DUNE Adaptive Grid Interface and Adaptive Multilevel Quadrature

Amir Peiraviminaei
apminaei@zedat.fu-berlin.de

March 2, 2019

1 Introduction

Partial differntial equations (PDEs) are used to model many phenomena in fluid dynamics, quantum mechanics, etc. . Simulation of PDEs is a challenging, high demanding task. There are large number of methods for solving PDEs such as finite element, finite volume and finite difference method as well as gridless methods. Many computer codes have implemented for each of these methods based on a particular set of features.

DUNE, the Distributed and Unified Numerics Environment is a modular toolbox for solving PDEs with grid base methods. Dune grid includes some grid managers, some of them internally included but most need additional packages, ALUGrid, UGGrid, SPGrid which are publicly available. For abstract framework and definitions see [2, 3]. The DUNE grid interface is able to represent grids with hierarchically nested grids consists of a collection of $J+1$ grids that are subdivisions of nested domains

$$\Omega = \Omega_0 \supseteq \Omega_1 \supseteq ... \supseteq \Omega_{J-1} \supseteq \Omega_J. \tag{1}$$

The grid that discretizes Ω_0 is called the macro grid and its elements the macro elements. The grid Ω_{l+1} is obtained from the grid for Ω_l by possibly subdividing each of its elements into smaller elements. Thus, each element of the macro grid and the elements that are obtained from refining it form a tree structure. The grid discretizing Ω_l with $0 \leq l \leq J$ is called the level$-l-$grid and its elements are obtained from an $l-$fold refinement of some macro elements. Due to the nestedness of the domains we can partition the domain Ω into

$$\Omega = \Omega_J \bigcup \bigcup_{l=0}^{l=J-1} \Omega_l \backslash \Omega_{l+1}. \tag{2}$$

As a consequence of the hierarchical construction a computational grid discretizing Ω can be obtained by taking the elements of the level$-J-$grid plus the elements of the level$-J-1-$grid in the region $\Omega_{J-1}\backslash\Omega_J$ plus the elements of the level$-J-2-$grid in the region $\Omega_{J-2}\backslash\Omega_{J-1}$ and so on plus the elements of the level$-0-$grid in the region $\Omega_0\backslash\Omega_1$. The grid resulting from this procedure is called the leaf grid because it is formed by the leaf elements of the trees emanating at the macro elements.

Here, we will discuss about adaptive grid interface and adaptive multilevel quadrature implementation.

1.1 Grid adaptation

Adaptive mesh refinement can be used to enhance accuracy and reduce cost of the simulation. The grid interface provides several methods that allow the modification of the grid via refinement and coarsening procedures, if provided by the grid implementation.

The method $mark(ref, e)$ is used to mark an entity e for refinement ($ref = 1$) or coarsening ($ref = -1$). Once entities of a grid are marked, the adaptation is done in the following way:
1. Call the grid's method *preAdapt*(). This method prepares the grid for adaptation. It returns true if at least one entity was marked for coarsening.
2. If *preAdapt*() returned true, any data associated with entities that might be coarsened during the following adaptation cycle has to be projected to the father entities.
3. Call *adapt*(). The grid is modified according to the refinement marks.
4. If *adapt*() returned true, new entities were created. Existing data must be prolonged to newly created entities.
5. Call *postAdapt*() to clean up refinement markers.

1.2 Adaptive multilevel integration

The **global error** can be estimated by taking the difference of the numerically computed value for the integral on a fine and a coarse grid.
Let $I_f^p(\omega)$ and $I_f^q(\omega)$ be two integration formulas of different orders $p > q$ for the evaluation of the integral over some function f on the element $\omega \subseteq \Omega$. If we assume that the higher order rule is locally more accurate then

$$\bar{\epsilon}(\omega) = |I_f^p(\omega) - I_f^q(\omega)|. \tag{3}$$

is an estimator for the **local error** on the element ω.
Refinement Strategy
If the estimated global error is not below a user tolerance the grid is to be refined in those places where the estimated local error is "high".
Consider an element ω and its father element ω^- , i. e. the refinement of ω^- resulted in ω. Moreover, assume that ω^+ is a (virtual) element that would result from a refinement of ω. Then it can be shown that under certain assumptions the quantity

$$\epsilon^+(\omega) = \frac{\bar{\epsilon}(\omega)^2}{\bar{\epsilon}(\omega^-)} \tag{4}$$

is an estimate for the local error on ω^+ , i. e. $\bar{\epsilon}(\omega^+)$.
Another idea to determine the refinement threshold is to look simply at the maximum of the local errors on the current mesh and to refine only those elements where the local error is above a certain fraction of the maximum local error. By combining the two approaches we get the threshold value κ actually used in the code:

$$\kappa = \min\{\max_\omega \epsilon^+(\omega), \frac{1}{2}\max_\omega \bar{\epsilon}(\omega)\} \tag{5}$$

Algorithm

The complete multigrid integration algorithm then reads as follows:
- Choose an initial grid.
- Repeat the following steps:

1. Compute the value I, global integral of the function on the current grid.
2. Compute the estimate E for the global error(the difference between integral over the current grid and the coarse grid).
3. If $E < tol.I$ we are done.
4. Compute the threshold κ as defined above.
5. Refine all elements ω where $\bar{\epsilon}(\omega) \geq \kappa$.

