

---

# An efficient implementation of an adaptive and parallel grid in DUNE

Adrian Burri, Andreas Dedner, Robert Klöfkorn<sup>1</sup>, Mario Ohlberger<sup>2</sup>

Abteilung für Angewandte Mathematik, Universität Freiburg,  
Hermann-Herder-Str. 10, D-79104 Freiburg i. Br., Germany  
Email: [alugrid@mathematik.uni-freiburg.de](mailto:alugrid@mathematik.uni-freiburg.de),  
DUNE website: <http://dune.uni-hd.de>

**Summary.** In this contribution we describe and evaluate an efficient implementation of an adaptive and parallel grid (**ALUGrid**) within the Distributed and Unified Numerics Environment **DUNE**. A generalization of the serial grid interface of **DUNE**, described in [1], to the adaptive and parallel case is discussed and example computations using the grid interface are presented. The computations are compared with computations of the original code, which was optimized for the specific example problem studied here.

## 1 Introduction

In [1] a serial version of a generic grid interface was introduced that was realized within the Distributed and Unified Numerics Environment **DUNE**. One of the major goals of such an interface based numerics environment is the separation of data structures and algorithms. For instance, the problem implementation can be done on the basis of the interface independent of the data structure that is used for a specific application. Moreover such a concept allows a reuse of existing codes beyond the interface. Up to now, within **DUNE**, there are five implementations of the grid interface, for example the interface implementation for the PDE software toolbox **UG** [2], for the Finite Element toolbox **ALBERTA** [8], and an implementation for a structured grid. Some of these implementations can be used to perform parallel computations. In this paper we focus on the detailed description of the parallel part of the grid interface that provides the necessary functionality for parallel computations. As some of the packages are already endowed with a parallelisation concept, the interface has to support an efficient access to the already existing parallelisation concepts. In this contribution we focus on the description of an efficient

---

<sup>1</sup> The author was supported by the Bundesministerium für Bildung und Forschung under contract 03KRNCFR.

<sup>2</sup> The author was supported by the Landesstiftung Baden-Württemberg under contract 21-665.23/8.

implementation of the parallel interface for the adaptive and parallel **ALU-Grid** library [3, 9]. **ALU-Grid** is an addaptive, load balanced, unstructured grid implementation that was specifically designed for an efficient implementation of explicit finite volume schemes for nonlinear conservation laws. The goal of this contribution is to demonstrate that the parallel grid interface to **ALU-Grid** can be implemented in such an efficient way that the resulting adaptive and parallel computations based on the implementation in **DUNE** are competitive with computations of the original finite volume code in **ALU-Grid**.

The paper is organized as follows: in Section 2 we give an abstract definition of a parallel hierarchic grid and discuss the corresponding interface classes in **DUNE**. In addition, the specific features of the **ALU-Grid** library are discussed. In Section 3 the handling of arbitrary data during grid reorganization in the case of grid adaptation and dynamic load balancing is discussed and an efficient implementation is presented that avoids the usage of virtual functions in C++. Finally, in Section 4 a run time comparison between the original finite volume implementation in **ALU-Grid** and the interface based implementation in **DUNE** is given.

## 2 Design of the parallel Grid Interface

The **DUNE** grid interface is an interface for parallel grids. This means that a serial grid can be seen as a parallel grid which runs on one processor. Therefore the described functionality is provided by every grid implementing the interface and for some implementations, methods such as `loadBalance` just do nothing. This guarantees that code written for parallel applications can be used for serial calculations as well. Furthermore the part of the grid interface responsible for parallelisation should be such that the user can write code for parallel applications without much effort, i.e. without coding MPI commands. The intention of the design is to provide a parallel extension of the grid interface by adding only a minimum number of methods.

This section is split into three parts: first an abstract mathematical definition of the parallel extension of the **DUNE** grid is presented. Then in the second part the classes implementing the abstract definitions are described. The last part describes the features of the **ALU-Grid** library concerning the grid and the interpretation of the features in terms of the abstract definition of the **DUNE** grid interface.

