D1.1
External libraries integration

Project Title	dealii-X: an Exascale Framework for Digital Twins of the Human Body
Project Number	101172493
Funding Program	European High-Performance Computing Joint Undertaking
Project start date	1 October 2024
Duration	27 months

dealii-X has received funding from the European High-Performance Computing Joint Undertaking Programme under grant agreement Nº 101172493

Deliverable title	External libraries integration
Deliverable number	D1.1
Deliverable version	v1
Date of delivery	July 31, 2025
Actual date of delivery	July 29, 2025
Nature of deliverable	Dissemination, Exploitation, and Communication
Dissemination level	Public
Work Package	WP1
Partner responsible	INPT

Abstract	This document describes the integration of external libraries in the deal.II finite element library, namely the MUMPS sparse direct solver, the PSCToolkit iterative solver and the GMSH three-dimensional finite element mesh generator.
Keywords	API; solvers; mesh; parallelism; interfaces

Document Control Information

Version	Date	Author	Changes Made
0.1	15/07/2025	Alfredo Buttari	Initial draft
0.2	22/07/2025	Marco Feder	Preliminary experimental results
0.3	23/07/2025	Fabio Durastante	Integration of PSCToolkit
0.4	28/07/2025	Marco Feder	Integration of distributed GMSH capabilities
0.5	29/07/2025	Alfredo Buttari	Review and reference
0.6	29/07/2025	Marco Feder	Comments about memory consumption
1.0	29/07/2025	Martin Kronbichler	Final version

Approval Details

Approved by: M. Kronbichler
Approval Date: July 29, 2025

Distribution List

- Project Coordinators (PCs)
- Work Package Leaders (WPLs)
- Steering Committee (SC)
- European Commission (EC)

Disclaimer: This project has received funding from the European Union. The views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European High-Performance Computing Joint Undertaking (the “granting authority”). Neither the European Union nor the granting authority can be held responsible for them.

Contents

1 Introduction

The objective of the dealii-X Work Package 1 is to enhance the capabilities of the deal.II finite element library through the use of some external libraries, namely, the MUMPS and PSCToolkit solvers and the GMSH three-dimensional finite element mesh generator. MUMPS is a parallel direct solver for sparse linear systems, i.e., it achieves the solution by computing the factorization (LU, LDL^T or Cholesky, depending of the problem type) of the system matrix and using the resulting factors in forward/backward substitution operations. One of the objectives of WP1 is to assess the effectiveness of some of the recent advanced features of MUMPS such as the block low-rank approximations or the mixed-precision computations on problems issued from the simulation of human body organs. PSCToolkit implements multiple parallel iterative solvers of the Krylov family for the solution of large scale sparse linear systems. These methods are, by nature, scalable; furthermore, PSCToolkit provides efficient algebraic multigrid preconditioners to improve the convergence. Finally, WP1 deals with the integration of the GMSH three-dimensional finite element meshing package in order to optimize the mesh generation at large scale.

1.1 Purpose of the Document

This document describes the integration of the above-mentioned external packages, the corresponding API within the deal.II framework and the features that are exposed to enable experimenting with advanced features in the context of the dealii-X project.

2 MUMPS

2.1 Existing interface

The deal.II package used to support a direct MUMPS integration in the past. This support was subsequently remove in favor of an indirect use of MUMPS through the PETSc library. As a first step towards the objectives of WP1 of the dealii-X project, this support was introduced into the core library again, as documented in Pull Request #18255 (`https://github.com/dealii/dealii/pull/18255`), which has been subsequently merged in the main branch. This minimalistic interface includes a number of basic tests.

2.2 Implemented improvements in the MUMPS support

The basic deal.II MUMPS interface did not include support for distributed-memory parallelism, which means that the system matrix and right-hand side(s) were entirely assembled on the master process where the subsequent phases (symbolic analysis, factorization and backward/forward substitution) take place; not only this might be infeasible due to memory limitations but it severely limits the performance of the sparse direct solver. The MUMPS integration in deal.II was extended to support distributed memory parallelism through the use of distributed matrix and vector datatypes as defined in the deal.II PETSc and Trilinos wrappers. This allows for a completely parallel initialization of the MUMPS data structure (that includes the system matrix and the right-hand sides) prior to the symbolic analysis, numerical factorization and forward/backward substitution.