2 Implementation

Now we want to implement the algorithm for the function $f(x) = \frac{1}{\|x\|^2}$. The function $f(x)$ is singular at $x = 0$, figure (1).

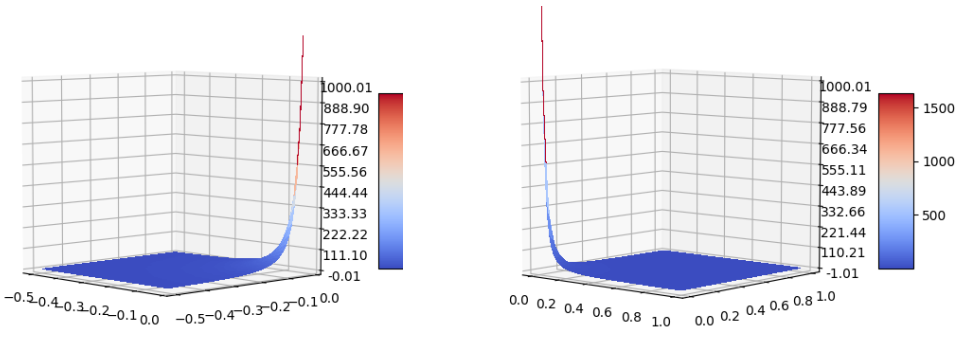

Figure 1: $f(x) = \frac{1}{\|x\|^2}$

The algorithm is realized in the following code. Complete project can be downloaded from www.github.com/apminaei

```
//Adaptive multilevel integration
#ifdef HAVE_CONFIG_H
# include "config.h"
#endif
# include <iostream>
# include <array>
# include <iomanip>
# include <dune/grid/io/file/vtk/vtkwriter.hh> // VTK output routines
# include <dune/common/parallel/mpihelper.hh> // include mpi helper class
# include <dune/grid/uggrid.hh>

# include "integrateentity.hh"
# include "gridviews.hh"

// ! adaptive refinement test
template < class Grid, class Functor >
void adaptiveintegration( Grid & grid, const Functor & f , const double tol)
```

```
{
// get grid view type for leaf grid
typedef typename Grid::LeafGridView GridView ;
// get iterator type
typedef typename GridView::template Codim <0>::Iterator ElementLeafIterator;

// get grid view on leaf part
GridView gridView = grid.leafGridView();

// algorithm parameters

const int loworder = 1;
const int highorder = 3;

// loop over grid sequence
double oldvalue = 10.0 ;
int count = 0 ;
for ( int k = 0; k < 10; k++)
{
// compute integral on current mesh
double value =0;
for ( ElementLeafIterator it = gridView.template begin<0>();
it != gridView.template end<0>(); ++it )
value += integrateEntity(*it ,f , highorder);

// print result
double estimated_error = std::abs( value - oldvalue );
// save value for next estimate
oldvalue = value ;
std::cout << " elements = "
<< std::setw(8) << std::right
<< grid.size(0)
<< " integral = "
<< std::scientific << std::setprecision(8)
<< value
<< " error = " << estimated_error
<< std::endl;

// check convergence
if ( estimated_error<= tol*value )
{
std::cout << "The number of refinement needed: " << count << std::endl;
break ;
}

// refine grid globally in first step to ensure
// that every element has a father
if ( k == 0)
{
grid.globalRefine(1);
continue ;
}

// compute threshold for subsequent refinement
double maxerror = -1E5 ;
double maxextrapolatederror = -1E5 ;
for ( ElementLeafIterator it = gridView.template begin <0>();
it != gridView.template end <0>(); ++it)
{
// error on this entity
double lowresult = integrateEntity(*it ,f , loworder );
double highresult = integrateEntity(*it ,f , highorder );
```

```
double error = std::abs( lowresult-highresult );

// max over whole grid
maxerror = std::max( maxerror , error );
// error on father entity
double fatherlowresult = integrateEntity( it-> father(), f, loworder);
double fatherhighresult = integrateEntity( it-> father(), f, highorder);
double fathererror = std::abs(fatherlowresult - fatherhighresult);

// local extrapolation
double extrapolatederror = error * error /( fathererror + 1E-30);
maxextrapolatederror = std::max( maxextrapolatederror, extrapolatederror);
}
double kappa = std::min( maxextrapolatederror , 0.5* maxerror );

// mark elements for refinement
for ( ElementLeafIterator it = gridView.template begin<0>();
it != gridView.template end<0>(); ++it )
{
double lowresult = integrateEntity (*it ,f , loworder );
double highresult = integrateEntity (*it ,f , highorder );
double error = std::abs( lowresult - highresult );
if ( error > kappa ) grid.mark (1 ,*it );
}

// adapt the mesh
grid.preAdapt();
grid.adapt();
grid.postAdapt();
++count;
}
writeAllGridViews(grid, "uggrid");
}
```