### 2.1 Abstract definition of the parallel grid

In the following we define a grid  $\mathcal{T}$  in mathematical terms. It is supposed to discretize a domain  $\Omega \subset \mathbb{R}^n$ ,  $n \in \mathbb{N}$ ,  $n > 0$ , with piecewise smooth boundary  $\partial\Omega$ . A grid  $\mathcal{T}$  consists of  $L + 1$  grid levels

$$\mathcal{T} = \{\mathcal{T}_0, \mathcal{T}_1, \dots, \mathcal{T}_L\}.$$

Each grid level  $\mathcal{T}_l$  consists of sets of grid entities  $\mathcal{E}_l^c$  of codimension  $c \in \{0, 1, \dots, d\}$  where  $d \leq n$  is the dimensionality of the grid:

$$\mathcal{T}_l = \{\mathcal{E}_l^0, \dots, \mathcal{E}_l^d\}.$$

Each entity set consists of individual grid entities which are denoted by  $\Omega_{l,i}^c$ :

$$\mathcal{E}_l^c = \left\{ \Omega_{l,0}^c, \Omega_{l,1}^c, \dots, \Omega_{l,N(l,c)-1}^c \right\}.$$

The number of entities of codimension  $c$  on level  $l$  is  $N(l, c)$  and we define a corresponding index set

$$I_l^c = \{0, 1, \dots, N(l, c) - 1\}.$$

**Definition 1.**  $\mathcal{T}$  is called a grid on  $\Omega$  if the following conditions hold:

1. (Tessellation). The entities of codimension 0 on level 0 define a tessellation of the whole domain:

$$\bigcup_{i \in I_0^0} \overline{\Omega_{0,i}^0} = \overline{\Omega}, \quad \forall i \neq j : \Omega_{0,i}^0 \cap \Omega_{0,j}^0 = \emptyset.$$

2. (Nestedness). Entities of codimension 0 on different levels form a tree structure. We require:

$$\forall l > 0, i \in I_l^0 : \exists! j \in I_{l-1}^0 : \Omega_{l,i}^0 \subset \Omega_{l-1,j}^0.$$

This  $\Omega_{l-1,j}^0$  is called father of  $\Omega_{l,i}^0$ . For entities with at least one side on the boundary this condition can be relaxed. We define the set of all descendant entities of codimension 0 and level  $l \leq L$  of an entity  $\Omega_{k,i}^0$  as

$$\mathcal{C}_L(\Omega_{k,i}^0) = \{\Omega_{l,j}^0 \mid \Omega_{l,j}^0 \subset \Omega_{k,i}^0, l \leq L\}.$$

3. (Recursion over codimension). The boundary of a grid entity is composed of grid entities of the next higher codimension, i. e. for  $c < d$  we have

$$\partial \Omega_{l,i}^c = \bigcup_{j \in I_{l,i}^{c+1} \subset I_l^{c+1}} \overline{\Omega_{l,j}^{c+1}}.$$

Grid entities  $\Omega_{l,j}^d$  of codimension  $d$  are points in  $\mathbb{R}^n$ .

4. (Reference elements and dimension). For each grid entity  $\Omega_{l,i}^c$  there is a reference element  $\omega_{l,i}^c \subset \mathbb{R}^{d-c}$  and a sufficiently smooth map

$$m_{l,i}^c : \overline{\omega_{l,i}^c} \rightarrow \overline{\Omega_{l,i}^c}$$

from the reference element to the actual element. Reference elements are convex polyhedrons in  $\mathbb{R}^{d-c}$ . The dimension of the grid  $d$  is the dimension of the reference elements corresponding to grid entities of codimension 0. For  $c = d$  the map  $m_{l,i}^d$  simply returns the corresponding point in  $\mathbb{R}^n$ .

5. (Nonconformity). Note that we do not require the mesh to be conforming in the sense that the intersection of the closure of two grid entities of codimension  $c$  is either zero or a grid entity with codimension greater than  $c$ . However, we require that all grid entities in  $\mathcal{E}_l^c$  are distinct, i. e. :

$$\forall i, j, c, l : \Omega_{l,i}^c = \Omega_{l,j}^c \Rightarrow i = j.$$

The set of all neighbors of an entity  $\Omega_{l,i}^0$  is represented by the set of all non empty intersections with that entity:

$$\mathcal{I}(\Omega_{l,i}^0) = \{\overline{\Omega_{l,i}^0} \cap \overline{\Omega_{l,j}^0} \mid \overline{\Omega_{l,i}^0} \cap \overline{\Omega_{l,j}^0} \neq \emptyset, i \neq j\}.$$

In the following we also use the notion of leaf entities which are all entities  $\Omega_{l,i}^c$  which are not further subdivided, i.e.,  $\mathcal{C}_L(\Omega_{l,i}^c) = \emptyset$  for all  $L \in \mathbb{N}$ .

The index set  $I_l^c$  is called LevelIndexSet. A similar index set  $I_{\text{leaf}}^c$  is also defined for the leaf entities.

