The Effect of Parallelization on a Tetrahedral Mesh Optimization Method

Domingo Benitez(✉), Eduardo Rodríguez, José M. Escobar, and Rafael Montenegro

SIANI Institute, University of Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, Spain
{dbenitez,erodriguez,jmescobar,rmontenegro}@siani.es

Abstract. A parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes is proposed in this paper. This algorithm is derived from a sequential mesh optimization method. We provide a detailed analysis of its parallel scalability and efficiency, load balancing, and parallelism bottlenecks using six benchmark meshes. In addition, the influence of three previously-published graph coloring techniques on the performance of our parallel algorithm is evaluated. We demonstrate that the proposed algorithm is highly scalable when run on a shared-memory computer with up to 128 Itanium 2 processors. However, some parallel deterioration is observed. Here, we analyze its main causes using a theoretical performance model and experimental results.

Keywords: Mesh optimization · Parallel performance evaluation

1 Introduction

Engineering design and analysis of real systems are becoming increasingly complex. The *80/20 design/analysis ratio* seems to be a common industrial experience: design and analysis account for about 80 % and 20 % of time, respectively. In the design process, there are many methods that are time consuming. On average, mesh generation accounts for 20 % of overall time and may take as much CPU-time as field solver, which may need of many man-months [18]. In our Meccano method for tetrahedral mesh generation, the most time-consuming phase is devoted to mesh optimization [15]. On the other hand, mesh generation tools frequently produce meshes with inverted and/or poorly shaped elements. So, untangling and/or smoothing techniques are applied to improve the mesh quality before or during the numerical analysis. In all these problems, improving the speed of mesh optimization with parallelism helps users solve problems faster.

In [8] we propose a simultaneous untangling and smoothing algorithm for tetrahedral meshes, in contrast with other techniques that require two separate and consecutive steps, one for untangling and another for smoothing. For very large tangled meshes, the runtime of our sequential algorithm may be long.

The improvement of our technique using parallel computation is not trivial because two vertices of the same tetrahedron cannot be simultaneously optimized on different processors. In this paper, we propose a parallel algorithm for simultaneously untangling and smoothing tetrahedral meshes with a number of vertices far greater than the number of available processors. Additionally, the performance of this algorithm on a shared-memory computer is analyzed. We show that our algorithm is highly scalable and compute-bound. However, the OpenMP thread scheduling overhead significantly deteriorates the parallel performance.

Section 2 summarizes our sequential approach to mesh untangling and smoothing. Section 3 describes its parallelization. The methodology we used to evaluate the performance of this parallel algorithm is explained in Sect. 4. Its results are presented in Sect. 5. Finally, Sect. 6 provides the main conclusions.

2 Our Approach to Tetrahedral Mesh Optimization

Let us consider $\mathcal{M}$ to be a tetrahedral mesh. Usual techniques to improve the quality of a mesh without inverted tetrahedra are based upon local smoothing [6]. 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. Such a function is based on a certain measurement of the quality of the local submesh $N_v \subset \mathcal{M}$ 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 [13] provide us an appropriate framework to define objective functions. In this paper, we use,

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

being n the number of elements in N_v, p is usually chosen as 1 or 2, $\eta_i = 1/q_i$ is the distortion of the i-th tetrahedron of N_v, and q_i is the mean ratio quality measure of a tetrahedron given by $q = 3\sigma^{\frac{2}{3}}/|S|^2$, 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 σ is the determinant of matrix S: $\sigma = det(S)$. Specifically, the weighted Jacobian matrix S is defined as $S = AW^{-1}$, being $A = (\boldsymbol{x}_1 - \boldsymbol{x}_0, \boldsymbol{x}_2 - \boldsymbol{x}_0, \boldsymbol{x}_3 - \boldsymbol{x}_0)$ the Jacobian matrix, and $\boldsymbol{x}_j, j = 0, \ldots, 3$ the coordinates of the vertices of the tetrahedron. The constant matrix W is derived from the ideal element. For more details, see [8].

