Daniel Arndt, Wolfgang Bangerth, Bruno Blais, Thomas C. Clevenger, Marc Fehling, Alexander V. Grayver, Timo Heister*, Luca Heltai, Martin Kronbichler, Matthias Maier, Peter Munch, Jean-Paul Pelteret, Reza Rastak, Ignacio Tomas, Bruno Turcksin, Zhuoran Wang, and David Wells

The `deal.II` library, Version 9.2

https://doi.org/10.1515/jnma-2020-0043
Received June 11, 2020; revised July 02, 2020; accepted July 02, 2020

Abstract: This paper provides an overview of the new features of the finite element library `deal.II`, version 9.2.

Keywords: software, finite elements, deal.II

Classification: 65M60, 65N30, 65Y05

1 Overview

`deal.II` version 9.2.0 was released May 20, 2020. This paper provides an overview of the new features of this release and serves as a citable reference for the `deal.II` software library version 9.2. `deal.II` is an object-oriented finite element library used around the world in the development of finite element solvers. It is available for free under the GNU Lesser General Public License (LGPL). Downloads are available at https://www.dealii.org/ and https://github.com/dealii/dealii.

The major changes of this release are:

- A new, parallel, fully distributed triangulation class (see Section 2.1);
- Substantially improved performance for very large computations on tens or hundreds of thousands of processor cores and up to trillions of unknowns (see Section 2.2);
- Better support for parallel hp-adaptive algorithms (see Section 2.3);
- Support for particle-based methods (see Section 2.4);
- Improved performance of the symbolic differentiation framework (see Section 2.5);

Daniel Arndt, Bruno Turcksin, Computational Engineering and Energy Sciences Group, Computational Sciences and Engineering Division, Oak Ridge National Laboratory, 1 Bethel Valley Rd., TN 37831, USA.
Wolfgang Bangerth, Department of Mathematics and Department of Geosciences, Colorado State University, Fort Collins, CO 80523-1874, USA.
Bruno Blais, Research Unit for Industrial Flows Processes (URPEI), Department of Chemical Engineering, Polytechnique Montréal, PO Box 6079, Stn Centre-Ville, Montréal, Québec, Canada, H3C 3A7.
Thomas C. Clevenger, School of Mathematical and Statistical Sciences, Clemson University, Clemson, SC, 29634, USA.
Marc Fehling, Institute for Advanced Simulation, Forschungszentrum Jülich GmbH, 52425 Jülich, Germany.
Alexander V. Grayver, Institute of Geophysics, ETH Zurich, Sonneggstrasse 5, 8092 Zürich, Switzerland.
***Corresponding author: Timo Heister,** School of Mathematical and Statistical Sciences, Clemson University, Clemson, SC, 29634, USA. Email: heister@clemson.edu
Luca Heltai, SISSA, International School for Advanced Studies, Via Bonomea 265, 34136, Trieste, Italy.
Martin Kronbichler, Peter Munch, Institute for Computational Mechanics, Technical University of Munich, Boltzmannstr. 15, 85748 Garching, Germany.
Matthias Maier, Department of Mathematics, Texas A&M University, 3368 TAMU, College Station, TX 77845, USA.
Peter Munch, Institute of Materials Research, Materials Mechanics, Helmholtz-Zentrum Geesthacht, Max-Planck-Str. 1, 21502 Geesthacht, Germany.
Jean-Paul Pelteret, Independent researcher.
Reza Rastak, Department of Civil and Environmental Engineering, Stanford University, Stanford, CA 94305, USA.
Ignacio Tomas, Sandia National Laboratories, Org 1442, P.O. Box 5800, MS 1320, Albuquerque, NM, 87185-1320, USA.
Zhuoran Wang, Department of Mathematics, Colorado State University, Fort Collins, CO 80523-1874, USA.
David Wells, Department of Mathematics, University of North Carolina, Chapel Hill, NC 27516, USA.

- Advances in SIMD capabilities and the matrix-free infrastructure (see Section 2.6);
- Advances in GPU support (see Section 2.7);
- Better use of modern C++ language features (see Section 2.8);
- Seven new tutorial programs (see Section 2.9).

These major changes are discussed in detail in Section 2. There are a number of other noteworthy changes in the current `deal.II` release that we briefly outline in the remainder of this section:

- `deal.II` had decent support for solving complex-valued problems for a while already (e.g., ones in quantum mechanics — like the equation used in the `step-58` tutorial program covered below — or for time-harmonic problems). However, there were two areas in which support was missing. First, the UMFPACK direct solver packaged with `deal.II` did not support solving complex-valued linear problems. This has now been addressed: UMFPACK actually can solve such systems, we just needed to write the appropriate interfaces. Second, the `DataOut` class that is responsible for converting nodal data into information that can then be written into files for visualization did not know how to deal with vector- and tensor-valued fields whose components are complex numbers. An example for this is to solve the time-harmonic version of the Maxwell equations that has the electric and magnetic fields as solution. This, too, has been addressed in this release.
- The new `DiscreteTime` class provides a more consistent, more readable, and less error-prone approach to control time-stepping algorithms within time-dependent simulations. While providing a rich read-only interface, the non-const interface of this class is designed to be minimal to enforce a number of important programming invariants, reducing the possibility of mistakes in the user code. For instance, `DiscreteTime` ensures that the final time step ends precisely on a predefined end time, automatically lengthening or shortening the final time step.
- A key component of `deal.II` are the `FEValues` and `FEFaceValues` classes that evaluate finite element functions at quadrature points located on cells and faces of a cell, respectively, see [13]. This release now contains a class `FEInterfaceValues` that considers the restriction of the shape functions from *both* sides of a face and allows evaluating jumps and averages of shape functions along this face. These are common components of the bilinear forms of discontinuous Galerkin schemes (as well as schemes for fourth-order equations, see the discussion of `step-47` below) and greatly simplify the implementation of these methods.
- Previously, the `parallel::distributed::ContinuousQuadratureDataTransfer` class, which transfers local quadrature point data onto the children of newly refined cells, did not allow different cells to store quadrature data of different lengths. This release lifts this restriction, which enables efficient quadrature data storage and data transfer within a triangulation containing multiple material models, each with their own number of local state variables and history variables. As an example, in a solid mechanics simulation, we can assign a hyper-elastic material model to a region of the mesh with no associated history variables, while in another part of the triangulation we incorporate an elasto-plastic constitutive model, requiring the storage of local plasticity data at quadrature points. During each refinement of the mesh, we can then use `ContinuousQuadratureDataTransfer` to transfer and interpolate plasticity data from parent cells to their children, ignoring the cells with the hyper-elastic constitutive model.

The changelog lists more than 240 other features and bugfixes.

2 Major changes to the library

This release of `deal.II` contains a number of large and significant changes that will be discussed in this section. It of course also contains a vast number of smaller changes and added functionality; the details of these can be found in the file that lists all changes for this release, see [49].

