MeshCleaner: A Generic and Straightforward Algorithm for Cleaning Finite Element Meshes

Gang Mei[1,2] · **Salvatore Cuomo**[3] · **Hong Tian**[4] · **Nengxiong Xu**[1,2] · **Linjun Peng**[5]

Received: 27 February 2017 / Accepted: 28 April 2017 / Published online: 6 May 2017

Abstract Mesh cleaning is the procedure of removing duplicate nodes, sequencing the indices of remaining nodes, and then updating the mesh connectivity for a topologically invalid Finite Element mesh. To the best of our knowledge, there has been no previously reported work specifically focused on the cleaning of large Finite Element meshes. In this paper we specifically present a generic and straightforward algorithm, MeshCleaner, for cleaning large Finite Element meshes. The presented mesh cleaning algorithm is composed of (1) the stage of compacting and reordering nodes and (2) the stage of updating mesh topology. The basic ideas for performing the above two stages efficiently both in sequential and in parallel are introduced. Furthermore, one serial and two parallel implementations of the algorithm MeshCleaner are developed on multi-core CPU and/or many-core GPU. To evaluate the performance of our algorithm, three groups of experimental tests are conducted. Experimental results indicate that the algorithm MeshCleaner is capable of cleaning large meshes very efficiently, both in sequential and in parallel. The presented mesh cleaning algorithm MeshCleaner is generic, simple, and practical.

Keywords Finite Element mesh · Data structure · Mesh topology · Parallel algorithm

✉ Gang Mei
gang.mei@cugb.edu.cn

[1] Department of Geological Engineering, Qinghai University, Xining, China

[2] School of Engineering and Technology, China University of Geosciences, Beijing, China

[3] Department of Mathematics and Applications "R. Caccioppoli", University of Naples Federico II, Naples, Italy

[4] Faculty of Engineering, China University of Geosciences, Wuhan, China

[5] Academician Pioneering Park, Dalian University, Dalian, China

1 Introduction

Finite Element meshes are widely used in the numerical simulation, shape modeling, computer graphics, etc. A topologically valid Finite Element mesh is composed of a set of nodes and a set of elements formed by nodes. In other words, there are at least two geometric primitives in any valid Finite Element mesh, i.e., the nodes and the elements. After some kinds of manipulations such as mesh Boolean operations, mesh refinement [2,10,16,26], mesh coarsening [1], mesh optimization [6,23] or mesh merging in parallel mesh generation [8,9,13,32], the nodes and/or the elements are removed or added, and the mesh topology is modified. In this case, the changed mesh needs to be cleaned to become valid again.

The reason why mesh cleaning is typically needed to be performed for Finite Element meshes is as follows. In general, when conducting the numerical analysis using Finite Element Method (FEM), duplicate nodes are not allowed since the system/global stiffness matrix will be singular if there are duplicate nodes; and in this case no correct or reliable analysis results can be achieved. Moreover, it would be better that all of the nodes in the same mesh are sequenced (i.e., continuously indexed). This can help reduce the computer memory requirement for storing the system stiffness matrix. Furthermore, it is much easier to assemble the system stiffness matrix due to easy access to the entries in both the element and system stiffness matrix. In summary, cleaned meshes is in general required in FEA. This is also the primary objective to carry out the presented work in this paper.

The primary objective of mesh cleaning is to make the changed mesh topologically valid again by merging potential duplicate vertices in the uncleaned mesh. In most cases, duplicate nodes, i.e., differently indexed nodes with the same coordinates and attributes, are not allowed. Hence, after conducting a specific kind of mesh modification, all nodes need to be merged, checked, and newly indexed. Furthermore, after the merging and re-indexing of all nodes, the mesh topology is also needed to be refreshed. In summary, in the process of mesh cleaning the vertices are needed to be merged and the topology is needed to be updated.

Nowadays, Finite Element meshes can be quite large. For example, in the CFD simulation of aircraft, the Finite Element models may be composed of one billion elements or even more. In this case, when needed to locally refine the large Finite Element mesh and then clean the refined mesh, the computational cost may be too expensive. In these cases, the computational efficiency needs to be improved. An effective and practical strategy to improve the computational efficiency is to perform the mesh cleaning in parallel on various parallel computing platforms such as multi-core CPUs, many-core GPUs, or even clusters. To the best of our knowledge, there has been no previously reported work specifically focused on the cleaning of large Finite Element meshes, especially on the parallel mesh cleaning on multi-core CPUs and/or many-core GPUs.

In this paper, we specifically present a generic and straightforward algorithm for cleaning large Finite Element meshes. The presented mesh cleaning algorithm is termed as MeshCleaner, and is implemented on multi-core CPU and/or many-core GPU architectures with the use of OpenMP [20], Thrust [11], and CUDA [19].

To demonstrate the effectiveness of the MeshCleaner, we apply it to combine large tetrahedral Finite Element meshes.

This paper is organized as follows. In Sect. 2, the motivation of cleaning Finite Element mesh is stated. In Sect. 3, the presented mesh cleaning algorithm, MeshCleaner, is described in details. Moreover, an application example of MeshCleaner is presented in Sect. 4 to demonstrate the effectiveness. In Sect. 5, the features of the algorithm MeshCleaner, including the advantages and shortcomings, are analyzed. Finally, the presented work is concluded in Sect. 6.

2 Problem Statement

In this section, for the sake of clarity when introducing our work, we specifically present several background concepts and definitions.

Mesh Representation. Typically, a mesh can be simply represented with two types of components, i.e., the list of nodes and the list of elements [12]. The list of nodes usually holds the index, coordinates, and attribute of each node, while the list of elements stores the index of each element, the indices (not the coordinates) of nodes forming the element, and the attribute of each element.

In this study, we specifically focus on those meshes represented with the above mesh representation form. The widely used `.OFF` format (Object File Format) is the simplest one of the above-mentioned mesh representation forms, and the mesh format employed in the famous tetrahedral mesh generator TetGen [28] also falls into this mesh representation form.

