6^{th} European Conference on Computational Mechanics (ECCM 6)
7^{th} European Conference on Computational Fluid Dynamics (ECFD 7)
11 ... 15 June 2018, Glasgow, UK

PARALLEL OPTIMIZATION OF TETRAHEDRAL MESHES

D. Benítez[1], **E. Rodríguez**[1], **J.M. Escobar**[1], **R. Montenegro**[1]

[1] University Institute for Intelligent Systems and Numerical Applications in Engineering (SIANI), University of Las Palmas de Gran Canaria, 35017 Las Palmas, Spain, domingo.benitez@ulpgc.es, www.dca.iusiani.ulpgc.es/proyecto2015-2017

Key words: Mesh Optimization, Mesh Smoothing, Mesh Untangling, Parallel Computing, Performance Evaluation.

Abstract. We propose a new algorithm on distributed-memory parallel computers for our simultaneous untangling and smoothing of tetrahedral meshes [9, 10]. A previous parallel implementation on shared-memory computers is analyzed in [1]. The new parallel procedure takes ideas from Freitag et al. strategy [11]. The method is based on: partitioning a mesh, optimizing interior vertices, optimizing boundary vertices of interior partitions, and communicating updated coordinates of boundary vertices. This paper presents performance evaluation results of our parallel algorithm. We apply the procedure in the mesh generation of several 3-D objects by using the Meccano method [4]. High levels of speed-up are obtained in the mesh optimization step of this method. However, several bottlenecks may limit the parallelism. We provide some hypotheses about the factors that cause more parallel overhead. The relative number of elements, that are located at the interfaces of the sub-domains of the object, is one of the more important aspects for the efficiency of the parallel mesh optimization.

1 INTRODUCTION

When a mesh is inverted, standard finite element simulation algorithms generally cannot obtain an appropriate numerical approach of problems based on partial differential equations (PDE). Thus, researchers recommend to untangle the mesh prior to analysis using software packages for finite element analysis (FEA).

Mesh optimization techniques reduce the total time to solve the problem and they improve the accuracy of results. Processing a mesh can spend up to 25% of the overall running time of a PDE-based application [5]. So, for large meshes, it is important that the operations of generation, warping, untangling and smoothing are performed in parallel.

There are several areas of research involving parallel processing of meshes. For example, many mesh processing techniques have been developed to generate meshes in parallel [7]. Additionally, parallel mesh warping algorithms have been developed which employ numerical optimization methods for use in computational simulations with deforming domains [15].

Other parallel algorithms have been proposed for mesh optimization. For distributed-memory architectures: Gorman et al. proposed a smoothing algorithm that maximizes the quality of the worst element by relocating each free vertex [12], and Sastry and Shontz proposed an algorithm that moves all mesh vertices to untangle the mesh and improve the quality of the worst quality elements [16]. In these works, maximum speedups range from 10×@12cores using OpenMP/MPI [12] to 50×@64cores using MPI [16].

For multi-core architectures, we proposed in [1] the first parallel mesh-untangling algorithm that simultaneously improves the quality of all inverted and non-inverted mesh elements. We achieved a maximum speedup of 67×@128cores for highly tangled tetrahedral meshes. For GPUs, Cheng et al. implemented a local optimization algorithm for smoothing 3D meshes on a heterogeneous GPU/multicore system [6], and Zhao et al. implemented a mesh optimization algorithm that improves the quality of 2D meshes by changing the mesh connectivity [17]. In both cases, maximum speedups on GPUs range from 21× [6] to 44× [17].

The main contributions of this paper are: (1) A new single-vertex optimization algorithm for simultaneous mesh untangling and smoothing on distributed-memory parallel computers is proposed in Section 2; (2) Section 3 shows that our parallel algorithm provides high performance for mesh optimization on two fixed tetrahedral meshes that are highly tangled.

