Gmsh-Fem: An Efficient Finite Element Library Based On Gmsh

A. Royer, E. Béchet, C. Geuzaine

INFORMATION

Keywords:
Finite Element Library
High-order Methods
Multi-threading
C++11

DOI: 10.23967/wccm-eccomas.2020.161

Published: 11/03/2021

14th World Congress on Computational Mechanics (WCCM)
ECCOMAS Congress 2020)
19–24 July 2020, Paris, France
F. Chinesta, R. Abgrall, O. Allix and M. Kaliske (Eds)

GMSH-FEM: AN EFFICIENT FINITE ELEMENT LIBRARY BASED ON GMSH

Anthony Royer[1], Eric Béchet[2] and Christophe Geuzaine[1]

[1]Université de Liège, Institut Montefiore B28, 4000 Liège (Belgium)
{anthony.royer, cgeuzaine}@uliege.be

[2] Université de Liège, Département Aérospatiale et Mécanique B52, 4000 Liège (Belgium)
eric.bechet@uliege.be

Key words: Finite Element Library, High-order Methods, Multi-threading, C++11

Abstract. GmshFem is an open source C++ finite element library based on the application programming interface of Gmsh. Both share the same design philosophy: to be fast, light and user-friendly. This paper presents the main principles of GmshFem, as well as some scalability results for high-order scalar and vector finite element assembly on multi-core architectures.

1 INTRODUCTION

A multitude of open source finite element codes and libraries are currently available, with varying degrees of generality, performance, robustness and user-friendliness. Amongst many others, one can cite e.g. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]... As with any software, most excel in one or two of these areas; but compromises are invariably necessary to balance e.g. generality and user-friendliness (the ability to deal easily with various partial differential equations and physical models, or to integrate the code in complex workflows) with raw performance. Robustness and code maturity is an additional parameter to consider, as is community support.

In this work we introduce a new open source C++ finite element library, called GmshFem (`https://gitlab.onelab.info/gmsh/fem`), that is based on the application programming interface (API) of the popular open source finite element mesh generator, pre- and post-processor Gmsh [15]. GmshFem exhibits its own particular blend of features to cover the aforementioned areas, as it follows the same design philosophy as Gmsh: to be fast, light and user-friendly [15].

To be fast, GmshFem is designed to gain the greatest benefits of multi-core CPUs by combining an efficient multi-threaded parallelization with SIMD vectorization. During all development, particular attention was paid to data localities resulting in excellent performance on multi-core chips. Moreover, native support for complex-arithmetic makes it a perfect framework for efficient time-harmonic finite element solvers.

To be light, GmshFem extensively relies on Gmsh to manage geometries (with direct access to the CAD boundary representation), meshes (structured, unstructured and hybrid with straight-sided or curved elements, in 1D, 2D and 3D), interpolation (arbitrary order Lagrange or hierarchical bases for $\mathcal{H}^1$ and $\mathcal{H}(\mathsf{curl})$ [16]) and integration (numerical quadratures on all element shapes: lines, triangles, quadrangles, tetrahedra, prisms, hexahedra, pyramids, ...). GmshFem relies on its own internal sparse matrix storage, but interfaces with Eigen [17] for dense linear algebra and external solvers like PETSc [18] and MUMPS [19] for large sparse solvers required for implicit formulations.

Strongly inspired by the design of GetDP [9], GmshFem relies for generality and user-friendliness on a symbolic, high-level description of weak formulations which allows to define the problem to solve (including boundary conditions, source terms, etc.) in a natural mathematical manner, and is amenable to scripting without pre- or re-compilation, like GetDP [9] or FreeFEM++ [8] but unlike most other libraries [3, 4, 7, 6, 10, 14]. Moreover, this genericity implies neither the hard-coding of particular classes of PDEs [5], nor the use of a full-blown external scripting language like Python [12, 7].

The paper is organized as follows. In Section 2 we start by describing how to express a finite element problem in GmshFem, using a simple Helmholtz problem as a guiding example. We then describe in Section 3 the assembly algorithm for classical continuous Galerkin finite elements, and analyze the parallel efficiency of the implementation. We conclude with general remarks and perspective for future work in Section 4.

