

Simulation Workflow - Build system, meshes and visualization

Dominic Kempf

August 31, 2021

Goals of this lecture

At the end of this lecture/exercise session you should

- ▶ understand the modular structure of Dune.
- ▶ have realized that build systems are your friend!
- ▶ have an overview of available grid implementations in Dune
- ▶ have seen several concepts to construct grids
- ▶ be able to visualize PDE solutions in ParaView

Dune is distributed as a zoo of projects. They fall into the following categories:

- ▶ *core modules*: Basic infrastructure and data types, Grid interface, geometries, basis functions and linear algebra
- ▶ *discretization modules*: **dune-pdelab**, dune-fem, dune-fufem, ...
- ▶ *Grid modules*: Additional grid managers
- ▶ *Extension modules*: Mixed bag of additional functionalities
- ▶ *User modules*

Each module...

- ▶ is a git repository in itself
- ▶ has a file *dune.module*, which describes its dependencies
- ▶ uses the build system from dune-common

New modules are created with a shell script `duneproject`

What is CMake anyway?

CMake...

- ▶ ... is an open source buildsystem tool developed at KITware.
- ▶ ... offers a one-tool-solution to all building tasks, like configuring, building, linking, testing and packaging.
- ▶ ... is a build system generator: It supports a set of backends called “generators”
- ▶ ... is portable
- ▶ ... is controlled by ONE rather simple language

We typically use the `Unix Makefiles` generator that generates Makefiles.

Where is the build system defined?

Each (sub)directory of the project contains a file `CMakeLists.txt`.
This file

- ▶ is written in the CMake language
- ▶ is run during configure
- ▶ recursively runs `CMakeLists.txt` files from subdirectories.

Some CMake commands everybody should know

- ▶ `add_subdirectory` lets the configure script recurse into a subdirectory. A `CMakeLists.txt` file is expected in that directory.
- ▶ `add_executable` adds build rules for a new executable.
- ▶ `dune_add_test` adds a test to the testing suite
- ▶ `dune_symlink_to_source_files` creates links from the source directory into the build directory

An example CMakeLists.txt file

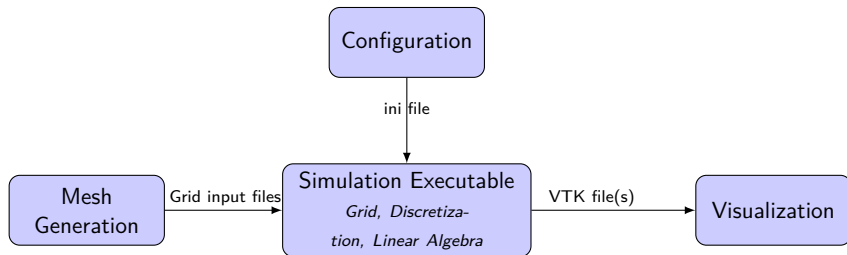
```
add_executable(mytarget mycode.cc)
dune_symlink_to_source_files(FILES mygrid.grid)

dune_add_test(SOURCE unittest.cc
              MPI_RANKS 4)

add_subdirectory(mysubdir)
```

For a reference of end-user CMake commands, see the build system documentation on the Dune website (**DEMOTIME!**).

Simulation workflow



Reading configuration through ini files

```
outputfilename = myfile.vtu  
[grid]  
lowerleft = -1. -1.  
upperright = 1. 1.
```

Dune style ini files support:

- ▶ Key/value pairs separated with a =
- ▶ Grouping of keys into sections to arbitrary depth

```
#include <dune/common/parameterTree.hh>  
#include <dune/common/parameterTreeParser.hh>
```

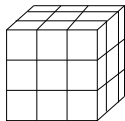
```
Dune::ParameterTree tree;  
Dune::ParameterTreeParser::readINITree(filename, tree)
```

Selecting a grid manager

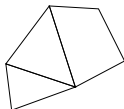
Dune offers a large variety of grid managers, which differ vastly in their feature set and their natural strengths. Here are some capabilities of grid implementations:

- ▶ structured vs. unstructured
- ▶ simplicial vs. quadrilateral vs. multi-geometry
- ▶ conforming vs. non-conforming
- ▶ parallel vs. sequential
- ▶ adaptive vs. non-adaptive (also: different refinement algorithms)
- ▶ different world dimensions (1, 2, 3, n)
- ▶ Surface/manifold grids

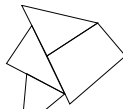
Dune allows for implementations of all sorts of grids through one common interface!