2.1 A new fully distributed triangulation class

Previously, all triangulation classes of `deal.II` had in common that the coarse grid is replicated by all processes in a parallel environment, and the actual mesh used for computations is constructed by repeated refinement. However, this has its limitations in many applications where the mesh comes from an external mesh generator in the form of a file that frequently already contains millions or tens of millions of cells. For such configurations, applications might exhaust available memory already while reading the mesh on each MPI process.

The new `parallel::fullydistributed::Triangulation` class targets this issue by distributing also the coarse grid. Such a triangulation can be created by providing to each process a `TriangulationDescription::Description` struct, containing (i) the relevant data to construct the local part of the coarse grid, (ii) the translation of the local coarse-cell IDs to globally unique IDs, (iii) the hierarchy of mesh refinement steps, and (iv) the owner of the cells on the active mesh level as well as on the multigrid levels. For the current release, triangulations set up this way cannot be adaptively refined after construction, though we plan to improve this for the next release.

The new fully distributed triangulation class supports 1D, 2D, and 3D meshes including geometric multigrid hierarchies, periodic boundary conditions, and hanging nodes.

2.2 Improved large-scale performance

Large-scale simulations with up to 304,128 cores have revealed bottlenecks in release 9.1 during initialization of a number of distributed data structures, due to the usage of expensive collective operations like `MPI_Allgather()` and `MPI_Alltoall()`. Typical examples are the pre-computation of the indices of those vector entries (or other linear index ranges) owned by each process, which were previously stored in an array on every process. This information is needed to set up the `Utilities::MPI::Partitioner` class. In release 9.2, we have replaced these functions in favor of consensus algorithms [39], which can be found in the namespace `Utilities::MPI::ConsensusAlgorithms` (short: `CA`). Now, only the locally relevant information about the index ranges is (re)computed when needed, which, for more than 100 MPI processes, uses point-to-point communications and a single `MPI_IBarrier()`.

Users can apply the new algorithms for their own dynamic-sparse problems by providing a list of target processes and pack/unpack routines either by implementing the interface `CA::Process` or by providing `std::function` objects to `CA::AnonymousProcess`.

By replacing the collective communications during set up and removing the arrays that contain information for each process (enabled by the application of consensus algorithms and other modifications — a full list of modifications leading to this improvement can be found online), we were able to significantly improve the set up time for large-scale simulations and to solve a Poisson problem with multigrid with 2.1×10^{12} unknowns on the SuperMUC-NG supercomputer with 304,152 cores [7, 8]. Figure 1 compares timings of simulations of various problem sizes (including set up) on 49,152 MPI ranks using a matrix-free solver [8, 46, 47]; this solver uses discontinuous elements of degree 5 in a geometric multigrid (GMG) scheme. The comparison between the previous release 9.1 and the current release 9.2 shows that while the scaling for the V-cycle had been very good before, many initialization routines have been considerably improved, especially the enumeration of unknowns on the multigrid levels and the setup of the multigrid transfer.

As part of this effort, we ran benchmarks on the TACC Frontera system, where we were able to apply the matrix-free geometric multigrid framework to a variable viscosity Stokes system and achieved weak and strong scaling up to 114K MPI ranks with up to 2.1×10^{11} unknowns. This is likely the largest block system currently solved with `deal.II` and required various optimizations and fixes on top of the ones mentioned above: (i) Bug fixes to concurrent point to point communications. (ii) Fixes to multigrid transfer with adaptive refinement and more than 4×10^9 unknowns. (iii) Fixes to index sets in block indices with more than 4×10^9 unknowns. (iv) Fixes to computations with more than 4×10^9 active cells. (v) Implementation of IDR(s) solvers to reduce memory overhead. For more details, see [19].

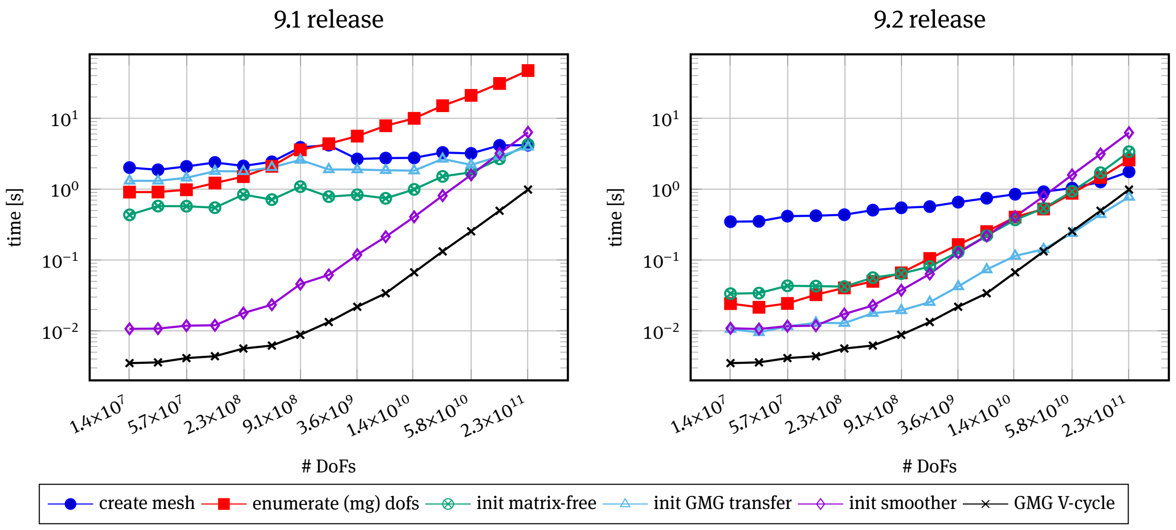

Fig. 1: Comparison of initialization costs of various data structures in the 9.1 release (left) and the new 9.2 release (right) when run on 49,152 MPI ranks.

2.3 Better support for parallel *hp*-adaptive algorithms

Since the previous release, `deal.II` has had support for *hp*-adaptive finite element methods on distributed memory systems [6]. We implemented the bare functionality for *hp*-adaptive methods with the objective to offer the greatest flexibility in their application. Here, reference finite elements still had to be assigned manually to each cell, which may not lead to an optimal mesh and is tedious.

With the current release, *hp*-adaptive finite element methods have been further expanded: New features like decision strategies have been added and the user interface has been overhauled, effectively making *hp*-methods more attractive to use. We introduced many new functions that automatize the general workflow for applying *hp*-decision strategies, which run on top of the previous low-level implementation for both serial and parallel applications.