2 SYMBOLIC DEFINITION OF THE PROBLEM

To illustrate how a finite element problem is set up in GmshFem, let us consider the scattering of a time-harmonic incident acoustic plane wave u_{inc} by a hard obstacle Ω_{scat} of boundary Γ_{scat}. To solve the associated Helmholtz equation in terms of the complex-valued scattered field u, the unbounded domain $\mathbb{R}^3 \backslash \Omega^-$ is truncated and a Sommerfeld absorbing boundary condition is imposed on the fictitious boundary Γ_{inf}, leading to the following problem on the bounded domain Ω of boundary Γ_{scat} and Γ_{inf} (the time dependence is assumed of the form of e^{-iwt}):

$$\begin{cases} (\Delta + k^2)u = 0 & \text{in } \Omega, \\ \partial_{\mathbf{n}} u = -\partial_{\mathbf{n}} u_{\text{inc}} & \text{on } \Gamma_{\text{scat}}, \\ \partial_{\mathbf{n}} u - iku = 0 & \text{on } \Gamma_{\text{inf}}, \end{cases} \tag{1}$$

where $i^2 = -1$, Δ is the Laplacian operator, $k = {}^{w}\!/_{c}$ is the wave number with c the speed of sound and $\partial_{\mathbf{n}}$ is the normal directional derivative with $\mathbf{n}$ the outgoing normal to Ω. In weak form, (1) writes: Find $u \in \mathcal{H}^1(\bar{\Omega})$ such that

$$\int_{\Omega} (k^2 u v - \mathsf{grad}\ u \cdot \mathsf{grad}\ v)\ \mathsf{d}\Omega - \int_{\Gamma_{\text{scat}}} \partial_{\mathbf{n}} u_{\text{inc}}\, v\ \mathsf{d}\Gamma_{\text{scat}} + \int_{\Gamma_{\text{inf}}} iku\, v\ \mathsf{d}\Gamma_{\text{inf}} = 0 \tag{2}$$

holds for every test function $v \in \mathcal{H}^1(\bar{\Omega})$.

The following sections describe how (2) is transcribed in the GmshFem library. All the classes and functions used in the examples below are defined in the `gmshfem` namespace, which has been omitted for conciseness (i.e. as if "`using namespace gmshfem;`" was used).

2.1 `Domain` class

In GmshFem, all the data related to the geometry, the mesh and the topology is manipulated with the help of `Domain` objects, which are based on the notion of physical group in Gmsh [15]. For example, each time a function is piece-wise defined over a part of a mesh, a `Domain` object is involved. Amongst other operations, `Domain` objects can be manipulated using binary operations of set algebra: the union, the intersection and the complement of domains are respectively defined using the bitwise OR operator `|`, the bitewise AND operator `&` and the bitwise NOT operator $\sim$. The bitwise OR assignment operator `|=` and the bitwise AND assignment operator `&=` are also overloaded.

To model our scattering problem, three discrete domains corresponding to the volume Ω, the surface of the scattering object Γ_{scat} and the fictitious boundary Γ_{inf} are needed. Listing 1 shows how to declare them: here these simple domains are directly linked to the physical groups defined in Gmsh, referenced by their dimension and their unique tag.

```
  domain::Domain omega(3, 3), gammaScat(2, 1), gammaInf(2, 2);
```

Listing 1: Domain needed to model the scattering problem.

2.2 `Function` class

All symbolic mathematical expressions in GmshFem are managed with the help of `Function` classes (e.g. `ScalarFunction`, `VectorFunction` or `TensorFunction`). These classes are templated on an algebraic structure (the real or complex set) with a desired precision (single or double precision). Functions store symbolic expressions in a tree structure that is evaluated at runtime, e.g. during pre-processing, assembly, residual calculation or post-processing.

On the one hand these classes are very versatile, which allows to write a wide variety of functions such as transcendental functions, interpolation functions, mesh coordinates evaluations, finite element field evaluations, etc. The C++ arithmetic operators are overloaded for these classes, which makes it possible to write expressions in a compact form close to their mathematical definition. New functions can of course be added by users, going as far as hard-coding the full terms necessary for e.g. assembling complex weak formulation or post-processing computations. This can for example be advantageously used for multilevel methods, where a function can embed another solver; or for optimization or inverse problems, where automatic differentiation engines can be efficiently interfaced.

On the other hand, the tree structure allows to write external parsers that can directly feed complex functional expressions to GmshFem, allowing to embed GmshFem in other codes and benefit from its high-performance C++ numerical kernel without having to recompile the library; and still keep the user-friendliness and flexibility of a scripted code, without incurring the runtime costs of scripted languages. This is directly inspired by the design of GetDP [9], where all expressions are analyzed once during a parsing phase, and executed at runtime later on.