Setting up MUMPS internal configuration parameters was not possible in the original deal.II MUMPS interface which, therefore, had to be extended to enable the use of the more advanced features of the MUMPS solver; these include the Block Low-Rank approximations, the GPU support and numerous other parameters that allow for fine-tuning both the performance and the numerical robustness of the solver. This extension was achieved by adding a `AdditionalData` object to the constructor of the MUMPS solver class that takes the desired values for selected solver parameters, following the common design patterns of the deal.II library.

The work related to above-mentioned extension is documented in pull request

#18497 (https://github.com/dealii/dealii/pull/18497) which was eventually merged in the master deal.II branch.

Furthermore, the deal.II documentation was updated with a detailed description of the MUMPS API within deal.II:

https://dealii.org/developer/doxygen/deal.II/classSparseDirectMUMPS.html

2.3 Preliminary experimental results

The new MUMPS interface has been validated through a series of tests to verify its correctness, using both serial and distributed sparse matrices. Such tests have been added to the deal.II test suite and are run automatically at each commit. In particular, the following additional steps have been carried out to validate the robustness of the current interface:

- Usage of Block-Low Rank [1] approximations as a preconditioner for a diffusion problem in 2D and 3D, on a sequence of parallel and adaptively refined meshes, both for PETSc and Trilinos distributed matrices, thereby confirming the correctness of the distributed memory parallelism support and the capabilities of BLR as a preconditioner.
- Integration into the *Brain application*, as added in pull request https://github.com/BRAINIACS-Group/ExaBrain/pull/1, serving as a drop-in replacement for the previous solver based on `Amesos_Superludist` from Trilinos. Preliminary results across several parameter configurations show that the new MUMPS solver reduces the total solver time by a factor ranging from 2 up to approximately 3.

References

[1] P. Amestoy, A. Buttari, J.-Y. L'Excellent, T. Mary: "Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures". *ACM Transactions on Mathematical Software*, 2019, 45 (1), pp.1-23. doi:10.1145/3242094

3 PSCToolkit

PSCToolkit is a high-performance parallel software package designed for solving large-scale sparse linear systems that commonly arise in scientific computing applications. It provides a comprehensive collection of Krylov subspace iterative methods, including the Conjugate Gradient (CG), the Generalized Minimal Residual (GMRES), and the BiConjugate Gradient Stabilized (BiCGSTAB) methods, among others. One of the key strengths of PSCToolkit lies in its efficient implementation of algebraic multigrid (AMG) preconditioners, which are used to significantly improve the convergence rate of iterative solvers for challenging problems such as those arising from discretized partial differential equations. The library is specifically designed to leverage modern parallel computing architectures, including both CPU and GPU systems, making it well-suited for exascale computing environments. PSCToolkit's modular design allows for easy integration with existing finite element frameworks while providing fine-grained control over solver parameters and preconditioning strategies.

PSCToolkit [3] is made of two main components:

PSBLAS A parallel sparse linear algebra library that provides basic operations for sparse matrices and vectors [1];

AMG4PSBLAS A suite of advanced algebraic multigrid and domain decomposition preconditioners specifically designed for use with PSBLAS [2].

The two libraries are designed to work together, with PSBLAS providing the foundational linear algebra operations and AMG4PSBLAS building on top of that to provide efficient preconditioning techniques. They are implemented in modern Fortran for the bulk of the code, with few routines of AMG4PSBLAS written in C, and a module of C/C++ CUDA code for GPU acceleration. To perform the integration with the deal.II library we exploit the already available C interfaces.

3.1 Improvement in the new RC for PSBLAS and AMG4PSBLAS

To prepare for the integration with deal.II, we have developed a new release candidate of PSBLAS and AMG4PSBLAS that includes several modifications to the han-

dling of the preprocessor directives. These modifications aim to improve compatibility and ease of use when integrating with other libraries and frameworks. Specifically, we have made the following changes:

- The preprocessor directives have been updated to have a `PSB_` prefix, which helps to avoid conflicts with other libraries and ensures that the directives are clearly associated with PSBLAS.
- The preprocessor directives have been moved to two dedicated header files for both PSBLAS and AMG4PSBLAS which are created at the compile time of the two libraries and are the `psb_config.h` and `amg_config.h` files, respectively. This allows for better organization and management of the preprocessor directives, making it easier to maintain and update them in the future.

As part of an ongoing effort to improve the ease of installation of the PSCToolkit libraries, the new RC also includes a new CMake module that simplifies the process of finding and linking the PSBLAS and AMG4PSBLAS libraries in CMake-based projects. Furthermore, we have added PSBLAS inside the Spack package manager, which allows for easy installation and management of the library in various environments; see https://packages.spack.io/package.html?name=psblas. We expect to add the AMG4PSBLAS library to Spack after the new release candidate is finalized; by the end of 2025.

