# Performance Evaluation of a Parallel Algorithm for Simultaneous Untangling and Smoothing of Tetrahedral Meshes

Domingo Benítez[1,2], Eduardo Rodríguez[1,2], José María Escobar[2], and Rafael Montenegro[2]

[1] Departamento de Informática y Sistemas, University of Las Palmas de Gran Canaria, Spain
[2] University Institute for Intelligent Systems and Numerical Applications in Engineering, SIANI, University of Las Palmas de Gran Canaria, Spain
{dbenitez,erodriguez,jmescobar,rmontenegro}@siani.es
http://www.dca.iusiani.ulpgc.es/proyecto2012-2014

**Abstract.** A new parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes is proposed in this paper. We provide a detailed analysis of its performance on shared-memory many-core computer architectures. This performance analysis includes the evaluation of execution time, parallel scalability, load balancing, and parallelism bottlenecks. Additionally, we compare the impact of three previously published graph coloring procedures on the performance of our parallel algorithm. We use six benchmark meshes with a wide range of sizes. Using these experimental data sets, we describe the behavior of the parallel algorithm for different data sizes. We demonstrate that this algorithm is highly scalable when it runs on two different high-performance many-core computers with up to 128 processors. However, some parallel deterioration is observed. Here, we analyze the main causes of this parallel deterioration.

## 1 Introduction

It is well known that the mesh and its quality can greatly impact the accuracy of simulations, as well as solver efficiency [1]. Frequently, the automatic mesh generation tools produce meshes with inverted or poorly shaped elements. To obtain high-quality meshes, often scientists must untangle and improve the quality of meshes before or during the numerical analysis [23]. We proposed a mesh optimization method that simultaneous untangle and smooth tetrahedral meshes [12,13].

When this method is applied using conventional non-parallel programs to large and/or tangled meshes, the wall-clock time may be extremely high. Our optimization method can be applied if higher performance could be achieved such as scientific and engineering applications that use 3D discretization methods to numerically solve partial differential equations [15,16].

J. Sarrate & M. Staten (eds.), *Proceedings of the 22nd International Meshing Roundtable*,
DOI: 10.1007/978-3-319-02335-9_32, 

In this paper, we propose a new parallel algorithm for simultaneously untangling and smoothing tetrahedral meshes. Its goal is to reduce execution time efficiently. In addition, a performance evaluation of the proposed parallel algorithm on many-core computers is described. It includes the analysis of the scalability, parallel efficiency, load balancing, performance bottlenecks, and influence of graph coloring algorithms on the performance of our new parallel algorithm.

Given that meshes, which are used for PDE simulations, are becoming larger all of the time, it is becoming necessary to have parallel algorithms for mesh generation and optimization. To the author's knowledge, there are currently no parallel mesh untangling algorithms or parallel simultaneous mesh untangling and smoothing techniques in the literature. Previous studies on parallel algorithms for mesh smoothing problems include the work of Jiao and Alexander [21] that is applied to triangulated surfaces. Their algorithm was implemented on distributed-memory computers with up to 128 processors and it was shown 43% of maximum parallel efficiency. Yeo et al [34] also proposed an algorithm for smoothing quad meshes that fits well into parallel streams and was mapped to a GPU. They applied their algorithm to real-time processing of surface models and showed a performance comparison with other GPU algorithms. Freitag et al [15] relied on theoretical shared-memory models without a real implementation on shared-memory multi-core systems, although they presented results on distributed-memory computers. Shontz and Nistor have published another similar study [32], which provides performance results for mesh simplification algorithms on GPUs. However, they do not mention if a graph coloring algorithm was used to find mesh vertices that have not computational dependency. Several mesh optimization algorithms are implemented in parallel routines of the petascale meshing software tools provided by the ITAPS project [19]. These tools are used in distributed-memory computers.

The rest of the paper is organized as follows. Section 2 summarizes the mathematical foundation of the 3D simultaneous untangling and smoothing algorithm. Section 3 describes the new parallel algorithm of this optimization method. The experimental methodology that we used to evaluate the performance of the parallel algorithm is explained in Section 4. Section 5 analyzes the performance scalability. Section 6 describes the influence of three previously published graph coloring algorithms on the parallel performance. Load unbalancing and other performance bottlenecks are studied in Section 7 and 8, respectively. Finally, the main conclusions and future work are discussed in Section 9.

## 2 Our Approach for Simultaneous Untangling and Smoothing of Tetrahedral Meshes

Let us consider $M$ to be a tetrahedral mesh. Usual techniques to improve the quality of a valid mesh, that is, one that does not contain inverted tetrahedra, are based upon local smoothing [10]. In short, these techniques consist of finding the new

position $\boldsymbol{x}_v$ that each inner mesh node $v$ must hold, in such a way that they optimize an objective function (boundary vertices are fixed during all the mesh optimization process). Such a function is based on a certain measurement of the quality of the local submesh $\mathcal{N}_v \subset \mathcal{M}$ that is formed by the set of tetrahedra connected to the *free node* $v$. As it is a local optimization process, we cannot guarantee that the final mesh is globally optimal. Nevertheless, after repeating this process several times for all the nodes of the mesh $\mathcal{M}$, quite satisfactory results can be achieved.

The algebraic quality metrics proposed by Knupp [24] provide us an appropriate framework to define objective functions. Specifically, the one used in this paper has the generic form,

$$K(\boldsymbol{x}_v) = \left(\sum_{i=1}^{n} [\eta_i(\boldsymbol{x}_v)]^p\right)^{\frac{1}{p}} \tag{1}$$

where $n$ is the number of elements in $\mathcal{N}_v$, $p$ is usually chosen as 1 or 2 and $\eta_i = 1 / q_i$ is the distortion of the *i-th* tetrahedron of $\mathcal{N}_v$ and $q_i$ is the chosen corresponding algebraic element quality measure. In particular, we have implemented the *mean ratio* quality measure of a tetrahedron given by $q = 3\ \sigma^{2/3} / |S|^2$, where $|S|$ is the Frobenius norm of matrix $S$ associated to the affine map from the ideal element (usually an equilateral tetrahedron) to the physical one, and $\sigma = \det(S)$. Specifically, the weighted Jacobian matrix $S$ is defined as $S = AW^{-1}$, being $A = (\boldsymbol{x_1}\text{-}\boldsymbol{x_0}, \boldsymbol{x_2}\text{-}\boldsymbol{x_0}, \boldsymbol{x_3}\text{-}\boldsymbol{x_0})$ the Jacobian matrix and $\boldsymbol{x_k}$, $k = 0,1,2,3$ the coordinates of the vertices of the tetrahedron. The constant matrix $W$ is derived from the ideal element (see [12]).