Listing 2 shows how the function $\partial_{\mathbf{n}} u_{\text{inc}}$ can be defined, when $u_{\text{inc}} = e^{ikx}$.

```
function::ScalarFunction<std::complex<double>> duInc =
   - k * function::sin(k * function::x<std::complex<double>>()) +
   im * k * function::cos(k * function::x<std::complex<double>>());
```

Listing 2: Definition of a scalar function used as boundary condition on the surface of the scattering object.

2.3 `Field` class

The `Field` class is designed to store information about a finite element field and its associated discrete function space. To write any finite element problem based on it weak form, one or several instantiations of the `Field` class are employed. Once the interpolation coefficients have been computed, these `Field` objects can be evaluated, for instance to be used in other formulations, to be saved as post-processing views with the Gmsh API, or to be exchanged across subdomains in domain decomposition algorithms.

A `Field` object has two templates: the first one indicates the algebraic structure and desired precision of the interpolation coefficients, and the second defines the mathematical nature of the field. For instance, 0-forms (scalar fields) are needed in our acoustic scattering example, while 1-forms will be needed for the electromagnetic test-case analyzed in Section 3.3. The simplest fields are instanciated in GmshFem with three parameters: a name, a domain of definition (through a `Domain` object), and the identifier of a discrete function space. For non-isoparametric interpolations, a fourth argument specifies the desired interpolation order. GmshFem currently supports both Lagrange basis functions and arbitrary order, hierarchical basis functions for $\mathcal{H}^1$ and $\mathcal{H}(\mathrm{curl})$ [16].

In our simple acoustic scattering example, the discrete scalar field u_h approximating the solution u of the weak form (1) is transcribed into a `Field` object templated over the body of complex numbers using double precision floating point arithmetic, as shown in Listing 3. It is defined over the closed discrete domain approximating Ω with its boundary, and is interpolated with hierarchical basis functions of order 6.

```
field::Field<std::complex<double>, form::Form0>
  u("u", omega|gammaScat|gammaInf, functionSpaceH1::HierarchicalH1, 6);
```

Listing 3: 0-form field defined to model the acoustic wave scattering problem.

2.4 `Formulation` class

The `Formulation` stores the symbolic representation of the weak formulation of the problem, and can evaluate linear and bilinear forms, store the corresponding matrix systems, and request their solution through external linear algebra packages [19, 18]. The `Formulation` class must be templated on the same algebraic structure and precision as the different objects it references, i.e. functions and fields.

For continuous Galerkin finite elements formulations, the `Formulation` object provides the `galerkin` function, whose two first `Function` arguments are the arguments of the inner product describing one term in the weak formulation: the first can involve any linear function of the unknown field, denoted by `dof()`; the second must involve a linear function of the test function, denoted by `tf()`. If the first argument involves an unknown field, it leads to the evaluation of a bilinear form; otherwise it leads to the evaluation of a linear form. The third argument specifies the `Domain` over which the integration is performed, and the fourth specifies the quadrature rule (e.g. `Gauss12` for a Gauss quadrature suited for integrating 12th order polynomials). An arbitrary number of `galerkin` terms can be specified: they are all summed to produce the final discrete weak formulation. This symbolic expression of the weak form is identical to the one introduced by GetDP [9], and allows to seamlessly handle coupled and mixed formulations. For simple implicit formulations, the `Formulation` provides three functions encapsulating the pre-processing phase, i.e. the identification of degrees of freedom and constraints (`pre`), the assembly of the linear system (`assemble`) and the solution of the linear system (`solve`).

In our acoustic scattering example, all objects are templated over the `std::complex<double>` type. Three bilinear terms (two in the volume, one on the artificial boundary) and one linear term (on the scatterer boundary) encode the discrete version of the weak formulation (2): see Listing 4.

```
problem::Formulation<std::complex<double>> formulation("helmholtz");

formulation.galerkin(equation::grad(equation::dof(u)),
                     equation::grad(equation::tf(u)), omega, "Gauss12");
formulation.galerkin(- k * k * equation::dof(u),
                     equation::tf(u), omega, "Gauss12");
formulation.galerkin(- im * k * equation::dof(u),
                     equation::tf(u), gammaInf, "Gauss12");
formulation.galerkin(-duInc, equation::tf(u), gammaScat, "Gauss12");

formulation.pre();
formulation.assemble();
formulation.solve();
```

Listing 4: Definition of the formulation for the acoustic wave scattering problem.

