A Flexible Shared-Memory Parallel Mesh Adaptation Framework

Lukas Gnam
Christian Doppler Laboratory for High Performance TCAD
Institute for Microelectronics
TU Wien, Austria
gnam@iue.tuwien.ac.at

Paul Manstetten
Institute for Microelectronics
TU Wien, Austria
manstetten@iue.tuwien.ac.at

Michael Quell
Christian Doppler Laboratory for High Performance TCAD
Institute for Microelectronics
TU Wien, Austria
quell@iue.tuwien.ac.at

Karl Rupp
Institute for Microelectronics
TU Wien, Austria
rupp@iue.tuwien.ac.at

Siegfried Selberherr
Institute for Microelectronics
TU Wien, Austria
selberherr@iue.tuwien.ac.at

Josef Weinbub
Christian Doppler Laboratory for High Performance TCAD
Institute for Microelectronics
TU Wien, Austria
weinbub@iue.tuwien.ac.at

***Abstract*—Numerical simulation is an important tool used in various fields of computational science and engineering. The models to be solved by simulation are predominantly based on equations which require the discretization of a spatial domain. The accuracy of the simulation results is heavily influenced by the properties of the underlying spatial discretization, the mesh. Thus, adapting a mesh to meet certain criteria is an integral step to achieve a desired accuracy and high computational performance. With the trend of more and more cores on a compute node, it is essential to efficiently exploit this available on-node parallelism of modern multi-core systems. Our work introduces a flexible shared-memory parallelized mesh adaptation framework. We show the integrability of available serial mesh adaptation algorithms and applicability to multi-region meshes. The first step of the framework is the partitioning of the initial mesh, where all the resulting partitions are subsequently assigned to independent sets using graph coloring algorithms. These sets are then processed in parallel using two different adaptation algorithms: A template-based algorithm and a Delaunay-based algorithm provided by the TetGen software. We perform benchmarks using a cube geometry with different mesh resolutions as well as a model of a microelectronic transistor device structure to demonstrate the scalability of our approach and investigate the resulting element quality. The obtained speedups for this inherently memory-bound problem and for constant problem sizes range between 4.6 and 8.6 using 16 threads.**

***Index Terms*—shared-memory, parallel meshing, flexible mesh adaptation, unstructured meshes**

I. Introduction

The use of unstructured meshes together with finite element or finite volume discretization methods is a popular approach for numerically solving partial differential equations in a wide variety of applications [1]–[3]. To obtain highly accurate solutions it is often necessary to globally or locally adapt the initial mesh by mesh refinement and by improving the element quality. Due to the potentially high number of mesh elements, the adaptation process can pose a severe computational bottleneck. This is especially the case if (i) the mesh adaptation step has to be conducted several times in order to obtain a proper mesh resolution in certain regions of interest, or (ii), the mesh has to be adapted after each solution step to properly capture the evolved solution-critical quantity distributions. This is a typical challenge found in semiconductor device simulations [4]: Fig. 1a shows an exemplary multi-material mesh of a double gate fin field-effect transistor (FinFET).

In the past, a lot of effort has been put into developing methods and algorithms to alleviate the bottleneck of mesh adaptation resulting in several parallelization approaches for distributed-memory computing [5]–[8].

However, the wide availability of on-node parallelism makes shared-memory approaches inherently attractive. Additionally, the number of applications requiring fast and accurate simulation results without having access to high-performance, distributed-memory computer clusters is increasing, e.g., medical analysis and treatments [10], [11].

Recently, mesh adaptation approaches emerged taking advantage of the available shared-memory parallelization on modern compute nodes [12], [13]. Commonly, these methods use fine-grained parallelism, where the implementation of the algorithms has to be carefully tailored to the shared-memory environment. Hence, it becomes rather challenging to integrate new algorithms or to adopt existing and well tested serial algorithms for utilization with these frameworks. Contrary to those approaches, we follow a coarse-grained parallelization approach offering the possibility to use available highly-tuned serial mesh adaptation algorithms in parallel.

This work is an extension of our previously presented and partly OpenMP shared-memory parallelized meshing framework for tetrahedral meshes [14]. In this initial work, the framework is capable of partitioning an input mesh into several sub-partitions which are subsequently distributed into independent sets using graph coloring techniques followed by a parallel refinement.

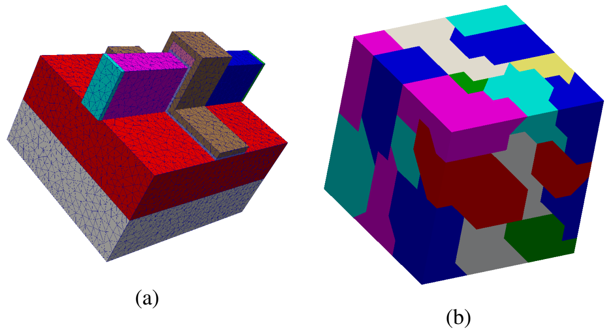

Fig. 1: (a) Tetrahedral mesh of an exemplary multi-material geometry of a double gate FinFET used in semiconductor device simulations. The different colors denote different material regions. (b) Example of possible independent sets of partitions denoted by the different colors. The mesh is partitioned using *mt-METIS* [17] and the partitions are colored using a greedy graph coloring algorithm.