Objective functions like (1) do not work properly when there are inverted elements ($\sigma < 0$). This is because they present singularities (barriers) when any tetrahedron of N_v changes the sign of its Jacobian matrix. In [8] 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 σ in the quality metrics by the positive and increasing function $h(\sigma) = \frac{1}{2}(\sigma + \sqrt{\sigma^2 + 4\delta^2})$. When a feasible region exists (subset of $\mathbb{R}^3$ where a free node v could be placed, being its 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 N_v. With this approach, we can use any standard and efficient unconstrained optimization method to find the minimum of the modified objective function [1]. In this way, our method allows simultaneous untangling and smoothing of tetrahedral meshes, in contrast with other techniques that require two separate and consecutive steps.

3 Parallel Algorithm for Mesh Untangling and Smoothing

Algorithm 1 shows our sequential method for simultaneous untangling and smoothing of tetrahedral meshes. It has the following inputs: $\mathcal{M}$ is a tetrahedral mesh, `IT` is a function that provides its total number of inverted tetrahedra, N_v is the set of tetrahedra connected to a free node v, and $\boldsymbol{x}_v$ is the initial position of v. The algorithm iterates over all the nodes and adjusts the spatial coordinates $\boldsymbol{x}_v$ of a free node v in each step; $\hat{\boldsymbol{x}}_v$ is its position after optimization, which is provided by the procedure `OptimizeNode`. Then, Q saves the lowest quality of a tetrahedron when the above-mentioned quality function (q) is used (it is 0 if any tetrahedron is tangled). The function called `quality` measures the increment in Q between successive iterations of mesh optimization. The mesh is optimized until it is completely untangled: `IT(`$\mathcal{M}$`)` $= 0$, and successive iterations increase the minimum mesh quality less than $\lambda = 5\,\%$: $\Delta Q < \lambda$. The algorithm also stops when the number of optimization iterations is larger than `maxIter`.

Algorithm 1. Sequential algorithm for the mesh optimization method

```
1: $Q \leftarrow 0$
2: $j \leftarrow 0$
3: **while** (IT($\mathcal{M}$) $> 0$ **or** $\Delta Q \geq \lambda$) **and** $j \leq$ maxIter **do**
4:     **for** *each vertex* $v \in \mathcal{M}$ **do**
5:         $\hat{\boldsymbol{x}}_v \leftarrow$ OptimizeNode($\boldsymbol{x}_v$,$N_v$)
6:     **end for**
7:     $\Delta Q \leftarrow$ quality($\mathcal{M}$)
8:     $j \leftarrow j + 1$
9: **end while**
```

Algorithm 2 is a parallel algorithm for our mesh optimization method. Its inputs $\mathcal{M}$, `IT`, N_v, $\boldsymbol{x}_v$, `OptimizeNode`, `quality`, λ, and `maxIter` have the same meanings as described for Algorithm 1. This algorithm has to prevent two adjacent nodes from being simultaneously optimized in parallel. On the contrary, new inverted mesh tetrahedra may be created [10]. Thus, when the sequential Algorithm 1 is parallelized, a computational dependency appears between adjacent vertices. This justifies the use of a graph coloring technique in our parallel algorithm to find mesh vertices that do not have computational dependency.

Algorithm 2. Parallel algorithm for the mesh optimization method

```
1: $I \leftarrow$ Coloring(G=(V,E))
2: $Q \leftarrow$ 0
3: $j \leftarrow$ 0
4: while (IT($\mathcal{M}$) $> 0$ or $\Delta Q \geq \lambda$) and $j \leq$ maxIter do
5:     for each independent set $I_i \in I$ do
6:         for all each vertex $v \in \mathcal{M}$ do in parallel
7:             $\hat{\boldsymbol{x}}_v \leftarrow$ OptimizeNode($\boldsymbol{x}_v$,$N_v$)
8:         end for
9:     end for
10:    $\Delta Q \leftarrow$ quality($\mathcal{M}$)
11:    $j \leftarrow j + 1$
12: end while
```

