Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 00 (2017) 000–000

Procedia Engineering

www.elsevier.com/locate/procedia

26th International Meshing Roundtable, IMR26, 18-21 September 2017, Barcelona, Spain

Using Graph Partitioning and Coloring for Flexible Coarse-Grained Shared-Memory Parallel Mesh Adaptation

Lukas Gnam[a,*], Josef Weinbub[a], Karl Rupp[b], Florian Rudolf[b], Siegfried Selberherr[b]

[a]*Christian Doppler Laboratory for High Performance TCAD, Institute for Microelectronics, TU Wien, Vienna, Austria*
[b]*Institute for Microelectronics, TU Wien, Vienna, Austria*

Abstract

We present a flexible coarse-grained shared-memory parallel mesh adaptation approach based on graph partitioning and graph coloring. The strength of our approach is the ability to use existing serial meshing algorithms for the overall parallel meshing workflow. Our approach does not require a fine-grained parallel meshing algorithm and thus reduces development effort for parallelizing serial meshing algorithms and enables flexibility in devising various parallel meshing workflows. We achieve speedups of 3.4 to 4.0 for two-dimensional and 4.3 to 4.4 for three-dimensional simplex volume mesh refinement tasks with up to 16 threads and compare our approach to the parallel meshing library PRAgMaTIc.

Keywords: Parallel meshing; shared-memory; coarse-grained; mesh refinement; graph coloring;

1. Introduction

Unstructured meshes are commonly used in conjunction with finite element or finite volume methods for the numerical solution of partial differential equations. To reduce the discretization error below that of the initially generated mesh, the initial mesh is usually locally or globally adapted [1,2]. This mesh adaptation is a critical step in simulation workflows regarding both optimal solution and computational performance and often poses an unwanted bottleneck, which is especially the case in large-scale problems in science and engineering. Therefore, a lot of work has been conducted to parallelize adaptive meshing methods for distributed-memory systems for large-scale finite elements problems [3,4]. However, as more and more cores per node become available, applications based on numerical simulations on shared-memory machines such as medical analysis and treatments [5,6] need to exploit on-node parallelism. Recent developments in shared-memory parallel mesh adaptation used fine-grained parallel methods and culminated in tools like PRAgMaTIc [7] and Omega_h [8]. Contrary to those approaches, we apply a coarse-grained parallelization technique, which enables the use of available serial meshing algorithms with little overhead whilst simultaneously offering considerable parallel efficiency. This paves the way for using highly tuned serial algorithms in a parallel manner.

[*] Corresponding author.
E-mail address: gnam@iue.tuwien.ac.at

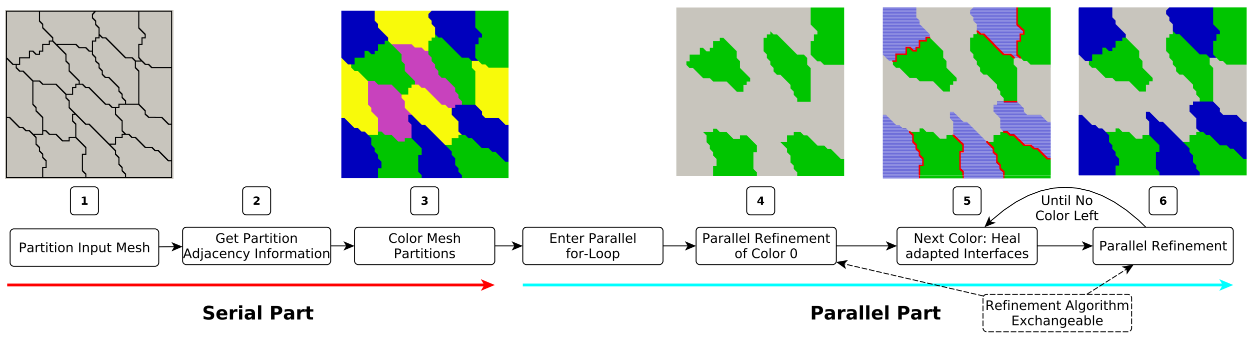
1
Partition Input Mesh
2
Get Partition Adjacency Information
3
Color Mesh Partitions
Enter Parallel for-Loop
4
Parallel Refinement of Color 0
5
Next Color: Heal adapted Interfaces
Until No Color Left
6
Parallel Refinement
Refinement Algorithm Exchangeable
Serial Part
Parallel Part

Fig. 1: Our parallel mesh refinement workflow: (1) Partitioning using METIS [12]. (2) Creation of adjacency information for the graph coloring. (3) Coloring of the partitions using a greedy coloring algorithm [13]. (4) Refinement of all partitions of color 0 in parallel. (5) Mesh healing prior to refinement of next color. (6) Refinement of all partitions of next color in parallel. The algorithm terminates once the partitions of the last color have been refined.

