronization ensures all of threads treat next elements simultaneously. As a result, the percentage of timing costs induced by synchronization operations is large enough to set obstacles for an efficient parallel execution. To reduce this percentage, we developed an improved version of Algorithm 2 (see Algorithm 3), following the suggestion by Remacle et al. [30] for their parallel Delaunay point insertion algorithm. In Algorithm 3, a thread treats $M_2$ elements simultaneously; therefore, the synchronization callings are reduced by nearly $M_2$ times. It is worthy of noting that the possibilities of overlaps between concurrently executed operations might increase along with the increase of $M_2$. Therefore, a suitable value of $M_2$ should balance its advantage and disadvantage. In the present study, $M_2$ is set to be 32 in default, although an in-depth study is necessary to evaluate the impact of different $M_2$ values on the parallel efficiency of Algorithm 3.
Algorithm 3. The improved parallel implementation of Algorithm 1.

1. Let \( T_{bad} \) be the set of elements to be improved
2. \( \text{hilbertSort}(T_{bad}) \)
3. \#pragma omp parallel num_threads(\( M_1 \))
   
   \begin{algorithmic}
   \STATE \( i = \text{omp\_get\_thread\_num}() \);
   \FOR \( (k = 0; k < \text{size}(T_{bad})/(M_1 \times M_2); k++) \)
   \STATE \( \text{index}[p] = k + i \times \text{size}(T_{bad})/(M_1 \times M_2) + p \times \text{size}(T_{bad})/M_2; \)
   \STATE \( \text{cavities}[i][p] = \text{the union of shells of non-boundary edges of } T_{bad}[\text{index}] \)
   \ENDFOR
   \STATE \#pragma omp barrier
   \FOR \( (p = 0; p < M_2; p++) \)
   \IF \( \text{noOverlap(cavities}[i][p]) \)
   \STATE \( \text{rmvAnElembyER}(T_{bad}[\text{index}][p], \text{cavities}[i][p]) // \text{remove an element by edge removal} \)
   \ENDIF
   \ENDFOR
   \STATE \#pragma omp barrier
   \ENDFOR
   \STATE \#pragma omp barrier
   \ENDalgorithmic}

3. The parallel mesh smoothing algorithm

To achieve the cost-effectiveness, we combine an optimization-based algorithm [20] with the Laplacian smoothing to reposition each interior mesh point that is included by at least one bad element (referred to as a bad point hereafter). To reposition a mesh point, we perform Laplacian smoothing firstly. If the improved ball (referring to all the elements incident at the point) no longer contains bad elements, the smoothing succeeds; otherwise, we perform the optimization-based smoothing. To save the smoothing time, a mesh point is flagged as \textit{smoothed} after a successful smoothing, and this flag is flushed only if the ball of the point is changed. In each smoothing cycle, all of non-smoothed bad points are treated only once. In each smoothing pass, the smoothing cycle is repeated until three indicators of the mesh quality are not improved further: (1) the quality of the worst tetrahedral \( (q_{\text{worst}}) \); (2) the number of bad elements \( (n_{bad}) \); and (3) the average quality of bad elements \( (q_{\text{aver}}) \). Note that the minimum sine of dihedral angles is used as a default quality measure in this study. The quality measure of a mesh is evaluated by a vector listing the quality of each tetrahedron contained by the mesh, in an order from the worst to the best. Since the worst tetrahedron in a mesh has far more influence than those average tetrahedra, the quality vectors of two meshes are compared lexicographically so that, for instance, an improvement in the second-worst tetrahedron improves the overall mesh quality even if the worst tetrahedron has not changed.

We parallelize the above mesh smoothing algorithm by following the suggestion of Freitag \textit{et al.} [11]. The parallel algorithm considers the region covered by elements adjacent to one single mesh point as an individual submesh, e.g., the shaded elements around a mesh point \( p \) in Figure 3. In order to avoid the synchronization costs required by the operations of repositioning adjacent mesh points, the mesh points are classified into many independent sets. The points belonging to the same set must not be adjacent to each other, as shown in Figure 3, where mesh points of different independent sets are differently coloured. Based on independent sets, the smoothing procedure is rescheduled as Algorithm 4, where the main computation lies in the inner loop described by Lines 5 and 6. This computation is parallelizable because the smoothing function callings in Line 6 can be executed concurrently.