2.5 Post-processing functions

Once a problem is solved, fields can be post-processed with the help of any `Function`, using a variety of operations. The simplest operation consists in exporting the data as a Gmsh post-processing view. Listening 5 shows two simple post-processing operations for our example problem, to export the field and its gradient and save them to disk in the default file format (a Gmsh `.msh` file).

```
post::save(u, omega, "u");
post::save(function::grad(u), omega, "grad_u");
```

Listing 5: Some example of post-processing operations.

The full code is available in Listing 6 in the appendix.

3 ASSEMBLY PROCESS ALGORITHM

For implicit, low order continuous Galerkin finite element formulations, the most time consuming part of the finite element process (CAD and meshing aside...) resides in the solution of the resulting large, sparse linear systems. For high-order finite element methods, however, which are increasingly used for complex simulations to alleviate the slow grid convergence of the state-of-the art (usually second order) methods provided by most industrial codes, the mere process of assembling the finite element matrices, or computing the residuals or the time iterates, rapidly becomes a bottleneck in the computer implementation.

While high-order finite elements naturally lead to increased arithmetic intensity, since the local element-wise matrices become larger and more dense, the number of quadrature points also dramatically increases. The best way to achieve good performance in such cases is to reformulate all the quadratures as dense matrix-matrix products [7, 20], and by pre-computing as many of the underlying matrices as possible. While a natural decomposition resides in the separation of the metric-dependent and metric-independent parts in the Galerkin terms [20], many codes trade-off accuracy and generality for performance, by assuming for example that all parts of the integrands are interpolated using the same bases as the unknown fields [7]. This is not suitable for e.g. strongly nonlinear problems or multiscale problems, though, as the coefficients don't have the same regularity. The cost of pre-computing and storing local matrices is exacerbated when using hierarchical bases, or vector-valued basis functions for e.g. $\mathcal{H}(\mathsf{curl})$ or $\mathcal{H}(\mathsf{div})$, where the basis functions depend on the orientation of the elements [16, 20]. Storing all unassembled matrices in such cases rapidly leads to prohibitive memory requirements, even in cases where the matrix is eventually factorized by direct linear solvers.

In GmshFem the efficient evaluation of linear and bilinear forms is based on a compromise between processing time and memory usage, where good parallel performance is obtained by the careful analysis the effect of spatial and temporal data locality.

3.1 Pre-processing Phase

Before any evaluation of linear or bilinear forms can take place, GmshFem performs a pre-processing phase, which builds a dictionary of degrees of freedom (identified by keys), based on the input mesh, the fields and their associated function spaces, and the finite element formulation. A unique tag is associated with each unknown DoF, which corresponds to an equation number. Tags are chosen for example such that "bubble" degrees of freedom (which only depend on the element, and are not shared between mesh entities) are explicitly identified, which helps assembling them without locks due to their one-element locality. For implicit formulations, the pre-processing phase also allocates the arrays used to store the finite element matrices in a compressed row storage (CRS) format. The sizes of these arrays are known by computing the pattern of the finite element matrix, by assuming that all local matrices (resulting from the integration over one element) are dense. The parallel efficiency of the pre-processing is limited by the performance of the hash map used to identify DoFs, and locks required by the calculation of the global matrix pattern.

3.2 Assembly Algorithm

The assembly process can be seen as an algorithm that combines information of different nature and stores the result in a finite element matrix. For instance, let us consider the following local stiffness matrix over the element Ω_e, where $f(\mathbf{x})$ is any function, ϕ_i or ϕ_j are the basis functions associated with degrees of freedom i or j, and $J = \partial x_i / \partial u_j$ is the Jacobian matrix mapping the reference coordinates ($\mathbf{u}$) of the reference element to the mesh coordinates ($\mathbf{x}$):

$$A_{ij} = \int_{\Omega_e} f(\mathbf{x}) (J^{-T} \mathsf{grad}\ \phi_i)^T (J^{-T} \mathsf{grad}\ \phi_j) \det J \ \mathsf{d}\Omega_e. \tag{3}$$

The integrand is a combination of geometric, i.e. metric-dependent, information (the Jacobian matrix J and its determinant $\det J$), metric-independent basis function data ($\mathsf{grad}\ \phi_i$ and $\mathsf{grad}\ \phi_j$), and arbitrary function evaluations ($f(\mathbf{x})$). This integral is evaluated using a numerical quadrature rule, leading to a weighted sum (with weights w_q) of evaluations of the integrand at integration points $\mathbf{x}_q$:

$$A_{ij} \approx \sum_{q=1}^{Q} f(\mathbf{x}_q) \left[J_q^{-T} \mathsf{grad}\, \phi_i(\mathbf{u}_q)\right]^T \left[J_q^{-T} \mathsf{grad}\, \phi_j(\mathbf{u}_q)\right] \det J_q w_q. \tag{4}$$

Four kinds of data are therefore needed to assemble such a term over an element:

1. Integration points and weights,
2. Geometric information represented by the Jacobian matrix and its determinant evaluated at integration points,
3. Basis functions evaluation at integration points,
4. Arbitrary function evaluated at the integration points.