Another addition to the RC stream was the introduction of the C interface to enable the creation of a parallel context in PSCToolkit from a pre-existing MPI communicator. This capability was already natively available in Fortran, but was not available from the C interfaces. Specifically, we have added the function:

```
void psb_c_init_from_fint(psb_c_ctxt *cctxt, psb_i_t f_comm);
```

The two RCs of PSBLAS and AMG4PSBLAS are available at the following links:

- PSBLAS: https://github.com/sfilippone/psblas3/archive/refs/tags/v3.9.0-rc3.tar.gz,
- AMG4PSBLAS: https://github.com/sfilippone/amg4psblas/archive/refs/tags/v1.2.0-rc3.tar.gz.

3.2 The interface with deal.II

The interface with deal.II we describe in the following is reported in the pull request https://github.com/dealii/dealii/pull/18662, which has yet to be merged in the main branch as of writing this document. The bulk of the interface is implemented in the two files

- `dealii/source/lac/psblas.cc`,
- `dealii/include/deal.II/lac/psctoolkit.h`,

in addition to the modification to the `CMakeLists.txt` files and scripts to enable the compilation of the new interface against the new PSCToolkit release candidate.

The interface is organized into four main namespaces: **Communicator** for MPI communication context management and descriptor operations, **Matrix** for sparse matrix creation and manipulation, **PSBVector** for vector operations and data distribution, and **Solvers** for preconditioners and Krylov solvers. The interface is conditionally compiled when `DEAL_II_WITH_PSBLAS` is defined, ensuring compatibility only when PSBLAS is available.

Communicator Operations: The `Communicator` namespace provides essential functions for managing PSBLAS communication contexts and descriptors. The basic initialization function `Init()` creates a new PSBLAS context over `MPI_COMM_WORLD` and sets the index base to 0 for C-style indexing. For integration with deal.II's MPI management, `InitFromMPI(MPI_Comm comm)` allows initialization from an existing MPI communicator, particularly useful when integrating with deal.II's `Utilities::MPI::MPI_InitFinalize` class. The function performs proper error checking for null communicators and converts the MPI communicator to Fortran-compatible format before initialization.

Listing 1: Core communicator functions

```
psb_c_ctxt *Init();
psb_c_ctxt *InitFromMPI(MPI_Comm comm);
void Info(psb_c_ctxt *cctxt, int *iam, int *nproc);
void Barrier(psb_c_ctxt *cctxt);
void Exit(psb_c_ctxt *cctxt);
```

The `Info` function retrieves the current process rank and total number of processes,

`Barrier` synchronizes all processes in the PSBLAS context, and `Exit` properly deallocates the context and associated resources; this function has to be used only if the PSBLAS context was created using `Init()` and not `InitFromMPI()`, since for the latter the context is handled by the deal.II MPI management and should not be explicitly freed.

Descriptors in PSBLAS define the data distribution pattern across processes, specifying which global indices are owned by each process. The `CreateDescriptor` function integrates with deal.II's `IndexSet` class, converting the locally owned indices into PSBLAS-compatible format through a process that extracts indices using `get_index_vector()`, converts deal.II's `types::global_dof_index` to PSBLAS's `psb_l_t`, allocates the descriptor using `psb_c_cdall_vl`, and ensures proper memory management with automatic cleanup.

Listing 2: Descriptor management functions

```
psb_c_descriptor *CreateDescriptor(IndexSet &index_set, psb_c_ctxt
    cctxt);
int DescriptorAssembly(psb_c_descriptor *cd);
int DescriptorFree(psb_c_descriptor *cd);
```

Matrix Operations: The `Matrix` namespace provides comprehensive functionality for creating, populating, and managing sparse matrices in PSBLAS format. The `CreateSparseMatrix` function creates a new double-precision sparse matrix and initializes it with the provided descriptor, allocating the matrix in remote format for efficient parallel assembly. Direct value insertion is supported through `InsertValue`, which allows insertion of matrix entries specified by row indices, column indices, and values.