Objective functions like (1) are appropriate to improve the quality of a valid mesh and avoid a valid mesh to be inverted, but they do not work properly when there are inverted elements ($\sigma < 0$). This is because they present singularities (barriers) when any tetrahedron of $\mathcal{N}_v$ changes the sign of its Jacobian matrix. In [12] we proposed a suitable modification of the objective function such that it is regular all over $\mathbb{R}^3$. It consists of substituting the term $\sigma$ in the quality metrics by the positive and increasing function $\mathrm{h}(\sigma) = \frac{1}{2}\left(\sigma + \sqrt{\sigma^2 + 4\delta^2}\right)$. When a feasible region exists (subset of $\mathbb{R}^3$ where $v$ could be placed, being $\mathcal{N}_v$ a valid submesh), the minima of the original and modified objective functions are very close and, when this region does not exist, the minimum of the modified objective function is located in such a way that it tends to untangle $\mathcal{N}_v$. In this way, we can use any standard and efficient unconstrained optimization method to find the minimum of the modified objective function [2]. In this paper, we have used the Newton unconstrained optimization method that is implemented with UNCMIN++ library [9,33].

## 3 A Novel Parallel Algorithm

In Algorithm 1, a sequential algorithm called `SUS` for our simultaneous untangling and smoothing of tetrahedral meshes can be seen. The inputs of the sequential

algorithm are the followings: $\mathcal{M}$ is a tangled tetrahedral mesh, `maxIter` is the maximum number of untangling and smoothing iterations, $N_v$ is the set of tetrahedra connected to the free *node* $v$, $\boldsymbol{x}_v$ is the initial position of the free node, $\hat{\boldsymbol{x}}_v$ is its position after optimization, which is implemented with the procedure `OptimizeNode`, `Q` measures the lowest quality of a tetrahedron of $\mathcal{M}$ when the above mentioned $q$ tetrahedron quality function is used, and `quality` is a function that provides the minimum quality of mesh $\mathcal{M}$ (it is 0 if any tetrahedral is tangled).

The output of the algorithm is an untangled and smoothed mesh $\mathcal{M}$, whose minimum quality must be larger than a user-specified threshold $\lambda$. This algorithm iterates over all the mesh vertices in some order and adjusts at each step the coordinates $\hat{\boldsymbol{x}}_v$ of the free node $v$. A code for this algorithm can be downloaded at [13].

In Algorithm 2, a novel parallel algorithm for our mathematical method for simultaneous untangling and smoothing of tetrahedral meshes is shown. The main procedure is called `pSUS`. Its inputs $\mathcal{M}$, `maxIter`, $\mathcal{N}_v$, $\boldsymbol{x}_v$, `OptimizeNode`, $\hat{\boldsymbol{x}}_v$, `Q`, $\lambda$, `quality` have the same meanings as described for Algorithm 1.

**Algorithm 1.** Sequential algorithm (`SUS`) for the simultaneous untangling and smoothing of a tetrahedral mesh $\mathcal{M}$.

```
1:  function OptimizeNode(x_v, N_v)
2:     Optimize objective function K(x_v)
3:  end function
4:  procedure SUS
5:     Q ← 0
6:     k ← 0
7:     while Q < λ and k < maxIter do
8:        for each vertex v ∈ M do
9:              x̂_v ← OptimizeNode(x_v, N_v)
10:       end do
11:       Q ← quality(M)
12:       k ← k+1
13:    end do
14: end procedure
```

The parallel algorithm has to prevent two adjacent vertices from being simultaneously untangled and smoothed on different processors. On the contrary, new inverted mesh elements may be created [15]. Thus, when the sequential Algorithm 1 is parallelized, a computational dependency appears between adjacent vertices because one vertex needs to be optimized after the other. This justifies the use in our parallel algorithm of a graph coloring algorithm to find vertices of a tetrahedral mesh $\mathcal{M}$ that have not computational dependency.

Graph coloring is implemented with procedure `Coloring`, which is expressed as follows. Let *G=(V,E)* be the graph associated to the tetrahedral mesh $\mathcal{M}$, where *V* is the set of vertices of the mesh (without information of vertex spatial coordinates) and *E* is the set of their edges, then `Coloring` is a procedure that is used

to color the graph G such that two adjacent vertices do not have the same color. Then, an *independent set*, or *color*, $I_i$ is a set of non-adjacent vertices (i.e., they do not share a common edge). That is, $v \in I_i \Rightarrow v \notin adj(I_i,G=(V,E))$, where $adj(I_i,G=(V,E))$ is the set of vertices that are adjacent to all vertex $j \in I_i$ being $j \neq v$. In this way the graph $G$ of a tetrahedral mesh $\mathcal{M}$ is colored and partitioned in a disjoint sequence of independent sets, $I=\{I_1,I_2,\ldots\}$.

We implemented three different and previously published graph coloring methods called "$C_1$", "$C_2$", and "$C_3$". $C_1$ is a vertex coloring method that has been used for parallel mesh smoothing by Freitag et al [15]. It requires the use of the asynchronous coloring heuristic proposed by Jones and Plassmann [22]. In particular, we used its serial version for $C_1$. This heuristic is based on Luby's Monte Carlo algorithm for determining the maximal independent set [26]. $C_2$ is a parallel version of $C_1$ that was also proposed in [22] for distributed-memory computers. We additionally adapted $C_2$ to multithread/multicore computers. $C_3$ is an iterative parallel greedy coloring algorithm that was proposed by Bozdag et al [3]. Section 6 compares the impact of these graph coloring algorithms on the performance of our parallel optimization method.

**Algorithm 2.** Parallel algorithm (pSUS) for the simultaneous untangling and smoothing of a tetrahedral mesh $\mathcal{M}$.

```
1:  procedure Coloring(G=(V,E))
2:    G=(V,E) is partitioned in a disjoint sequence of
      independent sets I={I1,I2,…} using C1, C2 or C3
      coloring algorithm
3:  end procedure
4:  function OptimizeNode(xv,Nv)
5:     Optimize objective function K(xv)
6:  end function
7:  procedure pSUS
8:     I ← Coloring(G=(V,E))
9:     k ← 0
10:    Q ← 0
11:    while Q < λ and k < maxIter do
12:      for each independent set Ii ∈ I do
13:        for each vertex v ∈ Ii in parallel do
14:           x̂v ← OptimizeNode(xv,Nv)
15:        end do
16:      end do
17:      Q ← quality(M)
18:      k ← k+1
19:    end do
20: end procedure
```

Our simultaneous untangling and smoothing algorithm optimizes in parallel the vertices of an independent set. The vertex set with the same color ($I_i$) is partitioned

among the available processors. Each processor optimizes its assigned set of vertices in a sequential fashion. At each sequential step, a processor applies the `OptimizeNode` function to a single vertex $v$. This optimization function implements the method described above in Section 2 to adjust the new position $\hat{\boldsymbol{x}}_v$ of each free vertex $v$ in its own submesh $\mathbb{N}_v$. The new vertex spatial position is available to other processors by writing to shared memory.

Each subsequent parallel phase optimizes another independent set of vertices. There are as many parallel phases as number of independent sets. As in the serial algorithm, after all vertices have been optimized, the mesh quality `Q` is measured. The mesh is successively untangled and smoothed until the mesh is completely untangled and successive iterations increase the minimum mesh quality less than 5%. The algorithm also stops when the number of untangling and smoothing iterations is larger than `maxIter`. Finally, if the mesh optimization procedure stops before `maxIter` is reached, the output of our parallel algorithm is a tetrahedral mesh $\mathcal{M}$ with a minimum quality greater than $\lambda$.