GmshFem retrieves the first three data directly from the Gmsh API. For efficiency, Gmsh and GmshFem deal with such data for groups (“buckets”) of elements of the same type, so that data can be accessed efficiently in contiguous chunks of memory, suitable for optimized vectorized operations. Three criteria define these buckets.

The first distinction is made on the type and geometrical order of the elements (e.g. lines, triangles, quadrangles, tetrahedra, ..., straight-sided or curved). Indeed, the integration points expressed in the reference coordinate system and their associated weights are identical for a given type if the same quadrature order is used. Furthermore, before assembling elements of the same type using the same quadrature order, basis functions can be pre-computed at integration points for all possible orientations. Then during the assembly process, each element has a tag that identify its orientation.

The second distinction is made by geometrical entity, for which metric-dependent information is computed in a single pass by Gmsh. Moreover, as weak form integrals are defined over geometric entities, mathematical functions appearing in their integrands can be pre-computed as well for all integration points at this stage.

The third distinction depends on the formulation. When a bilinear term is defined, it involves a pair of fields; an unknown field and test functions associated to the same field, or to another field. Once the problem is discretized, this pair corresponds to a block in the finite element matrix. Therefore, it is suitable to assemble terms pair by pair to avoid unnecessary displacements in memory that will negatively impact the cache efficiency of the program.

Once this data is pre-computed (in parallel) and stored in arrays, the assembly proceeds in parallel for elements belonging to the same bucket. Each thread is responsible of contiguous elements; and on each element, linear algebra operations are handled by the third part C++ template library for linear algebra, Eigen [17]. As threads assemble contiguous elements, they combine contiguous parts of the pre-computed data. Spatial memory locality is maximized as data needed to process an element is always close to each other and temporal memory locality is also maximized as needed data to assemble the next element is close to the data used to assemble the current one.

Finally, the local matrix elements are pushed into the global matrix stored in CRS format at locations given by pre-computed indexed arrays. An single atomic addition directive is applied to avoid race condition, except for bubble degrees of freedom as mentioned above. The whole assembly procedure is summarized in Algorithm 1.

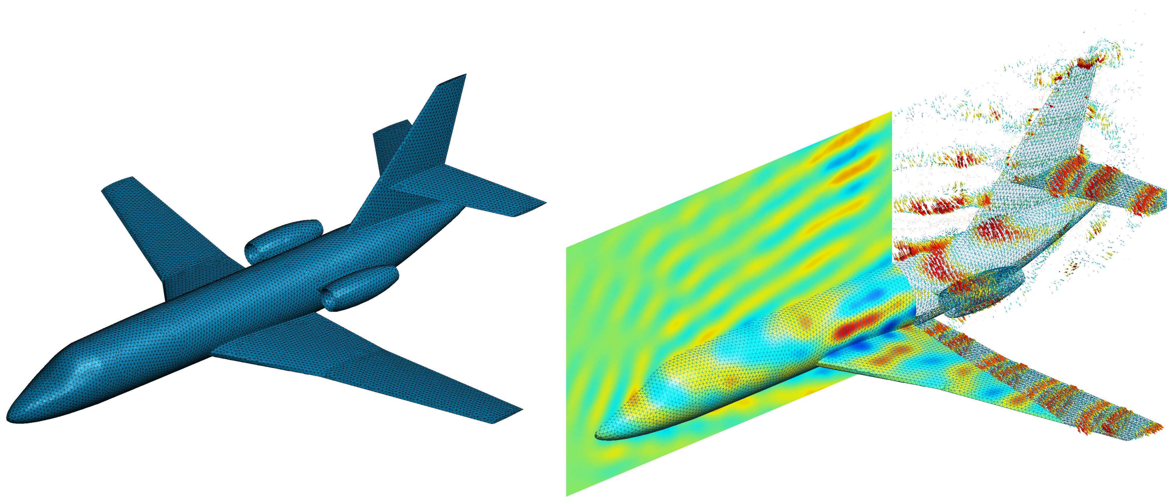

Figure 1: The Falcon plane used as the scatterer in our numerical test on the left. An acoustic and an electromagnetic scattered field computed on this mesh on the right. The front of the plane shows the acoustic scattered field of a 1m wavelength incident plane wave hitting the plane considered as a hard object; the back of the plane shows the electromagnetic scattered field of a 1m wavelength incident plane wave hitting the plane considered as a perfect electric conductor.