The interface is now as simple to use as the one for *h*-adaptive mesh refinement. Consider the following (incomplete) listing as an example: We estimate both error and smoothness of the finite element approximation. Further, we flag certain fractions of cells with the highest and lowest errors for refinement and coarsening, respectively (here: 30%/3%). From those cells listed for adaptation, we designate a subset for *h*- and *p*-adaptation. The parameters of the corresponding `hp::Refinement` function specify the fraction of cells to be *p*-adapted from those subsets flagged for refinement and coarsening, respectively (here: 90%/80%), while the remaining fraction will be *h*-adapted (here: 10%/20%).

C++ code

```
Vector<float> estimated_error_per_cell(n_active_cells);
KellyErrorEstimator::estimate(
dof_handler, ..., solution, estimated_error_per_cell, ...);
GridRefinement::refine_and_coarsen_fixed_number(
triangulation, estimated_error_per_cell, 0.3, 0.03);

Vector<float> estimated_smoothness_per_cell(n_active_cells);
SmoothnessEstimator::Legendre::coefficient_decay(
..., dof_handler, solution, estimated_smoothness_per_cell);
hp::Refinement::p_adaptivity_fixed_number(
dof_handler, estimated_smoothness_per_cell, 0.9, 0.8);
hp::Refinement::choose_p_over_h(dof_handler);

triangulation.execute_coarsening_and_refinement();
```

In particular, we implemented decision strategies based on refinement history and smoothness estimation, and made sure that they work for refinement as well as coarsening in terms of h- and p-adaptation in serial and parallel applications.

The former relies on knowing an estimate for the upper error bound [9, Thm. 3.4]. For successive refinements, we can predict how the error will change based on current error estimates and adaptation flags. In the next refinement cycle, these predicted error estimates allow us to decide whether the choice of adaptation in the previous cycle was justified, and provide a criterion for the choice in the next cycle [53].

In general, p-refinement is favorable over h-refinement in smooth regions of the finite element approximation [9, Thm. 3.4]. Thus, estimating its smoothness provides a suitable decision indicator for hp-adaptation. For this purpose, we express the finite element approximation in an orthogonal basis of increasing frequency, and consider the decay of their expansion coefficients as the estimation of smoothness. This has been implemented for both Fourier coefficients [14] and Legendre coefficients [26, 40, 41, 52].

2.4 Support for particle-based methods

Support for particles was originally introduced in `deal.II` version 9.0. These particles can be used as passive tracers, or as part of more complex models such as those based on Particle-In-Cells (PIC) approaches [28].

With the current release, support for particles has been further expanded: New parallel insertion mechanisms and a basic interface to post-process particles have been added, effectively making their usage more flexible and enabling a larger range of use cases (such as the immersed boundaries in step-70, see below).

Through the addition of the `Particles::ParticleHandler::insert_global_particles()` member function, particles can now be inserted in parallel from a vector of points even if these points do not lie on the subdomain from which the insertion is called. This operation requires extensive communication between the processes to locate the MPI process that owns the cell in which the particle is located. However, it is made significantly faster through the usage of bounding boxes that provide a coarse description of the geometrical shape of each subdomain. This function also takes care of transferring the properties attached to the particles to their new owner. This new capability enables particle generators that insert particles at the location of the support points (`Particles::Generator::dof_support_points()`) and at the quadrature points (`Particles::Generator::quadrature_points()`) of a possibly non-matching triangulation. Consequently, complex particle patterns can be inserted using unstructured grids generated outside of `deal.II`.

To visualize the motion of particles, the `Particles::DataOut` class was added to the library.

2.5 Improved performance of the symbolic differentiation framework

In the previous release we added support for symbolic expressions, leveraging the SymEngine library [63]. Although effective, evaluating lengthy expressions could be a bottleneck as this was performed using dictionary-based substitution. We have improved on this by implementing a `BatchOptimizer` class in the namespace `Differentiation::SD` that collects several `Expressions` and transforms them in such a way that the equivalent result is returned through a quicker code path. This may be done by simply using common subexpression elimination (CSE) for the dictionary-based expressions, by transformation to a set of nested `std::function` objects (the equivalent to `SymPy`'s 'lambdify', with or without using CSE), or by offloading these expressions to the `LLVM` just-in-time (JIT) compiler. Although each of these features is implemented and tested in the SymEngine library itself, the `BatchOptimizer` class provides both a uniform interface to their classes and a convenient interface for scalar expressions, as well as tensorial expressions formed using the `deal.II` tensor and symmetric tensor classes. It, like the `Expression` class, is also serializable.

The way the batch optimizer may be employed within a user's code is shown in the pseudo-code below. As per usual, one would first define some independent variables, and subsequently compute some symbolic expressions that are dependent on these independent variables. These expression could be, for example, scalar expressions or tensors of expressions. Instead of evaluating these expressions directly, the user would now create an optimizer to evaluate the dependent functions. In this example, the selected arithmetic type numerical result will be of type `double`, and the `LLVM` JIT optimizer will be invoked. It will employ common subexpression elimination and aggressive optimizations during compilation. The user then informs the optimizer of all of the independent variables and the dependent expressions, and invokes the optimization process. This is an expensive call, as it determines an equivalent code path to evaluate all of the dependent functions at once. However, in many cases each evaluation has significantly less computational cost than evaluating the symbolic expressions directly. Evaluation is performed when the user constructs a substitution map, giving each independent variable a numerical representation, and passes those to the optimizer. After this step, the numerical equivalent of the individual dependent expressions may finally be retrieved from the optimizer.

C++ code

```cpp
using namespace Differentiation::SD;

const Expression x("x");
const Expression y("y");
...
const auto f =calculate_f(x, y, ...); // User function
const auto g =calculate_g(x, y, ...); // User function
...

BatchOptimizer<double> optimizer (OptimizerType::llvm,
OptimizationFlags::optimize_all);
optimizer.register_symbols(x, y, ...);
optimizer.register_functions(f, g, ...);
optimizer.optimize();

const auto substitution_map
=make_substitution_map({x, ...}, {y, ...}, ...);
optimizer.substitute(substitution_map);

const auto result_f =optimizer.evaluate(f);
const auto result_g =optimizer.evaluate(g);
```

This expense of invoking the optimizer may be offset not only by the number of evaluations performed, but also by the amount of reuse each instance of a `BatchOptimizer` has. In certain circumstances, this can be maximized by generalizing the way in which the dependent expressions are formulated. For example, in the context of constitutive modelling the material coefficients may be made symbolic rather than encoding these into the dependent expressions as numerical values. The optimizer may then be used to evaluate an entire family of constitutive laws, and not a specific one that describes the response of a single material. Thereafter, serializing the optimizer instance and reloading the contents during subsequent simulations permits the user to skip the optimization process entirely. Serialization also enables these complex expressions to be compiled offline.