These independent sets of partitions are not connected and can therefore safely be processed in parallel (see Fig. 1b). Here, we present an extension of our initial approach by adding parallel partitioning and coloring algorithms. Furthermore, two different mesh adaptation approaches have been included: A template-based approach [13], [15] and a Delaunay-based approach provided by *TetGen* [16]. Additionally, we include several algorithms improving the mesh element quality, e.g., vertex smoothing. The high degree of integrability shows the flexibility of our framework, as available tools can be added to the framework. We compare the different algorithms with respect to their scaling capabilities and the resulting tetrahedral mesh quality using various tetrahedral volume meshes.

This paper is organized as follows: Section II describes the building blocks of our approach and presents the implemented algorithms. Section III reports the parallel performance and achieved mesh qualities using different example geometries.

II. Flexible Mesh Adaptation Framework

Our parallel mesh adaptation framework is comprised of multiple steps, where all computationally expensive parts are shared-memory parallelized. The full framework is depicted in Fig. 2, where the aforementioned computationally demanding steps are colored in purple. The initial mesh is partitioned into a user-defined number of partitions (Fig. 2a), which are subsequently colored to obtain independent sets of partitions (Fig. 2b). An exemplary coloring of partitions is shown in Fig. 1b. This enables the integration of different partitioning and coloring algorithms into the framework, as discussed in detail in the following subsection.

Based on these independent sets of partitions, any serial mesh adaptation algorithm can be integrated into the framework (Fig. 2c), and therefore executed in parallel, as no edges or facets are shared between the individual partitions of a set. The processing order of the independent set has implications on the adaptation process on the partition boundary: If the boundary has already been adapted during the processing of a neighboring partition, the adaptation is limited to the interior.

After the parallel processing of a set of partitions, the mesh adaptations on the partition interfaces are communicated to all affected neighboring partitions, yielding potentially non-conforming mesh partitions. Therefore, prior to the processing of a subsequent set of partitions, all non-conformities are resolved during a healing step (Fig. 2d). Subsection II-B describes the adaptation and healing process in detail, and shows the integration of two serial mesh adaptation algorithms into our framework.

A. Partitioning and Coloring

The first step in our mesh adaptation approach is the partitioning of the initial input mesh into a user-defined number of contiguous mesh partitions. To achieve this task we use *mt-METIS* [17], a multi-threaded version of the graph partitioning software *METIS* [18]. It converts the mesh into a graph and subsequently applies graph partitioning methods to create the desired number of mesh partitions. The partitioning software assigns each partition a dedicated identifier (Partition ID). Subsequently, the adjacency information of the partitions is created which is later used during the mesh healing and adaptation step.

In order to apply a serial adaptation algorithm on each partition in parallel, several graph coloring algorithms are used to create independent sets of the previously created mesh partitions [19]. The different colors used by the graph coloring algorithms are represented using unique numbers (Color ID). The coloring algorithms range from a serial greedy coloring approach to shared-memory parallel graph coloring algorithms as presented by Çatalyürek et al. [20] and algorithms creating balanced colorings as shown by Lu et al. [21]. One advantage of balanced coloring populations is the implicit load-balancing, which is especially advantageous in situations where the number of partitions is in the same order as the number of active threads.

Within our framework it is also possible to consider multi-material meshes. The material regions are used as initial input to the subsequent partitioning and coloring steps. Hence, the material interfaces are preserved throughout the adaptation process (see Fig. 1a).

B. Mesh Adaptation and Healing

The coloring process is followed by the parallel mesh adaption procedure. Each independent set of partitions, identified by a common Color ID, is processed in parallel. A dedicated mesh data structure is maintained for each partition. To keep a valid link between the initial mesh vertex indices (*global indices*) and the mesh vertex indices in the partitions (*local indices*), we store their connection using unordered associative containers.

Furthermore, a linear data array (*outbox*) stores the information regarding vertex insertions on the interface of neighboring partitions (see Fig. 3). If a vertex is inserted on such an

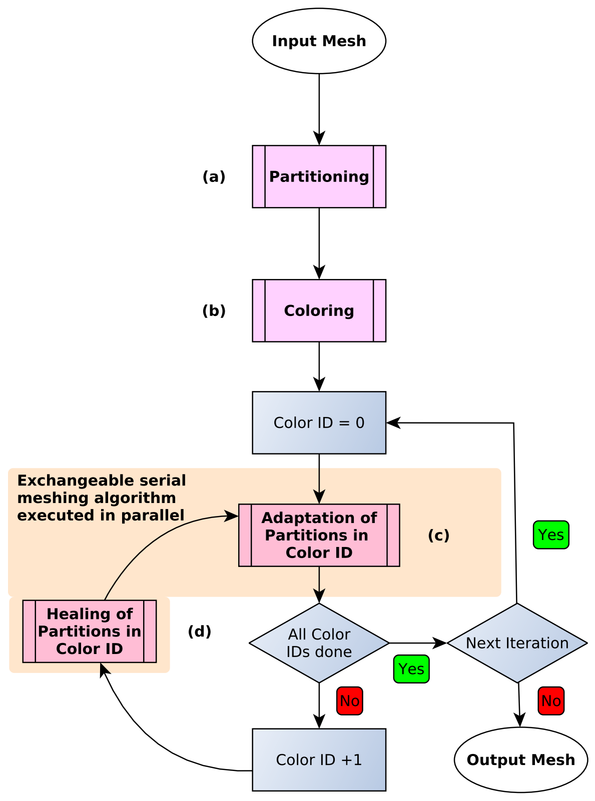
Input Mesh
(a)
Partitioning
(b)
Coloring
Color ID = 0
Exchangeable serial meshing algorithm executed in parallel
Adaptation of Partitions in Color ID
(c)
Yes
Healing of Partitions in Color ID
(d)
All Color IDs done
Next Iteration
Yes
No
No
Color ID +1
Output Mesh