On a shared memory computer, the schemes like Algorithm 4 can be parallelized easily. For instance, if OpenMP is adopted, a line such as `\#pragma omp parallel for` before Line 5 of Algorithm 1 will dispatch concurrent tasks onto available threads. Besides, a synchronization barrier is required after Line 8 to ensure all threads can enter the
smoothing step for the next independent set simultaneously.

Algorithm 4. The mesh smoothing algorithm based on independent sets.

1. \( k = 0 \)
2. Let \( S_0 \) be the initial set of mesh points marked for smoothing
3. \( \textbf{while} \ S_k \neq \emptyset \)
4. Choose an independent set \( I \) from \( S_k \)
5. \( \textbf{for each} \ v \in I \)
6. \( x' = \text{smooth}(x_v, \text{adj}(v)) \)
7. \( S_{k+1} = S_k \setminus I \)
8. \( k = k + 1 \)

It is beneficial to minimize the number of independent sets. However, it proves a NP-hard problem for the graph partitioning problem with this minimization goal [11]. In the present study, a rather simple heuristic algorithm is developed, which takes the following steps to color a graph:

- Step 1. Select an uncolored graph node as the seed and give it a new color.
- Step 2. Starting from the seed node, traverse other nodes according to the adjacency indices of nodes. For those uncolored nodes, if they are not directly adjacent to the nodes with the new color, color them with the new color.
- Step 3. If all graph nodes are colored, exit the routine; otherwise, go back to Step 1.

In our experience, the above algorithm usually needs about 10 colors to finish the coloring procedure of a mesh. This result is acceptable, although a smaller number of colors could be achieved by improving this coloring algorithm further. In terms of timing performance, the main issue is not due to the above coloring process, but due to the process of creating the node adjacency graph. Luckily, this more time-consuming process is parallelizable.

4. The overall parallel mesh improvement schedule

Algorithm 5 presents the proposed mesh improvement schedule, which combines smoothing and local reconnection schemes to improve the mesh quality. This schedule begins with a smoothing pass, and then executes a main loop of mesh improvement. In the main loop, a smoothing pass is followed after a pass of local reconnection to improve the mesh quality further. The main loop is ended when two subsequent combinational passes fail to make sufficient progress or the number of iteration steps exceeds a predefined threshold (in the present study, the default value of this threshold is 5). We gauge progress using three quality indicators mentioned in Section 3, i.e., \( q_{\text{worst}} \), \( n_{\text{bad}} \) and \( q_{\text{aver}} \).
Algorithm 5. The proposed mesh improvement schedule

\textbf{Algorithm 5. The proposed mesh improvement schedule}

\textbf{improveAMesh}($M$)  
\textbf{Input:}  
\textbf{Variables:}
\begin{itemize}
  \item $q_{\text{worst}}$, $q'_{\text{worst}}$, the quality of the worst tetrahedral
  \item $n_{\text{bad}}, n'_{\text{bad}}$, the number of bad elements
  \item $q_{\text{aver}}, q'_{\text{aver}}$, the average quality of bad elements
\end{itemize}
\begin{enumerate}
  \item $\text{failed} = 0$; $\text{itcount} = 0$
  \item Improve $M$ by the parallel mesh smoothing scheme
  \item Query the mesh quality and store the indicators in $q_{\text{worst}}, n_{\text{bad}}$ and $q_{\text{aver}}$, respectively
  \item \textbf{while} $\text{failed} < 3$ \&\& $++\text{itcount} <= 5$
  \item Improve $M$ by the parallel local reconnection scheme
  \item Improve $M$ by the parallel mesh smoothing scheme
  \item Query the mesh quality and store the indicators in $q'_{\text{worst}}, n'_{\text{bad}}$ and $q'_{\text{aver}}$, respectively
  \item \textbf{if} ($q'_{\text{worst}} < q_{\text{worst}} \| n'_{\text{bad}} > n_{\text{bad}} \| q'_{\text{aver}} < q_{\text{aver}}$) $\text{failed} = \text{failed} + 1$
  \item \textbf{else} $\text{failed} = 0$
  \item $q_{\text{worst}} = q'_{\text{worst}}; n_{\text{bad}} = n'_{\text{bad}}; q_{\text{aver}} = q'_{\text{aver}}$
\end{enumerate}

5. Numerical results

The tests presented here are conducted on a computer node of a Dawning cluster. This node is composed of two 8-core CPUs (CPU: 2.6GHz; Memory: 64GB). Four meshes of various magnitudes are selected in the tests, and these meshes are all the initial meshes output by our in-house Delaunay mesher [7], i.e., the default mesh improvement option of this mesher is switched off when we create these meshes.

