Exascale **Q**uantification of **U**ncertainties for **Te**chnology and Science Simulation

D2.2 First release of the octree mesh-generation capabilities and of the parallel mesh adaptation kernel

Document information table

Contract number:	800898
Project acronym:	ExaQUte
Project Coordinator:	CIMNE
Document Responsible Partner:	CIMNE
Deliverable Type:	Report
Dissemination Level:	Confidential
Related WP & Task:	WP 2 Tasks 2.1, 2.2
Status:	Final Version

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No **800898**

Authoring

Prepared by:				
Authors	Partner	Modified Page/Sections	Version	Comments
Alberto F. Martín	CIMNE			
Luca Cirrottola	INRIA			
Algiane Froehly	INRIA			
Contributors				

Change Log

Versions	Modified Page/Sections	Comments
V1.0	Created document, filling basic information	

Approval

Aproved by:				
	Name	Partner	Date	OK
Task leader	Alberto F. Martín	CIMNE	30.5.19	OK
WP leader	Algiane Froehly	INRIA	30.5.19	OK
Coordinator	Javier Principe	UPC	31.5.19	OK

Executive summary

This document presents a description of the octree mesh-generation capabilities and of the parallel mesh adaptation kernel. As it is discussed in Section 1.3.2 of part B of the project proposal there are two parallel research lines aimed at developing scalable adaptive mesh refinement (AMR) algorithms and implementations. The first one is based on using octree-based mesh generation and adaptation for the whole simulation in combination with unfitted finite element methods (FEMs) and the use of algebraic constraints to deal with non-conformity of spaces. On the other hand the second strategy is based on the use of an initial octree mesh that, after make it conforming through the addition of template-based tetrahedral refinements, is adapted anisotropically during the calculation.

Regarding the first strategy the following items are included:

- Description of the octree-based AMR kernel with space-filling curves.
- Description of a outer, wrapping AMR layer supporting FEM needs.
- Numerical results in a distributed environment.

Regarding the second strategy the following items are included:

- An outline of the anisotropic mesh adaptation algorithm.
- A description of the state of the art of the implementation.
- A discussion on the ongoing and future work.

Table of contents

1 Introduction **8**

2 Octree-based mesh generation and adaptivity **8**

2.1 Tree-based Adaptive Mesh Refinement and coarsening (AMR) endowed with Space-Filling Curves (SFCs) . . . 9

2.2 A general finite element (FE)-suitable parallel adaptive mesh data structure 11

2.3 Mesh handling primitives . . . 12

2.4 Numerical experiments . . . 13

2.4.1 Experimental environment . . . 13

2.4.2 partial differential equation (PDE) problem description . . . 14

2.4.3 Experiment design . . . 14

2.4.4 Numerical results . . . 15

2.5 Possible lines of extension . . . 16

3 Anisotropic mesh adaptation **17**

3.1 Outline of the algorithm . . . 17

3.2 State of art of the implementation . . . 19

3.2.1 Setting of the development framework . . . 19

3.2.2 Improvement of the Mmg library . . . 20

3.2.3 Core functions . . . 20

3.2.4 API functions . . . 20

3.3 Coupling with the *Kratos*multiphysics solver . . . 21

3.4 ongoing work . . . 21

A Software compilation and execution **22**

List of Figures

1 Single-octree mesh after 5 AMR sweeps. Left: FE solution. Right: partition of the mesh into 4 subdomains. . . . 15
2 Strong scaling test results on MN-IV. Poisson problem with $\mathcal{Q}_1(K)$ Lagrangian FEs. (a) 13 AMR levels, 49.8 Mdegrees of freedom (DOFs), up to 12.2K cores. (b) 16 AMR levels, 415.5 MDOFs, up to 30.7K cores. . . . 16
3 Mesh distribution on processors and subdivision into groups. . . . 17
4 Sequential remeshing of each group 18
5 Graph 18
6 Weighted graph 19
7 Final mesh after few steps of mesh adaptation 19
8 Instructions to compile and execute the FEMPAR's driver program that we used in order to obtain the computational results in Sect. 2.4. . . . 22

List of Tables

1 Nomenclature / Acronym list 7

Nomenclature / Acronym list

Acronym	Meaning
AMR	Adaptive mesh refinement and coarsening
SFC	Space-Filling curve
API	Application Programming Interface
OOP	Object Oriented Programming
BC	Boundary Condition
MPI	Message Pasing Interface

Table 1: Nomenclature / Acronym list

1 Introduction

This deliverable reports on the current state of mesh generation and adaptation capabilities developed during the project. A general discussion of the two strategies to be followed in the ExaQUte project was presented in Section 1.3.2 of part B of the project proposal. At this stage of the project, there has been no deviation from these strategies and the work proceeds as expected.

The first strategy is the use of octree-based mesh generation and adaptation for the whole simulation in combination with unfitted finite element methods (FEMs). The meshes produced by these algorithms are non-conforming, i.e one element may have more than one neighbor per face and the so-called hanging nodes appear. The FEM can be implemented imposing continuity weakly, using discontinuous Galerkin methods or algebraic constraints (as it is done in `FEMPAR`). By assuming this duty (imposing continuity) the solver provides an extra freedom that AMR algorithms can exploit to increase scalability. This framework is currently fully functional in `FEMPAR` and its `p4est` interface, as described in Section 2, although some improvements are still on the road-map.

The second strategy is to use anisotropic conforming meshes that permit the construction of a very accurate FE interpolation, capturing strong gradients near boundaries. An initial octree-based mesh is made conforming by subdivision of non-conforming cells and this is the starting point for the anisotropic AMR algorithms. After that, a metric field is produced by the application (*Kratos*) indicating the required refinement to the underlying AMR library (MMG). This interfacing, already developed, is currently under testing, profiling and robustification. The underlying anisotropic AMR algorithms are described in Section 3.