2 MESH UNTANGLING AND SMOOTHING ALGORITHM FOR DISTRIBUTED-MEMORY PARALLEL COMPUTERS

Our technique [9] for simultaneous untangling and smoothing of tetrahedral meshes consists of finding the new position (x_v) of a *free node* (v) by optimizing only one objective function (K). This function is based on a measurement of the quality of the local submesh (N_v), which is constituted by the set of elements connected to the free node v. After repeating this process several times for all free nodes of the mesh, quite satisfactory results can be achieved. In our case, the objective function is constructed as the L^1 norm of the vector $(\eta_1, \ldots, \eta_n)$:

$$K = \sum_{i=1}^{n} \eta_i(x_v) \qquad \eta_i(x_v) = \frac{||S_i||_F^2}{3\ h_i^{2/3}} \qquad h_i = \tfrac{1}{2}(\sigma_i + \sqrt{\sigma_i^2 + 4\delta^2}) \tag{1}$$

$$\sigma_i = det(S_i) \qquad \delta = \max\{10^{-3}\bar{\sigma}, \mathrm{Re}(10\sqrt{\epsilon(\epsilon - \sigma_{min})})\} \qquad \bar{\sigma} = \frac{1}{n}\sum_{i=1}^{n} |\sigma_i|$$

$$\sigma_{min} = \min\{\sigma_i\}_{i\in\{1,\ldots,n\}} \qquad \epsilon = 10^{11}\ DBL\_EPS$$

where $||S_i||_F$ is the Frobenius norm of matrix S associated to the affine map from the ideal element (usually an equilateral tetrahedron or triangle) to the physical one.

Specifically, the weighted Jacobian matrix S is defined as $S = A\ W^{-1}$, being $A = (\mathbf{x_1} - \mathbf{x_0}, \mathbf{x_2} - \mathbf{x_0}, \mathbf{x_3} - \mathbf{x_0})$ the Jacobian matrix and $\mathbf{x_k},\ k = 0 \ldots 3$, the spatial coordinates of the vertices of the tetrahedron. The constant matrix W is derived from the ideal element. DBL_EPS is upper bound on the relative error due to rounding in floating point arithmetic. The measurement of element distortion given by equation 1 (η) is taken as unified metric for both inverted and non-inverted elements. For more details, see [9, 10].

In this paper, we propose a new parallel algorithm for this simultaneous mesh untangling and smoothing technique on distributed-memory parallel computers. This algorithm is based on the idea from Freitag et al. [11]. The mesh is divided into a set of partitions. Interior and boundary vertices of partitions are treated separately. Additionally, the vertices that lie on the solid boundary are fixed during all the optimization process.

In each mesh sweep, the interior vertices of all partitions are optimized in parallel. For partition boundaries, independent sets of non-fixed vertices are created; each of them is optimized in parallel after all interior vertices have been optimized. Before another independent set is optimized, a synchronization/communication phase between partitions is required. These computation-synchronization-communication phases are repeated until all partition boundary vertices have been optimized. If the exit conditions are not reached, a new mesh sweep is done.

Algorithm 1 shows our parallel untangling and smoothing algorithm that includes one serial and three parallel phases. The serial phase involves: (a) reading the vertex coordinates and mesh elements (line 1), (b) dividing the mesh vertices into nC partitions P_i (line 2) and (c) distributing the partition information to nC cores (line 3).

In each partition, vertices are classified as interior, boundary or fixed. Interior vertices form elements whose all vertices belong to that partition. Boundary vertices form elements where at least one vertex belongs to other partition. Interior and boundary vertices that lie onto the solid boundary are fixed vertices, which are not optimized.

When boundary vertices are updated, the numerical kernel needs the spatial coordinates of all connected vertices. Thus, each partition requires information of adjacent vertices that resides in other partitions. This information is included in all partitions as a special type of vertex called *ghost*. Interior and non-ghost boundary vertices are optimized by the same processor.

For each partition P_i, the first parallel phase involves (lines 4 to 9): (a) coloring the non-fixed boundary vertices (line 5) [2]: $I_i = \{I_{ij}\}$, $I_{ij} \subset P_i$ is an independent set with color j, (b) finding the network of partitions that share boundary elements (line 5), (c) interchanging color information of boundary vertices with other partitions (line 6), (d) creating a list of boundary vertices that determines the order in which these vertices are optimized or received from boundary partitions in later parallel phases (line 7), and (e) computing the initial minimum quality Q_i of partition elements (line 8). Using the message passing interface (MPI) function `MPI_Allreduce()` at the end of parallel phase 1, a synchronization barrier ensures all partitions have completed these steps before continuing computation (line 9).