Cleaned Mesh. The cleaned mesh is the type of mesh with the following features:

1) There are no duplicate nodes with the same coordinates;
2) The indices of all nodes are sequenced (i.e., continuously listed);
3) All elements are represented by the already cleaned nodes.

The above features of cleaned meshes are typically required in Finite Element Analysis (FEA). In general, when conducting the numerical analysis using Finite Element Method (FEM), duplicate nodes are not allowed since the system / global stiffness matrix will be singular if there are duplicate nodes; and in this case no correct or reliable analysis results can be achieved. Moreover, it would be better that all of the nodes in the same mesh are sequenced (i.e., continuously indexed). This can help reduce the computer memory requirement for storing the system stiffness matrix. Furthermore, it is much easier to assemble the system stiffness matrix due to easy access to the entries in both the element and system stiffness matrix. In summary, cleaned meshes is in general required in FEA. This is also the primary objective to carry out the presented work in this paper.

The mesh cleaning is a procedure specifically for generating cleaned mesh. There are two major sub-procedures in the mesh cleaning.

- Sub-procedure 1: the generation of cleaned nodes by merging and reordering all vertices;
- Sub-procedure 2: the generation of cleaned elements by updating the mesh topology using the already cleaned nodes

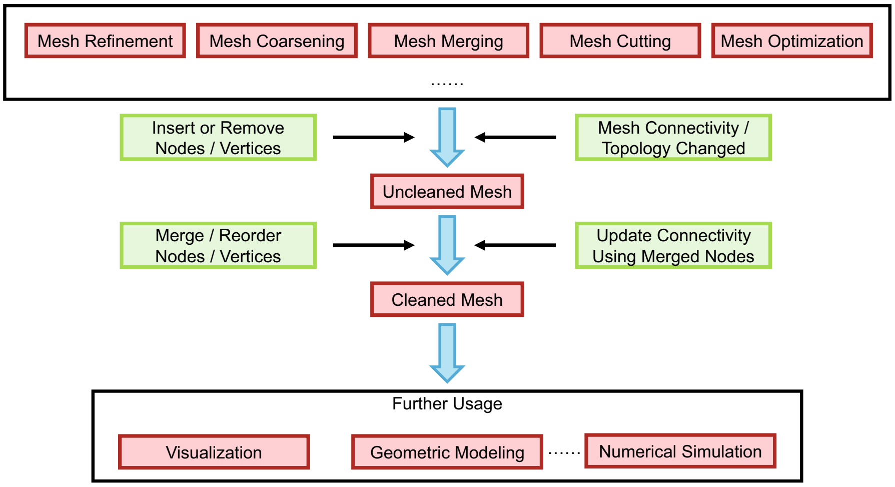


Fig. 1 Motivation of performing mesh cleaning

After performing the mesh cleaning, cleaned meshes can be obtained and then used for further mesh manipulation such as mesh refinement [16,26], mesh coarsening [1] or used as the computational mesh models for numerical analysis. The motivation of conducting the mesh cleaning is illustrated in Fig. 1.

3 The Proposed Algorithm: MeshCleaner

3.1 Overview

In general, a mesh can be described using a list of vertices and a list of elements. Correspondingly, the merging of meshes typically consists of two stages: (1) the compacting of all vertices by removing duplicate vertices and then reordering the IDs of the remaining nodes, and (2) the refreshing of all elements by updating the nodal IDs in each element; see an extremely simple illustration of mesh cleaning in Fig. 2. In the following content, we will describe the basic ideas behind the above two stages in detail.

3.2 The Basic Ideas Behind Merging and Reordering Nodes

The first task of mesh cleaning is to compact the vertices by deleting duplicate vertices and renumbering the remaining points. After performing some mesh modifications such as local mesh refinement or cutting part of meshes, the vertices are newly added or removed. In this case, duplicate vertices may exist, and the IDs of the vertices are probably no longer numbered continuously. However, in a topologically valid mesh, duplicate vertices are not allowed, and also the IDs of all vertices are typically continuous for easy accesses. Therefore, the new list of vertices after carrying out mesh operations needs to be first compacted.

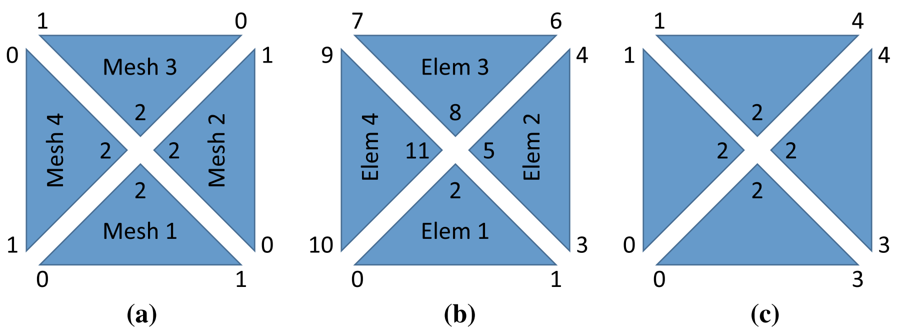
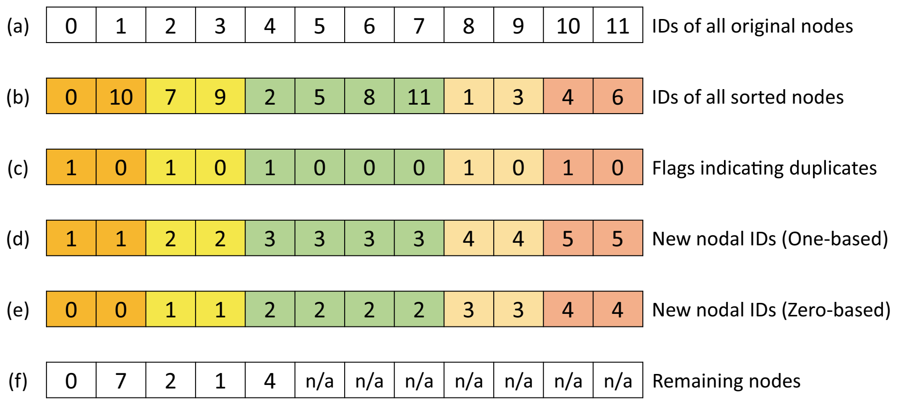