2 Octree-based mesh generation and adaptivity

In this section, we present a software framework, being developed as one of the ExaQUte project outcomes, which is suitable for scalable mesh generation, adaptation, and partitioning in parallel distributed-memory computing environments. In particular, we briefly overview its main design concept (Sections 2.1-2.3), report on the advances and a selected set of computational results so far (Section 2.4), and discuss possible lines of extension. This module is released as part of the `FEMPAR` scientific software project [2, 3]. In Appendix 2.4.2, we provide instructions to compile and execute, within a Dockerized computing environment, the software used to generate the computational results in Section 2.4. A full description of the adaptive mesh data structures and algorithms in the framework, and a comprehensive set of numerical experiments can be found in [5].

The main mesh data structure is designed such that: (a) it supports AMR (a.k.a. h-adaptivity) on computational domains that can be geometrically discretized as multiple connected adaptive trees—*forest-of-trees*—, thus being suitable for the efficient modeling of multi-scale problems; (b) it equips the FE discretization module with the geometrical data and the topological relationships among the mesh cells and the lower-dimension objects laying at their boundary (e.g., vertices, edges and faces in 3D) required to support the construction of *generic* conforming FE spaces of arbitrary order, thus covering a wide range of applications governed by PDEs; (c) it is highly scalable on current petascale distributed-memory supercomputers, thus suitable for large-scale simulations.

In order two achieve these goals the framework follows a two-layered meshing approach,

namely an inner, light-weight specialized meshing engine that handles a *forest-of-trees*, and an outer representation of the adaptive mesh suitable for the implementation of generic adaptive FE spaces. The ideas underlying the outer layer are general in that they apply for general cell topologies and an arbitrary number of dimensions. In Section 2.1, we briefly overview the inner layer, and in Sect. 2.2, the outer.

2.1 Tree-based AMR endowed with SFCs

Tree-based AMR with SFCs was originally proposed in [9] for the particular case of octrees, and extended to general trees in [12]. AMR with SFCs can be seen as a two-level decomposition of the computational domain Ω, referred to as macro and micro level, resp. In the macro level, we assume that there is a partition $\mathcal{C}_h$ of Ω into cells $K \in \mathcal{C}$ such that each of these cells can be expressed as a homeomorphism Φ_K over a set of admissible reference polytopes [3]. The mesh $\mathcal{C}_h$, referred to as the coarse mesh, it is assumed to be a *conforming mesh*, and can be generated from an standard (sequential) unstructured mesh generator. In the micro level, each of the cells of $\mathcal{C}_h$ becomes the root of an adaptive tree. This two-tier adaptive structure is referred to as *forest-of-trees*.

A particular adaptive mesh $\mathcal{T}_h$ of Ω to be used for FE discretization is defined as the union of all leaf cells (i.e., cells with no children) in the forest. Forest-of-trees can be exploited to consider more complex physical domains than single-tree approaches, i.e., conforming coarse meshes $\mathcal{C}_h$ with multiple cells. In order for forest-of-trees meshing to be efficient, scalable, and light-weight, the number of cells in the coarse unstructured mesh must be *small*. In any case, for PDEs posed on *complex* domains, i.e., problems for which it is not possible to fulfill this last requirement while still resolving the geometry/topology of the domain with a *small* coarse mesh, one can use a single-tree mesh (i.e., a coarse mesh composed of a single cell, such as a quadrilateral, or hexaedron) as a background mesh to be used in combination with unfitted (a.k.a. immersed or embedded) FE discretization methods (as, e.g., the finite cell [17], cutFEM [7], or agFEM methods [4]). In such a case, the framework presented in this report can be used to control geometry approximation errors by adaptation in regions of high geometric variability. This is indeed the approach pursued in ExaQUte for the description of complex geometries.

The process that generates a forest-of-trees is essentially characterized by defining two complementary ingredients, namely a *refinement rule* and an *SFC index*.

Refinement rule. A *refinement rule* prescribes how one may subdivide a cell K into finer cells that span the same region as K. K is referred to as the *parent* cell, and the latter ones, (its) *children* cells. The refinement rule is usually defined at the reference polytope and then mapped to the physical space using Φ_K. The reverse application of the refinement rule, i.e., the replacement of the children cells by its parent, is referred to as coarsening. Mesh generation in this context is a hierarchical process based on the recursive application of refinement and coarsening. At each level in the hierarchy, some cells are marked for refinement, and some other for coarsening. A cell marked for refinement is replaced by its children cells following the refinement rule. On the other hand, if all children cells of a given parent are marked for coarsening, they are collapsed into the parent cell. As a result of this process for all cells in $\mathcal{C}_h$, we obtain a *forest* of tree-like refinement structures rooted at every $K \in \mathcal{C}_h$.

SFC index. The SFC index is formally defined as a map among the set composed of all *constructable cells*, i.e., the cells that can be potentially constructed by means of the recursive application of the refinement rule to the forest roots up to a prescribed maximum level of refinement, and the set of natural numbers. A key motivation when choosing an SFC index is that mesh queries that locally operate on a cell take *constant time*, independently of the level of refinement of the cell. Local cell functions include computing the SFC index of a cell, determining the SFC index of its parent, children, and neighbors with the same refinement level, computing vertices coordinates, etc. Besides, the exploitation of the SFC index has to result in significant memory savings with respect to unstructured meshing, in which a list of neighbors has to be stored, as well as the coordinates of all vertices in the mesh, for each cell. On the other hand, the SFC index must be chosen such that the refinement rule preserves the ordering induced by the SFC index, i.e., the new cells generated by means of refinement have an SFC index inbetween the ones of those already existing in the forest. Therefore, the leaves of the forest at hand can be *uniquely arranged* in a linear array *in ascending order by their SFC index*, so that preserving this order when replacing a parent cell by its children or vice-versa becomes easy to implement (and efficient). Apart from providing an efficient storage layout, SFCs also provide a linear runtime solution to the partitioning problem of the geometrical domain among processors. In particular, a cell-based partition of $\mathcal{T}_h$ is simply generated by dividing the leaves in the linear ordering into as many equally-sized segments as processors involved in the computation. This in turn circumvents the high computational cost and lack of parallel scaling proneness typically associated with dynamic load balancing via unstructured graph partitioning algorithms [14].[1]