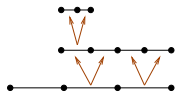
structured



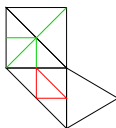
conforming



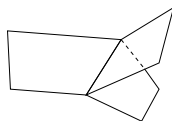
non conforming



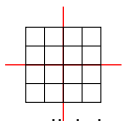
nested, 1D



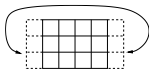
red-green, bisektion



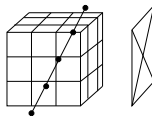
manifolds



parallel data decomposition



periodic



mixed dimensions

A selection of available Implementations of the Dune grid interface:

- ▶ YaspGrid (in dune-grid, structured, parallel, n -dimensional, tensorproduct grid)
- ▶ OneDGrid (in dune-grid, adaptive, parallel, 1D)
- ▶ GeometryGrid (in dune-grid, meta-grid, applies discrete mesh transformation)
- ▶ dune-uggrid (2D/3D, unstructured, parallel, multi-geometry)
- ▶ dune-alugrid (2D/3D, unstructured, parallel, simplex/cube)
- ▶ dune-multidomaingrid (meta-grid, partition grid into sub-domains)
- ▶ dune-subgrid (meta-grid, part of the grid hierarchy as grid in itself)
- ▶ opm-grid (3D, corner-point geometries)
- ▶ dune-foamgrid (1D/2D in 3D manifold grid)
- ▶ dune-prismgrid (meta-grid, extrudes surface mesh)
- ▶ dune-polygongrid (polygonal geometries)
- ▶ ...

StructuredGridFactory

There is a utility class, the StructuredGridFactory, which makes it easy to create structured grids (also for unstructured grid managers). It provides two static functions:

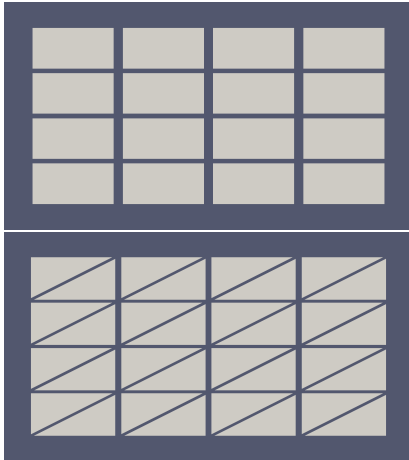
- ▶ `createCubeGrid` creates a grid with cubes with given domain size and number of elements in each direction.
- ▶ `createSimplexGrid` creates a grid with simplices which are generated by subdividing the cubes.

These return a `unique_ptr` to a grid object.

```
Dune::FieldVector<GridType::ctype, dim> lowerleft(0.0);
Dune::FieldVector<GridType::ctype, dim> upperright(1.0);
auto N = Dune::filledArray<dim, unsigned int>(4);

auto grid = Dune::StructuredGridFactory<GridType>::
    createCubeGrid(lowerleft, upperright, N);
```

Structured grids



GmshReader

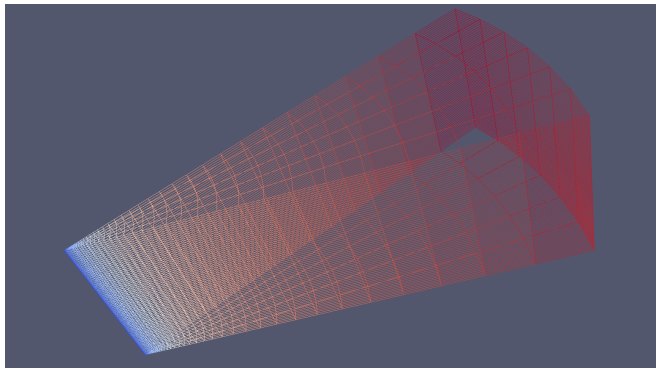
In DUNE there is a mesh reader interface, which can read meshes generated with Gmsh (available at <http://www.geuz.org/gmsh/>).

- ▶ Can only be used with grid managers that support unstructured grids (however, gmsh can generate structured grids, which are stored in an unstructured format)
- ▶ Supports simplex grids in 2D and 3D
- ▶ Gmsh can use geometries from CAD programs
- ▶ Gmsh supports a quadratic boundary approximation, which is translated into boundary segments by the reader.