Fig. 2 An extremely simple example to illustrate the procedure of mesh cleaning. **a** Multiple meshes, **b** combined and uncleaned mesh, **c** required and cleaned mesh



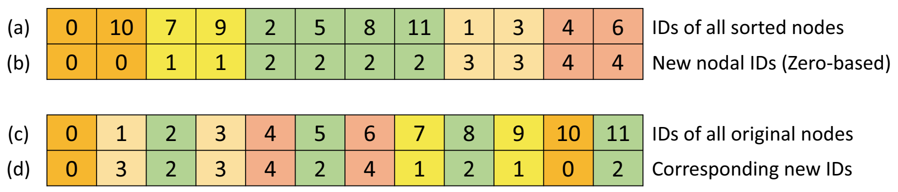
(a)	0	1	2	3	4	5	6	7	8	9	10	11	IDs of all original nodes
(b)	0	10	7	9	2	5	8	11	1	3	4	6	IDs of all sorted nodes
(c)	1	0	1	0	1	0	0	0	1	0	1	0	Flags indicating duplicates
(d)	1	1	2	2	3	3	3	3	4	4	5	5	New nodal IDs (One-based)
(e)	0	0	1	1	2	2	2	2	3	3	4	4	New nodal IDs (Zero-based)
(f)	0	7	2	1	4	n/a	n/a	n/a	n/a	n/a	n/a	n/a	Remaining nodes

Fig. 3 The basic ideas behind the merging and reordering of nodes

In the algorithm MeshCleaner, the compacting of vertices can be quite efficiently realized by utilizing the parallel primitives of sorting and scanning. The workflow of compacting the vertices are listed as follows.

1) Sort all nodes ascendingly according to the nodal coordinates.
2) Check duplicate nodes by comparing the coordinates of any pair of sorted adjacent nodes.
3) Mark each duplicate node with a specific flag.
4) Assign the new and continuous IDs for the sorted and checked nodes.
5) Remove the duplicate nodes and keep the unique ones.

Here we employ a simple illustration to demonstrate the above workflow of compacting vertices. As shown in Fig. 2, after combing four quite simple triangular meshes, there are totally 12 vertices before mesh cleaning. These 12 vertices are first simply numbered continuously without taking duplications into account; see Fig. 3a. In the first step, those 12 vertices are sorted according to the coordinates; see Fig. 3b. Then, each pair of adjacent vertices is checked to identify whether they are duplicate nodes. Obviously, there are several pairs of duplicate nodes, e.g., the nodes 0 and 10. We use specific flags to indicate the duplicates: 0 if duplicate and 1 if not. In this case, an array of flags consisting of ones and zeros is formed; see Fig. 3c. We then perform an inclu-



(a)	0	10	7	9	2	5	8	11	1	3	4	6	IDs of all sorted nodes
(b)	0	0	1	1	2	2	2	2	3	3	4	4	New nodal IDs (Zero-based)
(c)	0	1	2	3	4	5	6	7	8	9	10	11	IDs of all original nodes
(d)	0	3	2	3	4	2	4	1	2	1	0	2	Corresponding new IDs

Fig. 4 The basic ideas behind the updating of mesh topology

sive scanning for the array of flags to obtain the new nodal IDs; see Fig. 3d. Noticeably, the new nodal IDs, in this case, are one-based; and the corresponding zero-based IDs can be easily achieved by subtracting the value of ones; see Fig. 3e. It should be noticed that those new nodal IDs will further be used to determine the relationship between the old nodal IDs and the new nodal IDs. Finally, the required list of cleaned nodes can be obtained by removing the duplicate vertices according to the nodal flags; see Fig. 3f.

The above workflow of merging and reordering vertices can be easily parallelized. First, the sorting of vertices according to the coordinates can be obviously parallelized using the parallel merging sorting algorithm [24]. Then, each pair of adjacent vertices can be checked independently, and thus the checking for pairs of adjacent vertices can be conducted in parallel. Third, the inclusive scanning that is employed to calculate the new nodal IDs can also be carried out in parallel using the prefix sum algorithm [27]. Also, the transforming of one-based nodal IDs to zero-based ones can be quite easily parallelized.

3.3 The Basic Ideas Behind Updating Mesh Topology

The updating of mesh topology is much easier than that of merging vertices. After merging and reordering the nodes, each of the remaining nodes receives a new ID. That is, there is a corresponding relationship between the old IDs and the new IDs of all nodes. In this sense, for each node in an element, its new ID can be quite easily determined according to the relationship between the new and old IDs. The relationship can be obtained by re-sorting all the sorted nodes along with the new nodal IDs by using the old nodal IDs as the keys for sorting.

This can be also quite easily illustrated using an extremely simple example; see Figs. 2 and 4. After obtaining the zero-based new nodal IDs, there is, in fact, a one-to-one relationship between the old nodal IDs (Fig. 4a) and the new nodal IDs (Fig. 4b). However, when updating the mesh topology by refreshing the nodal indices in all elements, the access to the new nodal ID of a specific node, for example, the node with the old ID 9, needs to perform a searching in the array of old IDs of the first sorted nodes (Fig. 4a). This updating of mesh topology that needs searching new nodal ID is computationally inefficient.

To improve the computational efficiency of updating the mesh topology, we avoid the searching of the new ID for each node. Our solution is that: we re-sort all the sorted nodes along with the new nodal IDs by using the old nodal IDs as the keys for sorting; see the re-sorted results in Fig. 4c, d. After re-sorting, a new relationship is between the old nodal IDs and the new nodal IDs. In this case, the new ID for each node can be quite easily accessed by exploiting the new relationship.

The updating of mesh topology can also be easily conducted in parallel. First, the re-sorting of the sorted nodes can also be parallelized using the parallel merging sort by keys. Second, the refreshing of the nodal IDs in an element can be obviously performed independently; and this means the refreshing for all elements can be carried out in parallel.