We implemented graph coloring with procedure `Coloring`, which is expressed as follows. Let us consider $G = (V, E)$ to be the graph associated to the tetrahedral mesh $\mathcal{M}$, where V is its set of vertices (without spatial information), and E is the set of their edges, then `Coloring` is a procedure to color $G = (V, E)$. An *independent set* or *color*, I_i, is a set of non-adjacent vertices: $v \in I_i \rightarrow \ v \notin adj(I_i, G = (V, E))$, being $adj(I_i, G = (V, E))$ the set of vertices that are adjacent to all vertex $z \in I_i$, $z \neq v$. In this way, the graph $G = (V, E)$ of a tetrahedral mesh $\mathcal{M}$ is partitioned in a disjoint sequence of colors, $I = \{I_1, I_2, \ldots\}$.

Three different and previously published coloring methods called *C1*, *C2*, *C3* were implemented. *C1* is a sequential coloring method that has been used for mesh smoothing [10]. It requires the use of the asynchronous coloring heuristic proposed in [12]. This heuristic is based on Luby's Monte Carlo algorithm for determining the maximal independent set [14]. *C2* is a parallel version of *C1* for shared-memory computers that was proposed by Jones and Plassmann [12] for distributed-memory computers. *C3* is an parallel greedy coloring algorithm that was proposed by Catalyurek et al. [5]. Section 5 compares the impact of these graph coloring methods on the performance of our parallel optimization method.

The vertex set with the same color (I_i) is equipartitioned among the available processors. Each processor optimizes its assigned set of vertices in a sequential fashion. At each sequential step, `OptimizeNode` procedure is applied to a single vertex v with the method described in Sect. 2. The new vertex spatial position is available to other processors by writing to shared memory. Each subsequent parallel phase optimizes another color until all vertices are optimized. Finally, the exit criteria are the same as the ones used for the sequential algorithm.

Previous studies on parallel algorithms for mesh optimization include the work of Freitag et al. [10], which relies on a theoretical shared-memory model with results using distributed-memory computers. Another similar study was published by Shontz and Nistor [17], which provides performance results for mesh simplification algorithms on GPUs although they do not use graph coloring algorithms to find mesh vertices that do not have computational dependency.

Table 1. Description of the tangled tetrahedral meshes

NAME	m	T	IT	DEGREE	OBJECT
"m = 6358"	6358	26446	2215	26	Bunny
"m = 9176"	9176	35920	13706	26	Tube
"m = 11525"	11525	47824	1924	26	Bone
"m = 39617"	39617	168834	83417	26	Screwdriver
"m = 201530"	201530	840800	322255	26	Toroid
"m = 520128"	520128	2201104	1147390	26	HR toroid

Legends. NAME: name of the tetrahedral mesh. m: total number of mesh vertices. T: total number of mesh tetrahedra. IT: total number of inverted tetrahedra. DEGREE: maximum vertex degree. OBJECT: short description of the real object that is meshed.

4 Experimental Methodology

Our experiments were conducted on a HP Integrity Superdome node with 128 Itanium 2 cores, multithreading disabled, and 1TB NUMA shared memory. Algorithms 1 and 2 were applied on six tangled tetrahedral meshes (see Table 1). All these meshes were constructed with a tool that applies an strategy for adaptive mesh generation based on the *Meccano* method [15], using surface triangulations from different repositories as input data [16]. We also used the Intel C++ compiler 11.1 with `-O2` on a Linux system. The source code of the parallel version included OpenMP directives, which were disabled when the serial version was compiled [9]. Both versions were profiled with PAPI [4], which uses performance counter hardware [11]. All versions were run with processor and memory binding enabled, and the processors were not shared among other user or system level workloads. For each mesh, we run the parallel version using a given maximum number of threads between 1 and 128. Since our algorithms are CPU-bound, there is little sense in using more threads than available cores. We use the following performance metrics: wall-clock time, true speedup, parallel efficiency, and load balancing, which were averaged over more than 30 runs. Each run was divided into two phases. The first phase loops over all vertices repetitively and

Table 2. Best execution times for the complete parallel algorithm (Algorithm 2)