An easy GmshReader Example

```
#include <dune/grid/uggrid.hh>
#include <dune/grid/io/file/gmshreader.hh>

typedef Dune::UGGrid<2> GridType;
auto grid = Dune::GmshReader<GridType>::read(mshfile);
```



Attaching data to the grid in gmsh

GMSH allows you to define physical entities for cells and boundary segments. The `GmshReader` can read those when parsing the msh file and store them in a `std::vector<int>`. The user takes responsibility of those data structures.

```
#include<dune/grid/uggrid.hh>
#include<dune/grid/io/file/gmshreader.hh>

typedef Dune::UGGrid<2> GridType;

std::vector<int> boundaryPhysicalEntities;
std::vector<int> elementPhysicalEntities;

auto grid = Dune::GmshReader<GridType>::
    read(mshfile, boundaryPhysicalEntities, elementPhysicalEntities);
);
```

What is a tensor product grid?

A tensor product grid is a special kind of a non-equidistant structured cube grid. For each direction $i \in \{0, \dots, d-1\}$ we define a monotonuous sequence of coordinates:

$$\left(x_j^i\right)_{j=0}^{N_i}$$

From those coordinates we define the set of grid vertices V as the tensor product of those coordinate sequences:

$$V = \left\{ (x_{i_0}^0, \dots, x_{i_{d-1}}^{d-1}) \mid i_j \in \{0, \dots, N_j\} \forall j = 0, \dots, d-1 \right\}$$

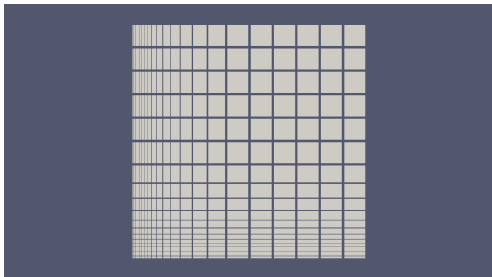
The resulting grid has

$$N = \prod_{i=1}^{d-1} N_i$$

cells.

Tensorproduct grids in simulations

Tensorgrids combine the performance advantages of a structured grid with an unstructured grids capability to have different resolutions in different parts of the domain.



TensorGridFactory

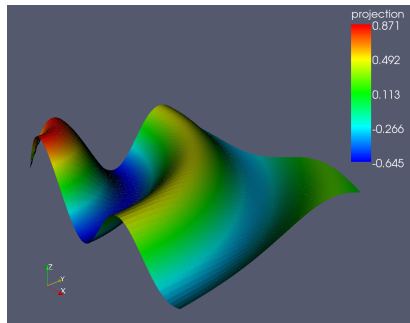
YaspGrid provides a natural implementation of a tensor product grid. The coordinate sequences may be provided manually through `std::vector<ctype>`. `TensorGridFactory` provides convenient methods to fill such coordinate sequences:

- ▶ `void setStart(int d, ctype value)`
- ▶ `void fillIntervals(int d, int n, ctype h)`
- ▶ `void fillRange(int d, int n, ctype end)`
- ▶
 - `void geometricFillIntervals(int d, int n, ctype ratio, ctype h0`
- ▶ `void geometricFillRange(int d, int n, ctype end, ctype h)`
- ▶ ...

The `TensorGridFactory` is compatible with all unstructured grid managers.

Visualization

- ▶ DUNE uses Paraview as a visualization program.
- ▶ Paraview uses the VTK (Visualization Toolkit) file format.
- ▶ Paraview can be obtained for free at <http://www.paraview.org>



VTKWriter

- ▶ To use the VTK-writer you have to
`#include<dune/grid/io/file/vtk/vtkwriter.hh>`
- ▶ There are two different ways to use the VTKWriter:
 1. Pass the data as a vector to the methods `addCellData` or `addVertexData`.
This is especially useful if your scheme already stores the solution in such a vector, i.e. cell- or vertex-centered Finite-Volume schemes.
 2. Define your own `VTKFunction` and pass it to `addCellData` or `addVertexData` as appropriate.
This is the more general approach and usually done for Finite-Element or DG schemes.
- ▶ After attaching zero or more data fields the data file(s) can be written with the `write` method of the `VTKWriter`.