3.4 Serial and Parallel Implementations

The mesh cleaning algorithm, MeshCleaner, is implemented on both multi-core CPU and many-core GPU platforms by exploiting the OpenMP API [20], Thrust library [11], and CUDA parallel computing model [19]. In this subsection, we will introduce more details on one serial and two parallel implementations.

It should be also noted that the source code of the above three implementations is available at: https://figshare.com/s/0cd4233b8bf10ba4fd0e.

3.4.1 Serial Implementation with STL

The serial implementation of the algorithm, MeshCleaner, is developed by coherently according to the basic ideas behind the merging of vertices and the updating of mesh topology; see Figs. 3 and 4. For the sake of simplicity, we develop this implementation by employing the vector containers and efficient algorithms such as sorting that are provided in the `C++ STL` (Standard Template Library).

The serial implementation is rather simple with the use of STL. First, we create two vector containers to store the original nodes and elements before mesh cleaning, then use the `std::sort()` function to sort all nodes according to the nodal coordinates. Third, we check the duplicate nodes by comparing the coordinates of any pair of adjacent nodes and accordingly set the flags for indicating the duplicates. According to the flags indicating duplicates, the new nodal IDs can be very easily obtained by summarization. After obtaining the new nodal IDs, to form the relationship between the new IDs and the old IDs, we use the `std::sort` function again to sort the old IDs along with the new IDs. Finally, we perform a loop over all the elements to update the nodal IDs in each element according to the formed relationship between the old IDs and new IDs.

3.4.2 Parallel Implementation on Multi-core CPU

As mentioned in Sects. 3.2 and 3.3, both of the two major stages, i.e., the merging of vertices and the updating of mesh topology can be quite easily parallelized. To improve the computational efficiency, we develop a parallel implementation on multi-core CPU by exploiting the interface OpenMP [20] and the library Thrust [4,5]. Note that in the library Thrust the parallel computing primitives such as sorting and scanning can be executed on either multi-CPUs or many-core GPUs [3].

This parallel implementation developed on multi-core CPU is quite similar to the serial implementation version described in Sect. 3.4.1. First, we create two Thrust vector containers rather than STL vector containers to store the original nodes and elements before mesh cleaning; then use the `thrust::sort()` rather than the `std::sort()` function to sort all the nodes according to the nodal coordinates in parallel. Third, we check the duplicate nodes in parallel by using the OpenMP directive "`#pragma omp parallel for`" and accordingly set the flags for indicating the duplicates. According to the flags indicating duplicates, the new nodal IDs can be very easily obtained by performing a parallel inclusive scanning procedure using the efficient function `thrust::inclusive_scan()`. After obtaining the new nodal IDs, to form the relationship between the new IDs and the old IDs, we use the `thrust::sort()` rather than the `std::sort()` function again to sort the old IDs along with the new IDs. And finally, we update the nodal IDs in each element in parallel according to the formed relationship between the old IDs and new IDs by using the OpenMP `parallel_for` directive.

3.4.3 Parallel Implementation on Many-Core GPU

We also implement the mesh presented algorithm, MeshCleaner, on a single CUDA-enabled GPU. The GPU implementation can also be roughly divided into two major stages: (1) the compacting/merging of all vertices in parallel, and (2) the updating of mesh topology in parallel. Several quite efficient parallel primitives such as GPU-accelerated sorting [24] and scanning [27], which are provided by the thrust library [4,5] in the CUDA programming model, are directly used in our GPU implementation.

After reading the uncleaned mesh data, we first allocate the global memory on the device and then transfer the mesh data on the host to the device. Second, We sort all the original and uncleaned vertices according to the coordinates in parallel by employing the quite efficient parallel sorting primitive `thrust::sort()`. Third, we design a CUDA kernel to specifically check whether any pair of adjacent points duplicate, and meanwhile mark the duplicate vertices with specific flag value. In this CUDA kernel, each thread is responsible for checking one pair of adjacent points and setting the flag value. Then, we use the parallel primitive `thrust::inclusive_scan()` to obtain the one-based new IDs of the sorted vertices. Moreover, another quite simple CUDA kernel is designed to transfer all the one-base new IDs into to zero-based IDs (Fig. 3e). We then re-sort all the previously sorted vertices according to the old nodal IDs using the efficient parallel primitive `thrust::sort()`. And in this case, the relationship between the old nodal IDs and the new nodal IDs is created. The duplicate vertices can be removed in this time. According to this relationship, we invoke another CUDA kernel to update the mesh topology, where each thread in the CUDA kernel takes the responsibility to update the nodal IDs in only one element.

4 Application Examples

To evaluate the performance of the algorithm MeshCleaner, in this section we will provide three groups of experimental tests by applying our implementation to combine several parts of tetrahedral meshes into an entire one.

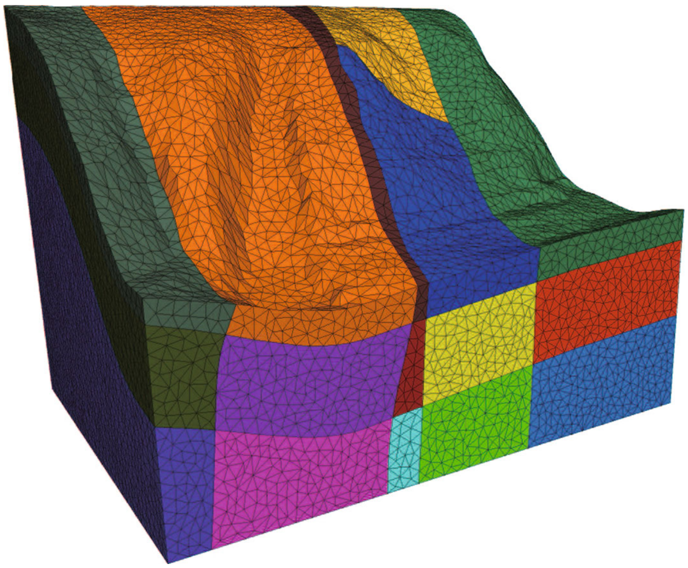


Fig. 5 One of the mesh models in the first group of testing data

4.1 Experimental Environment

The experimental tests are performed on a machine featured with an Intel i7-3610QM processor (2.30 GHz), 6 GB of memory and an NVIDIA GeForce GTX660M graphics card. The graphics card GTX 660M has 2 GB of RAM and 384 cores. All the experimental tests have been evaluated using the Visual Studio 2010 and CUDA toolkit version 8.0 on Window 7 Professional.