Fig. 2: Diagram of the fully shared-memory parallelized mesh adaptation framework depicting the single modules, with the exchangeable blocks shown in purple enabling, for example, the integration of different serial meshing algorithms. Since the color identifiers (Color IDs) are denoted using unique integers, the loop starts with Color ID 0.

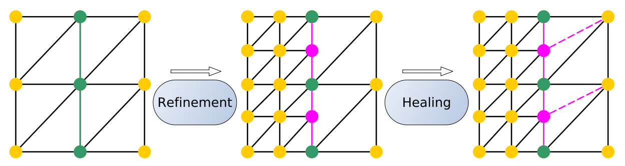

Fig. 3: Schematic showing our refinement and healing procedure using a so-called *outbox* for each partition. In the first step, the left partition refines its elements including the interface (green edges) to its neighbor (purple vertices and edges). Afterwards, the healing procedure integrates the inserted vertices (purple vertices) into its own partition (dashed purple edges) and subsequently refines the affected elements to sustain conformity.

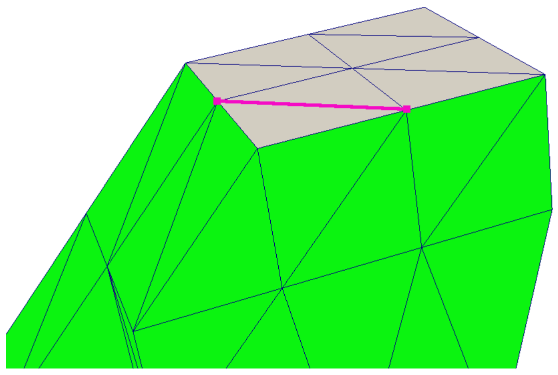

Fig. 4: The green triangles are part of two different partitions as well as the two purple vertices; the gray triangles are only part of a single partition. Since we store only to which partitions the vertices belong and do not store any explicit edge or facet data, it can happen that an edge is mistakenly marked as part of an interface to a neighboring partition (purple edge).

interface edge or facet, the inserting partition writes the Partition ID of the involved neighbor, the two global indices of the vertices comprising the corresponding edge, as well as the own local vertex index of the newly inserted vertex into its own *outbox*. This information is accessed by the neighboring partitions during the subsequent step of mesh healing.

We limit the adaptation of the partition interfaces to those partitions which have a smaller Color ID than all their neighboring partitions owning the same edge or facet. This prevents the partition interfaces from being adapted by more than one partition and leads to a homogeneous refinement across the partition interfaces.

In order to check if an edge is part of an interface to a neighboring partition, we store the associated Partition IDs of each vertex. Therefore, an edge may be mistakenly marked as an interface edge, as shown in Fig. 4. The owning partition still records the corresponding data (Partition ID of the neighboring partition, the two global vertex indices of the edge, and the local index of the added vertex) in its *outbox*. A subsequently processed neighboring partition (from the next Color ID) inserts this vertex marked wrongly as an interface vertex. To resolve this issue, a check for dangling vertices after the healing procedure is conducted, which removes all mistakenly inserted vertices. This approach circumvents the adaption of the data structures and algorithms which do not store edges or facets explicitly, while simultaneously requiring negligible overhead, as the detection of dangling vertices is computationally trivial.

We provide two different options for mesh refinement. A basic mesh element subdivision scheme [15], which has been successfully implemented in the shared-memory parallel software *PRAgMaTIc* by Rokos et al. [13]. It operates by subdividing an edge into two edges by vertex insertion, if an edge exceeds a user-prescribed maximum length threshold. Each mesh element is subsequently refined based on the number and position of the subdivided edges using predefined templates. Higher mesh resolutions are achieved by iterating several times over all mesh elements, since an element can only be refined once in a single iteration.

In our framework, we use a serial version of this template-based refinement algorithm, but apply it in parallel to all partitions with identical Color ID to achieve parallelism.

As an alternative mesh refinement algorithm, we integrate the serial Delaunay-based mesh generator *TetGen* [16] into our framework via the provided application programming interface. *TetGen* is, up to now, not able to perform anisotropic mesh refinement, i.e., refinement depending on the direction of the relevant mesh edge [16]. Therefore, it can be beneficial to apply a template-based approach, since it is able to conduct anisotropic mesh adaptation [13]. The template-based mesh refinement requires additional quality improvement steps after the element subdivision. To achieve this, we use the element swapping and smoothing operations provided by *PRAgMaTIc* in our coarse-grained parallel approach. To ensure proper element quality with the Delaunay-based approach, we use the available quality optimization options included in *TetGen* during the refinement process.