VTK-Export - Elementdata

```
template<class GridView>
void elementdata(const GridView& gridview)
{
    // allocate a vector for the data
    std::vector<double> solution(gridview.size(0));

    // iterate through all entities of codim 0
    for (const auto& e : elements(gridview))
    {
        // get global coordinate of cell center
        auto global = e.geometry().center();
        // evaluate function and store value
        solution[gridview.indexSet().index(e)] = exp(global[0]*global[0]);
    }

    // generate a VTK file
    Dune::VTKWriter<GridView> vtkwriter(gridview);
    vtkwriter.addCellData(solution, "data");
    vtkwriter.write("elementdata", Dune::VTK::appendedraw);
}
```

VTK-Export - Vertexdata I

```
#include<dune/grid/io/file/vtk/vtkwriter.hh>

template<class GridView, class Functor>
void vertexdata(const GridView& gridview, const Functor& f) {
    // allocate a vector for the data
    std::vector<double> solution(gridview.size(GridView::dimension));

    // iterate through all entities of codim dim
    for (const auto& v : vertices(gridview))
    {
        // evaluate functor and store value
        solution[gridview.indexSet().index(v)] = f(v.geometry().corner(0)

    }

    // generate a VTK file
    Dune::VTKWriter<GridView> vtkwriter(gridview);
    vtkwriter.addVertexData(solution, "data");
    vtkwriter.write("vertexdata", Dune::VTK::appendedraw);
}
```


Defining a VTKFunction

An output field can also be created by defining a VTKFunction object, e.g. MyVTKFunction<GridView>.

- ▶ It must be derived from

`Dune::VTKWriter<GridView>::VTKFunction`

- ▶ It has to provide the following functions:

- ▶ the number of components (i.e. whether the plot value is scalar or e.g. a velocity vector):

```
virtual int ncomps() const;
```

- ▶ a function returning the value of the plot function for the component comp at position xi in entity e:

```
virtual double evaluate(int comp, const Entity& e,  
                        const Dune::FieldVector<ctype, Grid::dimension>& xi  
                        ) const;
```

- ▶ the name of the plot function to be written in the VTK-file:

```
virtual std::string name() const;
```

VTK-Export - VTKFunction I

```
#include <dune/grid/io/file/vtk/vtkwriter.hh>

template<class GridView> class XY_VTKFunction
: public Dune::VTKWriter<GridView>::VTKFunction
{
    dune_static_assert(GridView::dimension == 2,
        "Illegal_GridView_dimension");

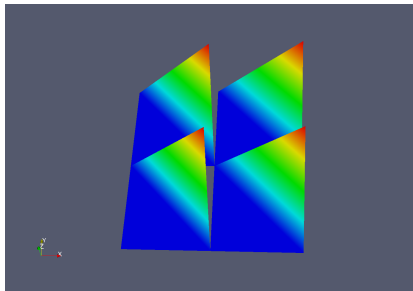
public:
    typedef typename GridView::template Codim<0>::Entity Entity;
    typedef Dune::FieldVector<typename GridView::ctype,
        GridView::dimension> CoordType;

    virtual int ncomps() const { return 1; }
    virtual std::string name() const { return "xy_elementwise"; }
    virtual double evaluate(int comp, const Entity &e,
        const CoordType &xi) const
    {
        auto coord=e.geometry().global(xi);
        return coord[0]*coord[1];
    }
};
```

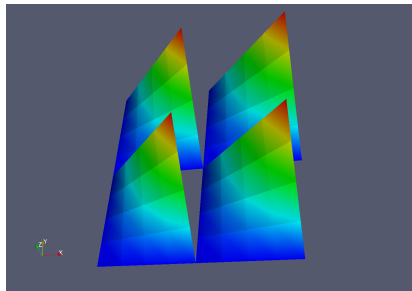
VTK-Export - VTKFunction II

```
template<class GridView>
void functiondata(const GridView& gridview) {
    Dune::VTKWriter<GridView> vtkwriter(gridview);
    vtkwriter.addVertexData(
        Dune::make_shared<XY_VTKFunction<GridView> >());
    vtkwriter.write("functiondata", Dune::VTK::ascii);
}
```

VTKWriter vs. SubsamplingVTKWriter I



Regular VTKWriter



SubsamplingVTKWriter

VTKWriter vs. Subsampling VTKWriter II

To visualize more complex functions (higher order or mesh elements other than triangles or tetrahedra), a subsampling VTKWriter is needed.

Necessary Changes:

```
#include <dune/grid/io/file/vtk/subsamplingvtkwriter.hh>  
Dune::SubsamplingVTKWriter<GridView> vtkwriter(gv, Dune::RefinementInt
```

This creates a subsampling VTKWriter and tells it to generate 2-times subrefined output. This happens “virtually”, i.e. without modifying the grid.