For parallel computation we use a domain decomposition strategy, in which each processor performs the simulation on a grid which covers only a part of the whole computational domain  $\Omega$ .

**Definition 2.** Let the domain  $\bar{\Omega}$  be decomposed into  $K$  disjoint partitions

$$\bar{\Omega} = \bar{\Omega}_1 \cup \dots \cup \bar{\Omega}_K \quad (1)$$

and let  $\mathcal{T}_k$  be a grid on  $\Omega_k$  for  $k = 1, \dots, K$  in the sense of Definition 1. The entity sets corresponding to  $\mathcal{T}_k$  are distinguished in the following by a subscript  $k$ .

1. (Border entities). For  $c = 1, \dots, d$  and  $l = 0, \dots, L$  we denote with  $\mathcal{E}_{k,l}^{b,c} \subset \mathcal{E}_{k,l}^c$  the set of border entities consisting of entities  $\Omega_{l,i}^c \subset \partial\Omega_k$  for which there exists at least one index  $k_i \in \{1, \dots, K\} \setminus \{k\}$  and an entity  $\Omega_{l,j_i}^c \in \mathcal{E}_{k_i,l}^c$  with  $\Omega_{l,i}^c = \Omega_{l,j_i}^c$ . Note that for border entities with codimension  $c > 1$  there can exist a large number of copies in the other partitions but for border entities  $\Omega_{l,i}^1 \in \mathcal{E}_{k,l}^{b,1}$  the indices  $k_i, j_i$  are unique.
2. (Ghost entities). The set  $\mathcal{E}_{k,l}^{g,0} = \{\Omega_{l,N(l,0)}^0, \dots, \Omega_{l,N(l,0)+N_g(l,0)}^0\}$  consists of ghost entities  $\Omega_{l,i}^0 \subset \Omega \setminus \Omega_k$  with  $\bar{\Omega}_{l,i}^0 \cap \bar{\Omega}_k \in \mathcal{E}_{k,l}^{b,1}$  and for which there exists an index  $k_i \in \{1, \dots, K\} \setminus \{k\}$  and an entity  $\Omega_{l,j_i}^0 \in \mathcal{E}_{k_i,l}^0$  with  $\Omega_{l,i}^0 = \Omega_{l,j_i}^0$ . Note that as for border entities of codimension one the indices  $k_i, j_i$  are unique. The entity  $\Omega_{l,j_i}^0$  is the master entity of the ghost entity  $\Omega_{l,i}^0$ .
3. (Interior entities). All entities that are neither border entities nor ghost entities are called interior entities.

## 2.2 An adaptive parallel extension of the DUNE grid interface

According to the abstract description of the parallel grid in 2.1, the grid interface consists of the following classes:

1. **Grid**(*dim*, *dimworld*, ...)
 

This class corresponds to the grid  $\mathcal{T}_k$  on  $\Omega_k$  that is processed on processor  $k$ . It is parameterized by the grid dimension  $d = \text{dim}$  and the space dimension  $n = \text{dimworld}$ . The grid class provides iterators for the access to its entities. For grid adaption, load balancing and the communication in the parallel case, the following methods are provided:

  - a) **myRank**(): Gives the processor number  $k$ .
  - b) **mark**(*ref*, *en*): Marks the entity *en* for refinement or coarsening.
  - c) **adapt**(*data*): Modifies the grid  $\mathcal{T}_k$  with respect to the refinement marks. During this procedure the numerical *data* is projected to the new grid.
  - d) **loadBalance**(*data*): Calculates load of the grid  $\mathcal{T}_k$  and repartitions the parallel grid, if necessary. For any entity that is relocated the corresponding numerical *data* is of course also relocated.
  - e) **communicate**(*data*): Communicates *data* on the parallel grid and handles the unique mapping from ghost entities of the grid  $\mathcal{T}_k$  to its master entity on some grid  $\mathcal{T}_l$ .
2. **Entity**(*codim*, *dim*, *dimworld*, ...), **Geometry**(*dim*, *dimworld*, ...)
 

Grid entities  $\Omega_{l,i}^c$  of codimension  $c = \text{codim}$  are realized by the classes **Entity** and **Geometry**. The **Entity** class contains all topological information, while geometrical specifications are provided by the **Geometry** class. The affiliation of an entity to one of the partition types **interior**, **border**, or **ghost** is provided by the member function **partitionType**(). The method **state**() of the **Entity** class ( $c = 0$ ) determines whether an entity might be removed during the next grid adaptation. After an adaptation took place this method allows to detect whether an entity was refined or not.
3. **LevelIterator**(*codim*, *partitionType*, ...)
 