NAME	SRT	BPRT	BNC	BSU	BPE (%)	BCA	C	I	MMQ	AMQ
"m = 6358"	17.3	1.5	72	11.7	16.2	*C1*	29	25	0.13	0.66
"m = 9176"	37.3	1.2	88	31.9	36.3	*C3*	29	26	0.26	0.68
"m = 11525"	33.7	1.1	120	29.7	24.8	*C3*	10	38	0.11	0.65
"m = 39617"	87.4	1.6	128	54.9	42.9	*C1*	31	11	0.17	0.73
"m = 201530"	2505.4	81.3	128	30.8	24.1	*C2*	21	143	0.23	0.67
"m = 520128"	2259.7	41.9	120	54.0	45.0	*C3*	34	36	0.22	0.68

Legends. NAME: mesh name. SRT: serial runtime (s). BPRT: best parallel runtime (s); its number of cores: BNC, speedup: BSU, parallel efficiency: BPE, coloring algorithm: BCA, colors: C, untangling/smoothing iterations: I, minimum mesh quality: MMQ, and average mesh quality: AMQ.

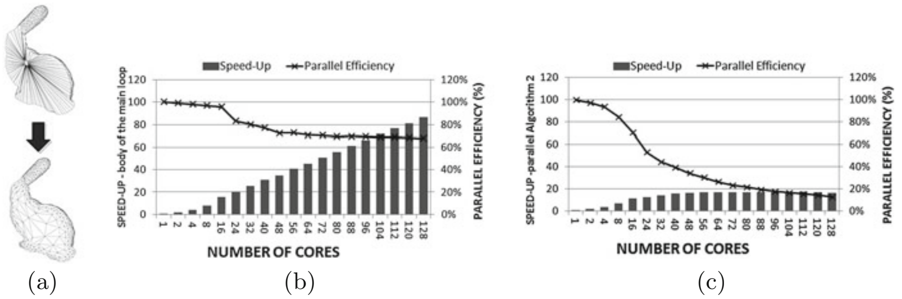

Fig. 1. (a) The tangled (upper) and untangled (lower) meshes called "m = 6358". True speedup and parallel efficiency of: (b) body of the main loop (line 7 of Algorithm 2), (c) complete parallel Algorithm 2, when the graph coloring technique is *C3*.

completely untangles a mesh; at the same time, the mesh is also smoothed. The second phase smoothes the mesh until the exit criteria is met. Table 2 shows the number of untangling and smoothing iterations (I) for each mesh.

5 Performance Evaluation

5.1 Performance Scalability

When the execution time of the main mesh optimization procedure is profiled in both versions of our algorithms, line 5 of sequential Algorithm 1 vs. line 7 of parallel Algorithm 2, we achieve results for true speedup as depicted in Fig. 1(b) when the "m = 6358" mesh is optimized. As can be seen, the true speedup linearly increases as the number of cores increases. Figure 1(b) also shows the parallel efficiency of the main mesh optimization procedure of Algorithm 2 (line 7). Note that up to 128 cores, the parallel efficiency is always above 67 %. Similar results were obtained for the rest of meshes. These results indicate that the main computation of our parallel algorithm is highly scalable. This performance scalability is caused by the parallel processing of independents sets of vertices (colors) that is made in each optimization iteration of the Algorithm 2 (line 7).

When the execution times of the complete sequential and parallel algorithms are profiled, we obtained results for true speedup as depicted in Fig. 1(c) when the "m = 6358" mesh is optimized. Note that in this case, the speedup does not increase so linearly as when the main mesh optimization procedures of algorithms are profiled, and maximum parallel efficiency is lower than 20 %. Table 2 shows the best results for all tetrahedral meshes. Note that the maximum parallel efficiency is 45 %, which was obtained when "m = 520128" mesh was optimized. Furthermore, the number of cores with best speedup depends on the benchmark mesh. In some cases, these highest performance results are obtained when the number of cores is lower than maximum (see column BNC in Table 2).