2.6 Advances in SIMD capabilities and the matrix-free infrastructure

The class `VectorizedArray<Number>` is a key component to achieve the high node-level performance of the matrix-free algorithms in `deal.II` [45, 46]. It is a wrapper class around a short vector of n entries of type `Number` and maps arithmetic operations to appropriate single-instruction/multiple-data (SIMD) concepts by intrinsic functions. The class `VectorizedArray` has been made more user-friendly in this release by making it compatible with the STL algorithms found in the header `<algorithm>`. The length of the vector can now be queried by `VectorizedArray::size()` and its underlying number type by `VectorizedArray::value_type`. The `VectorizedArray` class now supports range-based iteration over its entries. In addition `deal.II` now supports ternary operations on vectorized data, where for a given binary comparison operation a true or false

Tab. 1: Supported vector lengths of the class `VectorizedArray` and the corresponding instruction-set-architecture extensions.

double	float	ISA
VectorizedArray<double, 1>	VectorizedArray<float, 1>	(auto-vectorization)
VectorizedArray<double, 2>	VectorizedArray<float, 4>	SSE2/AltiVec
VectorizedArray<double, 4>	VectorizedArray<float, 8>	AVX/AVX2
VectorizedArray<double, 8>	VectorizedArray<float, 16>	AVX-512

Tab. 2: Comparison of relevant SIMD-related classes in `deal.II` and `C++23`.

VectorizedArray (`deal.II`)	std::simd (`C++23`)
VectorizedArray<Number>	std::experimental::native_simd<Number>
VectorizedArray<Number, size>	std::experimental::fixed_size_simd<Number, size>

value is selected. For example, the vectorized equivalent of `(left < right) ? true_value : false_value` can be expressed as

C++ code

```cpp
auto result =compare_and_apply_mask<SIMDComparison::less_than>(
left, right, true_value, false_value);
```

which compares every element of the vectorized array individually and selects the corresponding value from the `true_value`, or `false_value` array.

In previous `deal.II` releases, the vector length was set at compile time of the library to match the highest value supported by the given processor architecture. Now, a second optional template argument can be specified as `VectorizedArray<Number, size>`, where `size` explicitly controls the vector length within the capabilities of a particular instruction set. (A full list of supported vector lengths is presented in Table 1.) This allows users to select the vector length/ISA and, as a consequence, the number of cells to be processed at once in matrix-free operator evaluations. For example, the `deal.II`-based library `hyper.deal` [55], which solves the 6D Vlasov–Poisson equation with high-order discontinuous Galerkin methods (with more than a thousand degrees of freedom per cell), constructs a tensor product of two `MatrixFree` objects of different SIMD-vector length in the same application and benefits—in terms of performance—by the possibility of decreasing the number of cells processed by a single SIMD instruction.

The new interface of `VectorizedArray` also enables replacement by any type with a matching interface. Specifically, this prepares `deal.II` for the `std::simd` class that is slated to become part of the `C++23` standard. Table 2 compares the `deal.II`-specific SIMD classes and the equivalent `C++23` classes. These changes also prepare for specialized code paths exploiting vectorization within an element, see [46].

2.7 Advances in GPU support

For this release, the most noteworthy improvements in GPU support are the simplification of the kernel written by user, improvement of error messaging, and overlapping of computation and communication when using CUDA-aware MPI with the matrix-free framework.

In order to simplify user code, we now recompute local degree of freedom and quadrature point indices instead of having the user keeping track of them. A new option to overlap computation and communication is now available when using CUDA-aware MPI and matrix-free. The underlying idea is that only the degrees of freedom (DoFs) on the boundary of the local domain require communication. Before evaluating the matrix-free operator at these degrees of freedom, we need to communicate ghost DoFs from other processors. Similarly once the operator has been evaluated, we need to update the resulting global vector with values from other processors. The strategy that we are now using consists in splitting the DoFs into three groups: one group of DoFs that are on the local boundary and two groups each owning half of the interior DoFs. When evaluating the matrix-free operator, we start the MPI communication to get the ghosted DoFs and evaluate the operator on one of the two interior DoFs group. When the evaluation is done, we wait for the MPI communication to be over, and then evaluate the operator on the boundary DoFs group. When this evaluation is completed, we start the communication to update the global vector, we evaluate the operator on the last group of DoFs, and finally wait for the MPI communication to finish.

2.8 Expanded use of C++11 facilities

Certain types of quantities in a simulation are constants fully known at compile time. They can be precalculated and stored in the compiled executable in order to avoid unnecessary initialization during runtime. C++11 and later standards enable such computations by marking variables and functions with the `constexpr` keyword.

This optimization is now enabled for the class templates `Tensor` and `SymmetricTensor`. For instance, the linear mechanical constitutive model for isotropic elastic solids uses a constant fourth-order elasticity tensor $\mathbb{C} = \lambda \boldsymbol{I} \otimes \boldsymbol{I} + 2\mu\mathbb{I}$ which does not depend on the current state of strain. This tensor can now be statically initialized by defining it as `constexpr SymmetricTensor<4, dim>`. As another example, the lattice vectors in a crystal plasticity model are generally constant and known during compilation time, enabling their efficient definition as `constexpr Tensor<1, dim>`.

Declaring variables, functions, and methods as `constexpr` is a C++11 feature that was later expanded by the C++14 standard. Thus, parts of the `constexpr` support in `deal.II` depend on the C++ standard supported by the compiler used to install the library.

The next release of `deal.II` will require compiler support for the C++14 standard.

2.9 New and improved tutorial and code gallery programs

Many of the `deal.II` tutorial programs were revised in a variety of ways as part of this release. A particular example is that we have converted a number of programs to use range-based for loops (a C++11 feature) for loops over a range of integer indices such as loops over all quadrature points or all indices of degrees of freedom during assembly. This makes sense given that the range-based way of writing loops seems to be the idiomatic approach these days, and that we had previously already converted loops over all cells in this way.

In addition, there are a number of new tutorial programs:

- `step-47` is a new program that solves the biharmonic equation $\Delta^2 u = f$ with 'clamped' boundary condition given by $u = g$, $\partial u/\partial n = h$. This program is based on the C^0 interior penalty (C^0IP) method for fourth order problems [17]. In order to overcome shortcomings of classical approaches, this method uses C^0 Lagrange finite elements and introduces 'jump' and 'average' operators on interfaces of elements that penalize the jump of the gradient of the solution in order to obtain convergence to the H^2-regular solution of the equation.
  The C^0IP approach is a modern alternative to classical methods that use C^1-conforming elements such as the Argyris element, the Clough–Tocher element, and others, all developed in the late 1960s. From a twenty-first century perspective, they can only be described as bizarre in their construction. They are also exceedingly cumbersome to implement if one wants to use general meshes. As a consequence, they have largely fallen out of favor and `deal.II` currently does not contain implementations of these shape functions.
- `step-50` is a program that demonstrates the parallel geometric multigrid features in `deal.II` as described in [20]. The problem considered is a variable viscosity Laplace equation and it is solved with three different approaches: (i) using a matrix-based geometric multigrid based on Trilinos or PETSc; (ii) using a matrix-free geometric multigrid; (iii) using algebraic multigrid (Trilinos ML). The tutorial demonstrates the superiority of the matrix-free method for the problem under consideration, and shows that, for matrix-

based formulations, the performance of algebraic and geometric multigrid methods are roughly comparable.