The level iterator gives access to all grid entities on a specified level  $l$  of the partition *partitionType*, where *partitionType* is either **interior**, **border**, or **ghost**. This allows a traversal of the sets  $\mathcal{E}_l^c \setminus \mathcal{E}_l^{b,c}, \mathcal{E}_l^{b,c}, \mathcal{E}_l^{g,c}$ .
4. **LeafIterator**(*codim*, *partitionType*, ...)
 

The leaf iterator gives access to all grid entities of the partition *partitionType* that do not have any further children.
5. **HierarchicIterator**(*dim*, *dimworld*, ...)
 

Another possibility to access grid entities is provided by the hierarchic iterator. This iterator runs over all descendant entities with level  $l \leq L$  of a given entity  $\Omega_{k,i}^0$ . Therefore, it traverses the set  $\mathcal{C}_L(\Omega_{k,i}^0)$ .
6. **IntersectionIterator**(*dim*, *dimworld*, ...)
 

Part of the topological information provided by the **Entity** class of codimension 0 is realized by the intersection iterator. For a given entity  $\Omega_{l,i}^0$  the iterator traverses the set  $\mathcal{I}(\Omega_{l,i}^0)$ .

### 2.3 The ALUGrid library

The **ALUGrid** library [4] allows the use of both hexahedral and tetrahedral grids for simulations on 3d domains, i.e.,  $\Omega \subset \mathbb{R}^n, d = n = 3$  using the notation from Section 2.1. Together with local adaptivity and dynamic load-balancing this enables efficient simulations on arbitrary domains. In the following we describe the structure of the grid and the restrictions in comparison with the general definition of a parallel grid given above.

1. (Macro grid). The grid is initialized with the entities of codimension 0 on level 0 called the macro grid in the following. The entities of the macro grid are subsets in  $\Omega$ . In a parallel computation it is feasible for some of the grids  $\mathcal{T}_k$  to be empty; thus it is possible to start with an initial tessellation of the whole computational domain without having to perform an a-priori partitioning of the initial grid. The union of all the macro grid entities form a conform tessellation of  $\Omega$ .
2. (Restriction on non-conformity). If two leaf entities  $\Omega_{l_1, i_1}^0, \Omega_{l_2, i_2}^0$  have a codimension one intersection then  $|l_1 - l_2| \leq 1$ , i.e., only one level of non-conformity is admissible. For the parallel grid this restriction must also hold between all ghost entities and grid entities with codimension one intersection.
3. (Refinement/Coarsening). During a simulation, leaf entities  $\Omega_{l, i}^0$  can be marked for refinement or coarsening. Entities which are marked for refinement are decomposed into entities on the next level  $\Omega_{l+1, j_0}^0, \dots, \Omega_{l+1, j_q}^0$ . So far  $q = 8$  is implemented both for tetrahedral and hexahedral elements. If the non-conformity restriction is not satisfied, neighboring entities are also refined.  
Entities marked for coarsening are removed only if no violation of the non-conformity restriction occurs during the coarsening process.
4. (Load balancing). After each grid adaptation, the current load on each partition is estimated. The repartitioning of the grid is only performed on the macro grid level, i.e., only entities  $\Omega_{0, i}^0$  together with all  $\Omega_{l, j}^c \subset \Omega_{0, i}^0$  are moved between partitions. To perform load balancing a partitioning algorithm using the library METIS [6, 7] is utilized for the dual graph of the macro grid. Details can be found in [6, 7, 9].

## 3 Handling user data during grid reorganization

Most software for numerical simulations has its own data formats for storage of the numerical data, like for example `DOF_REAL_VEC` in **ALBERTA** [8]. This is a critical point because code once written using the data structure of a certain package is hardly portable. Therefore the general approach in **DUNE** is to separate the handling of numerical data from the grid. This means the interface has to provide some kind of identifier which allows to identify for example vertices or elements of the grid (see Definition 1). In **DUNE** this