Table 1 lists the timing and speed-up data when the proposed parallel mesh improvement algorithm is executed in different numbers of threads (referred to as $M_1$ hereafter). The total timing costs decrease continuously when the value of $M_1$ is doubled, but at a slower speed. The maximal speedup, i.e., 8.49, is achieved when the Shuttle’s mesh is improved in 16 threads. For each mesh input, Figure 5 draws a curve to show how the speedup values vary against $M_1$.

To analyze the factors that may do harm to parallel efficiency when $M_1$ increases, Table 2 lists the timing data of the main steps involved in the proposed parallel algorithm, and the F16 mesh is selected in this analysis.

\begin{itemize}
  \item \textit{Graph creation and coloring}. As mentioned in Section 3, the process of creating node adjacency graphs is more time-consuming than the process of coloring these graphs. This step totally consumes about 105.38s when executed in sequential. The time cost is reduced to 77.63s, 41.69s, 25.68s and 15.90s when $M_1$ increases to 2, 4, 8 and 16, respectively.
  \item \textit{Parallel smoothing}. In general, the speedup achieved in this step is acceptable: a speedup of 9.78 times could be achieved when $M_1=16$. The major factor that limits a better parallel efficiency for this step is that the fraction of sequential codes is still too high. How to parallelize those parallelizable sequential codes will be the focus on this step in our next-step study.
  \item \textit{Hilbert sorting}. Although this step is presently executed in sequential, the fraction of its timing cost is rather small. For instance, it only consumes about 2.5% of the total time when $M_1=16$. Thus, we do not think it is a major factor impacting the overall parallel efficiency, although it might be necessary to parallelize this step when more threads are involved in the parallel mesh improvement.
  \item \textit{Parallel edge removal}. Although the speedup achieved in this step is acceptable (a speedup of 6.88 times is achieved when $M_1=16$), the parallel efficiency of this step is not as good as the parallel meshing smoothing step. This is mainly because more synchronization costs are involved in this step. Meanwhile, a single calling of the
edge removal operation is much faster than that of the smoothing operation. How to reduce the synchronization costs further would be the focus on the efficiency improvement of this step.

![Fig. 4. Test mesh inputs: (a) Two spheres; (b) Shuttle; (c) London Tower bridge; (d) F16 aircraft.](image)

Table 1. General timing performance data

<table>
<thead>
<tr>
<th>Inputs</th>
<th>#Elements</th>
<th>#Bad elements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two spheres</td>
<td>6,431,731</td>
<td>793,132</td>
</tr>
<tr>
<td>Shuttle</td>
<td>13,322,593</td>
<td>1,728,705</td>
</tr>
<tr>
<td>Tower bridge</td>
<td>51,191,207</td>
<td>6,600,397</td>
</tr>
<tr>
<td>F16 aircraft</td>
<td>60,268,861</td>
<td>7,423,946</td>
</tr>
</tbody>
</table>

To verify the feasibility of the new idea for parallel local reconnection, we collect the times of reporting overlaps by Line 11 of Algorithm 3 (\(n_{\text{overlap}}\)). In accordance with our expectation, the times of reporting overlaps is negligible, compared with the huge numbers of edge removal callings in these mesh improvement processes. For instance, when \(M_1=16\) and \(M_2=1\), \(n_{\text{overlap}}\) is equal to 5 and 6 for the two sphere model and the F16 aircraft model, respectively. If we increase \(M_2\) up to its default value (i.e., 32), \(n_{\text{overlap}}\) increases to 263 and 313 for these two models, respectively.

In terms of practical applications, the proposed parallel mesh improvement algorithm has achieved acceptable parallel efficiency. When applied in a meshing pipeline for large-scale meshes, the proposed algorithm can reduce
the timing cost consumed in the step of quality improvement remarkably. For instance, our sequential in-house Delaunay mesher consumes about 280s to create the initial mesh of the F16 aircraft model, which is composed of about 60 millions of elements. However, a sequential run of the proposed mesh improvement algorithm consumes 1453.3s, which is about 5.19 times slower than the sequential run of the Delaunay mesher. In this sense, the real performance bottleneck of this totally sequential meshing pipeline lies in the phase of quality improvement rather than mesh generation itself. However, if executed in 16 threads, the proposed mesh improvement algorithm only consumes 182.5 seconds, which is slightly fewer than the time cost consumed by the in-house mesher.