Non-conformity and 2:1 balanced forest-of-trees Tree-based meshes provide multi-resolution capability by local adaptation, i.e., neighboring cells in $K \in \mathcal{T}_h$ might be located at different refinement levels. However, these meshes are (potentially) *non-conforming*, i.e., they contain the so-called *hanging Vertices, Edges, and Faces (VEFs)*. These occur at the interface of neighboring cells with different refinement levels. For FE applications, mesh non-conformity hardens the construction of conforming FE spaces, and the subsequent steps in the simulation [5]. However, common practice in order to significantly alleviate this extra complexity consists on enforcing the so-called 2:1 balance constraint (a.k.a. balance or mesh regularity condition). In a nutshell, geometrically neighboring cells may differ at most by a single level of refinement. In particular, one enjoys the following two cornerstone benefits: (a) *only* single-level (a.k.a. direct) hanging node constraints are required when constructing global conforming FE spaces; (b) in a distributed-memory context, all such constraints can be resolved locally with a single layer of ghost cells [5].

Forest-of-trees handler operations In practice, parallel tree-based AMR using SFCs is a specialized feature that applications typically outsource to an external adaptive meshing engine. In the interface among these, a set of core application-level operations have been identified as cornerstone in order to fully realize this functionality in numerical applications. These are briefly outlined in the sequel. (a) `New`: creates a new uniformly

[1] Dynamic load balancing is the ability of an adaptive mesh to be re-distributed in the presence of an unacceptable amount of load imbalance, e.g., the one generated by means of AMR in a highly localized region of Ω.

refined forest, up to a user-provided level, from a data structure describing the connectivity of mesh cells in $\mathcal{C}_h$; (b) `Adapt`: refines and coarsen the current forest accordingly to a user-provided criterion; (c) `Partition`: redistributes the forest leaves among processors for dynamic load balancing; (d) `Balance`: ensures the 2:1 balance condition among neighboring cells by local refinement where required; (e) `Ghost`: creates a data structure that contains layer of off-processor leaf cells that are neighbors of cells in the current processor; (f) `Iterate`: given a 2:1 balanced forest, provides a mechanism to iterate over its leaf cells, and *a subset of* the inter-cell interfaces, while letting applications get local neighborhood information of each interface visited, including both conforming and non-conforming cell interfaces.

State of the art software packages There are essentially two software packages around that are grounded on the tree-based AMR with SFCs approach. As an evidence of the potential of this approach, both libraries have been shown to efficiently scale up to hundreds of thousands of computational cores. On the one hand, `p4est` [9, 13] provides parallel forest-of-octrees (thus restricted to either quadrilaterals or hexahedra) grounded on the standard $1{:}2^d$ isotropic refinement rule and the Morton SFC index [9]. On the other hand, `t8code` [12] is an ambitious, on-going software effort, which provides abstract tree-based AMR with SFCs algorithms for the operations outlined above. By means of implementing a set of low-level local-to-cell operations, these algorithms may be extended to work with general polytopes (e.g., lines, simplices, bricks, prisms, pyramids, etc.). For triangles and tetrahedra, it is grounded on Bey's red-refinement rule, and the recently proposed Tetrahedral Morton (TM) SFC index [8]. In its current status, `t8code` only provides facet-oriented variants of the main operations above (e.g., `Balance` is restricted to $d-1$-balance), and thus cannot be readily used for the implementation of generic FEs. For this reason, in this report we restrict to `p4est` as a practical demonstrator.

2.2 A general FE-suitable parallel adaptive mesh data structure

As mentioned above, our approach for scalable mesh handling follows a two-layered meshing approach, namely an inner light-weight layer encoding the forest-of-trees, already overviewed in Sect. 2.1, and an outer rich FE-suitable mesh representation, that we briefly overview in this section. In particular, we have developed an specification for a distributed adaptive mesh representation that supports generic FE spaces built atop. The sort of data it needs to handle, and how it is internally laid out, has resulted from our experience in accommodating the requirements of a wide range of state-of-the-art FE discretizations within a single framework [3]. The concepts underlying this specification are not tailored to a particular tree-based AMR with SFCs technology or cell topology.

The adaptive mesh $\mathcal{T}_h$ is distributed among processors such that each processor owns a set of local cells, and a layer of off-processor cells which are in touch with the local cells of the processor. Using this overlapped mesh partition information, and a description of neighboring relationships between cells in the adapted mesh (apart from hierarchical relationships) provided by the inner layer, the outer layer mesh data structure is able to reconstruct a suitable-for-mesh-non-conformity set of topological and neighboring relationships among the mesh cells and the lower-dimension objects laying at their boundary. It also identifies the mesh objects into those which which are interior to the local portion of the processor, lay at the processor boundaries with other processors, or at the exterior,

while gluing together the mesh objects which lay at processor boundaries by means of a distributed algorithm which generates a global labeling of the mesh objects across the whole domain. It also classifies these objects into regular and hanging, is able to describe the cells in the vicinity of objects, including the coarser cells and objects on which these objects are in turn hanging in the case of non-conforming cell interfaces, thus providing the required mesh information to a FE discretization module for the set up of hanging DOF constraints. We refer to [5, Sect. 3] for a detailed description of the outer layer mesh data structure in our framework.

2.3 Mesh handling primitives

Users of our FE framework are provided with a distributed mesh data structure that hides the high complexity underlying the specialized tree-based AMR engine. To this end, it is equipped with three high-level mesh handling primitives:

- `Create`. Creates a new, yet unadapted forest-of-trees mesh provided a coarse mesh $\mathcal{C}_h$. The coarse mesh can be either described manually by the user for simple domains, or generated by an unstructured mesh generator and imported into the framework.

- `Refine_and_coarsen`. Adapts the forest-of-trees mesh based on cell flags set by the user, while transferring data that the user might have attached to the mesh objects (i.e., cells and VEFs) to the new mesh objects generated after mesh adaptation. The mesh data structure provides a set of mechanisms that, prior to the execution of this primitive, lets them walk through over the mesh cells and associate a per-cell flag that tells the framework whether a cell is to be refined, coarsened, or kept as it is.