Listing 3: Matrix creation and value insertion

```
psb_c_dspmat* CreateSparseMatrix(psb_c_descriptor *cd);
int InsertValue(psb_i_t nz, const psb_l_t *irw, const psb_l_t *icl,
                const psb_d_t *val, psb_c_dspmat *mh, psb_c_descriptor
                    *cdh);
```

The interface provides two overloaded `distribute_local_to_global` functions that integrate with deal.II's finite element assembly process. The first handles matrix-only distribution, while the second handles combined matrix and vector distribution. These functions perform the critical task of adding local finite element contributions to the correct positions in the global sparse matrix structure by converting deal.II's

local DOF indices to PSBLAS format, flattening the dense local matrix into coordinate format (COO), allocating memory for index and value arrays, performing parallel insertion using PSBLAS routines, and ensuring automatic memory cleanup.

Listing 4: Local-to-global distribution functions

```
int distribute_local_to_global(
    const std::vector<types::global_dof_index>& local_dof_indices,
    const FullMatrix<double>& cell_matrix,
    psb_c_dspmat *mh, psb_c_descriptor *cdh);

int distribute_local_to_global(
    const std::vector<types::global_dof_index>& local_dof_indices,
    const FullMatrix<double>& cell_matrix,
    const Vector<double>& cell_rhs,
    psb_c_dspmat *mh, psb_c_dvector *vec, psb_c_descriptor *cdh);
```

The selection of the actual sparse matrix format is then handled on the PSBLAS side.

Matrix assembly and management are handled through `AssembleSparseMatrix`, which checks if the matrix is already assembled to avoid redundant operations, and `FreeSparseMatrix`, which properly deallocates matrix resources.

Vector Operations: The `PSBVector` namespace manages distributed vectors, providing functionality for creation, data distribution, and assembly. The function `CreateVector` creates new vectors initialized with descriptors, while `AssembleVector` finalizes vector assembly and `FreeVector` handles proper deallocation.

The `distribute_local_to_global` function for vectors distributes local contributions from the finite element assembly of deal.II to the global PSBLAS vector, handling the conversion between deal.II and PSBLAS data formats.

Listing 5: Vector operations

```
psb_c_dvector* CreateVector(psb_c_descriptor *cd);
int AssembleVector(psb_c_dvector *vector, psb_c_descriptor *cd);
int FreeVector(psb_c_dvector *vector, psb_c_descriptor *cd);
void distribute_local_to_global(
    const std::vector<types::global_dof_index>& local_dof_indices,
    const Vector<double>& cell_rhs,
    psb_c_dvector *vec, psb_c_descriptor *cdh);
```

Solver Infrastructure: The `Solvers` namespace provides high-level interfaces

to PSBLAS's preconditioners and Krylov solvers. Preconditioner management includes creation through `CreateBasePreconditioner` with support for various types, building through `BasePrecBuild`, and cleanup through `FreeBasePreconditioner`.

Listing 6: Preconditioner and solver management

```
psb_c_dprec* CreateBasePreconditioner(psb_c_ctxt cctxt, const char *
    ptype);
int BasePrecBuild(psb_c_dprec *ph, psb_c_dspmat *mh, psb_c_descriptor
    *cdh);
int FreeBasePreconditioner(psb_c_dprec *ph);
```

Solver options are managed through a dedicated structure `psb_c_SolverOptions` containing iteration counts, maximum iterations, trace frequency, restart parameters, stopping criteria, convergence tolerance, and final error values. The interface provides comprehensive management through creation, configuration, printing (both to log and file), and cleanup functions. Default values are chosen: maximum iterations of 1000, convergence tolerance of 10^{-6}, relative residual norm stopping criterion $\|\mathbf{r}\|_2/\|\mathbf{b}\|_2$, GMRES restart of 30, and trace output every iteration.

Listing 7: Solver options structure and management

```
typedef struct {
    int iter, itmax, itrace, irst, istop;
    double eps, err;
} psb_c_SolverOptions;

psb_c_SolverOptions* CreateSolverOptions();
void SetSolverOptions(psb_c_SolverOptions *opt, int itmax, int itrace,
                      int irst, int istop, double eps);
```

The core solver interface is provided through `BaseKrylov`, which provides access to PSBLAS's Krylov solvers supporting methods such as CG for symmetric positive definite systems, GMRES for general systems, BICGSTAB for bi-conjugate gradient stabilized, and BICGSTABL with restart parameters.

Listing 8: Krylov solver interface

```
int BaseKrylov(const char *method, psb_c_dspmat *ah, psb_c_dprec *ph,
               psb_c_dvector *bh, psb_c_dvector *xh,
               psb_c_descriptor *cdh, psb_c_SolverOptions *opt);
```

Error Handling and Integration: The PSCToolkit interface implements comprehensive error handling throughout all operations with null pointer validation before

accessing PSBLAS objects, preventing segmentation faults and providing meaningful error messages. Every PSBLAS function call is monitored for error return codes, with detailed error reporting through deal.II's logging system (`deallog`). The interface ensures proper memory management with automatic cleanup of temporary arrays, proper deallocation of PSBLAS objects, and protection against double-free operations. against double-free operations.