We investigated the causes of this degradation with a performance model. *SpeedUp* is modeled as the ratio between the sequential (T_S) and the parallel execution times when k cores are activated ($T_{P,k}$):

$$SpeedUp = \frac{T_S}{T_{P,k}} \tag{2}$$

being $T_{P,k}$ the sum of the idealized parallel execution time (T_k, without overhead and with perfect load balancing) and the average performance overhead (O_k) [3]:

$$T_{P,k} = T_k + O_k \quad ,, \quad T_k = \frac{T_S}{k} \quad \rightarrow \quad SpeedUp = \frac{k}{1 + \frac{O_k}{T_k}} \tag{3}$$

$$T_k = \frac{N_k}{IPC_k \times f} \ ,, \ O_k = \frac{N_{O,k}}{IPC_{O,k} \times f} \quad \rightarrow \quad SpeedUp = \frac{k}{1 + \frac{N_{O,k} \times IPC_k}{N_k \times IPC_{O,k}}} \tag{4}$$

N is the total number of executed instructions excluding no-operation instructions: N_k during mesh optimization tasks, and $N_{O,k}$ during parallel thread scheduling; IPC is the number of executed instructions per clock cycle during the runtime: IPC_k during mesh optimization tasks, and $IPC_{O,k}$ during parallel thread scheduling; and f is the fixed clock speed of 1.6 GHz. N and IPC are obtained during profiling with performance counter hardware [7,11].

In Fig. 2, real speedups (graphic marks) are overlapped with predictions of our performance model (lines) when three benchmark meshes are optimized. Real data were obtained using dynamic OpenMP thread scheduling and coloring algorithm *C3*. The average precision of the performance model for all meshes was 95.6 %. Based on these results, we can conclude that for our complete parallel Algorithm 2, true speedup and parallel efficiency deteriorate as the number of cores increases because they tend to be dominated by the loop-scheduling overhead. We note that this parallel overhead is caused by the Itanium 2 instructions ($N_{O,k}$) that are generated when the OpenMP directive that implements line 6 is compiled. Figure 2 also shows that the speedup is maximum at certain number of processors. Our model indicates that it is due to the quadratic polynomial form of the overhead time (O_k) as the number of threads increases.

5.2 Load Balancing

The error produced by the above-described model may be caused by other sources of performance degradation; for example, load imbalance between processors. The load imbalance of k cores (L_k) is measured as follows:

$$L_k = 100\,\% \times \frac{t_{max} - t_{min}}{t_{avg}} \quad ,, \quad t_{max} \geq t_{avg} \geq t_{min} > 0 \tag{5}$$

where t_{max}, t_{avg}, t_{min} are respectively the maximum, average and minimum execution times of the parallel threads without parallel loop-scheduling overhead (line 7 of Algorithm 2). As L_k is smaller, the difference between maximum and minimum thread execution time is smaller than the average execution time of

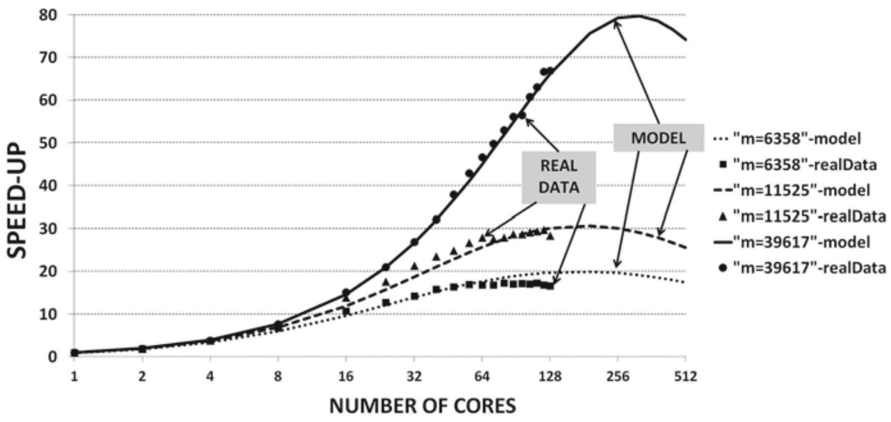

Fig. 2. Comparison of real data and data predicted by the performance model for the true speedup of parallel Algorithm 2.

threads. In these cases, threads tend to be stalled less time because load is more balanced, and so performance is better.