- `Redistribute`. Dynamically balances the computational load by redistributing the forest-of-trees mesh among processors involved in the parallel simulation. The default criteria is to balance the number of cells in each processor. Alternatively, the user might associate to each cell a partition weight. In this case, the primitive balances the sums of the cell partition weights among processors. The data that the user might have attached to the mesh objects (i.e., cells and VEFs) is also migrated among processors. This latter operation only involves non-blocking point-to-point message exchanges among processors (see discussion below).

The implementation of these mesh handling primitives is grounded on the following tree-based AMR operations; see Sect. 2.1. `Create` invokes `New` and `Ghost`, `Refine_and_coarsen` invokes `Adapt`, `Balance`, and `Ghost`, and `Redistribute` invokes `Adapt`, `Partition`, and `Ghost`. We note that these three mesh handling primitives also have to: (a) retrieve from the inner layer a description of neighboring relationships between cells in the adapted mesh; (b) call the algorithm which reconstructs the topological and neighboring relationships among the mesh cells and the lower-dimension objects that our FE-suitable adaptive mesh representation keeps track off; see Sect. 2.2. The process underlying (a) is implemented by means of invocations to `Iterate` (Sect. 2.1).

As mentioned above, the `Redistribute` mesh handling primitive is in charge of migrating the data that the user might have attached to the mesh objects (i.e., cells and VEFs). The underlying tree-based AMR engine may have a built-in mechanism that lets

users to attach, cell-wise, application-related data to the forest leaves, and migrate them *in one-shot* during the call to `partition`. This is indeed the approach followed by [9] atop `p4est` in order to migrate, not only mesh-related data, but that within any other data structure built on top of the mesh (such as, e.g., the nodal values of FE functions involved in the simulation). However, in our framework, we preferred to implement the communication pattern among processors required to carry out the data migration. This full control over data migration provides greater flexibility as we do not require the data to be migrated (e.g., FE function nodal values, or application-related cell-wise data arrays) to be fully available whenever the user calls to `Redistribute`.

The execution of the communication pattern required for data migration is easily implemented in terms of non-blocking point-to-point MPI communication primitives. The determination of such pattern is more involved. To this end, `Redistribute` performs a (temporary) copy of the forest-of-trees data structure right before calling `partition`. This copy consumes only a (temporary) small amount of extra memory due to the low memory footprint of tree-based AMR with SFCs. We stress that neither the full forest-of-trees mesh connectivity, nor the data structure storing ghost cells has to be copied. Then, `Redistribute` invokes `partition` on one of these two copies, so that it ends up with the same forest-of-trees, but (potentially) distributed differently among the processors involved. The algorithm that builds the communication pattern in turn relies on an algorithm that takes as input two instances of the forest-of-trees data structure. Let us refer to these as the "first" and "second" forest-of-trees, although we stress that they correspond to the same forest, but distributed differently among processors. On each processor p, the algorithm performs two passes over the leaf cells locally owned by p in the first forest. In a first pass, the algorithm determines, for each of these leaves, whether the leaf is still locally owned by p in the second forest, or by a remote processor q. This can be determined, for each leaf, in $\mathcal{O}(\log_2(P))$ operations [9]. A second pass lets processor p know the number and list of remote processors q that now own leaves in their local portion of the second forest, and how many leaves per processor. It can be easily seen that the send side of the communication pattern is determined if we call the algorithm with the first and second forest being the forest before and after calling `partition`, resp., while the receive side, the other way round.

2.4 Numerical experiments

In this section we leverage the mesh generation, adaptation, and partitioning capabilities developed so far for the numerical FE solution of a benchmark PDE problem. We will in particular focus on the parallel strong scalability of the mesh-handling related algorithms, although the one of the other stages in the FE simulation pipeline are reported as well. The data structures and algorithms for numerical integration, assembly and preconditioned iterative solution of the discrete system of linear equations resulting from FE discretization are part of `FEMPAR` as well, and rely on the services provided by the FE-suitable adaptive mesh representation presented in this report.

2.4.1 Experimental environment

The numerical experiments are run at the Marenostrum-IV (MN-IV) supercomputer, hosted by BSC. This petascale machine is equipped with 3,456 compute nodes interconnected together with the Intel OPA HPC network. Each node has 2x Intel Xeon Platinum

8160 multi-core CPUs, with 24 cores each (i.e. 48 cores per node) and 96 GBytes of RAM. With respect to the software, we used `FEMPAR`, linked against `p4est` v2.2 as its forest-of-octrees manipulation engine. These software were compiled with Intel v18.0.1 compilers using system recommended optimization flags and linked against the Intel Message Passing Interface (MPI) Library (v2018.1.163) for message-passing and the BLAS/LAPACK and PARDISO available on the Intel MKL library for optimized dense linear algebra kernels and sparse direct solvers, resp. All floating-point operations were performed in IEEE double precision.

2.4.2 PDE problem description

We consider the solution of a Poisson problem a 3D cube domain $\Omega = [0,1]^3$ with homogeneous Dirichlet boundary conditions (strongly) imposed over the entire domain boundary. The problem is discretized using a global Lagrangian FE space $\mathcal{V}_h \subset H_0^1(\Omega)$. The weak formulation of this problem reads: find $u_h \in \mathcal{V}_h$ such that

$$(\boldsymbol{\nabla} u_h, \boldsymbol{\nabla} v_h) = (f, v_h), \qquad \forall v_h \in \mathcal{V}_h. \tag{1}$$

The right-hand-size $f(x,y,z)$ is a piece-wise function defined as:

$$f(x,y,z) = \begin{cases} 1 & z > \frac{1}{2} + \frac{1}{4}\sin(4\pi x)\sin(4\pi y) \\ -1 & z \leq \frac{1}{2} + \frac{1}{4}\sin(4\pi x)\sin(4\pi y) \end{cases}.$$

The discontinuity in f leads to a solution that is non-smooth along a sinusoidal surface through the domain, so that very localized AMR is required in order to reduce the error in that area while keeping the computational requirements reasonably low.