4.2 Experimental Testing Data

Three groups of datasets have been created for testing. Each group of testing data consists of three or four tetrahedral meshes that are generated from the same geological model. The geological models are created by using our own geological modeling software ROCKModel [30,31]. There are several separated subregions/blocks in a geological model. Each block is tetrahedralized individually by the famous tetrahedral mesh generator TetGen [28]; and then the entire mesh model of the geological model is generated by combing those tetrahedral meshes in all subregions. Obviously, there are many duplicate vertices along the boundaries of the subregions, and mesh cleaning needs to be performed. More details on the three groups of testing data are described as follows.

4.2.1 The First Group of Testing Data

The first group of testing data consists of three mesh models with different sizes of nodes and elements. Those three mesh models are created by meshing the same

Table 1 Details on the mesh models in three groups of testing data

Group	Size	Num. of blocks	Num. of node	Num. of element	Figures
1st	Size 1 (0.95M)	18	173,906	950,846	Fig. 5
	Size 2 (3.24M)	18	561,948	3,237,982	–
	Size 3 (9.32M)	18	1,551,401	9,325,636	–
2nd	Size 1 (0.32M)	13	66,512	316,332	–
	Size 2 (1.05M)	13	196,083	1,053,906	Fig. 6
	Size 3 (2.31M)	13	397,341	2,309,047	–
	Size 4 (8.70M)	13	1,475,943	8,698,098	–
3rd	Size 1 (0.63M)	3	113,572	629,559	Fig. 7
	Size 2 (2.14M)	3	367,602	2,138,802	–
	Size 3 (6.17M)	3	1,009,115	6,116,335	–

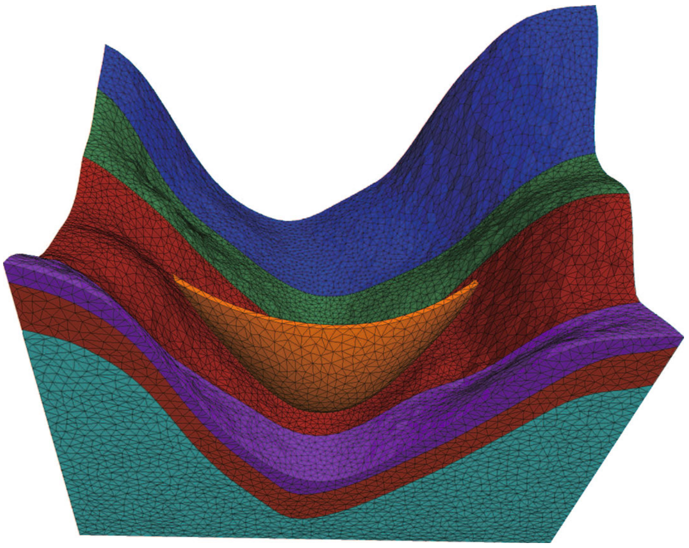


Fig. 6 One of the mesh models in the second group of testing data

geological slope model with different meshing configurations; see Fig. 5 for one of the mesh models. The numbers of subregions/blocks, nodes, and elements in each mesh model are listed in Table 1.

4.2.2 The Second Group of Testing Data

The second group of testing data consists of four mesh models with different sizes of nodes and elements. Those four mesh models are created by meshing the same geological dam model with different meshing configurations; see Fig. 6 for one of the mesh models. The numbers of subregions/blocks, nodes, and elements in each mesh model are listed in Table 1.

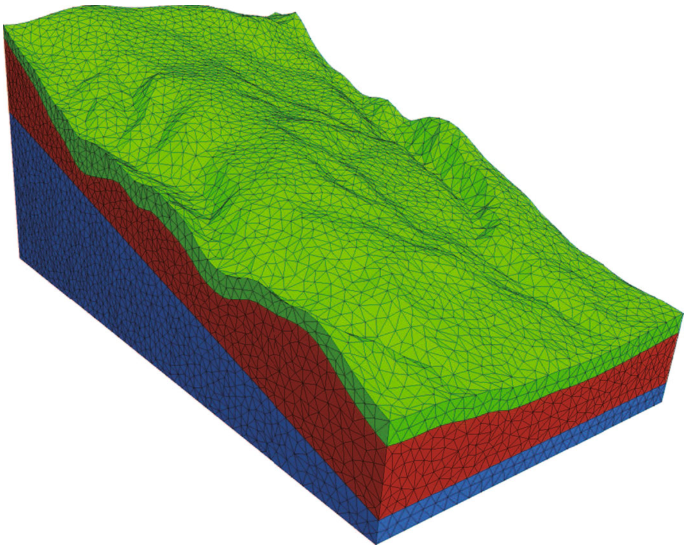
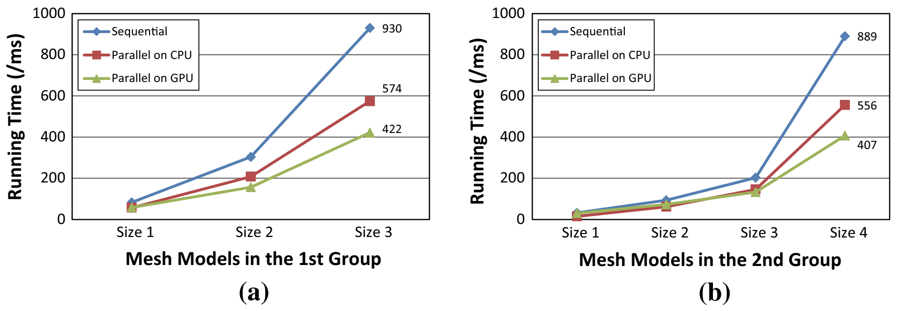