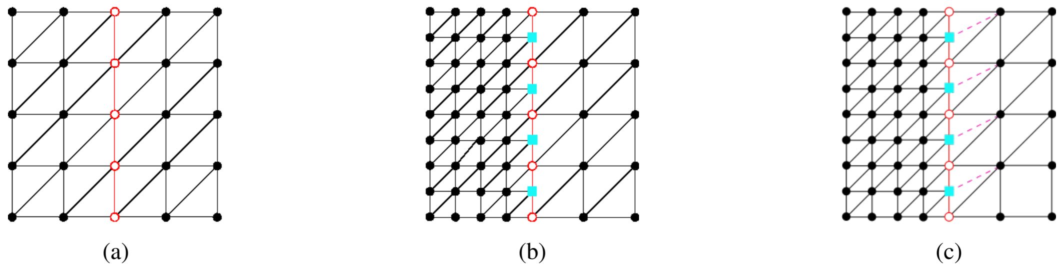

Fig. 2: (a) An example of two partitions and their interface in red (hollow vertices). (b) After the refinement of the left partition an adapted interface is created (turquoise rectangular vertices). (c) Before the right partition can be refined, its mesh has to be healed (dotted purple edges).

As a starting point in our mesh refinement pipeline we use graph partitioning and coloring methods. These are widely applied techniques to parallelize, for example, linear algebra methods for multi- and many-core architectures, but also for large-scale distributed-memory environments [9,10]. We apply these techniques to enable a coarse-grained parallel mesh adaptation workflow: Parallelism in our approach stems from the fact that the original mesh is partitioned into an independent set of submeshes on which the parallel units can operate. This is similar to methods which adapt independent sets of vertices or edges in parallel, as for example in [7].

2. Our Approach

Our parallel mesh refinement workflow is depicted in Fig. 1. This approach is suitable for arbitrary element types; here we demonstrate the practical feasibility for triangular and tetrahedral meshes. For better comparison, we consider the parallel refinement part using the same algorithm [11] as PRAgMaTIc to fathom the principle feasibility of our method.

Our pipeline starts with partitioning the input mesh using METIS [12] into contiguous mesh partitions (also called *submeshes*), as indicated in Fig. 1 (1). METIS assigns each partition a so-called partition identifier (ID), which is a unique integer. The second step in our pipeline is the creation of the adjacency information of all submeshes, as this is required as input for the subsequent coloring algorithm (Fig. 1 (2)). The last step in the serial section is the coloring itself. We use a greedy-coloring algorithm [13] to assign each partition a color (i.e., a unique, incremented integer), such that no neighboring partitions sharing an interface have the same color. In our workflow an interface consists of all edges shared by two neighboring partitions. An example interface of neighboring partitions is shown in Fig. 2a. The resulting coloring of the partitions is indicated in Fig. 1 (3), where the green partitions have color 0, the blue partitions color 1, the yellow partitions color 2, and the purple partitions color 3. Furthermore, the partitioning and coloring provide implicit load-balancing, as partitions of similar size are created.

After the coloring, we start the mesh refinement process by iterating in parallel over all partitions with color 0 (green partitions in Fig. 1 (4)), and create a dedicated mesh data structure for each of the partitions (each thread *owns* its mesh data structure). Since we have to keep a link between the vertex indices from the input mesh (*global indices*) and the vertex indices in the partition (*local indices*), we store their connection using unordered associative containers. Additionally, every partition's data structure also contains a so-called *outbox*. We use the outbox for the refinement of interfaces between two neighboring partitions sharing an interface. It is a linear data array, where only the owning partition of the active color writes into its own outbox, thus avoiding race conditions. A partition is only allowed to refine edges (or facets) on an interface, if its own color is *smaller* than the color of its neighbor, thus ensuring the termination of the algorithm. Furthermore, if a thread has to insert a vertex on an interface the partition ID of the neighbor, the global indices of the two vertices comprising the split edge, and the local index of the inserted vertex are stored in the outbox. This avoids data races and inconsistencies and prevents the elements on the interface of being refined twice, once by each submesh of the respective interface. After the parallel refinement of all partitions of color 0 is finished, the partitions of the next color are processed.