means that each entity must provide a minimal set of indices (see [5] for detailed description). Lets assume for simplicity that each element of the grid, for example each tetrahedron or triangle, can be identified by a unique index  $i$ , with  $i \in \mathbb{N}$ . Using this the numerical data can be stored in vectors and can be accessed via these indices. This approach leads to difficulties if grid reorganization requires the projection of user data from the old to the new grid. For example during adaptation of the grid, when elements are created or removed, all user data, i.e., solution, right hand side, etc. has to be projected onto the new grid. Therefore, since the data is not stored together with the grid, the grid interface has to provide methods to handle the projection process for all persistent data. Using the interface method `state()` defined for entities of codimension 0 — which identifies whether an entity will be coarsened or was refined — one can separate the restriction/prolongation process from the grid. This means that before the grid is adapted, all persistent data which belong to leaf entities of the grid has to be projected to their father; after grid adaptation the data is projected onto the new entities. Of course data is not modified if an entity is not changed. Unfortunately this method is very expensive in terms of CPU time when grid adaptation takes up a substantial part of the overall computational cost, e.g., in a time explicit finite-volume scheme.

We adopt a different strategy to project the relevant data onto the new grid. Using a call-back functionality during element creation or removal the data can be projected more efficiently. Of course all projections have to be done simultaneously for an element-children tuple. This means all data which have to be projected to the new grid should be available in a list-like structure. For the adaptation step we pass an object through the interface to the grid which provides a `prolong` and a `restrict` method. An easy but unefficient implementation of such a mechanism would use the virtual function concept of C++. Since efficiency is a primary goal in numerical software, a different approach relying on the template mechanism of C++ is used. The following code snippet explains how the required functionality for prolongation/restriction can be achieved, for simplicity showing only the method `prolong`.

```
// project data from father to son entity
template <class VectorType>
class SimpleProlongation
{
    VectorType & vec;
public:
    SimpleProlongation(VectorType & v) : vec(v) {}

    void prolong(Entity & father, Entity & son)
    {
        vec[son.index()] = vec[father.index()];
    }
};

template <class A, class B>
class CombinedProlongationOperator
{
    A & _a;
    B & _b;
};
```

```

public:
  // stores the references to the objects that should be combined
  CombinedProlongationOperator( A & a , B & b ) : _a(a) , _b(b) {}

  void prolong ( Entity & father , Entity & son )
  {
    _a.prolong(father , son);
    _b.prolong(father , son);
  }
};

```

Now the algorithm looks the following way:

```

// somewhere in the implementation
template <class GridType>
void algorithm ( GridType & grid )
{
  // vector storing the unknown density
  vector< double > density;

  // carray is an array of fixed length
  typedef carray<double,3> double_3;
  // vector storing the unknown velocity
  vector< double_3 > velocity;

  // the CombinedProlongationOperator is parameterized by
  // the SimpleProlongation classes
  typedef CombinedProlongationOperator<
    SimpleProlongation< vector<double> > ,
    SimpleProlongation< vector<double_3> > >
    > CombinedProlongationType;

  SimpleProlongation< vector<double> > prolongDensity( density );
  SimpleProlongation< vector<double_3> > prolongVelocity( velocity );

  CombinedProlongationType rpData ( prolongDensity , prolongVelocity );

  for ( ... ) // all timesteps
  {
    // start adaptation process, entities are already marked
    // when an element is refined then the method
    // prolong of the class CombinedProlongationOperator
    // is called and the data is prolonged
    grid.adapt( rpData );
  }
}

```

In this example the **class A** is of the type `SimpleProlongation<vector<double> >` and the **class B** is of the type `SimpleProlongation<vector<double_3> >`. Because all types are known at compile time the compiler is able to inline function calls for optimization. Both approaches allow to combine different types of objects, as long as they all provide a `prolong` method with exactly the same parameter list. As in the approach using the virtual functions, more than two objects can be combined, because **class A** or **class B** could also be of the type `CombinedProlongationOperator`. This example only shows the basic idea. The current implementation of the code uses a more sophisticated implementation which is available in an example code on the web [4].

Except for grid adaptation, two further situations occur where the above described functionality is needed. These are the communication and the load balancing procedure. During both steps, access to the user data is needed and the access needs to be as efficient as possible. The following examples

shortly delineate the idea of the concept as used in the communication step. For further details the interested reader should refer to [4].

```
// read and write data from/to message buffer
template <class EntityType, class VectorType>
class ReadWriteData
{
    VectorType & vec;
public:
    ReadWriteData ( VectorType & v ) : vec(v) {}

    void readData ( MessageBuffer & buf , EntityType & e ) {
        buf.readObject( vec[e.index()] );
    }

    void writeData ( MessageBuffer & buf , EntityType & e ) {
        buf.writeObject( vec[e.index()] );
    }
};
```

Here the message buffer is implemented as an object stream. Before a communication step, the data of all entities located at the process boundary is inserted in the stream. The interprocess communication consists of exchanging the stream objects, from which the data is then extracted in the same order as it was inserted on the other process. Because **ALUGrid** guarantees an iteration order which is the same on each side of a process border, the data of the entities get inserted in the right place. An example of the message buffer implementation can be found in the code of **ALUGrid**. Here we only need to know that the methods `readObject` and `writeObject` just read and write data to and from the object stream.