Fig. 7 One of the mesh models in the third group of testing data



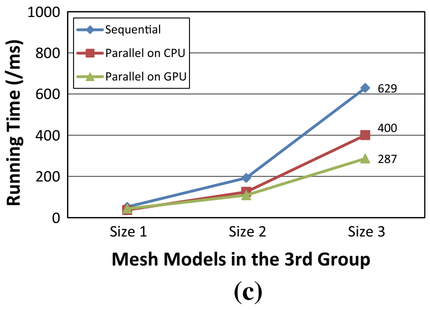


Fig. 8 Comparison of the running time of three implementations for three groups of testing data. **a** Running time for the first group of testing data, **b** running time for the second group of testing data, **c** running time for the third group of testing data

Table 2 Experimental results of the first group of testing data

Meshes in 3rd group	Num. of nodes and elements			Running time (/ms)			MFLOPS		
	Nodes uncleaned	Nodes cleaned	Elements	Serial	Parallel on CPU	Parallel on GPU	Serial	Parallel on CPU	Parallel on GPU
Size 1 (0.95M)	173,906	156,498	950,846	83	57	58	83.36	124.60	165.97
Size 2 (3.24M)	561,948	522,960	3,237,982	304	208	156	87.81	134.40	200.15
Size 3 (9.32M)	1,551,401	1,472,816	9,325,636	930	574	422	93.49	161.33	205.85

Table 3 Experimental results of the second group of testing data

Meshes in 3rd group	Num. of nodes and elements			Running time (/ms)			MFLOPS		
	Nodes uncleaned	Nodes cleaned	Elements	Serial	Parallel on CPU	Parallel on GPU	Serial	Parallel on CPU	Parallel on GPU
Size 1 (0.32M)	66,512	56,301	316,332	32	15	30	62.50	91.39	126.08
Size 2 (1.05M)	196,083	175,703	1,053,906	93	62	73	86.09	130.56	170.23
Size 3 (2.31M)	397,341	376,961	2,309,047	203	146	133	89.53	139.60	191.38
Size 4 (8.70M)	1,475,943	1,398,367	8,698,098	889	556	407	93.06	158.76	202.87

Table 4 Experimental results of the third group of testing data

Meshes in 3rd group	Num. of nodes and elements			Running time (/ms)			MFLOPS		
	Nodes uncleaned	Nodes cleaned	Elements	Serial	Parallel on CPU	Parallel on GPU	Serial	Parallel on CPU	Parallel on GPU
Size 1 (0.63M)	113,572	105,164	629,559	52	36	43	62.09	94.29	142.66
Size 2 (2.14M)	367,602	348,894	2,138,802	192	125	108	79.13	129.08	184.22
Size 3 (6.17M)	1,009,115	971,081	6,116,335	629	400	287	86.86	146.66	208.22

4.2.3 The Third Group of Testing Data

The third group of testing data consists of three mesh models with different sizes of nodes and elements. Those three mesh models are created by meshing the another geological slope model with different meshing configurations; see Fig. 7 for one of the mesh models. The numbers of subregions/blocks, nodes, and elements in each mesh model are also listed in Table 1.

4.3 Experimental Results

The computational performance of applying the algorithm MeshCleaner for cleaning three groups of combined meshes is presented in this subsection. Note that the running time presented in this work includes the overhead of transferring data between the host side (CPU) and the device side (GPU). However, the running time consumed for inputting the testing data from files on the disk and outputting the results to files on the disk is ignored.

The running time and FLOPS [7,21] of the three implementations for the three groups of testing data is listed in Tables 2, 3, and 4, respectively. Also, the running time for the same group of testing data is compared in Fig. 8.

According to the experimental results, it can be observed that: (1) all of the three implementations are quite efficient to clean large Finite Element meshes since the execution time is less than 1 second for all test data and implementations; (2) the parallel implementation on multi-core CPU is approximately 1.5 times faster than the fast serial implementation; and (3) the parallel implementation on many-core GPU is approximately 2.0 times faster than the fast serial implementation.

It can be easily learned that both of the two parallel implementations are faster than the serial implementation due to the use of multiple threads. However, the advantage of the parallel implementation over the serial implementation is not evident. We will analyze this in the Discussion section.

5 Discussion

In this paper, we have presented a specific algorithm, termed as MeshCleaner, for cleaning large Finite Element meshes. The mesh cleaning algorithm, MeshCleaner, is composed of two major stages, i.e., the compacting and reordering of all nodes and the updating of the mesh topology. We have introduced our basic ideas for performing the above two stages efficiently, both in serial and in parallel.

We have developed three efficient implementations of the algorithm MeshCleaner. The first and serial implementation is developed on the CPU with the use of STL. The second and parallel implementation is realized by exploiting the power of multi-core CPU with the aid of OpenMP and Thrust. The third and parallel one is developed on many-core GPU by using the CUDA programming model.

To evaluate the performance of our algorithm, we have created three groups of testing data and applied our algorithm to the real-world applications. In the following subsections, we will analyze the experimental results, capabilities, and limitations of the proposed algorithm.

5.1 Performance Comparison of Three Implementations

In Sect. 4.3, we have evaluated the computational performance of three implementations using three groups of testing data. It has been observed that: (1) those three implementations are quite efficient to clean large Finite Element meshes since the execution time is less than 1 second for all test data and implementations; (2) the parallel implementation on multi-core CPU is about 1.5 times faster than the fast serial implementation; and (3) the parallel implementation on many-core GPU is approximately 2.0× faster than the fast serial implementation.

Both of the two parallel implementations are faster than the serial implementation due to the use of multiple threads. However, the advantage of the parallel implementation over the serial implementation in terms of the computational efficiency is not significant. This is probably due to the following reasons. In the algorithm MeshCleaner, the most computationally expensive procedure is the sorting of nodes. In the serial implementation, by employing the function `std::sort()`, the computational performing has already been highly optimized and improved. When exploiting the power of multi-core CPU or many-core GPU to speed up the sorting of nodes in parallel, the performance gain in sorting nodes is not evident.