Before the refinement of the next color (marked as blue in Fig. 1 (3)) can start, each partition has to check the outboxes of its neighbors to see if vertices have already been inserted into one of its interfaces. Each partition iterates the outboxes and checks if one of the inserted partition IDs matches its own partition ID. If this is the case, the split edge in the local partition of the blue color can be found using the global indices in the outbox and the index mapping included in the mesh data structure. The coordinates of the inserted vertices are retrieved using the local index in the outbox and the partition ID to which it belongs, and the vertex is added to the local partition. To prevent non-conformities we have to conduct a so-called mesh healing step, where we use the vertices on the interface to create new elements in the partition (see Fig. 1 (5) and Fig. 2). Afterwards, the mesh refinement starts again for each partition with the active color as depicted in Fig. 1 (6). This is performed until all partitions of all colors are processed.

First investigations of our method show that the graph partitioning itself creates significant overhead, whereas the contributions of the subsequent gathering of the partitions adjacency information and the graph coloring can be neglected, because we color 4 096 partitions at most, which does not pose too much computational burden for the adjacency and coloring algorithms. Therefore, parallel graph partitioning methods will even further improve the parallel speedup with respect to Amdahl's law [14].

Additionally, due to the modular nature of our approach it is not necessary to develop fine-grained parallel algorithms for meshing in a shared-memory environment. Thus, the particular sequential meshing algorithm is exchangeable as long as the implementation provides information about changes on the mesh boundaries. This is a major difference to other parallel meshing frameworks, e.g., PRAgMaTIc, where the adaptation algorithms themselves follow a fine-grained approach, thus leading to much more specific implementations.

3. Results

A particular strength of our approach is flexibility, i.e., existing serial algorithms can be used in the parallel meshing workflow. In this section we pick the serial version of the refinement algorithm in PRAgMaTIc (version 1.3) and show that we are able to achieve a parallel performance comparable to the parallel algorithm in PRAgMaTIc. We implemented our workflow into the free open source meshing library ViennaMesh [15]. The benchmarking results were obtained on a single node of the Vienna Scientific Cluster 3 (VSC-3). A VSC-3 node consists of two Intel Xeon E5-2650v2 Ivy Bridge EP processors with 128GB of main memory. The code was compiled using Intel's C++ compiler, version 17.0.4, with optimization level -O3.

We evaluate the performance of our approach using two- (2D) and three-dimensional (3D) box geometries with uniform refinement but different resolutions for each case. In the 2D case we use a rectangular box of 4 004 001 vertices and 8 000 000 triangular cells (referred to as *2000x2000*) and a rectangular box with 9 006 001 vertices and 18 000 000 triangular cells (referred to as *3000x3000*). For the 3D tests the smaller cube mesh is comprised of 132 651 vertices and 750 000 elements (referred to as *50x50x50*), whereas the bigger cube mesh has 636 056 vertices and 3 684 750 elements (referred to as *85x85x85*). We conducted performance benchmarks using 1, 2, 4, 8 and 16 threads, with the number of mesh partitions ranging from 2 to 4 096. As speedup measure we use the ratio of the execution time in serial to the execution time with 2, 4, 8 and 16 threads. The presented results are the average execution times and speedups of three benchmark repetitions.

The results depicted in Fig. 3 were obtained for the refinement of 2D triangular meshes using 256 mesh partitions and the 3D tetrahedral meshes consisted of 1 024 mesh partitions, as those proofed to be the best configurations.

The speedup results for mesh refinement in two dimensions are better than those achieved with PRAgMaTIc for the 3000x3000 mesh, with a maximum speedup of 4.0 using our method compared to 3.1 with PRAgMaTIc. For the smaller 2000x2000 mesh we obtain parallel speedups of up to 3.4, which is the same for PRAgMaTIc. In three dimensions our speedup results are better for both test meshes, 4.3 for the 50x50x50 and 4.4 for the 85x85x85 mesh compared to 3.4 for both meshes with PRAgMaTIc. Furthermore, we compare the total execution times for the refinement, which are shown in Fig. 3 on the right. In two dimensions the smaller test mesh needs 6.4 seconds and the bigger one 13.0 seconds for refinement with 16 threads. PRAgMaTIc finishes this task in 10.6 and 22.9 seconds. As depicted for the 3D case, the execution times for the 50x50x50 test mesh are slightly better than PRAgMaTIc's, 2.0 versus 2.7 seconds. For the 85x85x85 mesh our approach takes 9.1 seconds for refinement whereas PRAgMaTIc takes 15.5 seconds.