For the load balancing process almost the same functionality is needed. Instead of entities on a process boundary, all children of a macro grid entity marked for relocation to another processor are considered. After inserting the entity's refinement information and data into the stream, it is sent to processor  $k$ . On processor  $k$  first the entity tree below the macro entity is recreated by refining the macro entity as described by the refinement information. Afterwards the data from the other processor is extracted on the newly created entities. To this end, the method `xtractData` is called which makes a hierarchical walk over the restored tree calling the method `readData`.

```
// Inline and Xtract Operator for exchanging
// leaf data during load balancing
template <class GridType, class ReadWriteDataType>
class InlineXtractData // pack/unpack data to/from message buffer
{
    GridType & grid;

    typedef typename GridType :: Entity EntityType;
    ReadWriteDataType & rwData;
public:
    InlineXtractData( GridType & g, ReadWriteDataType & rwd ) :
        grid(g) , rwData(rwd) {}

    void inlineData ( MessageBuffer & buf , EntityType & e ) {
        typedef typename EntityType :: HierarchicIterator HierarchicIterator;
        for( HierarchicIterator it = e.hbegin( grid.maxlevel() ); ... )
        {
            if((*it).isLeaf()) { rwData.writeData(buf, *it); }
        }
    }
};
```

```

    }
  }
  void xtractData ( MessageBuffer & buf , EntityType & e ) {
    typedef typename EntityType :: HierarchicIterator HierarchicIterator ;
    for( HierarchicIterator it = e.hbegin( grid.maxlevel() ); ... )
    {
      if((*it).isLeaf()) { rwData.readData(buf, *it); }
    }
  }
};

```

## 4 Example computation and performance evaluation

Since **ALUGrid** was designed with explicit finite volume schemes in mind, we base our efficiency test of the parallel **DUNE** interface on an explicit first order finite volume scheme for the Euler equations of gas dynamics. Using the notation from Definition 1 the scheme for evolving the piecewise constant discrete solution  $\{U_i^n\}_{i,n}$  on the leaf entities  $\Omega_{l,i}^0$  from a time level  $t^n$  to a time level  $t^{n+1} = t^n + \Delta t^n$  reads as follows:

$$U_i^{n+1} = U_i^n - \frac{\Delta t^n}{|\Omega_{l,i}^0|} \sum_{\Omega_{k,j}^0} G_{ij}(U_i^n, U_j^n)$$

where the sum is taken over all leaf entities  $\Omega_{k,j}^0$  which have a codimension one intersection with  $\Omega_{l,i}^0$ . The conservative quantities are  $U = (\rho, \rho u, \rho v, \rho w, \rho e)$  and the function  $G$  is a Riemann-solver based numerical flux function. A detailed description of the algorithm can be found in [3]. Basically the algorithm consists of five steps — assuming the data  $\{U_i^n\}_i$  is given:

1. (Communication). The data is exchanged from master entities to the ghost entities on the other processors.
2. (Flux evaluation). For each pair of entities  $\Omega_{l,i}^0, \Omega_{k,j}^0$  with codimension one intersection and  $i < j$  the numerical flux  $G_{ij}(U_i^n, U_j^n)$  is evaluated and the sum  $V_i^n := \frac{1}{|\Omega_{l,i}^0|} \sum_{\Omega_{k,j}^0} G_{ij}(U_i^n, U_j^n)$  is computed for each leaf entity. During this step, the maximal admissible local time-step sizes  $\Delta_{ij}^n$  are also computed.
3. (Global time step). The minimum time step size  $\Delta t^n = \min_{(i,j)} \Delta_{ij}^n$  is computed using a global communication step between all processors.
4. (Evolution). The conservative quantities at the next time level are constructed:  $U_i^{n+1} = U_i^n + \Delta t^n V_i^n$ .
5. (Adaptation and load balancing). The grid entities are refined, coarsened, and the grid is repartitioned with respect to the new solution.