Parallel phases 2 and 3 use the same `OptimizeNode()` procedure that implement our single-vertex optimization method (equation 1), but the exit conditions for each of them change (lines 10 to 26). Mesh is simultaneously untangled and smoothed in phase 2 ($z = untangling$). It is finished when there is no invalid element ($Q_m > \lambda_{untangling} > 0$, $Q_m = min\{Q_i\}$) or the number of iterations is larger than the input parameter $N_{maxIter,untangling}$.

Mesh smoothing is implemented in phase 3 ($z = smoothing$) only if the mesh is completely untangled. The exit condition for mesh smoothing depends on other two input parameters: maximum number of iterations ($N_{maxIter,smoothing}$) or minimum quality of output mesh ($Q_m > \lambda_{smoothing}$). After a variable number of mesh sweeps, the output of our parallel algorithm provides untangled and smoothed mesh partitions.

Algorithm 1 - Parallel algorithm for simultaneous mesh untangling and smoothing.

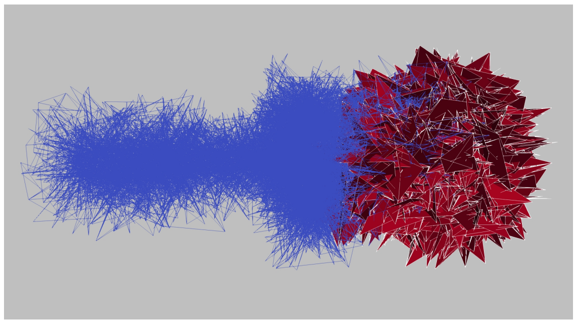
```
1:  Read the vertex and element information of a mesh (M)              ▷ Serial phase: begin
2:  $P_i \leftarrow Partition(M)$                                      ▷ $Nodes(M) = \{P_i\}_{i \in \{1,\dots,nC\}}$
3:  Distribute each partition $P_i$                                    ▷ Serial phase: end
4:  for $P_i$ in parallel do                                           ▷ Parallel phase 1: begin
5:      $I_i \leftarrow Coloring(P_i)$ ▷ It colors boundary vertices, $I_i = \{I_{ij}\}$, and finds the communication graph
6:      Send-Receive colors of vertices that belong to shared boundary elements to/from other partitions
7:      Store the color-based order of boundary vertices
8:      $Q_i \leftarrow Q(P_i)$                                        ▷ $Q_m = min\{Q(P_i), i \in (1,\dots,nC)\}$
9:  Synchronization MPI_Allreduce()                                    ▷ Parallel phase 1: end
10: for $z = \{untangling, smoothing\}$ do
11:     if $z = smoothing$ & $Q_m \leq \lambda_{untangling}$ then
12:         break                          ▷ Mesh could not be untangled and algorithm finishes
13:     for $P_i$ in parallel do    ▷ Parallel phase 2: z=untangling; Parallel phase 3: z=smoothing
14:         $k \leftarrow 0$
15:         while $Q_i \leq \lambda_z$ & $k < N_{maxIter,z}$ do
16:             for each interior free vertex $v \in P_i$ do           ▷ Interior vertex processing: begin
17:                 $\hat{x_v} \leftarrow OptimizeNode(x_v, N_v)$
18:             Synchronization MPI_Allreduce()                        ▷ Interior vertex processing: end
19:             for each boundary independent-set $I_{ij} \in P_i$ do  ▷ Boundary vertex processing: begin
20:                 for each boundary free vertex $v \in I_{ij}$ do
21:                     $\hat{x_v} \leftarrow OptimizeNode(x_v, N_v)$
22:                 MPI_Send-MPI_Receive updated coordinates of vertices $v \in I_{ij}$
23:             Synchronization MPI_Allreduce()                        ▷ Boundary vertex processing: end
24:             $Q_i \leftarrow Q(P_i)$                                ▷ $Q_m = min\{Q(P_i), i \in (1,\dots,nC)\}$
25:             Synchronization MPI_Allreduce()
26:             $k \leftarrow k + 1$
```