## 4 Experimental Methodology

Our experiments were conducted on two different many-core high-performance computers. One of them is a HP Integrity Superdome node that contains 128 Itanium2 Montvale cores with 1.6 GHz clock speed, multithreading disabled, and 1024 GB NUMA (Non-Uniform Memory Access) shared memory. It is a node of the Finis Terrae supercomputer [14]. The other parallel computer is called Manycore Testing Lab, which has been set up by Intel to work with 40 Westmere 2.27 GHz cores and 252 GB NUMA shared memory [18]. Both computers use Linux systems with kernels 2.6.16.53-0.8-smp and 2.6.18-194.11.4.el5-smp respectively.

The sequential and parallel versions of our 3D untangling and smoothing method were applied on six different tangled benchmark meshes. Their descriptions can be seen in Table 1. The minimum mesh quality of all meshes is 0 because at least one tetrahedron is inverted. The average mesh quality is obtained by summing the quality of all valid tetrahedra and dividing the sum by the number of valid and inverted tetrahedra. All these tetrahedral meshes were constructed with a tool that applies an automatic strategy for adaptive tetrahedral mesh generation based on the meccano method [6,7,28,29]. Some of them were generated using, as input data, the surface triangulations obtained from different Internet repositories [30]. Note that as the meccano method uses Kossaczky's algorithm [25], the maximum vertex degree (node valence) coincides in all the benchmark meshes.

To compile our programs, we used the same Intel C++ compiler version 11.1 on both parallel computers, but targeted to the respective processor architecture. This compiler generates more efficient programs for our algorithms than GNU GCC compiler, which is usually included in Linux distributions.

**Table 1.** Description of the tangled benchmark tetrahedral meshes. The quality of inverted elements is considered zero. So, the minimum quality is zero for all meshes.

| Name | Number of vertices (m) | Number of tetrahedra | Average mesh quality | Number of inverted tetrahedra | Maximum vertex degree | Object |
|---|---|---|---|---|---|---|
| "m=6358" | 6358 | 26446 | 0.2618 | 2215 | 26 | Bunny |
| "m=9176" | 9176 | 35920 | 0.1707 | 13706 | 26 | Tube |
| "m=11525" | 11525 | 47824 | 0.2660 | 1924 | 26 | Bone |
| "m=39617" | 39617 | 168834 | 0.1302 | 83417 | 26 | Screwdriver |
| "m=201530" | 201530 | 840800 | 0.2409 | 322255 | 26 | Toroid |
| "m=520128" | 520128 | 2201104 | 0.0657 | 1147390 | 26 | HR toroid |

The source code of the parallel version of our new algorithm includes OpenMP directives, which were disabled when the sequential version was compiled. After some test runs, we decided to use the dynamic OpenMP thread scheduling method [8] because it provides the best performance results for our parallel algorithm. In all cases, the compiler optimization flag "-O3" was enforced. All versions were run with additional software optimization based on hardware binding: processor and memory. For each benchmark mesh we run the parallel version multiple times using a given maximum number of active threads between 1 and 128 when the Itanium2-based computer is used and between 1 and 40 when the Westmere-based computer is used. Since our algorithms are CPU-bound, there is little sense in using more threads than available cores. Thus, we activate the same number of cores as the number of threads. No operating system code was executed during our experiments and the processors were not shared among other user level workloads.

The sequential and parallel versions of our source code were profiled with the Performance Application Programming Interface API [5], which uses performance counter hardware of processors. The information provided from these performance counters was used to calculate the following quantitative performance metrics: wall-clock time, parallel overhead time, true speed-up, parallel efficiency, and load balancing. Each metric is averaged over more than 30 independent runs.

Each run is divided into two phases. The first of them completely untangles a mesh. This phase loops over all the mesh vertices repetitively. During this phase, untangled tetrahedra are smoothed too. The number of untangling iterations depends on the mesh. The second phase is focused on smoothing the mesh until successive smoothing iterations increase the minimum mesh quality less than 5%.

Each run is characterized by the number of untangling and smoothing iterations, which is represented by "U&S". The results of the experimental setup, described in this section, are discussed in the following four sections.

## 5 Performance Scalability

The qualitative metric called performance scalability informs about the improvement of an algorithm when the amount of processors increases. It is usually based on the quantitative metric $S_{Nc}$ called *speed-up*, which is defined as the ratio of the sequential execution time $t_S$ to the parallel execution time $t_{Nc}$ when $N_c$ processors are used: $S_{Nc} = t_S / t_{Nc}$.

When the execution time of the main mesh optimization procedure alone is profiled in both versions of our algorithms, line 9 of serial Algorithm 1 vs. line 14 of parallel Algorithm 2, we obtain values for speed-up as illustrated in Fig. 1(b) (black bars labeled INSIDE). Each bar represents to speed-up for a given number of available threads/cores. The results shown in Fig. 1(c) were obtained when the "m=39617" mesh was optimized with our sequential and parallel algorithms using $C_3$ coloring algorithm and the parallel computer with 128 Itanium2 processors. Due to paper limitations, we can present detailed performance results for only one benchmark mesh and one graph coloring algorithm.

As it can be seen in Fig. 1(b), the speed-up of the inside part of the mesh optimization procedure linearly increases as the number of cores increases. In order to measure how well this linear increase of speed-up encompasses the increase of cores, the quantitative performance metric called *parallel efficiency* ($E_{Nc}$) is used: $E_{Nc} = 100\% \times S_{Nc} / N_C$, where $N_C$ is the number of cores. If speed-up ($S_{Nc}$) is not superlinear, maximum value of $E_{Nc}$ is 100% [8]. The utilization of the processors and speed-up scalability will be better as long as parallel efficiency is higher.

Figures 1(b) and 1(e) also show the parallel efficiency of the main mesh optimization procedure of Algorithm 2 (dotted line labeled INSIDE). Note that up to $N_C = 128$ cores, the parallel efficiency is always above 76% when up to 128 processors are used. These results indicate that the main computation of our parallel algorithm is highly scalable. This scalability is caused by the parallel processing of an independent set of vertices in the main loop of the Algorithm 2 (lines 13-15).

When the execution times of the complete sequential (procedure `SUS` of Algorithm 1) and parallel (procedure `pSUS` of Algorithm 2) algorithms are profiled, we obtained values for speed-up as illustrated in Fig. 1(b) and 1(e) (grey bars labeled OUTSIDE). Note that in this case, the speed-up of the complete algorithm does not increase linearly as when the main mesh optimization procedures of algorithms are profiled, and the parallel efficiency is above 50% when up to 128 processors are used.

We observe that as the number of threads/cores is larger, the OpenMP loop-scheduling overhead increases the number of executed instructions to synchronize and manage parallel threads. These additional instructions are not involved in the main computation of our parallel algorithm. As Amdahl's law describes [4], speed-up and parallel efficiency deteriorate as the number of threads increases because they tend to be dominated by this parallel overhead.