Similar conclusions can be drawn between the two parallel implementations. The parallel implementation on many-core GPU is only slightly faster than the parallel version on multi-core CPU. This is because the most computational intensive procedures in MeshCleaner are the sorting and inclusive scanning of nodes. The above two procedures have already been highly optimized on multi-core CPU, and no significant improvement in the efficiency can be achieved when mapping the parallel sorting and scanning from multi-core CPU to many-core GPU.

5.2 The Simplicity of the Algorithm MeshCleaner

The presented algorithm MeshCleaner is quite simple in the aspects of algorithmic design and implementation. The only complex procedure is the sorting of nodes according to nodal coordinates, which has the complexity of $O(n\log n)$. Most of other procedures are quite straightforward and have the linear complexity. In addition, the sorting of nodes can be quite easily realized by adopting existing, quite efficient functions such as `std::sort` and `thrust::sort()`. And most of other procedures are quite suitable to be performed in parallel because of very less data dependency. In summary, the algorithm MeshCleaner is straightforward and easy to implement in both sequential and parallel.

5.3 The Generality of the Algorithm MeshCleaner

The algorithm MeshCleaner is generic and applicable for any type of valid Finite Element meshes. As described in Sect. 3, there are two major stages in in the algorithm

MeshCleaner: (1) the compacting and reordering of nodes, and (2) the updating of the mesh topology. The first stage is generic for most Finite Element mesh since the most basic geometric primitive is the node/vertex. Any operations that are conducted for nodes/vertices in a mesh can also be directly applied in another mesh, while the only difference is that in some meshes the nodes are two-dimensional and in some of other meshes the nodes are three or even higher dimensional. The second stage is also generic for valid Finite Element meshes. In the second stage, it is only needed to learn which nodes form an element; and then the IDs of those nodes in an element is updated from the old, uncleaned indices to the new, cleaned indices. Obviously, no specific type of element is needed at this stage. In summary, the algorithm MeshCleaner is generic and can be applied to clean valid Finite Element meshes.

5.4 The Shortcomings of the Algorithm MeshCleaner

Although the algorithm MeshCleaner is straightforward and generic, it has shortcomings when implementing it on multi-core CPU and/or many-core GPU. The most obvious shortcoming is the implementation of sorting nodes. The sorting of nodes is the most computationally expensive procedure in MeshCleaner. By employing the function `std::sort()`, the sorting of nodes is still quite efficient even performed in sequential. However, when needed to sort extremely large size of nodes, for example, 1 billion nodes [15], the sorting only in sequential would probably not be satisfied in terms of speed. Thus, typically the sorting needs to be performed in parallel, which calls for highly efficient parallel sorting approaches and implementations. In our parallel implementations, we directly employ the function `thrust::sort()` to sort nodes in parallel. The parallel sorting provided by Thrust library is quite flexible and easy to use, but it has been reported that the parallel sorting primitive in Thrust is not as efficient as several of its counterparts [14,22,25]. Hence, performance gains probably could be achieved by improving the efficiency of parallel sorting procedure in MeshCleaner.

5.5 Outlook and Future work

As analyzed in Sect. 5.4, the most computationally intensive procedure in MeshCleaner is the sorting of nodes. And performance gains probably could be achieved by improving the efficiency of parallel sorting procedure in MeshCleaner. Thus, future work is planned to focus on this issue.

Another future work is probably to optimize the GPU implementation by taking the data layouts for efficiently representing meshes on the GPU. Data layout is the form for organizing data. There are two commonly used data layouts, i.e., the Structure-of-Arrays (SoA) and the Array-of-Structures (AoS) [17,18]. For the sake of easy programming, the data layout AoS is adopted in our three implementations. In future, we plan to re-write all the three implementations by using the layout SoA and then evaluate the computational performance.

Our algorithm MeshCleaner is currently implemented and evaluated on a single GPU and a single machine. To be able to handle extremely large size mesh models such as those composed of billions of elements [15,29], we also plan to extend our algorithm to run on multi-GPUs or even Clusters.

6 Conclusion

In this paper, we have presented a generic and straightforward algorithm, MeshCleaner, specifically for cleaning large Finite Element meshes. The presented mesh cleaning algorithm, MeshCleaner, is composed of two major stages: (1) the compacting and reordering of all nodes and (2) the updating of the mesh topology. We have introduced our basic ideas for performing the two stages efficiently both in sequential and in parallel. We have developed one serial and two efficient parallel implementations of the algorithm MeshCleaner, where the two parallel implementations are developed on multi-core CPU and many-core GPU, respectively. To evaluate the performance of our algorithm, we have created three groups of testing data and applied our algorithm to the real-world applications. We have found that the algorithm MeshCleaner is capable of cleaning large meshes quite efficiently, both in sequential and in parallel. Our mesh cleaning algorithm is generic, simple, and practical.

Acknowledgements This research was supported by the Natural Science Foundation of China (Grant Nos. 11602235, 51674058 and 41602374), China Postdoctoral Science Foundation (2015M571081), and the Fundamental Research Funds for the Central Universities (2652015065).