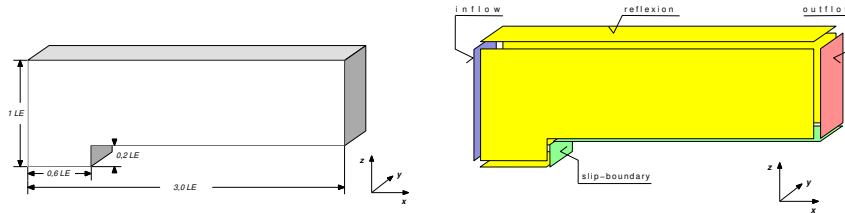
The main difference between the implementation of the scheme in **DUNE** compared to using **ALUGrid** directly concerns the storage of the data. In the **ALUGrid** implementation, the data (i.e.,  $U_i^n, V_i^n$ ) is stored directly in

the objects representing the grid entities. Therefore accessing data is very direct and efficient since it is loaded into the cache together with the geometric information. Also the reorganization of the grid during the adaption process is very efficient since storage space for the data is automatically allocated together with the geometric information for the new entities. Since grid adaptation is performed in each time step, the execution time for the grid modification is comparable to the cost of the numerical scheme (about 20% of the overall time). In this sense the explicit finite volume scheme is a very challenging problem for a grid interface like **DUNE** where data is managed independently of the grid.

As a test case we use the forward facing step benchmark problem [10] for a perfect gas law with  $\gamma = 1.4$ . The domain is shown in Figure 1. As initial data we use

$$U_i^0 = (1.4, 4.2, 0.0, 0.0, 8.8)$$

for all leaf entities. The Dirichlet data on the inflow boundary is also set to this value and remains constant over time. This leads to a Mach three flow in the "wind-tunnel".



**Fig. 1.** Setting for the Forward-Facing-Step problem.

Figure 2(top) shows the density of the solution at time  $t = 1.75$  together with the locally refined grid. The bottom part of Figure 2 shows the grid partitioning for the same point in time using  $K = 8$  processors. The evolution in the interval  $t \in [1.5, 2.0]$  of the grid size (average number of leaf elements together with the number of leaf elements in the largest and smallest partition) is plotted in Figure 3(left). The points of grid redistribution can be clearly distinguished where maximum and minimum size are almost identical to the average number. The CPU time for the flux calculation (step 2 of the algorithm) in the same time interval is shown on the right of Figure 3. The total runtime per time-step is also shown. It is clearly visible that doing a grid redistribution at times causes the total runtime per time-step to increase on average with the total number of elements. The peaks in the total runtime show the computational cost of the redistribution step, which increases the total runtime of these time-steps by merely 20%.

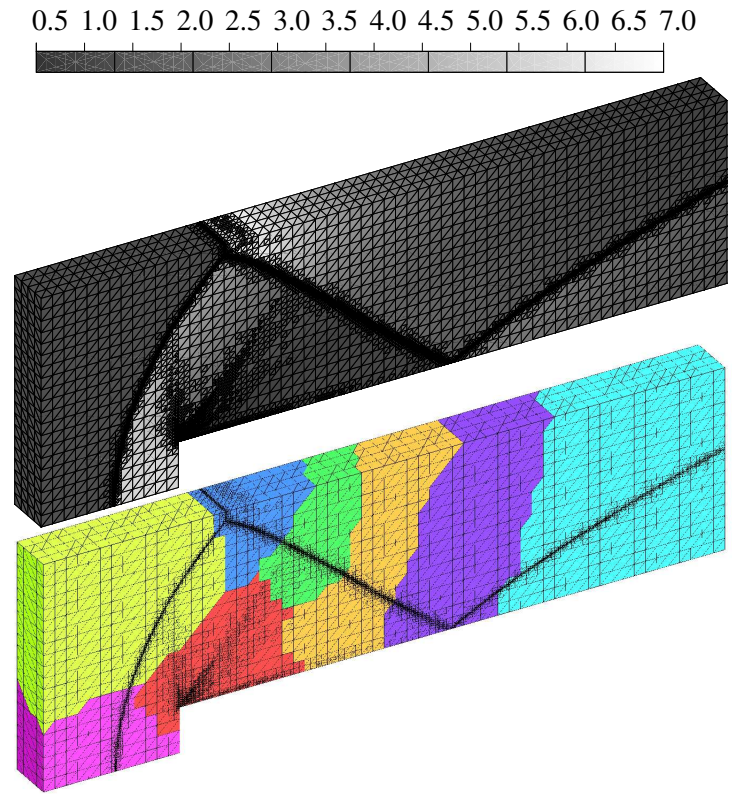


Fig. 2. Density (top) and partitioning (bottom) at  $t = 1.75$ .