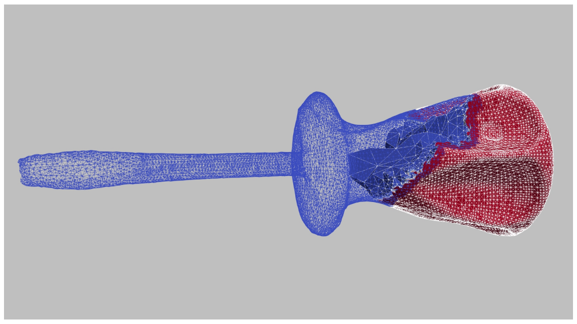
3 EXPERIMENTAL METHODOLOGY AND RESULTS

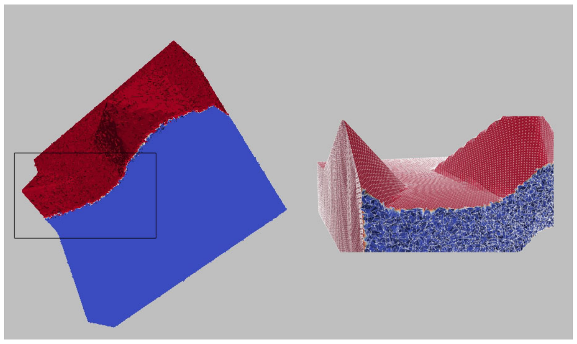
Our experiments were conducted on a cluster with 27 compute nodes that are organized in 7 BullxR424E2 servers. They are interconnected with InfiniBand QDR 4× network. Each node integrates two Intel Xeon E5645 sockets (6 cores each, 2.4 GHz), and 48 GB of DDR3/1333 MHz RAM. So, up to 324 cores, 12 cores per compute node were used in parallel experiments. We activated multiples of 12 cores to completely occupy different numbers of compute nodes. Only one compute node was employed when less than 12 cores were occupied. Multiple runs were conducted on 1, 2, 4, 12, 24, 48, 96, 192, 216, 312, 324 cores.

To compile our sequential and parallel programs on a Linux system, we used gcc 4.8.4 and Open MPI 1.6.5, respectively. The sequential version involves no locks, no synchronization, no partitioning, and no extra overhead that is inherent in MPI parallel programs. We also used `O2` compilation flag, double-precision floating-point arithmetic and Mesquite 2.99 C++ library [3]. Mesquite was extended to support our sequential and parallel algorithms that simultaneously perform mesh untangling and smoothing.

The sequential and parallel codes were applied on two tangled tetrahedral meshes called *Screwdriver* and *Egypt* (see Figure 1). All meshes were obtained by using an automatic strategy for adaptive tetrahedral mesh generation based on the Meccano method [4, 14]. Note in Figure 1 that input meshes are heavily tangled. The size of each input mesh was fixed during all parallel experiments. We used Metis 5.1.0 to partition the meshes [13] before the parallel optimization. So, the size of each partition generally decreases as the number of partitions increases for the same fixed-size benchmark mesh.


(a) Screwdriver: input mesh, elements = 1.69 10^5, inverted elements: 49%.


(a) Screwdriver: output optimized mesh.


(b) Egypt: input mesh, elements = 1.0 10^7, inverted elements: 46%.

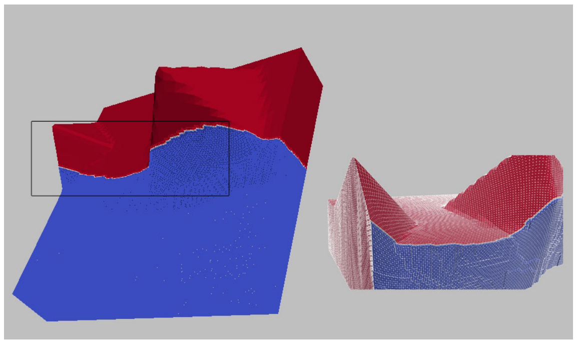

(b) Egypt: output optimized mesh (left) and a detail view (right).