2.4.3 Experiment design

We start with a coarse mesh that is obtained after uniformly refining the root octant of the octree up to four times. This mesh has 16 cells (hexahedra) per coordinate direction, and 4096 cells in total. Then, the Poisson problem is solved on a hierarchy of meshes where the mesh at a given level is obtained from the one at the previous one by means of the `Refine_and_coarsen` mesh handling primitive presented in Sect. 2.3. With load balancing in mind, the mesh is dynamically redistributed at each level by means of the `Partition` primitive right after each call to `Refine_and_coarsen`. For a given (fixed) number of AMR levels, we measure the elapsed time (i.e., wall clock time) spent in each of the stages in the simulation of process, *aggregated across all levels*, and evaluate at which rate they are reduced with increasing number of CPU cores (i.e., strong scalability test). The number of AMR levels is adjusted such that a "sufficiently large" problem size at the last level is obtained for the CPU core range on which we run the strong scaling test.

The decision of which cells to be refined, coarsened, or untouched (previous step required for the `Refine_and_coarsen` primitive) is performed as follows. Given the solution of the linear system, each processor computes independently of each other an error indicator for its local cells using the a-posteriori error estimator in [15]. Then, given user-defined refinement and coarsening fractions, denoted by α_r and α_c, resp., we find the thresholds θ_r and θ_c such that the *total* number of cells with error indicator larger (resp., smaller)

than θ_r (resp., θ_c) is (approximately) a fraction α_r (resp., α_c) of the *total* number of cells. Besides, we set $\alpha_r = 0.15$ and $\alpha_c = 0.03$, so that, the number of cells is at least doubled at each AMR level (assuming that no cells can be coarsened). The actual number of cells at each mesh in the hierarchy, however, also depends on the algorithm within `p4est` that 2:1 balances the forest-of-octrees, as it may need to apply more refinement in order to keep this constraint. Fig. 1 shows the adapted octree mesh after the application of 5 AMR sweeps, the computed FE solution for a 2D variant of Prob. (1), an the partition of the mesh into 4 subdomains.

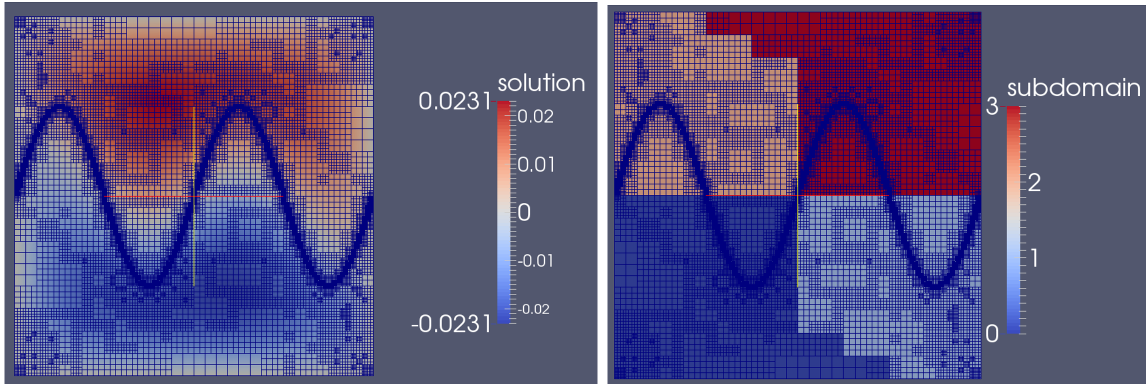

Figure 1: Single-octree mesh after 5 AMR sweeps. Left: FE solution. Right: partition of the mesh into 4 subdomains.

2.4.4 Numerical results

Fig. 2 shows the strong scalability of the algorithms and data structures subject to study for problem (1) discretized with trilinear ($\mathcal{Q}_1(K)$) Lagrangian FEs. In particular, the curve labeled as Mesh includes the calls to the `Refine_and_coarsen` and `Partition` primitives; see Sect. 2.3. For readability purposes, Fig. 2 also provides the ideal strong scaling slope (solid black line). The more parallel a given strong scaling curve is to the ideal slope the more strongly scalable the corresponding stage is. Two different global problem sizes suitable for the $[48, 12228]$ and $[384, 30672]$ CPU cores range were tested. In particular, the results in Fig. 2 (a) correspond to a problem in which the AMR hierarchy has 13 levels, resulting into a 49.8 MDOFs problem size at the last level, while in Fig. 2 (b), we report the ones for a 16 AMR levels hierarchy, and a 415.5 MDOFs problem size.[2] *These DOFs counts, and the ones in the rest of the paper, include both regular and hanging DOFs.* The averaged load per core at the last level ranges, in Fig. 2 (a), from 1.04M DOFs/core to 4.06K DOFs/core, and, in Fig. 2 (b), from 1.08M DOFs/core to 13.53K DOFs/core.

Overall, the computational time reduction with the number of cores is very significant for the Mesh stage. The parallel efficiency only decays to 16% and 26% for the largest number of cores tested in Fig. 2 (a) and Fig. 2 (b), respectively, despite the Mesh

[2] We stress that this is by no means a parallel scaling limit of the algorithms proposed, but the largest number of cores we can exploit on MN-IV provided the access constraints that we have to this supercomputer. We expect the algorithms proposed to be able to scale up to hundreds of thousands of cores in the solution of hundred of billions DOFs problem sizes.