Figure 4 shows the major contributions to the total execution time of our method. The execution times for creating the adjacency information of the partitions and the coloring process are not shown as they are negligible (cf. Section 2). The green bar shows the overhead of the steps not explicitly given in the figures. This includes the setup of the dedicated mesh data structures as well as the creation and population of the associated containers. The mesh healing process has only a small impact on the total execution time whereas the serial mesh partitioning step with METIS produces most of the overhead for higher partition numbers. In two dimensions with 32 partitions the execution time for this step is nearly as large as the refinement itself for both test meshes. For the 3D meshes a similar behavior can be observed with 128 partitions. The contribution of METIS to the total execution time increases with the partition number: At 4 096 partitions it ranges from 43 to 64% for our test meshes. This is an expected behavior, because the graph partitioning problem is NP-complete [16]. Additionally, our investigations show that the relative contribution of the overhead decreases with increasing partition numbers, but still contributes 12 to 26 % to the total execution time for 4 096 partitions, which is similar to the refinement process itself.

4. Summary

We have shown that our flexible coarse-grained parallelization method using an existing serial refinement algorithm yields speedups of 3.4 to 4.0 in two dimensions and 4.3 to 4.4 in three dimensions. These speedups are better than those of PRAgMaTIc, with exception of smaller 2D meshes, whereas in terms of refinement execution times our method outperforms PRAgMaTIc in all large and 3D cases.

Acknowledgements

The financial support by the *Austrian Federal Ministry of Science, Research and Economy* and the *National Foundation for Research, Technology and Development* is gratefully acknowledged. The computational results presented have been achieved using the Vienna Scientific Cluster (VSC).

References

[1] B.N. Davis et al., Pure App. Geophys., (2016), 4055-4074
[2] J. Southern et al., J. Comput. Sci. (2012), 8-16
[3] A. Loseille et al., Proc. IMR24 (2015)
[4] M.O. Freitas et al., Eng. Comput. 32 (2016), 655-674
[5] L. Perillo et al., Europ. J. Orthod. 37 (2015), 56-59
[6] G.R. Joldes et al., Med. Image Anal. (2009) 912-919
[7] G. Rokos et al., Facing the Multicore-Challenge III (2013), 143-144; https://github.com/meshadaptation/pragmatic
[8] D. Ibanez, et al., Proc. IMR25 (2016); https://github.com/ibaned/omega_h
[9] G. Ballard et al., ACM Trans. Parallel. Comput. 3 (2016), 18:1-18:34
[10] K. Giario et al., Discrete Appl. Math. 157 (2009), 3625-3630
[11] X. Li et al., Comput. Methods Appl. Mech. Eng. (2005), 4915-4950
[12] G. Karypis et al., SIAM J. Sci. Comput. 20 (1999), 359-392
[13] A. Gyarfas et al., J. Graph Theory 12 (1988), 217-227
[14] G.M. Amdahl, Proc. of AFIPS Conf. (1967), 483-485
[15] F. Rudolf et al., J. Comput. Appl. Math. 270 (2014), 166-177; https://github.com/viennamesh/viennamesh-dev
[16] G. Karypis et al., J. Parallel Distrib. Comput. 48 (1998), 96-129