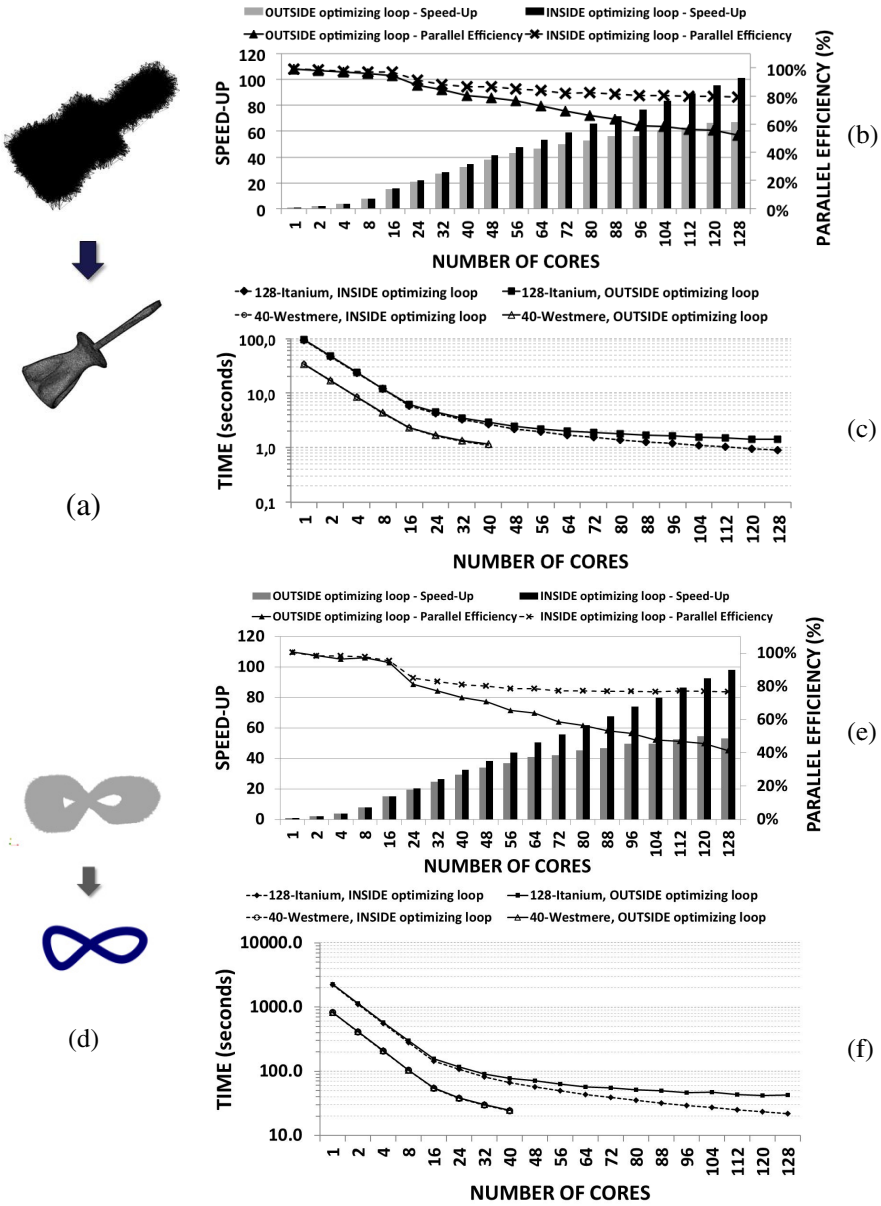


**Fig. 1** "m=39617": (a) Tangled (up) and untangled (down) mesh. (b) Speed-up and parallel efficiency of the line 14 of Algorithm 2 (label: INSIDE) and the complete Algorithm 2 (label: OUTSIDE) on the Itanium2 computer. (c) Execution time of the line 14 (label: INSIDE) and the complete parallel algorithm (label: OUTSIDE) when 40 Westmere and 128 Itanium2 cores are respectively used. "m=520128": (d), (e) and (f) show the respective results for this mesh. $C_3$ coloring algorithm and dynamic thread scheduling were used.

This scalability deterioration of our parallel algorithm is mainly due to the parallel loop-scheduling overhead that is incurred when the threads are scheduled and launched during runtime. After some experiments, we observed that the best OpenMP directive for line 13 of Algorithm 2 uses dynamic thread scheduling with one mesh vertex per chunk: `#pragma omp parallel for schedule (dynamic, 1)`. The small chunk size that achieves the best load balancing is justified by the fact that the time to optimize each vertex varies significantly.

Using data collected from some of the hardware counters of processors during runtime, we observe that as the number of threads/cores is larger, the above mentioned OpenMP directive makes the number of executed instructions to schedule and launch parallel threads to be larger. These additional instructions are not involved in the main computation of our parallel algorithm. As Amdahl's law describes [4], speed-up and parallel efficiency deteriorate as the number of threads increases because they tend to be dominated by this parallel overhead. Thus, the main performance overhead is due to the implementation tool that is used in developing our parallel programs.

Computer performance comparisons are frequently done using wall-clock time metric. Thus, Fig. 1(c) and 1(f) show the runtime of our parallel algorithm when the "m=39617" and "m=520128" meshes are respectively used. For the both cases, $C_3$ coloring algorithm and two many-core computers are used. First of all, note that when the number of cores is smaller than 32, the differences in runtime between the inner loop of the parallel algorithm (data labeled INSIDE) and the complete parallel algorithm (data labeled OUTSIDE) are inappreciable. As the number of cores is larger, the runtime performance of the complete parallel algorithm decreases, but not as much as the runtime of the inner loop. However, the larger the number of vertices, the lower the parallel overhead tends to be. These conclusions are valid for the two computers used in the experiments.

Additionally, we observe in most of cases that for the complete parallel Algorithm 2, the Westmere-based computer provides higher performance than Itanium2-based computer although more Itanium2 cores are used than Westmere cores: 128 vs. 40, respectively. This is mainly due to three causes: the low parallel efficiency of Algorithm 2 when Itanium2 processors are used, the lower clock speed of Itanium2 processors, and the inefficiency of compiler in exploiting the instruction level parallelism of the very-long-instruction-word of Itanium2 instructions.

However, when the inner loop of the parallel algorithm is studied, the Itanium2-based computer provides lower execution time than the other parallel computer. This is due to the high parallel efficiency of the main computation of our algorithm that efficiently reduces the runtime as the number of cores increases. In Fig. 1(b) and 1(e), it is shown that the parallel efficiency of the main computation of our parallel algorithm for the "m=39617" mesh is 76% when 128 Itanium2 cores are used. Thus, the lower clock speed of Itanium2 processors is balanced by a larger number of cores.