– `step-58` is a program that solves the nonlinear Schrödinger equation, which in non-dimensional form reads

$$-\mathrm{i}\frac{\partial\psi}{\partial t} - \frac{1}{2}\Delta\psi + V\psi + \varkappa|\psi|^2\psi = 0$$

augmented by appropriate initial and boundary conditions and using an appropriate form for the potential $V = V(\mathbf{x})$. The tutorial program focuses on two specific aspects for which this equation serves as an excellent test case: (i) Solving complex-valued problems without splitting the equation into its real and imaginary parts (as `step-29` does, for example). (ii) Using operator splitting techniques. The equation is a particularly good test case for this technique because the only nonlinear term, $\varkappa|\psi|^2\psi$, does not contain any derivatives and consequently forms an ODE at each node point, to be solved in each time step in an operator splitting scheme (for which, furthermore, there exists an analytic solution), whereas the remainder of the equation is linear and easily solved using standard finite element techniques.

– `step-65` presents `TransfiniteInterpolationManifold`, a manifold class that can propagate curved boundary information into the interior of a computational domain by transfinite interpolation [32]. This manifold is a prototype for many other manifolds in that it is relatively expensive to compute the new points, especially for higher order mappings. Since typical programs query higher-order geometries in a large variety of contexts, the contribution of the mapping to the run time can be significant. As a solution, the tutorial also presents the class `MappingQCache`, which samples the information of expensive manifolds in the points of a `MappingQ` and caches it. The tutorial shows that this makes all queries to the geometry very cheap.

– `step-67` is an explicit time integrator for the compressible Euler equations discretized with a high-order discontinuous Galerkin (DG) scheme using the matrix-free infrastructure. Besides the use of matrix-free evaluators for systems of equations and over-integration, it also presents `MatrixFreeOperators::CellwiseInverseMassMatrix`, a fast implementation of the action of the inverse mass matrix in the DG setting using tensor products. Furthermore, this tutorial demonstrates the usage of new pre and post operations, which can be passed to `MatrixFree::cell_loop()`, to schedule operations on sections of vectors close to the matrix-vector product to increase data locality.

– `step-69` presents a first-order scheme solving the compressible Euler equations of gas dynamics with a graph-viscosity stabilization technique. Beside the usual conservation properties of mass, momentum, and total energy, the method also guarantees that the constructed solution obeys pointwise stability constraints (in particular positivity of density and internal energy, and a local minimum principle on the specific entropy). As such step-69 is strictly speaking more a collocation-type discretization than a variational formulation, even though it is implemented with finite elements.
The time-update at each node requires the evaluation of a right-hand side that depends (nonlinearly) on information from the previous time-step that spans more than one cell. Therefore, assembly loops operate directly on the sparsity graph in order to retrieve information from the entire stencil associated with each node. From a programming perspective, `step-69` features a number of techniques that are of interest for a wider audience: It discusses a hybrid thread and MPI parallelized scheme with efficient MPI node-local numbering of degrees of freedom. It showcases how to perform asynchronous write-out of results using a background thread with `std::async`, and discusses a simple but effective checkpointing and restart technique.

– `step-70` solves a fluid structure interaction problem on non-matching parallel distributed triangulations, showing the usage of the `particles::ParticleHandler` class for two different tasks: (i) to track the position of quadrature points of a non-matching solid grid immersed in a fluid grid, and (ii) to track the position of a collection of massless tracers.
The program considers a mixing problem in the laminar flow regime. Such problems occur in a wide range of applications ranging from chemical engineering to power generation (e.g. turbomachinery). Mixing problems are particularly hard to solve numerically, because they often involve a container (with fixed boundaries, and possibly complex geometries such as baffles), and one (or more) immersed and rotating

impellers, making it difficult to use Arbitrary Lagrangian Eulerian formulations, where the fluid domain — along with the mesh! — is smoothly deformed to follow the deformations of the immersed solid. We show how to solve such problems using a penalization technique where the flow problem is solved in the union of the fluid and solid domain, and the movement of the solid is imposed weakly using a Nitsche-like penalization.
The `particles::ParticleHandler` class is used in this context to allow integration of the fluid basis functions on the solid domain (which is immersed and non-matching with regard to the fluid grid). This is achieved by attaching the information required to perform the integration on the solid grid, to the property field of the particles associated with the quadrature points of the solid, and advecting those particles according to the solid velocity. In this program, we externally prescribe rather than solve for the the solid velocity; however, the program can easily be extended to allow for the solution of coupled elasticity equations on the solid domain.

There are also new programs in the code gallery (a collection of user-contributed programs that often solve more complicated problems than tutorial programs, and intended as starting points for further research rather than as teaching tools):
– A program based on the XBraids library to solve time-dependent problems in a parallel-in-time fashion. This program was contributed by Joshua Christopher.
– A program that solves the biphasic nonlinear poro-viscoelasticity equations based on Ogden hyperelasticity, to explore the porous and viscous contributions in brain mechanics applications. This program was contributed by Ester Comellas and Jean-Paul Pelteret. The program also serves as a demonstration of the automatic differentiation capabilities of `deal.II`.

2.10 Python interfaces

Initial support for Python has existed in `deal.II` since version 9.0. The present release significantly extends the Python interface. Specifically, a large number of methods from classes such as `Triangulation`, `CellAccessor`, `TriaAccessor`, `Mapping`, `Manifold`, `GridTools` can now be invoked from Python. We have focused on methods and functions that are widely used when a mesh is created and parameters related to the boundary, manifold, and material identifiers are assigned. The following listing gives an idea of how such code looks:

Python code

```
import PyDealII.Release as dealii

triangulation =dealii.Triangulation('2D')
triangulation.generate_hyper_shell(center =dealii.Point([0, 0]),
inner_radius =0.5, outer_radius =1.,
n_cells =0, colorize =True)

triangulation.refine_global(2)

for cell in triangulation.active_cells():
cell.material_id =1 if cell.center().x >0 else 2

for face in cell.faces():
if face.at_boundary() and face.boundary_id ==1:
cell.refine_flag ='isotropic'

triangulation.execute_coarsening_and_refinement()
```

The mesh that results from this code is shown in Fig. 2.