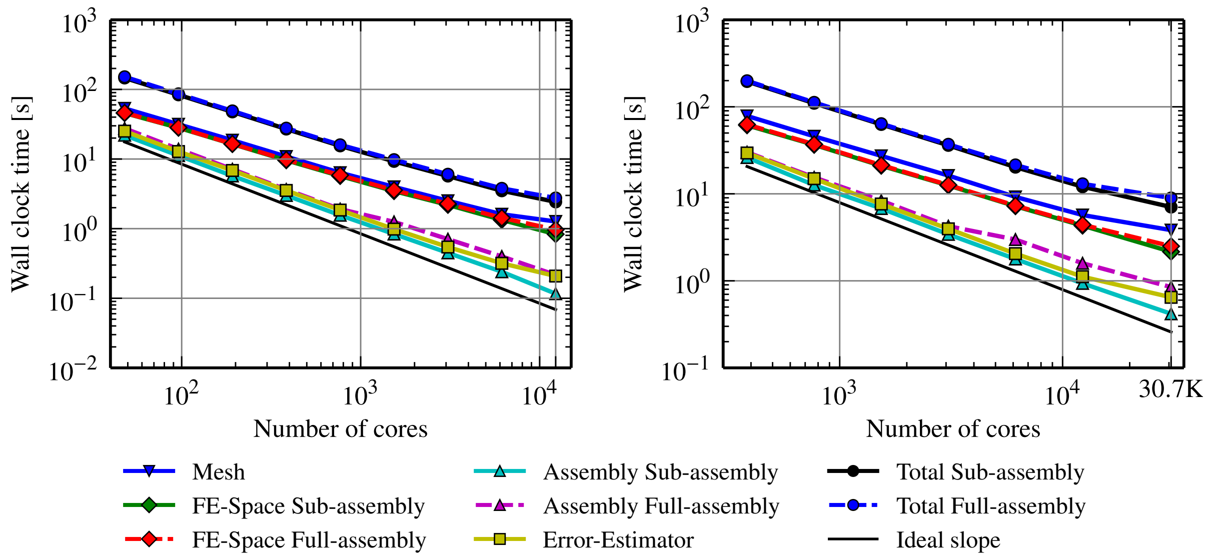

Figure 2: Strong scaling test results on MN-IV. Poisson problem with $\mathcal{Q}_1(K)$ Lagrangian FEs. (a) 13 AMR levels, 49.8 MDOFs, up to 12.2K cores. (b) 16 AMR levels, 415.5 MDOFs, up to 30.7K cores.

stage involves significantly more communication volume (relative to local work) than, e.g., Assembly Sub-Assembly.

2.5 Possible lines of extension

With the octree-based adaptive meshing capabilities in its current status we are able to cover, among others, the following scenarios: (1) PDEs in which the area that requires high-resolution is known *a priori*, the presented adaptive mesh data structure can be adapted before starting the actual numerical simulation; PDEs in which solution features evolve spatially over time, and that are known only during the actual numerical simulation, it can be adapted dynamically, driven by a posteriori error estimates [1] during the actual simulation; PDEs posed on *complex* domains, and in combination with embedded FE discretization methods, as, e.g., the finite cell [17], cutFEM [7], or agFEM methods [4], they can be used to control geometry approximation errors by adaptation in regions of high geometric variability. While the first two families of embedded FE solvers can be implemented using the current services of our adaptive mesh data structure, the latter one will require additional services that we may consider as future lines of extension. AgFEM is relevant to ExaQUte as it has been proven to be a very promising approach for addressing the ill-conditioning issues associated to badly cut cells. In particular, given which cells of the adaptive mesh are exterior, cut, and interior to the embedded computational domain, the mesh data structure should be able to: (a) build the cell aggregates of agFEM methods [4]; (b) extend the notion of ghost cells to also consider root cells of cell aggregates which are owned by remote processors. Such cells are required in order to be able to resolve locally on each processor the agFEM methods DOF constraints [4].

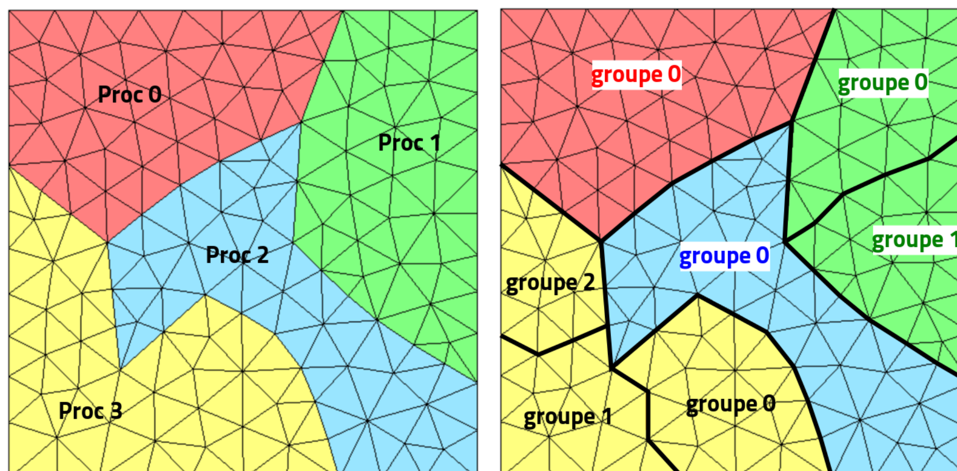

Figure 3: Mesh distribution on processors and subdivision into groups.

3 Anisotropic mesh adaptation

This section first provides the method of parallelization, then, it describes the state of art of the first release of ParMmg, including the API functions available to the project partners. It concludes with the ongoing work for the next months.

3.1 Outline of the algorithm

The user mesh (centralized on one processor or distributed on several ones) is divided into sub-meshes (here called *groups*). Groups are equidistributed on different processors, groups interfaces are fixed and each group is remeshed through a call to the sequential remesher Mmg [11]. In order to adapt on previously frozen regions, groups are modified so that old interfaces are moved inside the domain for the next mesh adaptation iteration. The aim of the algorithm is to converge on the full adapted mesh (where the effects of the previously freezed interfaces is not anymore appreciable). Similar algorithms have been successfully used by [6], [16], [10] (among others).

In particular, our algorithm is build by the following steps:

1. Root process distribute the mesh on all the available processes (optional, only for centralized initial meshes):
   (a) Dual graph construction (1 mesh element = 1 graph node);
   (b) Call to the graph partitionner (Metis) asking *nproc* colours (nproc being the number of processes). Metis colours the graph nodes;
   (c) Mesh subdivision into groups;
   (d) Group communication to each target process;
2. While the stopping criterion is not reached, on each processor:
   (a) Group subdivision into subgroups (3);
   (b) Sequential remeshing of each group, the interfaces between groups being freezed (4);

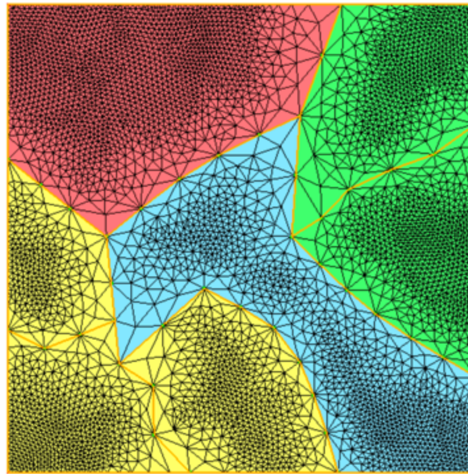

Figure 4: Sequential remeshing of each group .