## 6 Influence of Graph Coloring Algorithms on Parallel Performance

As described in Section 3, three different graph coloring algorithms were used by our parallel algorithm as software methods to identify independent sets of vertices. The main coloring routine is represented in Algorithm 2 by lines 1-3, 8. Many papers evaluate the performance of graph coloring algorithms on parallel computers [3,15,22], but their impacts on the performance of mesh optimization algorithms are rarely reported.

First of all, we confirmed the performance results published in previous papers for $C_1$, $C_2$, and $C_3$ coloring algorithms. Then, we investigated their influence on the performance of our parallel untangling and smoothing algorithm. Since the goal of graph coloring is discovering parallelism, the execution time involved in graph coloring is only considered when our parallel algorithm is run and not when the serial version is run.

The main conclusion of this part of our investigation is that the execution time of graph coloring algorithms has relatively low impact on the whole execution time of our parallel algorithm. When up to 128 Itanium2 processors and the six benchmark meshes are considered, the percentage of total runtime that is required by $C_1$, $C_2$, and $C_3$ coloring routines ranges respectively from 0.8% to 3.4%, from 2.4% to 12.9%, and from 0.1% to 1.9% (see Fig. 2). This means that the computational load required by our parallel algorithm is much heavier than required by graph coloring algorithms. Note that the lowest impact on total execution time is achieved by $C_3$ coloring algorithm.

However, the speed-up of our parallel algorithm depends on the selected coloring algorithm. Figure 3 shows the speed-ups achieved by our parallel algorithm for the mesh "m=39617" when all three coloring algorithms are used for different numbers of available threads/cores. In addition to these results, Figure 4 shows the speed-up for all six benchmark meshes when all 128 Itanium2 processors are working in parallel.

It can be also observed in Fig. 4 that low speed-up is obtained with smaller meshes. This means that parallelism is more difficult to exploit in smaller meshes than larger meshes. For a given number of threads, as the number of mesh vertices decreases, each thread tends to optimize fewer vertices. Thus, the loop-scheduling overhead is less balanced with optimization time in smaller meshes than larger ones. Note in Fig. 4 that our parallel algorithm achieves the best speed-up when $C_3$ coloring algorithm is used. This is due to that $C_3$ coloring algorithm clusters the vertices of a mesh using a less number of colors than $C_1$ and $C_2$ coloring algorithms. A lower number of vertex clusters allows a larger number of vertices to be processed in parallel.

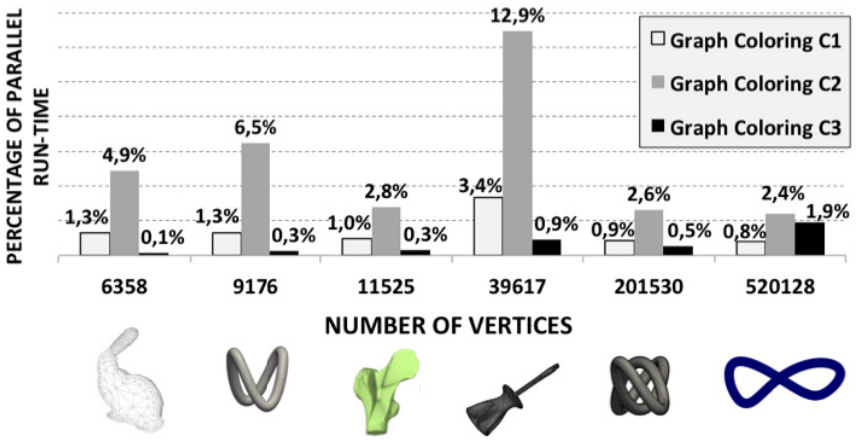


**Fig. 2** Percentage of total parallel runtime that is required by the three graph coloring algorithms $C_1$, $C_2$, and $C_3$ when the six benchmark meshes are untangled and smoothed and 128 Itanium2 processors are used. In all cases, the parallel algorithm uses OpenMP dynamic thread scheduling.

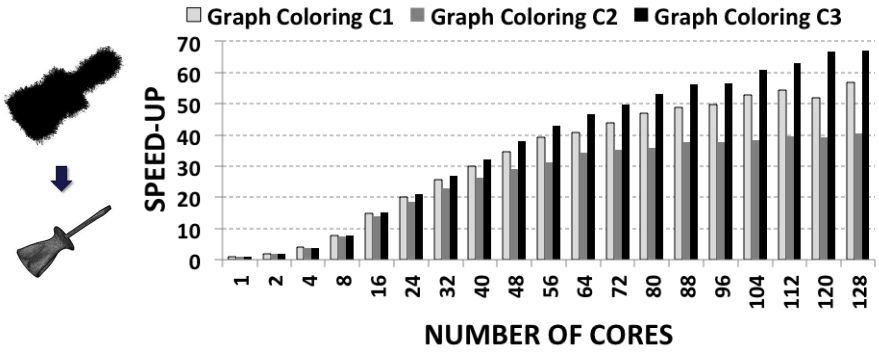


**Fig. 3** Speed-up achieved by the complete parallel algorithm (`pSUS`) when the three mesh coloring algorithms ($C_1$, $C_2$, and $C_3$) and the benchmark mesh "m=39617" are considered

However, we also remark that the speed-up is obtained as a ratio of serial and parallel runtimes. So, speed-up is a relative performance metric and the best speed-up does not mean the best performance because the reference performance of the serial algorithm may not be the best [27]. This occurs in our application. Being $C_3$ the graph coloring with the best speed-up, $C_1$ or $C_2$ provide in some cases the best performance because they need the smallest number of mesh sweeps (see Tables 2 and 3).

Hence, to compare the three different graph coloring algorithms, we also use absolute running times. Table 2 shows the best running time for each benchmark

mesh when the inner loop of the parallel algorithm (line 14 of Algorithm 2) is monitored. Similarly, the results for the complete parallel algorithm (procedure `pSUS` of Algorithm 2) are presented in Table 3. In these cases, the best performance coloring algorithm is not always the same.

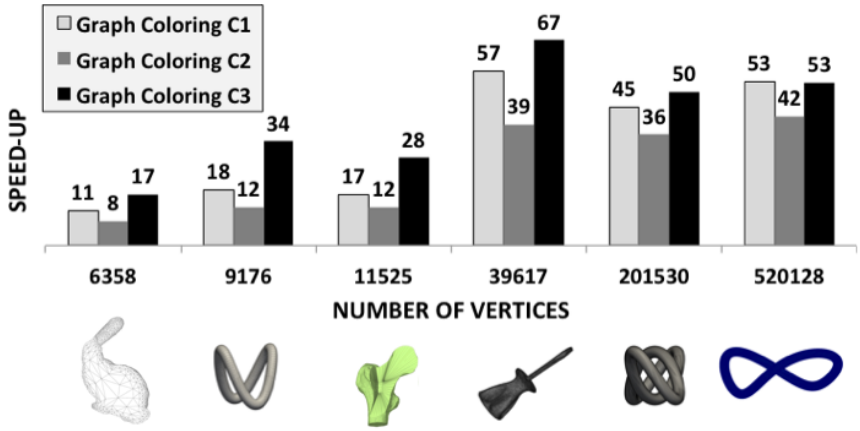


**Fig. 4** Speed-up achieved by the complete parallel algorithm (`pSUS`) for all six benchmark meshes when three graph coloring algorithms ($C_1$, $C_2$, and $C_3$) are used and all 128 shared-memory Itanium2 processors are active