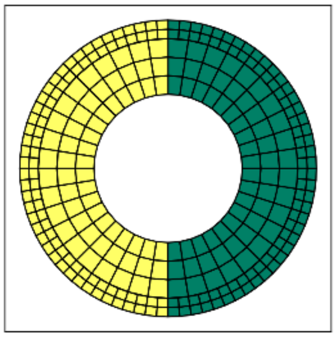

Fig. 2: The mesh generated by the Python code shown in the main text. Cells are colored by material id.

All triangulations created from within Python are serial. However, once the mesh is designed, the triangulation can be serialized along with the auxiliary information about possible refinement, boundaries, materials, and manifolds. This object can be easily deserialized within a C++ program for subsequent production runs. Furthermore, such a serialized triangulation can also be used in the construction of `parallel::shared`, `parallel::distributed`, and `parallel::fullydistributed` triangulations (see also Section 2.1).

To facilitate the illustration of the new Python bindings, tutorial programs step-49 and step-53 were replicated as Jupyter notebooks.

The introspective nature of the Python language makes it easy to infer the list of supported methods from the Python objects, for example by typing `dir(PyDealII.Release.Triangulation)`. The current Python interface does not yet provide access to `deal.II`'s finite element machinery, i.e., classes such as `DoFHandler`, `FE_*`, `FEValues`, etc.

2.11 Incompatible changes

The 9.2 release includes around 60 incompatible changes; see [49]. The majority of these changes should not be visible to typical user codes; some remove previously deprecated classes and functions; and the majority change internal interfaces that are not usually used in external applications. That said, the following are worth mentioning since they may have been more widely used:

– Two functions that provide information about all processes, namely
  – `DoFHandler::locally_owned_dofs_per_processor()`
  – `DoFHandler::locally_owned_mg_dofs_per_processor()`

  have been deprecated. As discussed in Subsection 2.2, `deal.II` by default no longer stores information for all processes on all processes, but only local or locally-relevant information. On the other hand, if necessary, global information can still be computed using, for example, calling `Utilities::MPI::Allgather(locally_owned_info(), comm)`.
– A corresponding change has been made in parallel triangulation classes: `parallel::TriangulationBase::compute_n_locally_owned_active_cells_per_processor()` can be used to obtain information about how many cells each process owns.

3 How to cite `deal.II`

In order to justify the work the developers of `deal.II` put into this software, we ask that papers using the library reference one of the `deal.II` papers. This helps us justify the effort we put into it.

There are various ways to reference `deal.II`. To acknowledge the use of the current version of the library, *please reference the present document*. For up to date information and a bibtex entry see:

https://www.dealii.org/publications.html

The original `deal.II` paper containing an overview of its architecture is [13]. If you rely on specific features of the library, please consider citing any of the following:

- For geometric multigrid: [20, 42, 43];
- For distributed parallel computing: [12];
- For *hp* adaptivity: [14];
- For partition-of-unity (PUM) and enrichment methods of the finite element space: [24];
- For matrix-free and fast assembly techniques: [45, 46];
- For computations on lower-dimensional manifolds: [25];
- For curved geometry representations and manifolds: [34];
- For integration with CAD files and tools: [34];
- For boundary element computations: [31];
- For `LinearOperator` and `PackagedOperation` facilities: [50, 51];
- For uses of the `WorkStream` interface: [65];
- For uses of the `ParameterAcceptor` concept, the `MeshWorker::ScratchData` base class, and the `ParsedConvergenceTable` class: [61];
- For uses of the particle functionality in `deal.II`: [28].

`deal.II` can interface with many other libraries:

- ADOL-C [33, 66]
- ARPACK [48]
- Assimp [62]
- BLAS and LAPACK [5]
- cuSOLVER [21]
- cuSPARSE [22]
- Gmsh [29]
- GSL [27]
- Ginkgo [30]
- HDF5 [64]
- METIS [44]
- MUMPS [2–4, 54]
- muparser [56]
- nanoflann [16]
- NetCDF [59]
- OpenCASCADE [57]
- p4est [18]
- PETSc [10, 11]
- ROL [60]
- ScaLAPACK [15]
- SLEPc [35]
- SUNDIALS [38]
- SymEngine [63]
- TBB [58]
- Trilinos [36, 37]
- UMFPACK [23]

Please consider citing the appropriate references if you use interfaces to these libraries. We note that the nanoflann and NetCDF interfaces are now deprecated and will be removed in `deal.II` version 9.3.

The two previous releases of `deal.II` can be cited as [1, 6].

4 Acknowledgments

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This document describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

`deal.II` is a world-wide project with dozens of contributors around the globe. Other than the authors of this paper, the following people contributed code to this release:

Pasquale Africa, Ashna Aggarwal, Giovanni Alzetta, Mathias Anselmann, Kirana Bergstrom, Manaswinee Bezbaruah, Benjamin Brands, Yong-Yong Cai, Fabian Castelli, Joshua Christopher, Ester Comellas, Katherine Cosburn, Denis Davydov, Elias Dejene, Stefano Dominici, Brett Dong, Luel Emishaw, Niklas Fehn, Isuru Fernando, Rebecca Fildes, Menno Fraters, Andres Galindo, Daniel Garcia-Sanchez, Rene Gassmoeller, Melanie Gerault, Nicola Giuliani, Brandon Gleeson, Anne Glerum, Krishnakumar Gopalakrishnan, Graham Harper, Mohammed Hassan, Nicole Hayes, Bang He, Johannes Heinz, Jiuhua Hu, Lise-Marie Imbert-Gerard, Manu Jayadharan, Daniel Jodlbauer, Marie Kajan, Guido Kanschat, Alexander Knieps, Uwe Köcher, Paras Kumar, Konstantin Ladutenko, Charu Lata, Adam Lee, Wenyu Lei, Katrin Mang, Mae Markowski, Franco Milicchio, Adriana Morales Miranda, Bob Myhill, Emily Novak, Omotayo Omosebi, Alexey Ozeritskiy, Rebecca Pereira, Geneva Porter, Laura Prieto Saavedra, Roland Richter, Jonathan Robey, Irabiel Romero, Matthew Russell, Tonatiuh Sanchez-Vizuet, Natasha S. Sharma, Doug Shi-Dong, Konrad Simon, Stephanie Sparks, Sebastian Stark, Simon Sticko, Jan Philipp Thiele, Jihuan Tian, Sara Tro, Ferdinand Vanmaele, Michal Wichrowski, Julius Witte, Winnifried Wollner, Ming Yang, Mario Zepeda Aguilar, Wenjuan Zhang, Victor Zheng.

Their contributions are much appreciated!

`deal.II` and its developers are financially supported through a variety of funding sources:

D. Arndt and B. Turcksin: Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U.S. Department of Energy.

W. Bangerth, T. C. Clevenger, and T. Heister were partially supported by the Computational Infrastructure in Geodynamics initiative (CIG), through the National Science Foundation under Award No. EAR-1550901 and The University of California — Davis.