However, to conduct the mesh refinement operations using the Delaunay-based approach on the interior of the partitions, it is necessary to first use the template-based element subdivision schemes for refining the boundary elements. This is due to the fact that our mesh healing data structures and procedures depend on the template-based refinement schemes and data structures. After the healing is successfully completed, we restrict the Delaunay approach to perform its mesh adaptation routines only in the interior of the mesh partition.

After a set of independent partitions is processed, each partition in the subsequently processed set heals its interface elements in order to sustain a conformal mesh (see Fig. 3). To detect interface adaptations, a partition checks all the *outboxes*, if vertices have been inserted into the interface from its neighboring partitions with a Color ID smaller than its own. The new interface vertices are accessible using the global indices stored in the *outbox* combined with the Partition ID of the originating partition and the local vertex index inside the originating partition.

After all *outboxes* have been processed, the template-based element subdivision schemes are used to heal the interface mesh elements. Afterwards, the parallel mesh refinement algorithm continues, either using the template-based or the Delaunay-based adaptation algorithm (see Fig. 2c).

III. Evaluation

In order to evaluate the performance and the resulting mesh quality of our framework we use two different resolutions of a three-dimensional cube mesh as shown in Fig. 1b. We refer to the two cube meshes as *small* and *big*, consisting of 6000 and 93750 tetrahedral elements, respectively. Additionally, we perform tests with a geometry taken from a semiconductor device simulation problem, i.e., a double gate FinFET (see Fig. 1a) [4]. The FinFET mesh initially consists of 837584 mesh cells. We compare the two aforementioned mesh adaptation algorithms: The template-based algorithm and *TetGen*. The number of partitions is set to 1024 for all cases and the coloring is performed using both an unbalanced and a balanced greedy approach [19].

A. Benchmark Platform

We use a single node on the Vienna Scientific Cluster (VSC-3)[1] for our benchmarks, which is equipped with two Intel Xeon E5-2650v2 Ivy Bridge EP processors running with a clock frequency of 2.6 GHz and a total of 256 GB of main memory. The benchmarks are compiled using Intel's C++ compiler version 17.0.4 and optimization level `-O3`. The threads are pinned using the provided thread-core affinity *GOMP_CPU_AFFINITY* variable, to avoid migration of OpenMP threads.

B. Adaptation Parameters

We perform the benchmarks for the template-based refinement algorithm using two iterations, with a length threshold of 25% of the initial edge length for the *small* and *big* cube meshes. For the Delaunay-based algorithm we apply a volume constraint to the *small* cube of 1% of the initial element volume for two iterations. The *big* cube mesh is adapted applying the same relative constraints. Additionally, the FinFET is also investigated using a single iteration of the template-based refinement algorithm with a length threshold of 25% applied to the original edge length. The Delaunay-based approach is used with a volume constraint of 13% of the element volume.

C. Partition Coloring

Fig. 5 depicts the number of partitions for each Color ID. The presented data indicates that the application of a greedy coloring for the partitions yields less partitions in higher Color IDs. Therefore, it is possible that the number of threads exceeds the number of partitions as the number of colors increases, leaving threads without work and affecting potential speedup. Nevertheless, the data in Fig. 6 proves that for the given test meshes the coloring has almost no effect, yielding similar runtime and speedup results. The reason for this behavior is that the highest Color IDs still offer a significant workload for the maximum number of threads we used in this study.

D. Strong Scaling Analysis

We assess the strong scalability of the presented framework by increasing the number of threads up to 16 physical cores, while keeping the problem size constant. The obtained speedups for our set of meshes using 1024 partitions are depicted in Fig. 6.

The *small* cube is refined using two iterations, yielding an increase in tetrahedral mesh elements from 6000 to about 315000. With 16 threads the maximum speedups obtained for the *small* cube mesh are 4.6 using the template-based approach and 7.8 using the Delaunay-based approach. Our study shows that for this small problem size the template-based

[1]https://vsc.ac.at

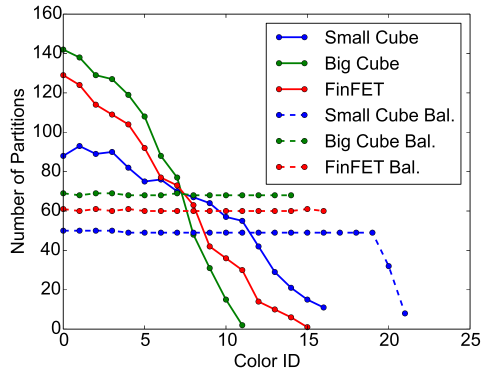

Fig. 5: The population of the different Color IDs obtained with both a balanced and an unbalanced greedy coloring approach for all test meshes.

mesh refinement algorithm delivers faster execution times than the Delaunay-based workflow.

The data for the *small* cube indicate that for smaller problem sizes the overhead introduced by our coarse-grain framework has significant influence on the overall performance: The runtime of the original serial *TetGen* software (1.29 seconds) is only outperformed when using more than eight threads.