The interface is designed to be integrated with deal.II's existing infrastructure through direct support for deal.II's `IndexSet` class enabling natural integration with parallel data distribution mechanisms, acceptance of deal.II's `FullMatrix<double>` and `Vector<double>` objects directly without manual data conversion, and compatibility with deal.II's MPI initialization through `Utilities::MPI::MPI_InitFinalize` ensuring consistent parallel environment management.

Usage: A typical PSCToolkit usage pattern follows the sequence of initializing the PSBLAS context from MPI communicator, creating descriptors to define data distribution using deal.II's IndexSet, assembling descriptors to finalize the distribution pattern, creating matrices and vectors, distributing local contributions through finite element assembly loops, assembling global objects to finalize parallel assembly, setting up preconditioners, configuring solvers with convergence criteria and methods, solving the system through Krylov solvers, and cleaning up all allocated resources.

Performance considerations include minimizing dynamic memory allocation during assembly by pre-allocating arrays based on degrees of freedom per cell, leveraging PSBLAS's native parallel assembly routines to minimize communication overhead, and ensuring optimal numerical performance through direct integration with PSBLAS's optimized sparse matrix formats.

Testing: The interface has been validated through a series of tests to verify its correctness within the deal.II framework. Test covers the creation of PSBLAS contexts, descriptors, matrices, and vectors, as well as the distribution of local contributions from deal.II's finite element assembly to the global PSBLAS matrix and vector structures. These tests ensure that the interface correctly handles parallel assembly, data distribution, and solver operations.

Extensions: Future extensions will include integration with AMG4PSBLAS for algebraic multigrid preconditioning through additional wrapper functions, and exposure of PSBLAS's GPU capabilities through additional configuration options and

specialized matrix formats.

References

[1] S. Filippone and M. Colajanni, ”PSBLAS: A library for parallel linear algebra computation on sparse matrices,” ACM Transactions on Mathematical Software, vol. 26, no. 4, pp. 527-550, 2000.

[2] P. D’Ambra, F. Durastante, and S. Filippone, ”AMG preconditioners for linear solvers towards extreme scale,” SIAM Journal on Scientific Computing, vol. 43, no. 5, pp. S679-S703, 2021.

[3] P. D’Ambra, F. Durastante, and S. Filippone. ”Parallel Sparse Computation Toolkit.” Software Impacts 15 (2023): 100463.

4 GMSH

4.1 Existing interface

The `GridIn` class in deal.II provides functionality for reading mesh data from several formats, allowing users to read and to import complex geometries into deal.II for their simulations. Among the many supported formats, the Gmsh format [1] is particularly useful for its ability to handle complex geometries and its widespread use in the scientific computing community. So far, external meshes are first read as serial triangulations, and then partitioned across all participating processes. From these objects, an object of type `parallel::fullydistributed::Triangulation` (hereafter abbreviated as `p::f::T`) can be created, which stores the relevant part of the mesh on each MPI process.

On the other hand, the Gmsh library provides the capability to partition a given mesh into multiple subdomains. To give a concrete example, if a user wants to partition their mesh file, named `brain.msh`, into N multiple subdomains, they can use the Gmsh command line switches as follows:

Listing 9: Partitioning a .msh file via command line switches

```
gmsh brain.msh -part N -part_ghosts -part_split -save brain.msh
```

This would generate N files, `brain_1.msh`, `brain_2.msh`, ..., `brain_N.msh`, each one describing mesh partitions, including the relevant ghost information for each partition. Sample partitions generated when N=4 for the brain mesh are shown in Figure 1.

4.2 Improvements in handling partitioned meshes

The ability to *directly* create a `p::f::T` object based on the mesh partitions defined in partitioned Gmsh files was not previously supported by deal.II. This task can be achieved by exploiting suitable calls to the Gmsh API, so that Gmsh can be directly embedded in the source code. More precisely, they allow local and ghost information for every partition to be retrieved from the Gmsh files, and the necessary data structures needed to create a `p::f::T` object can be constructed locally to each

process. In this way, it is possible to avoid loading the entire mesh. An example of this new workflow is sketched in Listings 10.