Algorithm 1 Pseudo-code of the assembly algorithm.

```
for all elementTypes do
    basisFunctionsData ← ComputeNeededBasisFunctions();
    entities ← GetEntitiesHavingCurrentElementType();
    for all entities do
        precomputedNeededFunctions();
        geometricData ← ComputeNeededGeometricData();
        fieldPairs ← GetFieldPairsDefinedOverCurrentEntity();
        for all fieldPairs do
            dofIndices ← ComputeDOFIndices();
            AssembleAndStoreInMatrix(basisFunctionsData, geometricData, dofIndices);
        end for
    end for
end for
```

3.3 Parallel Efficiency

The parallel efficiency is studied on a toy 3D scattering problem, consisting in bouncing a plane acoustic or electromagnetic wave on a simplified Falcon airplane, considered respectively as a hard acoustic scatterer or a perfect electric conductor. The acoustic scattering problem corresponds to (1). The elec-

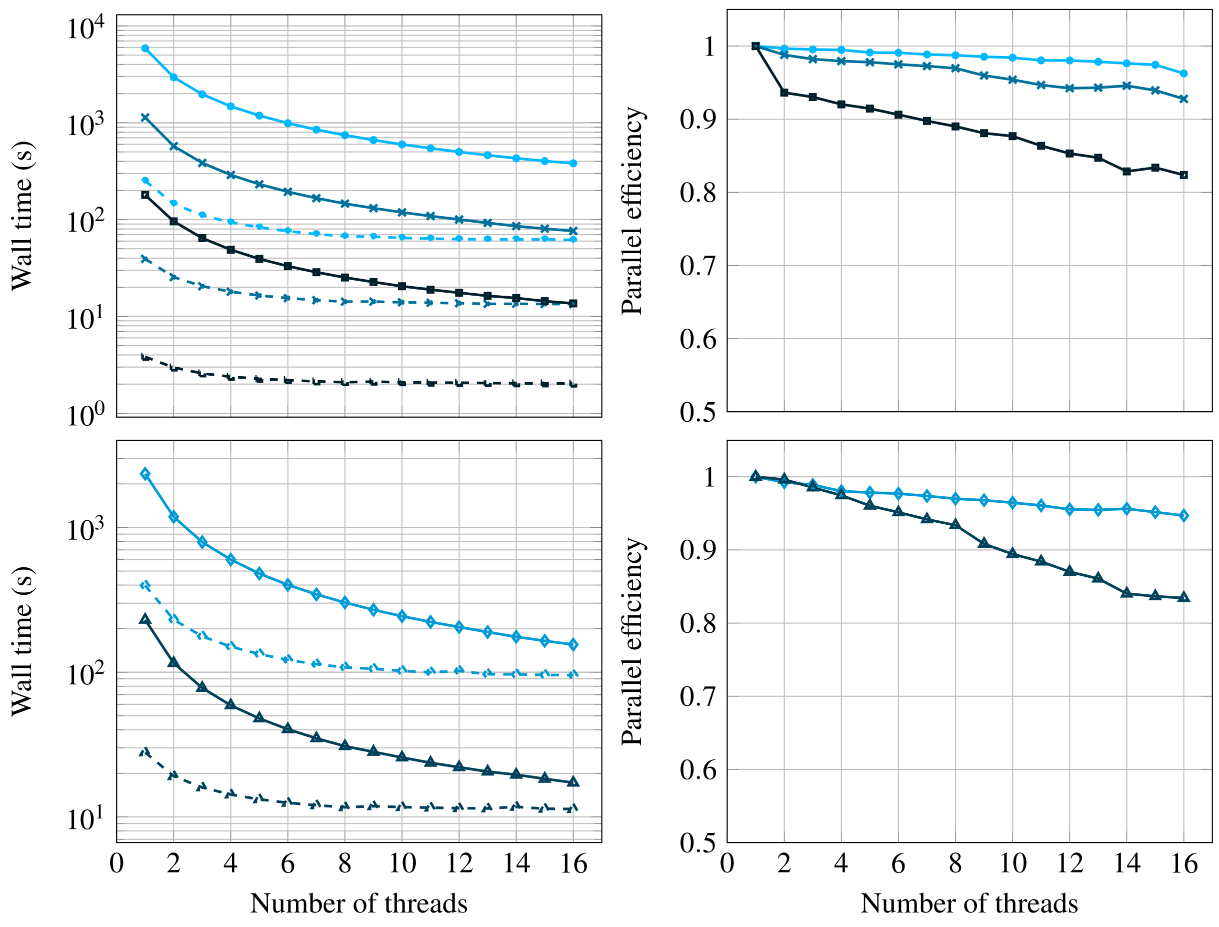

Figure 2: Wall time (left) and parallel efficiency (right) of the assembly process for the acoustic (top) and electromagnetic (bottom) case. The acoustic case uses basis functions of order 2 (squares), 4 (crosses) and 6 (circles) and the electromagnetic case uses basis functions of incomplete order 2 (diamonds) and incomplete order 3 (triangles). The dashed curves in the wall time graphs represent the pre-processing time, while the plain ones represent the assembly time.

tromagnetic problem is formulated in terms of the electric field $\mathbf{e}$:

$$\begin{cases} (\mathsf{curl} - k^2)\mathbf{e} = 0 & \text{in } \Omega, \\ \gamma^T(\mathbf{e}) = -\gamma^T(\mathbf{e}_{\text{inc}}) & \text{on } \Gamma_{\text{scat}}, \\ \gamma^t(\mathsf{curl}\ \mathbf{e}) + ik\gamma^T(\mathbf{e}) = 0 & \text{on } \Gamma_{\text{inf}}, \end{cases} \tag{5}$$

where $\gamma^T : \mathbf{v} \mapsto \mathbf{n} \times (\mathbf{v} \times \mathbf{n})$ is the tangential component trace operator and $\gamma^t : \mathbf{v} \mapsto \mathbf{n} \times \mathbf{v}$ the the tangential trace operator. The Silver-Müller radiation condition is enforced on the fictitious boundary Γ_{inf} to ensure that the electric field $\mathbf{e}$ is outgoing.