Interestingly, for the *small* cube mesh we observe a superlinear speedup applying the Delaunay algorithm with two threads. The reason for this behavior is that this algorithm usually has a complexity of $O(n \log n)$: A decomposition into N partitions yields a theoretical acceleration of

$$\frac{n \log n}{\frac{n}{N} \log \frac{n}{N}} = N \frac{1}{1 - \frac{\log N}{\log n}}, \tag{1}$$

with the right side being larger than 1. Therefore, a decomposition into N parts could give a speedup larger than N. However, as non-perfect load balancing or different geometrical complexity of the partitions strongly influence the workload, superlinear speedup is rarely observed [22].

For the *big* cube and the FinFET the number of resulting tetrahedral mesh elements is increased from 93750 to about 5500000, and from 837584 to about 6600000 elements, respectively. Two iterations are used for the *big* cube mesh, whereas for the FinFET only one iteration is used, due to the fact that a second iteration is not supported with the current implementation of the *TetGen* software. The speedups for the *big* cube mesh using 16 threads peak at 5.2 and 8.6 for the template and Delaunay approach, respectively. Regarding the FinFET, maximum speedups of 5.1 for the template-based and 6.8 for the Delaunay-based approach are obtained. The runtime of the serial original *TetGen* software (FinFET: 32.2 s, *big* cube: 21.6 s) is already outperformed using two threads (26.3 s) and four threads (14.8 s) for the FinFET and *big* cube mesh, respectively. The template-based workflow is slower for the *big* cube and the FinFET mesh, since more time is needed for swapping and smoothing operations compared to the Delaunay-based approach.

For all three investigated test meshes the speedup using up to eight threads is strongly limited by memory bandwidth [8]. Additionally, for higher thread counts the performance is heavily influenced by effects occurring due to non-uniform memory access (NUMA).

E. Mesh Quality

In order to investigate the influence of the constrained interface element adaptation process on the resulting element quality of the adapted meshes, we use two different quality metrics [23]. The first metric is the edge ratio which is defined as

$$R_{\text{Edge}} = \frac{L_{\text{max}}}{L_{\text{min}}}, \tag{2}$$

where L denotes the maximum and minimum edge length of a tetrahedron. A ratio of 1 denotes optimal tetrahedral element quality, i.e., an equilateral tetrahedron. As second quality metric we use the minimum dihedral angle present in a tetrahedral element, where an angle of about $70.53°$ represents the optimal element quality. Fig. 7 shows the resulting element qualities obtained for our meshes. The blue bars depict the element quality resulting from the template-based algorithm, the green bars show the quality obtained applying the Delaunay-based algorithm in our framework, and the pink bars denote the element quality obtained with the original serial *TetGen* software.

The template-based algorithm splits only elements already present in the mesh by inserting vertices on their edges and subsequently aims to improve the elements by applying smoothing and swapping operations. On the contrary, the Delaunay-based mesh refinement approach has more freedom in choosing the ideal position for inserting new vertices, since it is not limited to the element edges for vertex insertion. Thus, the resulting element quality of the template-based approach is inferior compared to its competitor. We have observed this behavior in all three investigated meshes (see Fig. 7). The quality results with respect to the edge ratio show that the additional freedom of the Delaunay approach results in lower edge ratios for all investigated test meshes compared to the template approach. The same holds for the minimum dihedral angle distributions. The depicted accumulations of elements with similar minimum dihedral angles of 35.3°, 45.0°, 54.7°, and 60.0° in both cube meshes result from the minimum dihedral angle distribution in the initial meshes peaking at the mentioned angles. These peaks are preserved in our framework due to the constrained boundary adaptation.

For the *small* cube mesh the original serial *TetGen* software yields a better distribution of minimal dihedral angles and edge ratios, showing that, if the number of elements in the partitions is too small, the resulting element quality using our parallel framework is limited. The element qualities obtained for the other two meshes evaluated in this study using the presented parallelization framework are similar to the original serial *TetGen* software, proving that the constraints for the mesh adaptation on interface elements has little effect on the resulting mesh quality.

IV. Summary

We present a flexible shared-memory parallel mesh adaptation approach, capable of applying available serial mesh adaptation algorithms in parallel. We evaluate the approach by using two different mesh adaptation approaches and test geometries, a cube and a FinFET, and analyze the resulting mesh element quality with respect to its edge ratio and minimum dihedral angle. Additionally, a strong scaling analysis shows the parallel performance. The maximum speedup achieved for this inherently memory-bound problem is 8.6 for 16 threads using the Delaunay-based approach. Additionally, our parallel workflow is able to significantly outperform the original serial *TetGen* software for big test meshes.

Additionally, our investigations show that the Delaunay-based meshing technique included in *TetGen* produces higher quality meshes than a template-based approach and that the Delaunay-based approach is able to outperform the template-based approach regarding performance in some cases. However, for anisotropic mesh adaptation the template-based solution is attractive. Possible future extensions could focus on investigating other mesh healing and domain decomposition approaches.

Acknowledgment

The financial support by the *Austrian Federal Ministry for Digital and Economic Affairs* and *the National Foundation for Research, Technology and Development* is gratefully acknowledged. Additionally, the financial support by the Austrian Science Fund (FWF) through project P29119-N30 is gratefully acknowledged. The computational results presented have been achieved using the Vienna Scientific Cluster (VSC).