We note that the load imbalance of our parallel algorithm for the six benchmark meshes when all graph coloring algorithms and up to 128 Itanium 2 processors are used is mainly caused by the number of active threads. The higher the number of active threads, the higher the load imbalance. 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 imbalance was observed [2].

5.3 Parallelism Bottlenecks

We applied the profiling methodology described in [7] to analyze the performance inefficiencies caused by bottlenecks in processor functional units, cache memories, and NUMA shared memory. When up to 128 Itanium 2 processors are used, the stall cycles of parallel threads are in the range [29 %. . . 58 %]. These stall cycles are related to double-precision floating-point arithmetic units (from 70 % to 27 % of stall cycles), data loads (from 16 % to 55 %), and branch instructions (from 5 % to 14 %). Stall cycles due to data load instructions are mainly caused by cache memory latencies. NUMA memory latencies cause less than 1 % of data stall cycles, which is corroborated by monitoring the NUMA memory bandwidth usage that was never higher than 5 %. Thus, our parallel algorithm is compute bound and memory binding techniques have low impact on performance.

Another performance bottleneck was identified in the machine code that is generated by the compiler for Itanium 2 processors. On average, 40 % of executed instructions are no-operation instructions. This is caused by the long instruction format where up to three explicit instructions are included [11]. When the compiler cannot schedule a bundle of at least three instructions, no-operation instructions are used to fill some instruction slots. Enhancing the instruction level parallelism of our main optimization procedure may improve thread performance. However, it is a time-consuming task because the algorithms would have to be hand coded. Conventional x86 architectures do not suffer from this performance bottleneck because their instruction formats only include one instruction [2].

5.4 Influence of Graph Coloring Algorithms on Parallel Performance

Many papers evaluate the performance of graph coloring algorithms on parallel computers [5,10,12]. However, for the authors' knowledge, their impacts on the performance of algorithms that use these coloring algorithms is rarely reported.

First of all, we confirmed the performance results published in previous papers for *C1*, *C2* and *C3* coloring algorithms. Then, we investigated their influence on the performance of our parallel optimization algorithm. Since graph coloring aids in discovering parallelism, the time involved in graph coloring is only considered when profiling our parallel algorithm and not the sequential algorithm. When up to 128 processors and the six benchmark meshes are used, the percentage of total runtime that is required by the graph coloring methods *C1*, *C2* and *C3* ranges respectively from 0.8 % to 3.4 %, from 2.4 % to 12.9 %, and from 0.1 % to 1.9 %. This means that the computational load required by our parallel algorithm is much heavier than required by these graph coloring algorithms.

However, the total execution time depends on the selected coloring algorithm. We note that our parallel algorithm achieves the best performance on the Itanium2-based computer when the *C3* graph coloring technique is used. This is due to the lowest number of colors in which *C3* groups the vertices of each mesh with respect to the other graph coloring methods *C1* and *C2*. A lower number of vertex groups allow a larger number of vertices to be processed in parallel.

6 Conclusions and Future Work

We have proposed a new parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes. It is based on successive optimization iterations. In each of them, the spatial coordinates of independent sets of vertices are modified in parallel. The performance evaluation of this algorithm on a 128-core shared-memory computer using six meshes shows that it is a scalable and efficient parallel algorithm. It is due to the graph coloring method that is used to identify independent sets of vertices without computational dependency. We have additionally analyzed the causes of the parallel performance deterioration. We conclude that it is mainly due to loop-scheduling overhead of the OpenMP programming methodology. When analyzing hardware usage, we observe that our parallel algorithm is compute-bound because it uses the functional units and cache memory during 99 % of runtime. Finally, we also investigated the influence of three graph coloring methods on the performance of our parallel algorithm. They have low impact on the total execution time. However, the performance of our parallel algorithm depends on the selected coloring method. In this paper, we have shown that the *C3* coloring method [5] allows our parallel algorithm to achieve the highest parallel performance on a Itanium2-based computer.