The wall clock time and the parallel efficiency for different orders of hierarchical $\mathcal{H}^1$ (for the acoustic problem) and $\mathcal{H}(\mathsf{curl})$ (for the electromagnetic problem) elements is reported on Figure 2. (Computations were run on 2.0GHz Intel SandyBridge CPUs, using an unstructured mesh made of 219.477 first-order tetrahedra and 42.569 first-order triangles.) The parallel efficiency for the incomplete 3th order elements and for the 4th and 6th order elements is excellent, around or above 95% for 16 threads. It degrades slightly for lower orders, down to about 82% for 16 threads at orders 2. The wall clock time of the assembly process for $\mathcal{H}(\mathsf{curl})$ elements is of course higher at the same order than for $\mathcal{H}^1$ due to the larger number of DoFs, but the scaling is very similar. These results are coherent with the observations made above: on the one hand, the arithmetic intensity increases with the order, which leads to better parallel efficiency; and on the other hand, the ratio of "bubble" degrees of freedom increases with the element order, leading to a decrease of locking events. Finally, the pre-processing time for $\mathcal{H}^1$ elements is always much smaller than the assembly time, while it could still be improved for $\mathcal{H}(\mathsf{curl})$ elements.

4 CONCLUSIONS

In this paper we introduced GmshFem, an open source C++ finite element library based on the Gmsh API. After an introduction to the basic philosophy of the software, its parallel efficiency was demonstrated for high-order scalar and vector problems.

While GmshFem is still very much a work in progress, it is already used to solve extreme-scale finite element problems on massively parallel, distributed computer architectures in the context of high-frequency, time-harmonic acoustic, elastic and electromagnetic wave problems in conjunction with new optimized, high-order domain decomposition methods [21, 22, 23]. Work is ongoing on the integration of discontinuous Galerkin methods and the efficient offloading of linear algebra on GPUs.