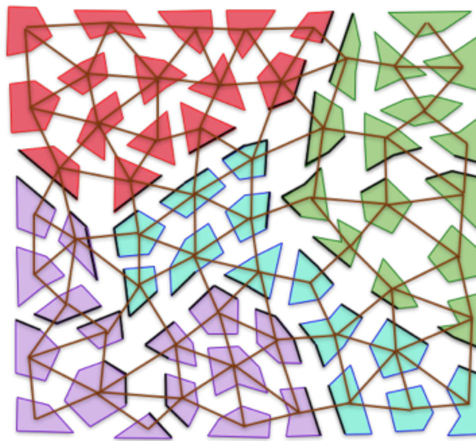

Figure 5: Graph .

(c) Input size map interpolation (because this size map is modified by Mmg);

(d) Load balancing:

i. New subdivision into smaller groups (smaller granularity for the graph partitionner);

ii. Metis graph construction with groups as graph nodes (5);

iii. Adding of weights to the nodes / edges of the graph to ensure the displacement of the previously freezed interfaces inside the new groups (6);

iv. Graph gathering on root process;

v. New group coloration using Metis;

vi. Check of the colour contiguity and correction of the colours if needed (to ensure the remesher robustness);

vii. Merging of groups of same colours;

viii. Group communication to the target process.

(e) Go back to step 2;

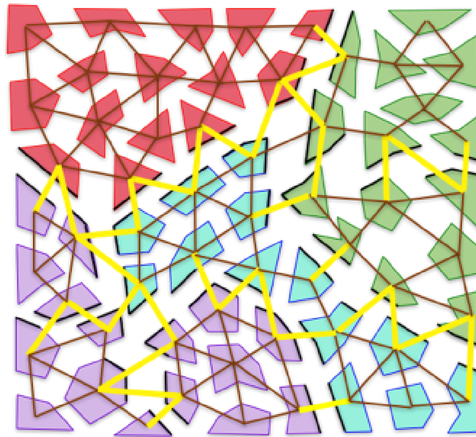

Figure 6: Weighted graph .

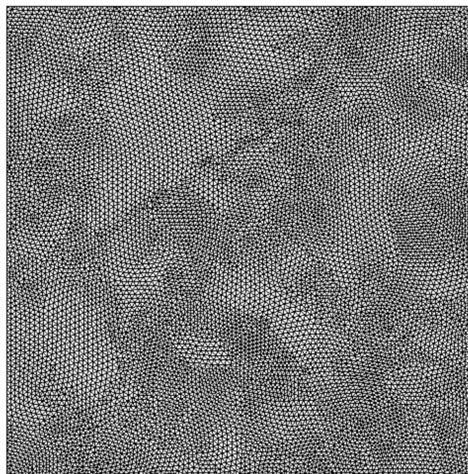

Figure 7: Final mesh after few steps of mesh adaptation .

3. Groups packing (deletion of empty positions in the meshes);
4. Group merging over each process;
5. Centralization if asked by the user:
   (a) Groups gathering over the root process;
   (b) Groups merging and packing (7).

3.2 State of art of the implementation

3.2.1 Setting of the development framework

At the beginning of the project, we give a particular attention to the setting of a framework favorable to collaborative development:

- Continuous integration with CTest runned by Jenkins and on cloudstack slaves;
- Automatic compilation using CMake;

- Merge Request git workflow;

3.2.2 Improvement of the Mmg library

Modification of the management of the mesh adaptation near freezed boundaries to avoid the mesh degenerescence (leading to very poor qualities and long CPU times):

- Computation of the size to impose at required nodes as a mean of the length of required edges passing through the node;
- Size map smoothing from required entities toward non required ones;

3.2.3 Core functions

All the needed modules have been implemented in C99 and all the parallel mesh adaptation kernel functions benefits from the updates available in the last release of the Mmg remesher.

3.2.4 API functions

The implemented API functions aim to provide the project partner with the tools to couple their computational mechanics solvers with the parallel mesh adaptation library. The requirements for these API functions can be found in the Deliverable 2.1 As such, the API functions fall into the three main categories described in the following paragraphs. Basic tutorials are available with the source code, and a complete documentation is underway on the gitlab project webpage.

Functions to initialize and recover a sequential or distributed mesh
These functions closely match the analogous API functions available in the Mmg remeshing library to initialize pointers to Mmg data structures and to set/get mesh entities (nodes, elements, boundary triangles) one-by-one or by arrays. The user is required to set mesh elements (tetrahedra) and nodes, together with boundary triangles on each local mesh.

Functions to set and get the interface entities of a distributed mesh
In order to be able to work both with node-centered and element-centered computational mechanics solvers, two separate sets of API functions are available to initialize either interface faces or nodes (if both are provided, nodes information are discarded). The user is asked to provide an array with the indices of interface faces/nodes in the local input mesh, together with an array with the global indices of the same entities (provided in the same order). This information is internally used to reconstruct the communication graph among processes.

Functions to check the correctness of the set interface entities against input data are also provided.

Main parallel mesh adaptation function
Two library functions are provided in order to run the parallel mesh algorithm starting from a sequential or a distributed mesh.

3.3 Coupling with the *Kratos*multiphysics solver

The parallel remeshing library, with the previously described API functions, has been made available to the project partner CIMNE, where the *Kratos*developers team is taking care of the coupling with the *Kratos*multiphysics solver.

3.4 ongoing work

The short term roadmap is:

- profiling and speedup;
- robustification.