References

1. Alhadeff, A., Leon, S.E., Celes, W., Paulino, G.H.: Massively parallel adaptive mesh refinement and coarsening for dynamic fracture simulations. Eng. Comput. **32**(3), 533–552 (2016). doi:10.1007/s00366-015-0431-0
2. Antepara, O., Lehmkuhl, O., Borrell, R., Chiva, J., Oliva, A.: Parallel adaptive mesh refinement for large-eddy simulations of turbulent flows. Comput. Fluids **110**, 48–61 (2015). doi:10.1016/j.compfluid.2014.09.050
3. Barlas, G.: Chapter 7—the thrust template library. In: Barlas, G. (ed.) Multicore and GPU Programming, pp. 527–573. Morgan Kaufmann, Boston (2015). doi:10.1016/B978-0-12-417137-4.00007-1
4. Bell, N., Hoberock, J.: Chapter 26–thrust: a productivity-oriented library for CUDA. In: Hwu, W.M.W. (ed.) GPU Computing Gems, Jade Edition, Applications of GPU Computing Series, pp. 359–371. Morgan Kaufmann, Boston (2012). doi:10.1016/B978-0-12-385963-1.00026-5
5. Bell, N., Hoberock, J., Rodrigues, C.: Chapter 16-thrust: a productivity-oriented library for CUDA. In: Kirk, D.B., Hwu, W.M.W. (eds.) Programming Massively Parallel Processors, 2nd edn, pp. 339–358. Morgan Kaufmann, Boston (2013). doi:10.1016/B978-0-12-415992-1.00016-X
6. Chen, J., Zheng, J., Zheng, Y., Xiao, Z., Si, H., Yao, Y.: Tetrahedral mesh improvement by shell transformation. Eng. Comput. (2016). doi:10.1007/s00366-016-0480-z
7. Cuomo, S., De Michele, P., Piccialli, F.: 3D data denoising via nonlocal means filter by using parallel gpu strategies. Comput. Math. Methods Med. **2014**, 14 (2014). doi:10.1155/2014/523862
8. Feng, D., Chernikov, A.N., Chrisochoides, N.P.: Two-level locality-aware parallel delaunay image-to-mesh conversion. Parallel Comput. **59**, 60–70 (2016). doi:10.1016/j.parco.2016.01.007
9. Freitas, M.O., Wawrzynek, P.A., Cavalcante-Neto, J.B., Vidal, C.A., Carter, B.J., Martha, L.F., Ingraffea, A.R.: Parallel generation of meshes with cracks using binary spatial decomposition. Eng. Comput. **32**(4), 655–674 (2016). doi:10.1007/s00366-016-0444-3
10. Hatipoglu, B., Ozturan, C.: Parallel triangular mesh refinement by longest edge bisection. SIAM J. Sci. Comput. **37**(5), C574–C588 (2015). doi:10.1137/140973840
11. Hoberock, J., Bell, N.: Thrust—a parallel algorithms library (2017). https://thrust.github.io/

12. Lage, M., Martha, L.F., Moitinho de Almeida, J.P., Lopes, H.: Ibhm: index-based data structures for 2d and 3d hybrid meshes. Eng. Comput. (2015). doi:10.1007/s00366-015-0395-0
13. Laug, P., Guibault, F., Borouchaki, H.: Parallel meshing of surfaces represented by collections of connected regions. Adv. Eng. Softw. **103**, 13–20 (2017). doi:10.1016/j.advengsoft.2016.09.003
14. Leischner, N., Osipov, V., Sanders, P.: GPU sample sort. In: 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–10 (2010). doi:10.1109/IPDPS.2010.5470444
15. Lo, S.: 3D delaunay triangulation of 1 billion points on a PC. Finite Elem. Anal. Des. **102C103**, 65–73 (2015). doi:10.1016/j.finel.2015.05.003
16. Lu, Q.K., Shephard, M.S., Tendulkar, S., Beall, M.W.: Parallel mesh adaptation for high-order finite element methods with curved element geometry. Eng. Comput. **30**(2), 271–286 (2014). doi:10.1007/s00366-013-0329-7
17. Mei, G., Tian, H.: Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. Springerplus **5**, 104 (2016). doi:10.1186/s40064-016-1731-6
18. Mei, G., Tipper, J.C., Xu, N.: A generic paradigm for accelerating laplacian-based mesh smoothing on the GPU. Arab. J. Sci. Eng. **39**(11), 7907–7921 (2014). doi:10.1007/s13369-014-1406-y
19. NVIDIA: CUDA (Compute Unified Device Architecture) (2017). http://www.nvidia.com/object/cuda_home_new.html
20. OpenMP_ARB: The OpenMP API Specification for Parallel Programming (2017). http://www.openmp.org/
21. Palma, G., Comerci, M., Alfano, B., Cuomo, S., Michele, P.D., Piccialli, F., Borrelli, P.: 3D non-local means denoising via multi-GPU. In: 2013 Federated Conference on Computer Science and Information Systems, pp. 495–498 (2013)
22. Ranokphanuwat, R., Kittitornkun, S.: Parallel partition and merge QuickSort (PPMQSort) on multicore CPUs. J. Supercomput. **72**(3), 1063–1091 (2016). doi:10.1007/s11227-016-1641-y
23. Sastry, S.P., Shontz, S.M.: A parallel log-barrier method for mesh quality improvement and untangling. Eng. Comput. **30**(4), 503–515 (2014). doi:10.1007/s00366-014-0362-1
24. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–10 (2009). doi:10.1109/IPDPS.2009.5161005
25. Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D., Dubey, P.: Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pp. 351–362. ACM, New York, NY, USA (2010). doi:10.1145/1807167.1807207
26. Schepke, C., Maillard, N., Schneider, J., Heiss, H.U.: Online mesh refinement for parallel atmospheric models. Int. J. Parallel Prog. **41**(4), 552–569 (2013). doi:10.1007/s10766-012-0235-4
27. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware. GH '07, pp. 97–106. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2007)
28. Si, H.: TetGen, a delaunay-based quality tetrahedral mesh generator. ACM Trans. Math. Softw. (2015). doi:10.1145/2629697
29. Soner, S., Ozturan, C.: Generating multibillion element unstructured meshes on distributed memory parallel machines. Sci. Program. (2015). doi:10.1155/2015/437480
30. Xu, N., Tian, H.: Wire frame: a reliable approach to build sealed engineering geological models. Comput. Geosci. **35**(8), 1582–1591 (2009). doi:10.1016/j.cageo.2009.01.002
31. Xu, N., Tian, H., Kulatilake, P.H., Duan, Q.: Building a three dimensional sealed geological model to use in numerical stress analysis software: a case study for a dam site. Comput. Geotech. **38**(8), 1022–1030 (2011). doi:10.1016/j.compgeo.2011.07.013
32. Yilmaz, Y., Ozturan, C.: Using sequential NETGEN as a component for a parallel mesh generator. Adv. Eng. Softw. **84**, 3–12 (2015). doi:10.1016/j.advengsoft.2014.12.013