A APPENDIX

```cpp
#include <gmshfem/GmshFem.h>
#include <gmshfem/Function.h>
#include <gmshfem/Formulation.h>

using namespace gmshfem;

int main(int argc, char **argv)
{
  common::GmshFem fem(argc, argv);

  gmsh::open("mesh.msh");
  domain::Domain omega(3, 3), gammaScat(2, 1), gammaInf(2, 2);

  double k = 1;
  std::complex<double> im(0, 1);
  function::ScalarFunction<std::complex<double>> duInc =
    - k * function::sin(k * function::x<std::complex<double>>()) +
    im * k * function::cos(k * function::x<std::complex<double>>());

  field::Field<std::complex<double>, form::Form0>
    u("u", omega|gammaScat|gammaInf, functionSpaceH1::HierarchicalH1, 6);

  problem::Formulation<std::complex<double>> formulation("helmholtz");
  formulation.galerkin(equation::grad(equation::dof(u)),
                       equation::grad(equation::tf(u)), omega, "Gauss12");
  formulation.galerkin(- k * k * equation::dof(u),
                       equation::tf(u), omega, "Gauss12");
  formulation.galerkin(- im * k * equation::dof(u),
                       equation::tf(u), gammaInf, "Gauss12");
  formulation.galerkin(-duInc, equation::tf(u), gammaScat, "Gauss12");

  formulation.pre();
  formulation.assemble();
  formulation.solve();

  post::save(u, omega, "u");
  post::save(function::grad(u), omega, "grad_u");
  return 0;
}
```

Listing 6: Full code for the small acoustic scattering example.

REFERENCES

[1] G. Dhondt and K. Wittig. CALCULIX: A free software three-dimensional structural finite element program. www.calculix.de.

[2] Electricité de France. Finite element *Code_Aster*, analysis of structures and thermomechanics for studies and research. Open source on www.code-aster.org, 1989–2017.

[3] D. Arndt, W. Bangerth, T. C. Clevenger, D. Davydov, M. Fehling, D. Garcia-Sanchez, G. Harper, T. Heister, L. Heltai, M. Kronbichler, R. M. Kynch, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells. The deal.II library, version 9.1. *Journal of Numerical Mathematics*, 27(4):203–213, 2019.

[4] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the Dune-Fem module. *Computing*, 90:165–196, 2010.

[5] P. Råback, P.-L. Forsström, M. Lyly, and M. Gröhn. Elmer - finite element package for the solution of partial differential equations, 2007. Poster presentation.

[6] C. PrudHomme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, and G. Pena. Feel++: A compu-

tational framework for galerkin methods and advanced numerical methods. In *ESAIM: Proceedings*, volume 38, pages 429–455. EDP Sciences, 2012.

[7] M. S. Alnæs, J Blechta, J Hake, A Johansson, B Kehlet, A Logg, C Richardson, J Ring, M. E. Rognes, and G. N. Wells. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 3(100):9–23, 2015.

[8] F. Hecht. New development in FreeFem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[9] P. Dular and C. Geuzaine. GetDP reference manual: the documentation for GetDP, a general environment for the treatment of discrete problems. `http://getdp.info`.

[10] Y. Renard and K. Poulio. GetFEM: Automated fe modeling of multiphysics problems based on a generic weak form language. 2020. hal-02532422.

[11] MFEM: Modular finite element methods library. `mfem.org`.

[12] J. Schöberl. C++11 implementation of finite elements in NGSolve. Technical report, Institute for Analysis and Scientific Computing, Vienna University of Technology, 2014.

[13] F. McKenna, M. H. Scott, and G. L. Fenves. Nonlinear finite-element analysis software architecture using object composition. *Journal of Computing in Civil Engineering*, 24:95–107, 2010.

[14] A. Halbach. Sparselizard - the user friendly finite element c++ library. `www.sparselizard.org`, 2017.

[15] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, 79(11):1309–1331, 2009.

[16] P. Solin, K. Segeth, and I. Dolezel. *Higher-order finite element methods*. CRC Press, 2003.

[17] G Guennebaud, B Jacob, et al. Eigen v3. `eigen.tuxfamily.org`, 2010.

[18] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, et al. PETSc users manual. 2019.

[19] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. MUMPS: a general purpose distributed memory sparse solver. In *International Workshop on Applied Parallel Computing*, pages 121–130. Springer, 2000.

[20] N. Marsic and C. Geuzaine. Efficient finite element assembly of high order Whitney forms. *IET Science, Measurement & Technology*, 9(2):204–210, 2015.

[21] Y. Boubendir, X. Antoine, and C. Geuzaine. A quasi-optimal non-overlapping domain decomposition algorithm for the Helmholtz equation. *Journal of Computational Physics*, 231(2):262–280, 2012.

[22] M. El Bouajaji, B. Thierry, X. Antoine, and C. Geuzaine. A quasi-optimal domain decomposition algorithm for the time-harmonic Maxwell's equations. *Journal of Computational Physics*, 294:38–57, 2015.

[23] A. Modave, A. Royer, X. Antoine, and C. Geuzaine. A non-overlapping domain decomposition method with high-order transmission conditions and cross-point treatment for Helmholtz problems. *Submitted to CMAME*, 2020.