W. Bangerth was also partially supported by award OAC-1835673 as part of the Cyberinfrastructure for Sustained Scientific Innovation (CSSI) program, DMS-1821210, and EAR-1925595.

B. Blais was partially supported by the National Science and Engineering Research Council of Canada (NSERC) through the RGPIN-2020-04510 Discovery Grant.

T. C. Clevenger was also partially supported EAR-1925575 and OAC-2015848.

A. V. Grayver was partially supported by the European Space Agency Swarm DISC program.

T. Heister was also partially supported by the National Science Foundation (NSF) Award DMS-2028346, OAC-2015848, EAR-1925575, and by Technical Data Analysis, Inc. through US Navy STTR Contract N68335-18-C-0011.

L. Heltai was partially supported by the Italian Ministry of Instruction, University and Research (MIUR), under the 2017 PRIN project NA-FROM-PDEs MIUR PE1, 'Numerical Analysis for Full and Reduced Order Methods for the efficient and accurate solution of complex systems governed by Partial Differential Equations'.

M. Kronbichler was supported by the German Research Foundation (DFG) under the project 'High-order discontinuous Galerkin for the exa-scale' (ExaDG) within the priority program 'Software for Exascale Computing' (SPPEXA) and the Bayerisches Kompetenznetzwerk für Technisch–Wissenschaftliches Hoch- und Höchstleistungsrechnen (KONWIHR) in the context of the project 'Performance tuning of high-order discontinuous Galerkin solvers for SuperMUC-NG'.

M. Maier was partially supported by ARO MURI Award No. W911NF-14-0247 and NSF Award DMS-1912847.

D. Wells was supported by the National Science Foundation (NSF) award OAC-1450327.

Z. Wang was partially supported by the National Science Foundation under award OAC-1835673.

The Interdisciplinary Center for Scientific Computing (IWR) at Heidelberg University has provided hosting services for the `deal.II` web page.

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing access to HPC resources that have contributed to the research results reported within this paper.

Clemson University is acknowledged for generous allotment of compute time on the Palmetto cluster.

References

[1] G. Alzetta, D. Arndt, W. Bangerth, V. Boddu, B. Brands, D. Davydov, R. Gassmoeller, T. Heister, L. Heltai, K. Kormann, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells, The `deal.II` library, Version 9.0, *J. Numer. Math.* **26** (2018), No. 4, 173–184.

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent, A fully asynchronous multifrontal solver using distributed dynamic scheduling, *SIAM J. Matrix Anal. Appl.* **23** (2001), No. 1, 15–41.

[3] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, Multifrontal parallel distributed symmetric and unsymmetric solvers, *Comput. Methods Appl. Mech. Engrg.* **184** (2000), 501–520.

[4] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, Hybrid scheduling for the parallel solution of linear systems, *Parallel Computing* **32** (2006), No. 2, 136–156.
[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, third ed, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.
[6] 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, *J. Numer. Math.* **27** (2019), No. 4, 203–213.
[7] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells, The deal.II finite element library: Design, features, and insights, *Computers & Mathematics with Applications* **in press** (2020).
[8] D. Arndt, N. Fehn, G. Kanschat, K. Kormann, M. Kronbichler, P. Munch, W. A. Wall, and J. Witte, ExaDG – high-order discontinuous Galerkin for the exa-scale, In: *Software for Exascale Computing – SPPEXA 2016–2019* (Eds. H.-J. Bungartz, W. E. Nagel, S. Reiz, B. Uekermann, and Ph. Neumann), Lecture Notes in Computational Science and Engineering, Vol. 136, Springer, Cham, 2020.
[9] I. Babuška and M. Suri, The *p*- and *h-p* versions of the finite element method, an overview, *Comp. Meth. Appl. Mechanics Engrg.* **80** (1990), No. 1, 5–26.
[10] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. May, L. Curfman McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Users Manual*, Argonne National Laboratory, Report No. ANL-95/11 - Revision 3.9, 2018.
[11] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. May, L. Curfman McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Web Page*, http://www.mcs.anl.gov/petsc, 2018.
[12] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, Algorithms and data structures for massively parallel generic adaptive finite element codes, *ACM Trans. Math. Softw.* **38** (2011), 14/1–28.
[13] W. Bangerth, R. Hartmann, and G. Kanschat, deal.II — a general purpose object oriented finite element library, *ACM Trans. Math. Softw.* **33** (2007), No. 4.
[14] W. Bangerth and O. Kayser-Herold, Data Structures and Requirements for *hp* Finite Element Software, *ACM Trans. Math. Softw.* **36** (2009), No. 1, 4/1–4/31.
[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.
[16] J. L. Blanco and P. K. Rai, *Nanoflann: a C++ Header-Only Fork of FLANN, a Library for Nearest Neighbor (NN) with KD-Trees*, https://github.com/jlblancoc/nanoflann, 2014.
[17] S. C. Brenner and L.-Y. Sung, C^0 interior penalty methods for fourth order elliptic boundary value problems on polygonal domains, *J. Sci. Comp.* **22-23** (2005), No. 1-3, 83–118.
[18] C. Burstedde, L. C. Wilcox, and O. Ghattas, `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees, *SIAM J. Sci. Comput.* **33** (2011), No. 3, 1103–1133.
[19] T. C. Clevenger and T. Heister, Comparison between Algebraic and Matrix-free Geometric Multigrid for a Stokes Problem, *submitted* (2019).
[20] T. C. Clevenger, T. Heister, G. Kanschat, and M. Kronbichler, *A Flexible, Parallel, Adaptive Geometric Multigrid Method for FEM*, arXiv:1904.03317, Report, 2019.
[21] *CuSOLVER Library*, https://docs.nvidia.com/cuda/cusolver/index.html.
[22] *CuSPARSE Library*, https://docs.nvidia.com/cuda/cusparse/index.html.
[23] T. A. Davis, Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method, *ACM Trans. Math. Softw.* **30** (2004), 196–199.
[24] D. Davydov, T. Gerasimov, J.-P. Pelteret, and P. Steinmann, Convergence study of the *h*-adaptive PUM and the *hp*-adaptive FEM applied to eigenvalue problems in quantum mechanics, *Adv. Modeling Simul. Engrg. Sci.* **4** (2017), No. 1, 7.
[25] A. DeSimone, L. Heltai, and C. Manigrasso, *Tools for the Solution of PDEs Defined on Curved Manifolds with Deal.II*, SISSA, Report No. 42/2009/M, 2009.
[26] T. Eibner and J. M. Melenk, An adaptive strategy for *hp*-FEM based on testing for analyticity, *Comp. Mechanics* **39** (2007), No. 5, 575–595.
[27] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, and R. Ulerich, *GNU Scientific Library Reference Manual (Edition 2.3)*, 2016.
[28] R. Gassmöller, H. Lokavarapu, E. Heien, E. Gerry Puckett, and W. Bangerth, Flexible and scalable particle-in-cell methods with adaptive mesh refinement for geodynamic computations, *Geochemistry, Geophysics, Geosystems* **19** (2018), No. 9, 3596–3604.
[29] C. Geuzaine and J.-F. Remacle, Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities, *Int. J. Numer. Methods Engrg.* **79** (2009), No. 11, 1309–1331.
[30] *Ginkgo: High-Performance Linear Algebra Library for Manycore Systems*, https://github.com/ginkgo-project/ginkgo.