Figure 1: Tangled benchmark meshes for 2-core experiments.

Figure 2: Optimized meshes after using Algorithm 1 and two partitions.

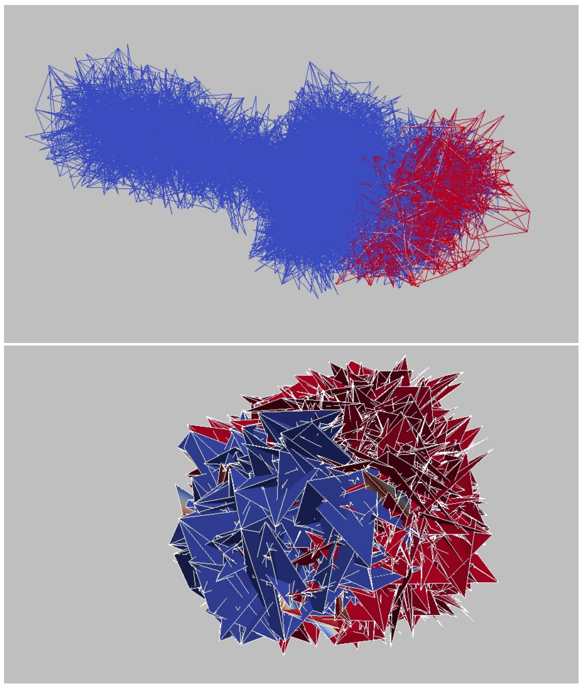


Figure 3: Partitions 0 (up) and 1 (down) of the tangled Screwdriver mesh for experiments with two cores.

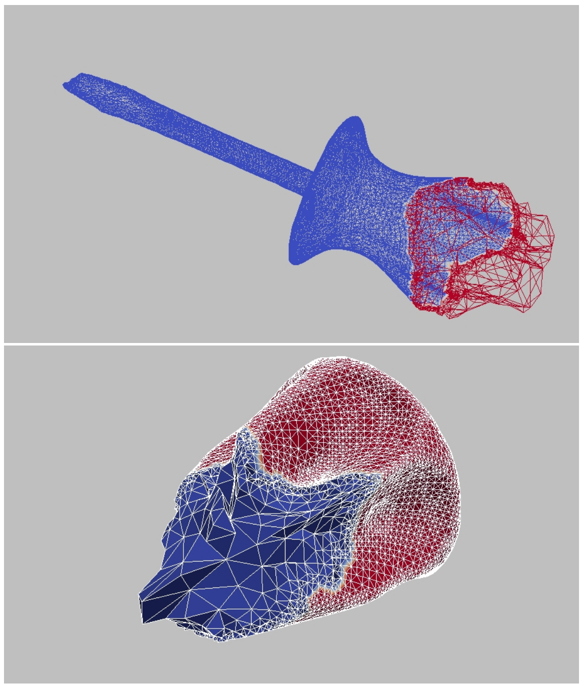


Figure 4: Interior and ghost tetrahedra of partitions 0 (up) and 1 (down) after using Algorithm 1 and two cores.

Figure 1 shows tangled meshes divided into the two partitions that were used in experiments with two cores. Figure 2 shows the output meshes after optimization. Figures 3 and 4 show more details of the two partitions for the Screwdriver mesh before and after optimization, respectively.

Additionally, we chose the following termination criteria in order to measure parallel performance: (a) untangling was stopped when all mesh elements were valid ($\lambda_{untangling} = 0$), (b) ten mesh sweeps were completed after untangling to improve the average mesh quality ($N_{maxIter,smoothing} = 10$). So, the total number of mesh sweeps in each experiment was the number of iterations to completely untangle all partitions plus ten. The average number of mesh sweeps was 20.3 and 13.1 for Screwdriver and Egypt meshes, respectively. The quality of the mesh elements was obtained by using the mean ratio quality metric. It takes value 1 for equilateral elements and 0 for tangled elements. The average and minimum qualities of optimized meshes do not depend on the number of partitions. Final average quality was 0.73 and 0.72, and final minimum quality was 0.16 and 0.20 for Screwdriver and Egypt meshes, respectively.