References

[1] T. Dias dos Santos, M. Morlighem, H. Seroussi, P. Remy Bernard Devloo, and J. Cardia Simões: Implementation and Performance of Adaptive Mesh Refinement in the Ice Sheet System Model (ISSM v4.14). Geoscientific Model Development Discussions, pp. 1–32, 2018, doi:10.5194/gmd-2018-194.

[2] P. Fromme, G.W. Blunn, W.J. Aston, T. Abdoola, J. Koris, and M.J. Coathup.: The Effect of Bone Growth onto Massive Prostheses Collars in Protecting the Implant from Fracture. Medical Engineering & Physics, vol. 41, pp. 19–25, 2017, doi:10.1016/j.medengphy.2016.12.007.

[3] A. Paluszny, and R.W. Zimmermann: Modelling of Primary Fragmentation in Block Caving Mines Using a Finite-Element Based Fracture Mechanics Approach. Geomechanics and Geophysics for Geo-Energy and Geo-Resources, vol. 3, no. 2, pp. 121–130, 2017, doi:10.1007/s40948-016-0048-9.

[4] A. Bojita, C. Boianceanu, E. Tomas, and V. Topa: A Study of Adaptive Mesh Refinement Techniques for an Efficient Capture of the Thermo-Mechanical Phenomena in Power Integrated Circuits. In: Proceedings of the 40th International Semiconductor Conference, 2017, pp. 205–208, doi:10.1109/SMICND.2017.8101201.

[5] A. Loseille, V. Menier, and F. Alauzet: Parallel Generation of Large-Size Adapted Meshes. Procedia Engineering, vol. 124, pp. 57–69, 2015, doi:10.1016/j.proeng.2015.10.122.

[6] M.O. Freitas, P.A. Wawrzynek, J.B. Cavalcante-Neto, C.A. Vidal, B.J. Carter, L.F. Martha, and A.R. Ingraffea: Parallel Generation of Meshes with Cracks Using Binary Spatial Decomposition. Engineering with Computers vol. 32, no. 4, pp. 655–674, 2016, doi:10.1007/s00366-016-0444-3.

[7] M. Greenwood, K.N. Shampur, N. Ofori-Opoku, T. Pinomaa, L. Wang, S. Gurevich, and N. Provatas: Quantitative 3D Phase Field Modelling of Solidification Using Next-Generation Adaptive Mesh Refinement. Computational Material Science, vol. 42, pp. 153–171, 2018, doi:10.1016/j.commatsci.2017.09.029.

[8] O. Meister, K. Rahnema, and M. Bader: Parallel Memory-Efficient Adaptive Mesh Refinement on Structured Triangular Meshes with Billions of Grid Cells. ACM Transactions on Mathematical Software, vol. 43, no. 3, pp. 19:1–19:27, 2017, doi:10.1145/2947668.

[9] J. Rodriguez, J. Weinbub, D. Pahr, K. Rupp, and S. Selberherr: Distributed High-Performance Parallel Mesh Generation with ViennaMesh. Lecture Notes in Computer Science, vol. 7782, pp. 548–552, 2013, doi:10.1007/978-3-642-36803-5_44.

[10] W. Wang, W. Zou, D. Hu, and J. Wang: Adaptive Mesh Refinement for Elastic Modulus Reconstruction in Elastography. Proceedings of the Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine, vol. 232, no. 2, pp. 215–22, 2018, doi:10.1177/0954411917752026.

[11] M. Namburi, S. Nagothu, C.S. Kumar, N. Chakrapani, C.H. Hanumantharao, and S.K. Kumar: Evaluating the Effects of Consolidation on Intrusion and Retraction Using Temporary Anchorage Devices - A FEM Study. Progress in Orthodontics, vol. 18, no. 2, pp. 18:1–18:11, 2017, doi:10.1186/s40510-016-0155-8.

[12] D. Ibanez, and M. Shephard: Mesh Adaptation for Moving Objects on Shared Memory Hardware. In: Proceedings of the 25th International Meshing Roundtable, 2016.

[13] G. Rokos, and G. Gorman: PRAgMaTIc – Parallel Anisotropic Adaptive Mesh Toolkit. In: Facing the Multicore-Challenge III: Aspects of New Paradigms and Technologies in Parallel Computing, Keller, R., Kramer, D., Weiss, J.-P., Eds. Springer, Berlin, 2013, pp. 143–144, doi:10.1007/978-3-642-35893-7_22.

[14] L. Gnam, J. Weinbub, K. Rupp, F. Rudolf, and S. Selberherr: Using Graph Partitioning and Coloring for Flexible Coarse-Grained Shared-Memory Parallel Mesh Adaptation. In: Proceedings of the 26th International Meshing Roudtable, 2017.

[15] R. Biswas, and R. Strawn: A New Procedure for Dynamic Adaptation of Three-Dimensional Unstructured Grids. Applied Numerical Mathematics, vol. 13, no. 6, pp. 437–452, 1994, doi:10.1016/0168-9274(94)90007-8.

[16] S. Hang: TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. ACM Transactions on Mathematical Software, vol. 42, no. 2, pp. 11:1–11:36, 2015, doi:10.1145/2629697.