Function integrateEntity uses quadrature rule and is defined as follows

```
// vi : set et ts =4 sw =2 sts =2:
# ifndef DUNE_ADAPTIVEGRIDINTERFACE_MULTILEVELQUADRATURE_INTEGRATEENTITY_HH
# define DUNE_ADAPTIVEGRIDINTERFACE_MULTILEVELQUADRATURE_INTEGRATEENTITY_HH

# include <dune/common/exceptions.hh>
# include <dune/geometry/quadraturerules.hh>

// ! compute integral of function over entity with given order
template < class Entity , class Function >
double integrateEntity( const Entity & entity , const Function & f , int p )
{
// dimension of the entity
const int dim = Entity::dimension ;

// type used for coordinates in the grid
typedef typename Entity::Geometry::ctype ctype ;

// get geometry
const typename Entity::Geometry geometry = entity.geometry();

// get geometry type
const Dune::GeometryType gt = geometry.type();

// get quadrature rule of order p
const Dune::QuadratureRule <ctype, dim>& rule
                        = Dune::QuadratureRules <ctype , dim >::rule( gt, p );
```

```
// ensure that rule has at least the requested order
if ( rule.order() < p )
DUNE_THROW ( Dune::Exception , "␣order␣not␣available␣" );

// compute approximate integral
double result =0;
for ( typename Dune::QuadratureRule < ctype , dim >::
                const_iterator i = rule.begin(); i != rule.end(); ++i )
{
double fval = f( geometry.global (i -> position()));
double weight = i -> weight();
double detjac = geometry.integrationElement(i -> position());
result += fval * weight * detjac ;
}

// return result
return result ;
}
# endif
```

When you run the project for $2D$ grid and $Tol = 10^{-3}$ and maximum refinement number 10, you get this message in the terminal

```
DimX=1, DimY=1, DimZ=1
DimX=1, DimY=1, DimZ=1
grid dimension is 2
elements =         4 integral = 1.71759049e+00 error = 8.28240951e+00
elements =        16 integral = 1.74014213e+00 error = 2.25516405e-02
elements =        23 integral = 1.75137595e+00 error = 1.12338163e-02
elements =        30 integral = 1.75699286e+00 error = 5.61690815e-03
elements =        37 integral = 1.75980131e+00 error = 2.80845407e-03
elements =        44 integral = 1.76120554e+00 error = 1.40422704e-03
The number of refinement needed: 4
```

$2D$ grid is refined first globally and then 4 times locally, figure (2)

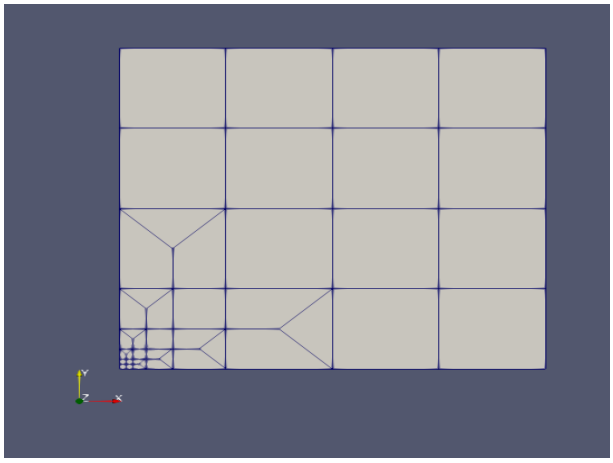

Figure 2: $2D$ leaf grid view

References

[1] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Nolte, M. Ohlberger , O. Sander. The Distributed and Unified Numerics Environment (DUNE) Grid Interface HOWTO. 2017.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. Computing, 82(2-3), 2008, pp. 103-119.

[3] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE. Computing, 82(2-3), 2008, pp. 121-138.