We measured the CPU times taken to execute the sequential and parallel codes. The execution time included the time to completely met the previously mentioned termination criteria. The sequential CPU time was 8.7 minutes and 6.4 hours for Screwdriver and Egypt meshes, respectively. *Parallel Speedup (S)* and *Parallel Efficiency (E)* were obtained as follows [8]:

$$S = \frac{t_s}{t_p} \qquad\qquad E = 100\% \; \frac{t_s}{nC \;\; t_p} \tag{2}$$

where t_p is the time taken for nC cores to complete the execution of Algorithm 1, and t_s is the time taken to complete the execution of the pure sequential algorithm using the natural vertex ordering of input meshes (see Figure 5).

In a previous study on shared-memory computers [1], we obtained greater parallel efficiency for the same number of cores and mesh. For example, taking Screwdriver mesh and 128 cores, parallel efficiency (E) was 52% and 35% using shared- and distributed-memory architectures, respectively.

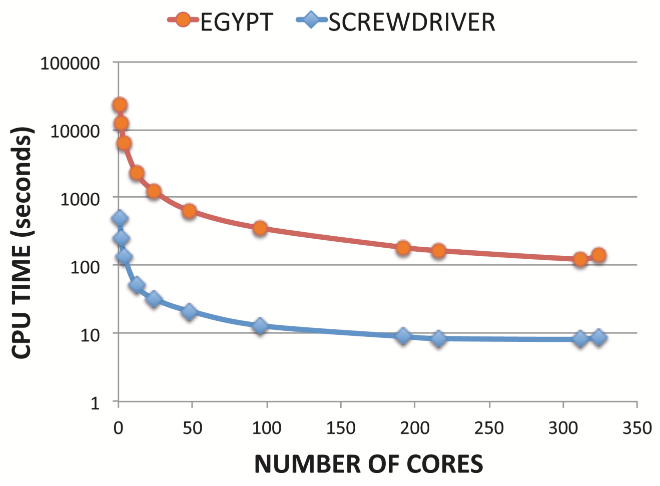
In general, we have observed that lower temporal overheads are produced in OpenMP than MPI for the same mesh and number of cores. However, higher speedup can be achieved for mesh optimization on distributed-memory architectures due to larger number of available cores. In this paper, we report a maximum speedup (S) of 186$\times$ using the Egypt mesh (see Figure 5). This result improves our previous study on parallel optimization when applied to tangled meshes. Comparing our performance results to previous studies on parallel mesh optimization using MPI, it can be observed that for similar meshes of approximately 10^7 elements and occupying 64 cores, we achieved a speedup of 51$\times$ that is similar to the results reported in [16].

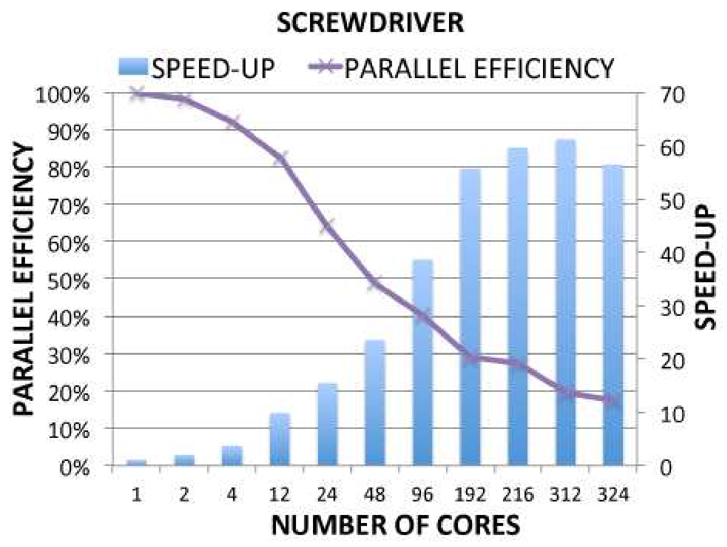
As the number of partitions is increased, the speedup of our algorithm also increases for all benchmark meshes only up to 312 cores. However, for 324 cores, our algorithm exhibits lower speedups than using 312 cores for Screwdriver and Egypt meshes (see Figure 5). Several bottlenecks may limit the parallelism.