[17] D. LaSalle, and G. Karypis.: Multi-Threaded Graph Partitioning. In: Proceedings of the 27th IEEE International Symposium on Parallel and Distributed Processing, 2013, pp. 225–236, doi:10.1109/IPDPS.2013.50.

[18] G. Karypis, and V. Kumar: A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing, vol. 20, no. 1, pp. 359–392, 1999, doi:10.1137/S1064827595287997.

[19] L. Gnam, S. Selberherr, and J. Weinbub: Evaluation of Serial and Parallel Shared-Memory Distance-1 Graph Coloring Algorithms. Lecture Notes in Computer Science, vol. 11189, pp. 106–114, 2019, doi: 10.1007/978-3-030-10692-8_12.

[20] Ü. Çatalyürek, J. Feo, H. Gebremedhin, M. Halappanavar, and A. Pothen: Graph Coloring Algorithms for Multi-Core and Massively Multithreaded Architectures. Parallel Computing, vol. 38, no. 10, pp. 576–597 (2012), doi:10.1016/j.parco.2012.07.001.

[21] H. Lu, M. Halappanavar, D. Chavarría-Miranda, A. Gebremedhin, A. Panyala, and A. Kalyanaraman: Algorithms for Balanced Graph Colorings with Applications in Parallel Computing. IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 5, pp. 1240–1256, 2017, doi:10.1109/TPDS.2016.2620142.

[22] E.G. Ivanov, H. Andrä, and A. Kudryavtsev: Domain Decomposition Approach for Automatic Parallel Generation of Tetrahedral Grids. Computational Methods in Applied Mathematics, vol. 6, pp. 178–193, 2006, doi:10.2478/cmam-2006-0009.

[23] M.P. Knupp, C.D. Ernst, D.C. Thompson, C.J. Stimpson, and P.P. Pebay: The Verdict Geometric Quality Library. Sandia National Laboratories, (2006), doi:10.2172/901967.