Furthermore, Table 1 displays, for the `brain.msh` file and its 4 partitions, the size (in MB) of the original and partitioned mesh files produced through the Gmsh command line interface. The memory consumption associated with the triangulation data structure in deal.II, both for the non-partitioned case and for the partitioned case, is shown in Table 2, highlighting how the creation of the triangulation data structure from the partitioned mesh files allows for a more efficient use of resources, as only the relevant parts of the mesh are loaded into memory on each process.

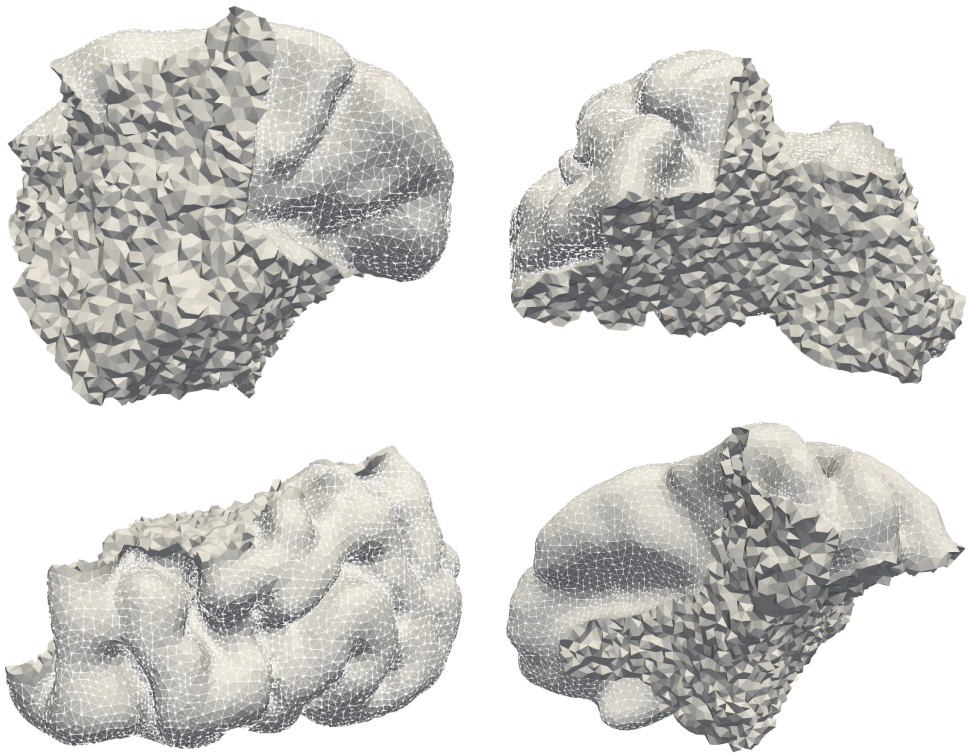

Figure 1: Visualizations of the partitions when N=4, for the `brain.msh` mesh file comprising $2{,}375{,}673$ tetrahedra.

A possible workflow for this feature is summarized in the following code snippet:

Listing 10: Creating a `p::f::T` object from partitioned Gmsh files

```
// Assume partitioned Gmsh files defined in MESH_DIR
static constexpr unsigned int dim = 3; // Dimension of the mesh
MPI_Comm mpi_comm = MPI_COMM_WORLD; // MPI communicator
parallel::fullydistributed::Triangulation<dim> tria(mpi_comm);
GridIn<dim>::read_partitioned_msh(tria,
                                  mpi_comm,
                                  MESH_DIR "/brain");
```

Mesh Data	Size (MB)
Original mesh file	`67.8`
Partitioned mesh files	`18.3, 17.9, 17.9, 18.0`

Table 1: File sizes (in MB) for the `brain.msh` and its 4 partitions.

Memory Consumption	Size (MB)
Non-partitioned triangulation	`155.0`
Per process (`p::f::T`)	`45.0, 44.0, 44.0, 44.0`

Table 2: Memory consumption of the triangulation data structure in deal.II for the brain mesh (4 partitions).

The integration of this new feature into deal.II is documented in pull request #18759 (https://github.com/dealii/dealii/pull/18759), and is currently under review.

References

[1] Geuzaine, C. and Remacle, J.-F., Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. Int. J. Numer. Meth. Engng., 79: 1309-1331, 2009.

5 Conclusion

The integration described in this document lays the basis for several application simulations with the deal.II library and MUMPS, PSCToolkit and Gmsh. It is planned to further enhance the capabilities of the software layers that connect these projects with deal.II and the application solvers, and to continue the work on more sophisicated algorithms that can efficiently use exascale hardware.