Nevertheless, further improvement is necessary and possible, for instance, for the proposed parallel local reconnection algorithm. It was reported that in [30] that the parallel Delaunay triangulation algorithm, which has a similar code structure as Algorithm 3, could achieve a speedup of 18 by using 32 computer cores. In other words, the parallel efficiency is about 0.56. Our local reconnection algorithm could usually achieve parallel efficiency above 0.56 when the number of computer cores ($N_{core}$) is 8 or below. However, a remarkable drop of the parallel efficiency is observed when $N_{core}$ increases up to 16. Initial investigations reveal that at least two facts might account for this performance loss. Firstly, the average loads per thread decreases when $N_{core}$ increases such that the relevant ratio of synchronization costs becomes higher. Secondly, in order to protect shared data, a few frequently called functions used in the present parallel local reconnection algorithm includes undesirable ‘locks’. As a result, the performance drop due to intensive data competition becomes larger when $N_{core}$ increases. Presently, we are working on improving the implementations of these functions.

Stability is a key performance index for parallel mesh improvement algorithms, which evaluates whether a parallel algorithm could improve the quality of a mesh to a comparable level with the result of the sequential algorithm. To evaluate the stability of the proposed algorithm, Figure 6 presents the distributions of dihedral angles of different meshes of the two sphere model and the F16 aircraft model. It is observed that the curves for the meshes after parallel improvement almost overlap their counterparts for the meshes after sequential improvement. Meanwhile, the meshes after parallel improvement are much better than the initial meshes.

![Fig. 5. The curves showing how speedup values vary against the number of threads](image)

| Table 2. Breakdown of the timing performance data in the test for the F16 aircraft model |
|---------------------------------|-----------------|--------|--------|--------|--------|
| Main steps                      | Timing costs with various numbers of threads (seconds)   |
|                                | 1                | 2      | 4      | 8      | 16     |
| Smoothing                      | Smooth creation and coloring | -      | 77.63  | 41.69  | 25.68  | 15.90  |
|                                | Smoothing        | 1159.20| 618.76 | 330.16 | 187.65 | 118.57 |
| Edge removal                   | Hilbert sorting  | -      | 3.20   | 1.61   | 1.09   | 1.443  |
|                                | Edge removal     | 294.07 | 156.04 | 92.00  | 51.71  | 40.92  |
| Other steps                    | -                | 4.58   | 3.74   | 5.24   | 5.66   |
| Total                          | 1453.3           | 860.2  | 462.9  | 271.4  | 182.5  |
6. Conclusions

A multi-threaded parallel improvement algorithm is proposed for tetrahedral meshes, and different techniques are developed to parallelize the mesh smoothing scheme and the local reconnection scheme adopted in the algorithm. The parallelization of mesh smoothing operations is based on an existing data decomposition technique, which colors the dual graph of the mesh to subdivide mesh points into a few independent sets. The parallelization of local reconnection operation is based on a new data decomposition technique, which defines a feature point in the interior of each local reconnection operation, and sorts the features points along a Hilbert curve. The decomposition of this Hilbert curve could result in a load-balanced distribution of local reconnection operations on active threads. This distribution features its ability to ensure that the regions of local operations concurrently executed in different threads are usually separated far away, such that the possibility of interference is reduced to a very low level. As a result, even if the interference happens, we could simply withdraw those local reconnection operations that cause interference, and it was observed that no obvious negative impact was introduced by this simple strategy. Experiments show that the current version of this improver could achieve a speedup of about 8 on a 16 core computer. Meanwhile, the mesh quality achieved by the parallel improver is comparable to that achieved by its sequential counterpart.

Acknowledgements

The authors appreciate the joint support for this project by the National Natural Science Foundation of China (Grant Nos. 11172267, 11432013, 10872182), the Joint Fund of the National Natural Science Foundation of China and China Academy of Engineering Physics (Grant No. U1630121) and Zhejiang Provincial Natural Science Foundation (Grant No. LR16F020002 and Y1110038).

References


[18] George PL, Borouchaki H. Back to edge flips in 3 dimensions. In: 12th International Meshing Roundtable, Santa Fe, NM, USA, 2003; 393-402.