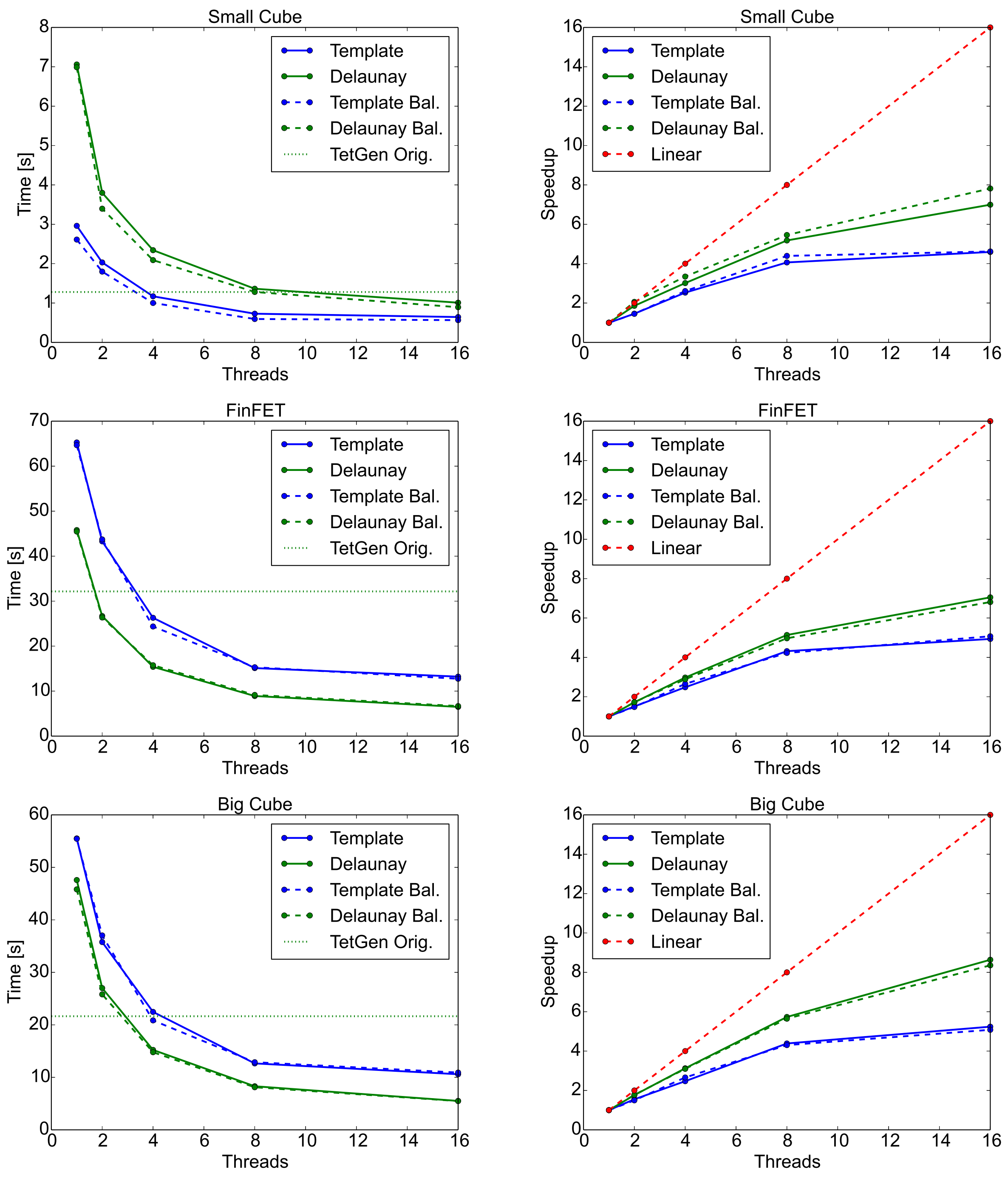

Fig. 6: Strong scaling results for the two cube geometries after two iterations using 1024 partitions and for the FinFET geometry after a single mesh adaptation iteration also using 1024 partitions. Dotted lines show the data from the balanced coloring prior to the application of the adaptation algorithms. Additionally, the left column shows the serial runtime of *TetGen* (*TetGen Orig.*).

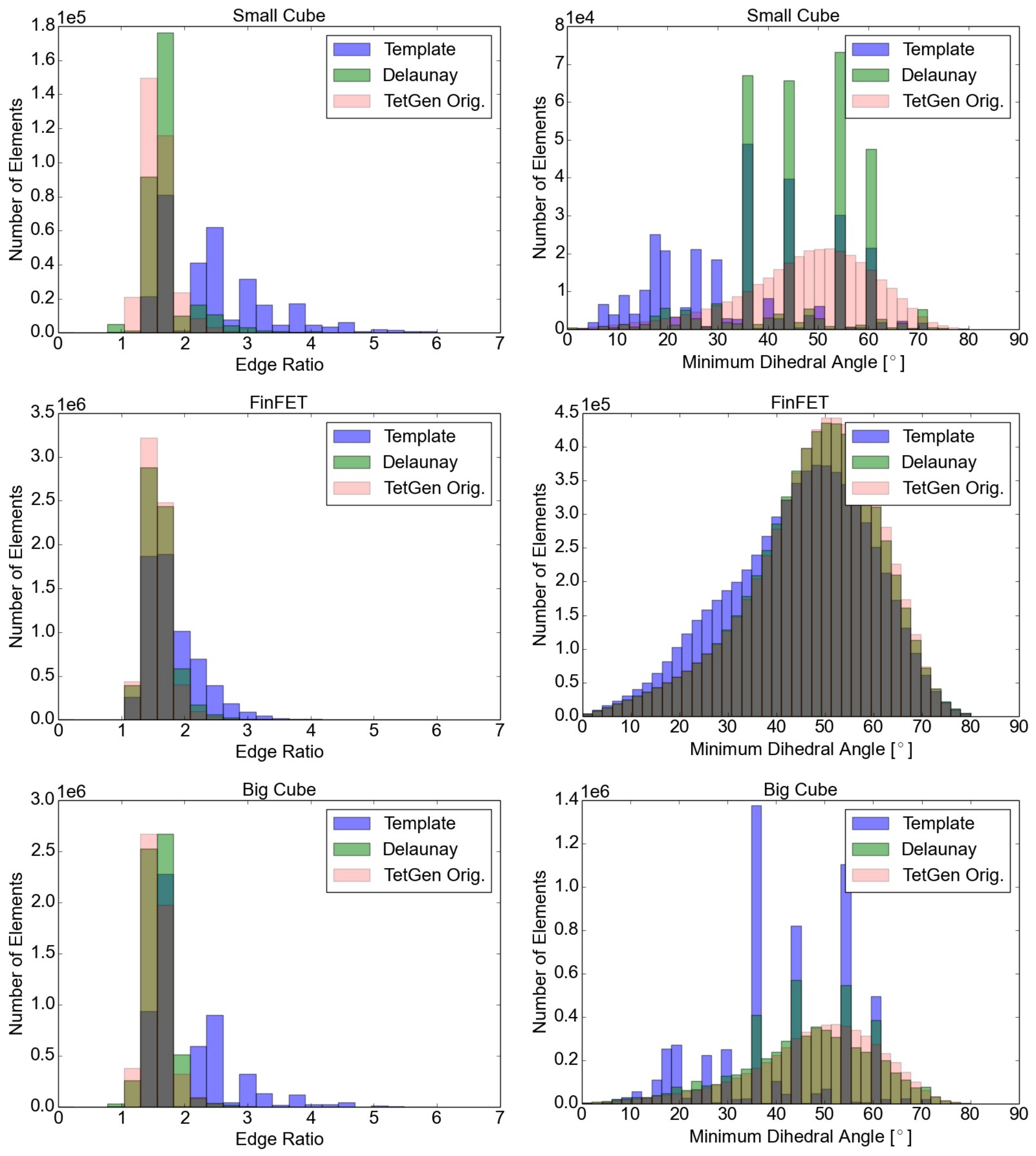

Fig. 7: Resulting mesh qualities for the *small* and *big* cube meshes after two refinement iterations using 1024 partitions and for the FinFET after one refinement iteration also using 1024 partitions. The left column depicts the edge ratios and the right column the minimum dihedral angles occurring in the tetrahedrons.