[31] N. Giuliani, A. Mola, and L. Heltai, π-BEM: A flexible parallel implementation for adaptive, geometry aware, and high order boundary element methods, *Adv. Engrg. Software* **121** (2018), No. March, 39–58.
[32] W. J. Gordon and L. C. Thiel, Transfinite mappings and their application to grid generation, *Appl. Math. Comput.* **10** (1982), 171–233.
[33] A. Griewank, D. Juedes, and J. Utke, Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++, *ACM Trans. Math. Software* **22** (1996), No. 2, 131–167.
[34] L. Heltai, W. Bangerth, M. Kronbichler, and A. Mola, *Using Exact Geometry Information in Finite Element Computations*, arXiv:1910.09824, Report, 2019.
[35] V. Hernandez, J. E. Roman, and V. Vidal, SLEPc: a scalable and flexible toolkit for the solution of eigenvalue problems, *ACM Trans. Math. Software* **31** (2005), No. 3, 351–362.
[36] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, An overview of the Trilinos project, *ACM Trans. Math. Softw.* **31** (2005), 397–423.
[37] M. A. Heroux et al., *Trilinos Web Page*, 2018, http://trilinos.org.
[38] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers, *ACM Trans. Math. Software* **31** (2005), No. 3, 363–396.
[39] T. Hoefler, C. Siebert, and A. Lumsdaine, Scalable communication protocols for dynamic sparse data exchange, *ACM Sigplan Notices* **45** (2010), No. 5, 159–168.
[40] P. Houston, B. Senior, and E. Süli, Sobolev regularity estimation for hp-adaptive finite element methods, In: *Numerical Mathematics and Advanced Applications* (Eds. F. Brezzi, A. Buffa, S. Corsaro, and A. Murli), pp. 631–656, Springer, Milan, 2003.
[41] P. Houston and E. Süli, A note on the design of *hp*-adaptive finite element methods for elliptic partial differential equations, *Comp. Meth. Appl. Mechanics Engrg.* **194** (2005), No. 2, 229–243.
[42] B. Janssen and G. Kanschat, Adaptive multilevel methods with local smoothing for H^1- and H^{curl}-conforming high order finite element methods, *SIAM J. Sci. Comput.* **33** (2011), No. 4, 2095–2114.
[43] G. Kanschat, Multi-level methods for discontinuous Galerkin FEM on locally refined meshes, *Comput. & Struct.* **82** (2004), No. 28, 2437–2445.
[44] G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM J. Sci. Comput.* **20** (1998), No. 1, 359–392.
[45] M. Kronbichler and K. Kormann, A generic interface for parallel cell-based finite element operator application, *Comput. Fluids* **63** (2012), 135–147.
[46] M. Kronbichler and K. Kormann, Fast matrix-free evaluation of discontinuous Galerkin finite element operators, *ACM Trans. Math. Soft.* **45** (2019), No. 3, 29:1–29:40.
[47] M. Kronbichler and W. A. Wall, A performance comparison of continuous and discontinuous Galerkin methods with fast multigrid solvers, *SIAM J. Sci. Comput.* **40** (2018), No. 5, A3423–A3448.
[48] R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, SIAM, Philadelphia, 1998.
[49] *List of Changes for 9.2*, https://www.dealii.org/developer/doxygen/deal.II/changes_between_9_1_1_and_9_2_0.html.
[50] M. Maier, M. Bardelloni, and L. Heltai, LinearOperator – a generic, high-level expression syntax for linear algebra, *Comp. Math. Appl.* **72** (2016), No. 1, 1–24.
[51] M. Maier, M. Bardelloni, and L. Heltai, *LinearOperator Benchmarks, Version 1.0.0*, 2016.
[52] C. Mavriplis, Adaptive mesh strategies for the spectral element method, *Comp. Meth. Appl. Mech. Engrg.* **116** (1994), No. 1, 77–86.
[53] J. M. Melenk and B. I. Wohlmuth, On residual-based a posteriori error estimation in *hp*-FEM, *Adv. Comp. Math.* **15** (2001), No. 1-4, 311–331.
[54] *MUMPS: a MUltifrontal Massively Parallel Sparse Direct Solver*, http://graal.ens-lyon.fr/MUMPS/.
[55] P. Munch, K. Kormann, and M. Kronbichler, *Hyper.deal: An Efficient, Matrix-Free Finite-Element Library for High-Dimensional Partial Differential Equations*, arXiv:2002.08110, Report, 2020.
[56] *Muparser: Fast Math Parser Library*, http://muparser.beltoforion.de/.
[57] *OpenCASCADE: Open CASCADE Technology, 3D Modeling & Numerical Simulation*, http://www.opencascade.org/.
[58] J. Reinders, *Intel Threading Building Blocks*, O'Reilly, 2007.
[59] R. Rew and G. Davis, NetCDF: an interface for scientific data access, *Computer Graphics and Applications, IEEE* **10** (1990), No. 4, 76–82.
[60] D. Ridzal and D. P. Kouri, *Rapid Optimization Library*, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Report, 2014.
[61] A. Sartori, N. Giuliani, M. Bardelloni, and L. Heltai, deal2lkit: A toolkit library for high performance programming in deal.II, *SoftwareX* **7** (2018), 318–327.
[62] T. Schulze, A. Gessler, K. Kulling, D. Nadlinger, J. Klein, M. Sibly, and M. Gubisch, Open asset import library (assimp), *Comp. Software* (2012), https://github.com/assimp/assimp.
[63] *SymEngine: Fast Symbolic Manipulation Library, Written in C++*, https://github.com/symengine/symengine,

http://sympy.org/.
[64] The HDF Group, *Hierarchical Data Format, Version 5*, 1997-2018, http://www.hdfgroup.org/HDF5/.
[65] B. Turcksin, M. Kronbichler, and W. Bangerth, *WorkStream* – a design pattern for multicore-enabled finite element computations, *ACM Trans. Math. Software* **43** (2016), No. 1, 2/1–2/29.
[66] A. Walther and A. Griewank, Getting started with ADOL-C, In: *Combinatorial Scientific Computing* (Eds. U. Naumann and O.Schenk), Chapman-Hall CRC Computational Science, pp. 181–202, 2012.