Our hypothesis states that one of them is caused by wait times originated from the ordering of vertex updating of partition boundaries. Two partitions can start processing their non-fixed boundary vertices in parallel. In any case, the ordered list of boundary vertices of a partition may indicate that should wait for updated boundary vertices from other partitions before optimization continues (Algorithm 1, line 22). During this waiting time, the vertex processing is interrupted at that partition.

Load imbalance is another performance bottleneck that includes the execution time due to processor overload when the concurrent vertex updating is not well balanced among partitions. In our parallel algorithm, this bottleneck is influenced by both the number of partitions and the ratio of function and gradient evaluations (FGE) between boundary and interior vertex processing. For example, the ratios of FGE for 324 cores are 0.17 for Egypt and 0.46 for Screwdriver. Consequently, parallel efficiency (E) drops at 324 cores from 51% for Egypt to 17% for Screwdriver.

Communications is another performance bottleneck caused by the transmission of updated coordinates of boundary vertices. Our hypothesis states that this bottleneck provides the lowest overhead time. For a given number of partitions, this overhead is caused by the dependence of communication time on the number of boundary vertices, mesh sweeps and interconnection bandwidth, in contrast to mesh optimization time that is





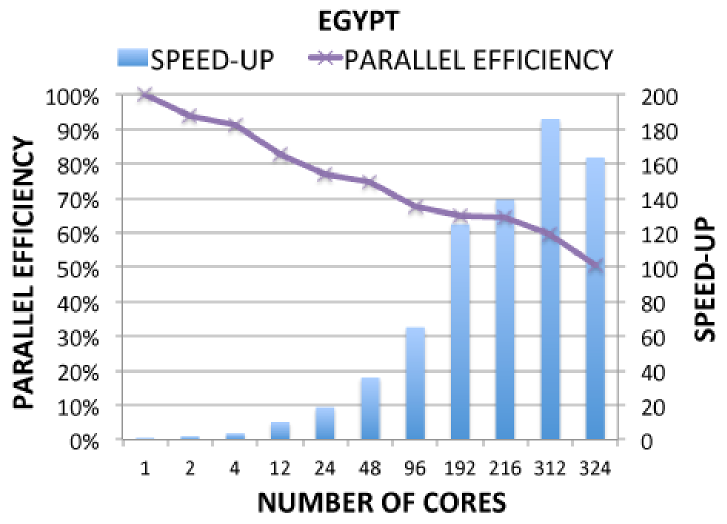


Figure 5: Parallel performance for Screwdriver and Egypt input meshes. Up: execution time. Center, down: speedup and parallel efficiency.

dependent on the number of FGE, number of elements per patch and tetrahedra processing rate. For example, for 324 partitions of Egypt mesh, 14% of vertices were partition boundary vertices, the total communication time was 0.2 s and the total execution time was 140.6 s.

The irregular number of FGE evaluations, when a fixed-size mesh is optimized in parallel, is another cause of performance deterioration. This occurs when the workload with larger number of partitions requires larger FGE. For example, using Egypt mesh, FGE is 2.00 10^9 and 2.12 10^9 for 312 and 324 partitions, respectively. This effect is due to the influence of the vertex processing order on the workload needed for the mesh optimization algorithm to converge. Note that a different number of partitions for parallel processing implies a different vertex processing order.

4 CONCLUSIONS AND FUTURE WORK

We proposed a new single-vertex parallel algorithm that simultaneously performs mesh untangling and smoothing on distributed-memory computers. In this paper, we obtain greater speedup than previous published works on parallel mesh optimization. In order to improve parallel performance, it is important to minimize the number of boundary vertices of partitions. One of our research goals is to study the influence of boundary vertices on parallel performance and bottlenecks when other domain partitioning methods and the Meccano parametric mesh [14] are used.