A Software compilation and execution

In order to illustrate the usage of the octree-based meshing capabilities implemented in this project task, and to obtain the computational results in Sect. 2.4, we developed a FEMPAR driver program. This driver program is distributed as part of the official Git repository of the FEMPAR software project[3]. In Fig. 8 we list the set of Linux shell commands to be used in order to compile and execute the driver program within a Dockerized computing environment.

```
1  # You must Sing up at https://hub.docker.com/ to get your_username_at_dockerhub
2  $ docker login --username=your_username_at_dockerhub
3  # Get Docker image from Docker Hub
4  $ docker pull fempar/fempar-env:gnu_debug_p4est-parallel
5  $ docker run -ti fempar/fempar-env:gnu_debug_p4est-parallel
6
7  # ****** COMPILATION INSTRUCTIONS ******
8  $ WORKDIR=/data
9  $ SOURCES_DIR=$WORKDIR/sources
10 $ FEMPAR_DIR=$WORKDIR/FEMPAR
11 $ git clone --branch preparing_fempar_dealii_comparison_experiment \
12     --recursive https://gitlab.com/fempar/fempar $SOURCES_DIR
13 $ cd $WORKDIR
14 # invokes CMake while setting up appropriate values for CMake variables
15 # run $SOURCES_DIR/Tools/configure -h to get an informative message on screen
16 $ $SOURCES_DIR/Tools/configure -s $SOURCES_DIR/SuperBuild -c GNU --with-tests
17 $ make
18
19 # ****** EXECUTION INSTRUCTIONS ******
20 $ cd $WORKDIR/FEMPAR/bin
21 # get informative message on screen
22 $ mpirun -np 1 par_test_h_adaptive_poisson -h
23 # run the program with:
24 #  x 5 MPI tasks (-np 5)
25 #  x 2D Poisson problem (--dim 2)
26 #  x Biquadratic Lagrangian FEs (-order 2)
27 #  x 7 AMR sweeps (-num_refs 7)
28 #  x Generate simulation data files for visualization
29 #    using e.g. ParaView (-wsolution .true.)
30 $ mpirun --oversubscribe -np 5 par_test_h_adaptive_poisson  -l 2 \
31     --dim 2 -order 2 -num_refs 7 -wsolution .true.
```

Figure 8: Instructions to compile and execute the FEMPAR's driver program that we used in order to obtain the computational results in Sect. 2.4.

References

[1] M. Ainsworth, J. T. J. T. Oden, and Wiley InterScience (Online service). *A posteriori error estimation in finite element analysis*. Wiley, 2000. ISBN 9781118032824.

[2] S. Badia, A. Martín, and J. Principe. FEMPAR Web page. http://www.fempar.org, 2018.

[3] S. Badia, A. F. Martín, and J. Principe. FEMPAR: An Object-Oriented Parallel Finite Element Framework. *Archives of Computational Methods in Engineering*, 25 (2):195–271, 2018. doi:10.1007/s11831-017-9244-1.

[4] S. Badia, F. Verdugo, and A. F. Martín. The aggregated unfitted finite element method for elliptic problems. *Computer Methods in Applied Mechanics and Engineering*, 336:533–553, 2018. doi:10.1016/j.cma.2018.03.022.

[3] Available at https://gitlab.com/fempar/fempar.

[5] S. Badia, A. F. Martín, E. Neiva, and F. Verdugo. A generic finite element framework on parallel tree-based adaptive meshes. *Arxiv*, 2019.

[6] P. Benard, G. Balarac, V. Moureau, C. Dobrzynski, G. Lartigue, and Y. D'Angelo. Mesh adaptation for large-eddy simulations in complex geometries. *International journal for numerical methods in fluids*, 81(12):719–740, 2016.

[7] E. Burman, S. Claus, P. Hansbo, M. G. Larson, and A. Massing. CutFEM: Discretizing Geometry and Partial Differential Equations. *International Journal for Numerical Methods in Engineering*, 104(7):472–501, 2015. doi:10.1002/nme.4823.

[8] C. Burstedde and J. Holke. A Tetrahedral Space-Filling Curve for Nonconforming Adaptive Meshes. *SIAM Journal on Scientific Computing*, 38(5):C471–C503, 2016. doi:10.1137/15M1040049.

[9] C. Burstedde, L. C. Wilcox, and O. Ghattas. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011. doi:10.1137/100791634.

[10] T. Coupez, H. Digonnet, and R. Ducloux. Parallel meshing and remeshing. *Applied Mathematical Modelling*, 25(2):153–175, 2000.

[11] C. Dapogny, C. Dobrzynski, and P. Frey. Three-dimensional adaptive domain remeshing, implicit domain meshing, and applications to free and moving boundary problems. Technical report, Mar. 2013. URL `https://hal.sorbonne-universite.fr/hal-00804636`.

[12] J. Holke. Scalable Algorithms for Parallel Tree-based Adaptive Mesh Refinement with General Element Types. 2018. URL `http://arxiv.org/abs/1803.04970`.

[13] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas. Recursive Algorithms for Distributed Forests of Octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015. doi:10.1137/140970963.

[14] G. Karypis and V. Kumar. Parallel Multilevel series k-Way Partitioning Scheme for Irregular Graphs. *SIAM Review*, 41(2):278–300, 1999. doi:10.1137/S0036144598334138.

[15] D. W. Kelly, J. P. De S. R. Gago, O. C. Zienkiewicz, and I. Babuska. A posteriori error analysis and adaptive processes in the finite element method: Part I—error analysis. *International Journal for Numerical Methods in Engineering*, 19(11):1593–1619, 1983. doi:10.1002/nme.1620191103.

[16] C. Lachat, F. Pellegrini, C. Dobrzynski, and G. Staffelbach. Fast parallel remeshing for accurate large-eddy simulations on very large meshes. Technical report, 2017.

[17] D. Schillinger and M. Ruess. The Finite Cell Method: A Review in the Context of Higher-Order Structural Analysis of CAD and Image-Based Geometric Models. *Archives of Computational Methods in Engineering*, 22(3):391–455, 2015. doi:10.1007/s11831-014-9115-y.