**Table 2** Best runtime for the main optimization procedure of the parallel algorithm (line 14, Algorithm 2)

| Name of tetrahedral mesh | Best runtime (sec.) | Best parallel computer | Best number of cores | Best coloring algorithm | Number of colors | Number of iterations (U&S) | Minimum mesh quality | Average mesh quality |
|---|---|---|---|---|---|---|---|---|
| m=6358 | 0.16 | Itanium2 | 128 | $C_1$ | 29 | 25 | 0.1319 | 0.6564 |
| m=9176 | 0.33 | Itanium2 | 128 | $C_1$ | 29 | 26 | 0.2560 | 0.6820 |
| m=11525 | 0.34 | Itanium2 | 128 | $C_2$ | 18 | 39 | 0.1104 | 0.6474 |
| m=39617 | 0.79 | Itanium2 | 128 | $C_1$ | 31 | 11 | 0.1698 | 0.7329 |
| m=201530 | 24.69 | Westmere | 40 | $C_2$ | 21 | 120 | 0.2275 | 0.6687 |
| m=520128 | 20.60 | Itanium2 | 128 | $C_1$ | 34 | 36 | 0.2232 | 0.6750 |

On the one hand, most of the time $C_1$ coloring algorithm provides the lowest running time when the inner loop is monitored (see Table 2). This is due to the number of iterations to completely untangle and smooth the mesh. When two coloring algorithms are compared on the same mesh, we observe that both the vertex partition and vertex ordering do not coincide. Thus, the number of simultaneous untangling and smoothing iterations depends on the selected coloring algorithm. In our experiments, there have been small differences in the running times of the

untangling and smoothing iterations of a mesh, and the best graph coloring algorithm has been that for which the number of iterations is minimum.

On the other hand, most of the times $C_2$ coloring algorithm provides the lowest running time when the complete algorithm is monitored (see Table 3). This is due to the same reasons as previously mentioned for the best coloring algorithm of the inner loop of the parallel algorithm.

**Table 3** Best runtime for the complete parallel algorithm (procedure `pSUS`, Algorithm 2)

| Name of tetrahedral mesh | Best runtime (sec.) | Best parallel computer | Best number of cores | Best coloring algorithm | Number of colors | Number of iterations (U&S) | Minimum mesh quality | Average mesh quality |
|---|---|---|---|---|---|---|---|---|
| m=6358 | 0.39 | Westmere | 40 | $C_3$ | 11 | 24 | 0.1289 | 0.6564 |
| m=9176 | 0.66 | Westmere | 40 | $C_2$ | 18 | 25 | 0.2629 | 0.6821 |
| m=11525 | 0.78 | Westmere | 40 | $C_3$ | 10 | 48 | 0.1106 | 0.6474 |
| m=39617 | 1.00 | Westmere | 40 | $C_2$ | 19 | 11 | 0.1586 | 0.7329 |
| m=201530 | 25.92 | Westmere | 40 | $C_2$ | 21 | 120 | 0.2275 | 0.6687 |
| m=520128 | 24.31 | Westmere | 40 | $C_2$ | 21 | 35 | 0.2234 | 0.6749 |

However, note that for a given mesh, the best graph coloring algorithm for the complete untangling and smoothing procedure does not coincide with the best one for the inner loop. Thus, the best coloring algorithm for a determined benchmark mesh depends on the part of algorithm that is analyzed.

Furthermore, note that the parallel computer that provides the highest performance (i.e. lowest runtime) for the inner loop is different from the parallel computer that provides the highest performance for the complete algorithm (see Tables 2 and 3). This is mainly due to the hardware architecture and the performance overhead of the thread scheduling technique used by compiler. Itanium2-based computer has larger number of cores than the Westmere-based computer: 128 vs. 40. Thus, the larger number of available processors benefits the scalable performance of the main loop of our parallel algorithm, although the clock speed of Itanium2 processors is lower than Westmere processors. However, Westmere-based computer provides lower running times than Itanium2-based computer for the complete parallel algorithm because the compiler for Itanium2 processor provides less efficient parallel code for our algorithm than the compiler for Westmere processor, incurring in more loop-scheduling overhead.

As different coloring strategies would lead to different ordering of vertex processing, it is clear that the quality of resulting meshes could depend on the coloring algorithm. We have found that in exceptional cases a benchmark mesh could not be untangled by using a specific coloring algorithm. Therefore, the convergence of our algorithm may depend on the ordering of vertices. This aspect has been analyzed in 2D problems in [31]. At present, there is no criterion to determine the optimal vertex ordering for a given mesh.

## 7 Load Imbalance

In parallel computing, load unbalancing between computing nodes also causes performance deterioration. As it is described in the previous section, the parallel performance of our parallel algorithm is affected by the mesh size in addition to the number of available parallel threads. So, we have quantitatively analyzed the load unbalancing caused by our parallel algorithm when the mesh size and the number of parallel threads vary. We focus on the experimental results obtained with the Itanium2-based computer because it has the largest number of processors.

In order to measure load unbalancing, we monitor the execution time of each parallel thread with the native hardware counter of Itanium2 called CPU_OP_CYCLES_ALL [17]. These threads are monitored when they execute the main optimizing procedure of Algorithm 2 (lines 12-16). Then, load imbalance ($L_{Nc}$) of $N_c$ parallel working threads is obtained as follows:

$$L_{N_c} = 100\% \times \frac{t_{\max} - t_{\min}}{t_{avg}} \tag{2}$$

where $t_{max}$, $t_{min}$, $t_{avg}$ are respectively the maximum, minimum and average execution times of the parallel working threads, supposing that $t_{max} \geq t_{avg} \geq t_{min} > 0$. As $L_{Nc}$ gets smaller, the difference between maximum and minimum thread execution time is comparatively smaller than the average execution time of threads. In these cases, threads tend to be stalled less time because load is more balanced among processors, and so, parallel performance is better.

Before doing the experiments described in this paper, we analyzed the impact of the OpenMP thread scheduling alternatives (static, dynamic, guided) on the performance of our parallel algorithm for each one of the above mentioned coloring algorithms and benchmark meshes. We observed that the dynamic thread scheduling with chunk size of one vertex per thread shows better performance for all cases. This means that the thread scheduler distributes to each thread the job of optimizing only one vertex. In this way, parallel performance is better because the load unbalancing caused by the OpenMP programming methodology is minimized.

Figure 5 shows the load unbalancing ($L_{Nc}$) of our parallel algorithm for the six benchmark meshes when the $C_3$ coloring algorithm and up to 128 shared-memory Itanium2 processors are used. Note that for each one of the benchmark meshes, the number of active threads causes the main impact on load unbalancing. The higher the number of active threads, the higher the load unbalancing. This means that as the loop-scheduling overhead instructions increase, the main computation of threads is more unbalanced. We tested our parallel algorithm on other x86 parallel computers, and the same effect on load unbalancing was observed.