The demonstrated scalability potential of our compute-bound parallel algorithm for shared-memory 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 non-consecutive mesh vertices are optimized by the same streaming multiprocessor.

Acknowledgments. This work has been supported by the Spanish *Sec. Estado Univ. e Inv.*, *Min. Economa y Competitividad* and *FEDER*, contract: CGL2011-29396-C03-01. It has been also supported by *Fondo Sec. CONACYT SENER Hidrocarburos*, contract: 163723, and two *CESGA ICTS* projects.

References

1. Bazaraa, M., Sherali, H., Shetty, C.M.: Nonlinear Programming. Wiley, New York (1993)
2. Benitez, D., Rodríguez, E., Escobar, J.M., Montenegro, R.: Performance evaluation of a parallel algorithm for simultaneous untangling and smoothing of tetrahedral meshes. In: Proceedings of the 22nd International Meshing Roundtable, pp. 579–598. Springer (2014)
3. Bronevetsky, G., Gyllenbaal, J., De Supinski, B.R.: CLOMP: accurately characterizing OpenMP application overheads. Int. J. Parallel Prog. **37**(3), 250–265 (2009)
4. Browne, S., Dongarra J., Garner N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: Proceedings of the ACM/IEEE Conference on Supercomputing. IEEE Computer Society (2000)
5. Catalyurek, U.V., Feo, J., Gebremedhin, A.H., Halappanavar, M., Pothen, A.: Graph coloring algorithms for multicore and massively multithreaded architectures. Parallel Comput. **38**(10–11), 576–594 (2012)
6. Dompierre, J., Labbé, P., Guibault, F., Camarero, R.: Proposal of benchmarks for 3D unstructured tetrahedral mesh optimization. In: Proceedings of the 7th International Meshing Roundtable, pp. 459–478. Sandia National Laboratories (1998)
7. Ekman, P.: Studying program performance on the Itanium 2 with pfmon. www.pdc.kth.se/~pek/ia64-profiling.txt (2003)
8. Escobar, J.M., Rodríguez, E., Montenegro, R., Montero, G., González-Yuste, J.M.: Simultaneous untangling and smoothing of tetrahedral meshes. Comput. Methods Appl. Mech. Eng. **192**, 2775–2787 (2003)
9. Escobar, J.M., Cascón, J.M., Rodríguez, E., Montenegro, R.: A new approach to solid modeling with trivariate T-splines based on mesh optimization. Comput. Methods Appl. Mech. Eng. **200**(45–46), 3210–3222 (2011)
10. Freitag, L., Jones, M.T., Plassmann, P.E.: A parallel algorithm for mesh smoothing. SIAM J. Sci. Comput. **20**(6), 2023–2040 (1999)
11. Intel: Intel Itanium 2 processor reference manual (251110-003). Intel (2004)

12. Jones, M.T., Plassmann, P.E.: A parallel graph coloring heuristic. SIAM J. Sci. Comput. **14**(3), 654–669 (1993)
13. Knupp, P.M.: Algebraic mesh quality metrics. SIAM J. Sci. Comput. **23**(1), 193–218 (2001)
14. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM J. Comput. **4**, 1036–1053 (1986)
15. Montenegro, R., Cascón, J.M., Escobar, J.M., Rodríguez, E., Montero, G.: An automatic strategy for adaptive tetrahedral mesh generation. Appl. Numer. Math. **59**(9), 2203–2217 (2009)
16. Shape repositories. www.cyberware.com, http://graphics.stanford.edu/data/3Dscanrep, www-roc.inria.fr/gamma/gamma/download/download.php
17. Shontz, S.M., Nistor, D.M.: CPU-GPU algorithms for triangular surface mesh simplification. In: Proceedings of the 21st International Meshing Roundtable, pp. 475–492. Springer (2013)
18. Von Cottrell, J.A., Hughes, T.J.R., Bazilevs, Y.: Isogeometric Analysis: Toward Integration of CAD and FEA. Wiley, New York (2009)