5 ACKNOWLEDGEMENTS

This work was partially supported by the Spanish ministry ”Ministerio de Economía y Competitividad” under contract CTM2014-55014-C3-1-R.

REFERENCES

[1] D. Benítez, E. Rodríguez, J.M. Escobar, R. Montenegro; Performance Evaluation of a Parallel Algorithm for Simultaneous Untangling and Smoothing of Tetrahedral Meshes; in Proceedings of the 22^{nd} International Meshing Roundtable (IMR), pp. 579-598, Springer, 2014.

[2] D. Bozdag, A. Gebremedhin, F. Manne, E. Boman, U. Catalyurek; A framework for scalable greedy coloring on distributed memory parallel computers; Journal of Parallel and Distributed Computing, 68(4):515-535, 2008.

[3] M. Brewer, L. Diachin, P. Knupp, T. Leurent, D. Melander; The Mesquite mesh quality improvement toolkit; in Proceedings of the 12^{th} International Meshing Roundtable (IMR), pp. 239-250, 2003.

[4] J.M. Cascón, E. Rodríguez, J.M. Escobar, R. Montenegro; Comparison of the meccano method with standard mesh generation techniques, Engineering with Computers, Vol. 31, pp. 161-174, 2015.

[5] Y. Che, L. Zhang, C. Xu, Y. Wang, W. Liu, Z. Wang; Optimization of a parallel CFD code and its performance evaluation on Tianhe1A; Computing and Informatics, 33(6)1377-1399, 2015.

[6] Z. Cheng, E. Shaffer, R. Yeh, G. Zagaris, L. Olson; Efficient parallel optimization of volume meshes on heterogeneous computing systems; Engineering with Computers, pp. 1-10, 2015.

[7] N. Chrisochoides; A survey of parallel mesh generation methods. Tech. Rep. SC-2005-09, Brown University, 2005.

[8] D.E. Culler, A. Gupta, J.P. Singh; Parallel Computer Architecture: A Hardware/Software Approach; Morgan Kaufmann Publishers, 1997, pp. 6, 230.

[9] J.M. Escobar, E. Rodríguez, R. Montenegro, G. Montero, J.M. González-Yuste; Simultaneous untangling and smoothing of tetrahedral meshes; Computer Methods in Applied Mechanics and Engineering, 192, 2775-2787, 2003.

[10] J.M. Escobar, E. Rodríguez, R. Montenegro, G. Montero, J.M. González-Yuste; SUS code: Simultaneous mesh untangling and smoothing code, http://www.dca.iusiani.ulpgc.es/SUScode, 2010.

[11] L. Freitag, M.T. Jones, P.E. Plassmann; A parallel algorithm for mesh smoothing; SIAM Journal on Scientific Computing, 20(6):2023-2040, 1999.

[12] G.J. Gorman, J. Southernb, P.E. Farrella, M.D. Piggotta, G. Rokosa, P.H.J. Kellya; Hybrid OpenMP/MPI Anisotropic Mesh Smoothing; Procedia Computer Science, Vol. 9, pp.1513-1522, 2012.

[13] G. Karypis; Metis: A software package for partitioning unstructured graph, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Users guide, v.5.1.0; University of Minnesota, 2013.

[14] R. Montenegro, J.M. Cascón, J.M. Escobar, E. Rodríguez, G. Montero; An automatic strategy for adaptive tetrahedral mesh generation; Applied Numerical Mathematics, 59(9):2203-2217, 2009.

[15] T. Panitanarak, S.M. Shontz; A parallel log barrierbased mesh warping algorithm for distributed memory machines; Engineering with Computers, 34:5976, 2018.

[16] S. Sastry, S.M. Shontz; A parallel log-barrier method for mesh quality improvement and untangling; Engineering with Computers 30(4):503-515, 2014.

[17] K. Zhao, G. Mei, N. Xu, J. Zhang; On the Accelerating of Two-dimensional Smart Laplacian Smoothing on the GPU; Journal of Information Computational Science, 12(13):5133-5143, 2015.