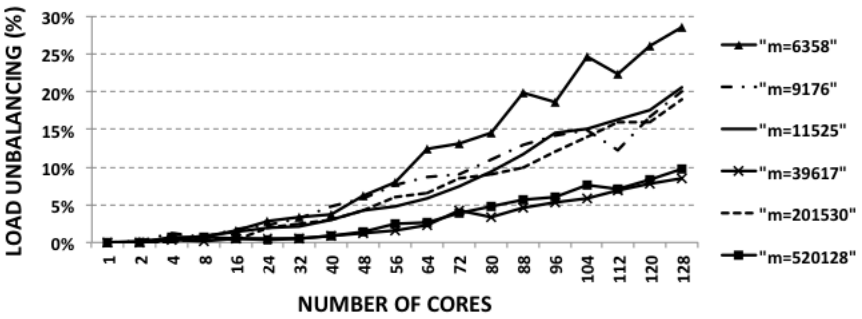


**Fig. 5** Load imbalance when meshes with different number of vertices (m) and up to 128 Itanium2 cores are used. $C_3$ coloring algorithm and dynamic thread scheduling are used.

## 8 Parallelism Bottlenecks

Previously, we have concluded that the OpenMP thread scheduling causes the main parallel inefficiency. This overhead increases the number of executed instructions that are used to synchronize and manage parallel threads. Additionally, in previous section, we have analyzed another parallel inefficiency caused by the load unbalancing of threads. We have also analyzed the computational inefficiencies in each thread that are caused by the main sequential computation of our parallel algorithm corresponding to line 14 of Algorithm 2. During runtime, computation bottlenecks are located in processor functional units, cache memories, and NUMA (Non-Uniform Memory Access) memory. We have measured the number of stalled cycles in these hardware units using the methodologies described in [11,20] and some performance counters of Itanium2 processor [17]. In this section, we also focus on experimental results obtained with Itanium2-based computer because it has the largest number of processors.

On average, when up to 128 shared-memory Itanium2 processors are used, the stall cycles of each parallel thread are in the range from 29% to 58%. These stall cycles are related to double-precision floating-point units (from 70% to 27% of stall cycles respectively), data loads (from 16% to 55%), and branch instructions (from 5% to 14%). Stall cycles due to data load are mainly caused by cache memory latencies. NUMA memory latencies cause less than 1% of data stall cycles when data are loading from memory hierarchy. We corroborated this aspect by monitoring the NUMA memory bandwidth usage, which was never higher than 5%. Thus, memory binding techniques have low impact on our parallel algorithm.

The main optimization procedure of our algorithms (`OptimizeNode`) is not mainly affected by the local cache memory latencies (Levels 1 and 2). When the objective function (Eq. 1) of this procedure is applied to one free node by a processor, it needs the local submesh $\mathcal{N}_v$ in addition to the free vertex and some constants. All these data are copied into the local cache memory of the processor when the respective thread is running. During the parallel computing of a color,

each free node is used and updated by only one processor, and the part of the cache memory that stores the submesh is not updated or invalidated by other processors. So, during the parallel processing of a color, there is no dependency between processors or memory cache invalidations.

However, the latency of the Level 3 cache memory does affect the performance of the optimization procedure. The shared Level 3 cache memory is distributed among all multicore chips. As the number of cores/threads increases, it is more probably that a free vertex, updated in the Level 3 cache memory of one multicore chip, is needed by a core of another chip. This causes higher performance degradation due to a larger number of Level 3 cache misses.

Additionally, the performance of the optimization procedure is not affected by the size of the data cache. The working-set of this procedure mainly depends on the number of submesh vertices, which is bounded by the maximum vertex degree. Note in Table 1 that the maximum vertex degree is 26, which is established by the meccano mesh generation method [6,7,28,29]. Thus, the maximum data cache size that is needed by a processor during the optimization of a free node is 3.4 KB approximately, which is about 10% of the available Level 1 data cache memory. All the above mentioned cache memory issues are not affected by the vertex ordering provided by the coloring algorithm.

Using the Itanium2 performance counter called NOPS_RETIRED [17], another performance bottleneck was identified in the machine code that is generated for our serial and parallel algorithms by the Intel compiler for Itanium2 processors. On average, 40% of executed instructions are no-operation instructions. This is caused by the Itanium2 instruction-set, which determines that up to three explicit instructions should be included in a very long instruction word instruction (VLIW). When the compiler analyzes dependencies of our OpenMP programs and it cannot schedule a bundle of at least three explicit instructions, no-operation instructions are automatically generated to fill some instruction slots of the respective VLIW instruction.

## 9 Conclusions and Future Work

We have proposed a new parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes. It is based on a successive set of mesh optimization iterations. In each of them, all the vertex coordinates are recalculated in parallel. The performance evaluation of this algorithm on two many-core shared-memory computers using six benchmark meshes with a wide range of sizes shows that it is a scalable and efficient parallel algorithm. For most of the processed meshes, we have observed that the parallelization of the body of the inner loop of our mesh optimization algorithm allows a reduction of about 1/100 of runtime related to the sequential implementation with 128 cores.

We additionally have analyzed the causes of parallel performance deterioration when OpenMP is used to implement the optimization loop of our algorithm. Using real data collected with some performance counters, we can conclude that the

performance overhead of our parallel algorithm is mainly due to loop-scheduling overhead of the OpenMP programming methodology. Moreover, the results indicate that mesh size positively influences on parallel performance, but the number of optimization iterations deteriorates performance more severely.

We also investigated the influence of three graph coloring algorithms on the performance of our parallel untangling and smoothing algorithm. They have low impact on the total execution time. However, the total execution time of our parallel algorithm depends on the selected coloring algorithm. In this paper, we have shown that there is not a unique coloring algorithm for our parallel algorithm that achieves the highest parallel performance.

When parallel load balancing is analyzed, we observed that load unbalancing is mainly caused by OpenMP loop-scheduling overhead. Thus, its negative impact increases for larger numbers of available parallel threads. When analyzing hardware usage, we observe that our parallel algorithm is processor-bound because it uses CPU and cache memory during 99% of runtime.

Our parallel algorithm is CPU bound and its demonstrated scalability potential for many-core architectures encourages us to extend our work to achieve higher performance improvements from GPUs. The main problem will be to reduce the negative impact of global memory random accesses when a same streaming multiprocessor optimizes non-consecutive mesh vertices.

Many parallel finite-element applications use a domain decomposition approach on distributed-memory computers. Our parallel algorithm needs to be adapted for a partitioned mesh on this type of computers. A distributed-memory strategy may sweep the partitioned mesh in three successive steps. First of all, the inner nodes of each subdomain can be sent to the same computing node. In this case, the boundary nodes of each subdomain should be also sent to its computing node. In a second step, each multi-core computing node colors and optimizes the inner vertices of its subdomain. Finally, subdomain boundary vertices and their current submeshes can be sent to a single multi-core computing node to optimize the subdomain boundary vertices.