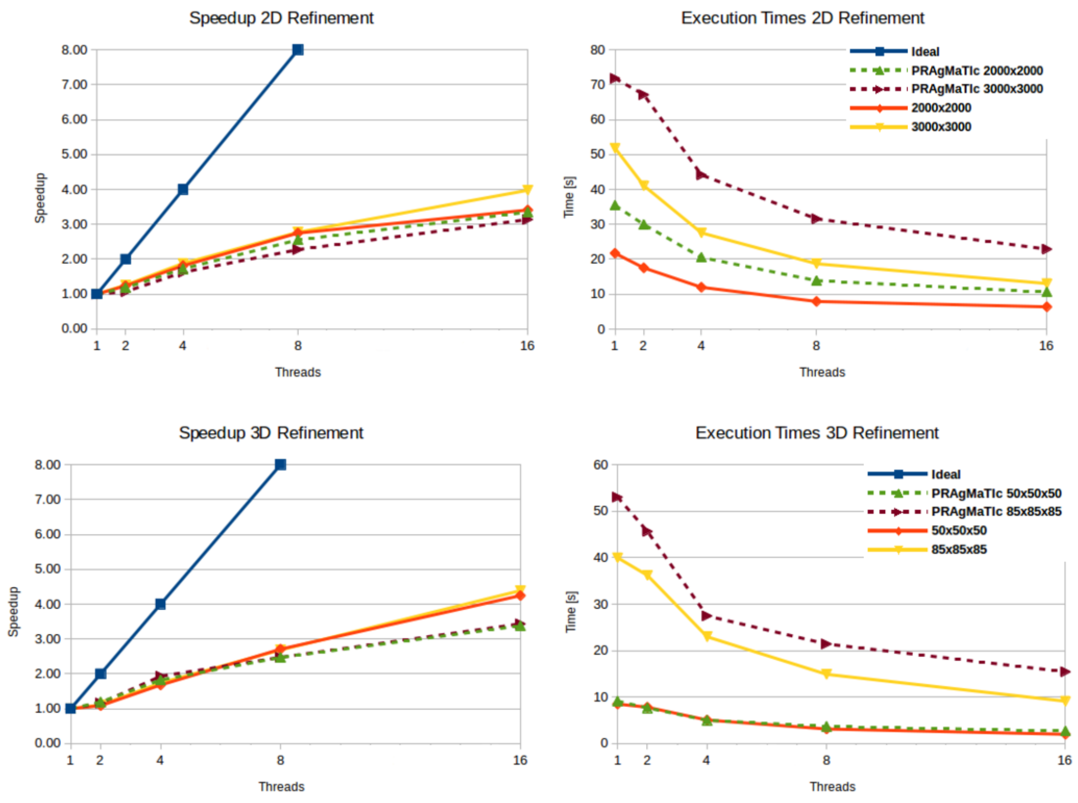

Fig. 3: Execution times of the refinement procedure. The solid lines represent our method and the dotted lines the results obtained using PRAgMaTIc. On the left the speedups and on the right the execution times for both, the 2D and 3D meshes are depicted. The results were obtained with 256 partitions in 2D and 1 024 partitions in 3D.

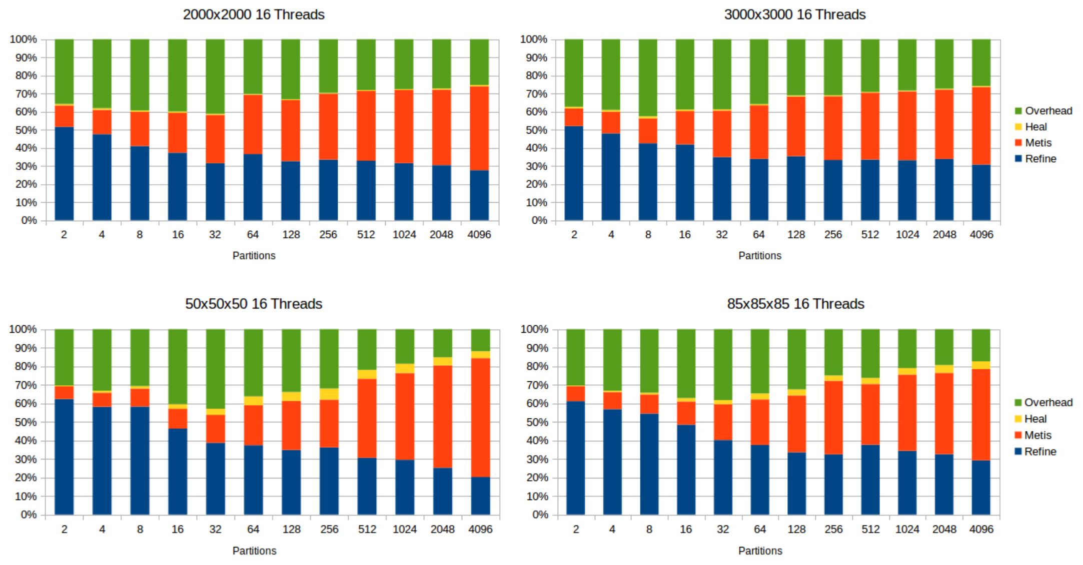

Fig. 4: Breakdown of the execution times for the 2D and the 3D meshes using 16 threads. The execution times for the creation of the adjacency information and the subsequent coloring are not shown, because their influence is negligible.