**Acknowledgments.** This work has been supported by the Spanish Government, Secretaría de Estado de Universidades e Investigación, Ministerio de Economía y Competitividad, and FEDER, grant contract: CGL2011-29396-C03-01. It has been also supported by Fondo Sectorial CONACYT SENER Hidrocarburos, grant contract: 163723, CESGA ICTS projects (ICTS198, ICTS216), and Intel. We thank to anonymous reviewers for their valuable comments and suggestions on this manuscript.

## References

1. Batdorf, M., Freitag, L.A., Ollivier-Gooch, C.: Computational study of the effect of unstructured mesh quality on solution efficiency. Presented at the 13th Annual AIAA Computational Fluid Dynamics Conference, AIAA (1997)

2. Bazaraa, M., Sherali, H., Shetty, C.M.: Nonlinear Programming. Theory and Algorithms. Wiley (1993)
3. Bozdag, D., Gebremedhin, A., Manne, F., Boman, E., Catalyurek, U.: A framework for scalable greedy coloring on distributed memory parallel computers. Journal of Parallel and Distributed Computing 68(4), 515–535 (2008)
4. Bronevetsky, G., Gyllenhaal, J.C., Supinski, B.R.: Clomp: Accurately characterizing OpenMP application overheads. Int. J. Parallel Programming 37(3), 250–265 (2009)
5. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: Proc. 2000 ACM/IEEE Conference on Supercomputing. IEEE Computer Society (2000)
6. Cascón, J.M., Montenegro, R., Escobar, J.M., Rodríguez, E., Montero, G.: A new meccano technique for adaptive 3-D triangulation. In: Proc. of the 16th International Meshing Roundtable, pp. 103–120. Springer, Berlin (2007)
7. Cascón, J.M., Montenegro, R., Escobar, J.M., Rodríguez, E., Montero, G.: The meccano method for automatic tetrahedral mesh generation of complex genus-zero solids. In: Proc. 18th International Meshing Roundtable, pp. 463–480. Springer, Berlin (2009)
8. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press (2007)
9. Dennis, J.E., Schnabel, R.B.: Numerical methods for unconstrained optimization and nonlinear equations. Society for Industrial and Applied Mathematics, SIAM (1996)
10. Dompierre, J., Labbé, P., Guibault, F., Camarero, R.: Proposal of benchmarks for 3D unstructured tetrahedral mesh optimization. In: Proc. of the 7th International Meshing Roundtable, pp. 459–478. Sandia National Laboratories, Dearborn (1998)
11. Ekman, P.: Studying program performance on the Itanium 2 with pfmon (2003), http://www.pdc.kth.se/~pek/ia64-profiling.txt
12. Escobar, J.M., Rodríguez, E., Montenegro, R., Montero, G., González-Yuste, J.M.: Simultaneous untangling and smoothing of tetrahedral meshes. Comp. Meth. Appl. Mech. Eng. 192, 2775–2787 (2003)
13. Escobar, J.M., Rodríguez, E., Montenegro, R., Montero, G., González-Yuste, J.M.: SUS Code - Simultaneous Mesh Untangling and Smoothing Code (2010), http://www.dca.iusiani.ulpgc.es/proyecto2012-2014/html/Software.html
14. FINIS TERRAE Supercomputer, http://archivo.cesga.es/content/view/917/115/lang,en
15. Freitag, L., Jones, M.T., Plassmann, P.E.: A parallel algorithm for mesh smoothing. SIAM J. Sci. Comput. 20(6), 2023–2040 (1999)
16. Frey, P.J., George, P.L.: Mesh Generation: Application to Finite Elements, 2nd edn. ISTE, London (2010)
17. Intel, Intel® Itanium® 2 Processor Reference Manual. For software development and optimization. Intel, Order Number: 251110-003 (2004)
18. Intel Manycore Testing Laboratory, http://software.intel.com/en-us/intel-manycore-testing-lab
19. Interoperable Technologies for Advanced Petascale Simulations, http://www.itaps.org/
20. Jarp, S.: A Methodology for using the Itanium 2 Performance Counters for Bottleneck Analysis. Tech-Report HP Labs (2002)
21. Jiao, X., Alexander, P.J.: Parallel feature-preserving mesh smoothing. In: Gervasi, O., Gavrilova, M.L., Kumar, V., Laganá, A., Lee, H.P., Mun, Y., Taniar, D., Tan, C.J.K. (eds.) ICCSA 2005. LNCS, vol. 3483, pp. 1180–1189. Springer, Heidelberg (2005)

22. Jones, M.T., Plassmann, P.E.: A parallel graph coloring heuristic. SIAM J. Sci. Comput. 14(3), 654–669 (1993)
23. Kim, J., Panitanarak, T., Shontz, S.M.: A multiobjective mesh optimization framework for mesh quality improvement and mesh untangling. International Journal of Numerical Methods in Engineering 94(1), 20–42 (2013)
24. Knupp, P.M.: Algebraic mesh quality metrics. SIAM J. Sci. Comput. 23(1), 193–218 (2001)
25. Kossaczky, I.: A recursive approach to local mesh refinement in two and three dimensions. J. Comput. Appl. Math. 55, 275–288 (1994)
26. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM Journal on Computing 4, 1036–1053 (1986)
27. Madden, P.H.: Dispelling the myths of parallel computing. IEEE Design & Test of Computers. IEEE Computer Society Digital Library. IEEE Computer Society (2012)
28. Montenegro, R., Cascón, J.M., Escobar, J.M., Rodríguez, E., Montero, G.: An automatic strategy for adaptive tetrahedral mesh generation. Appl. Num. Math. 59(9), 2203–2217 (2009)
29. Montenegro, R., Cascón, J.M., Rodríguez, E., Escobar, J.M., Montero, G.: The meccano method for automatic three-dimensional triangulation and volume parametrization of complex solids. In: Developments and Applications in Engineering Computational Technology, pp. 19–48. Saxe-Coburg Publications, Stirling (2010)
30. Shape benchmark repositories: Cyberware (`http://www.cyberware.com`), The Stanford 3D Scanning Repository (`http://graphics.stanford.edu/data/3Dscanrep`), GAMMA (`http://www-roc.inria.fr/gamma/gamma/download/download.php`)
31. Shontz, S.M., Knupp, P.: The effect of vertex reordering on 2D local mesh optimization efficiency. In: 17th International Meshing Roundtable, pp. 107–124 (2008)
32. Shontz, S.M., Nistor, D.M.: CPU-GPU algorithms for triangular surface mesh simplification. In: Jiao, X., Weill, J.-C. (eds.) Proceedings of the 21st International Meshing Roundtable, vol. 123, pp. 475–492. Springer, Heidelberg (2013)
33. Uncmin++ library, `http://www.smallwaters.com/software/cpp/uncmin.html`
34. Yeo, Y.I., Ni, T., Myles, A., Goel, V., Peters, J.: Parallel smoothing of quad meshes. Vis. Comput. 25(8), 757–769 (2009)