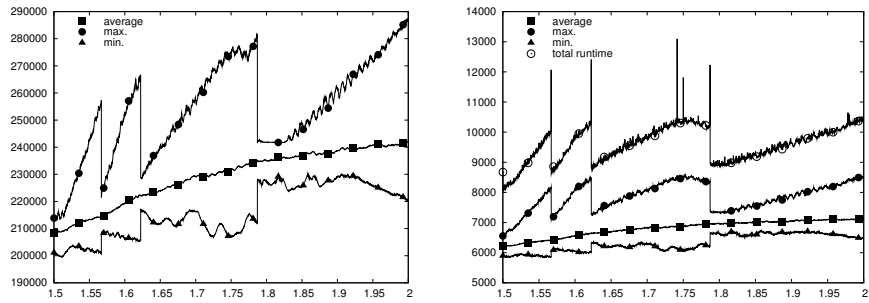


Fig. 3. Evolution of grid size (left) and corresponding runtimes (right) for  $t \in [1.5, 2]$

#### 4.1 Definition of performance measures

We define the size of the problem in a fixed time interval  $[t_{start}, t_{end}]$  as the average over all time-steps  $t^n$  with  $n \in N := \{m \mid t^m \in [t_{start}, t_{end}]\}$  of the number of leaf entities in the locally adapted grid at time  $t^n$ :  $\Sigma_K := \frac{1}{|N|} \sum_{n \in N} \sum_{k=1}^K S_k^n$  where  $S_k^n$  is the number of leaf entities of codimension zero at time-level  $t^n$  in the grid  $\mathcal{T}_k$ . To measure the efficiency we study the average runtime on  $K$  processors:  $\tau_K := \frac{1}{|N|} \sum_{n \in N} \tau_K^n$ ; here  $\tau_K^n$  is the total runtime per time-step. For a more detailed analysis we furthermore study the computational cost  $\tau_{s,K}^n$  on  $K$  processors of each time-step  $t^n$  for the steps  $s = 2, 4$ , and  $5$  of the algorithm described above; we set  $\tau_{s,K} := \frac{1}{|N|} \sum_{n \in N} \tau_{s,K}^n$ . Average values for the runtime per element are now easily defined as  $\eta_K := \frac{\tau_K}{\Sigma_K}$  and  $\eta_{s,K} := \frac{\tau_{s,K}}{\Sigma_K}$  for  $s = 2, 4, 5$ .

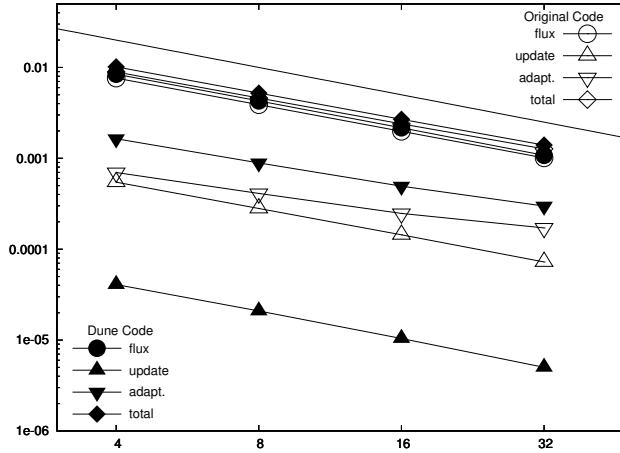
To estimate the parallel effectiveness of the **DUNE** interface we compute the speedup and the efficiency using the average total runtime per element  $\eta_K$ . The *speedup* from  $L$  to  $K > L$  processors is then given by  $S_{L \rightarrow K} := \frac{\eta_L}{\eta_K}$  and is in the optimal case equal to  $\frac{K}{L}$ ; the *efficiency*  $E_{L \rightarrow K} := \frac{L}{K} S_{L \rightarrow K}$  should therefore be approximately 1.

Note that the definition of speedup and efficiency differs from the standard definitions, where the speedup and efficiency would be defined as  $s_{L \rightarrow K} := \frac{\tau_L}{\tau_K}$  and  $e_{L \rightarrow K} := \frac{L}{K} s_{L \rightarrow K}$  respectively. For non-adaptive computations (i.e. with a fixed number of entities), the definitions  $S_{L \rightarrow K}$ ,  $s_{L \rightarrow K}$  and  $E_{L \rightarrow K}$ ,  $e_{L \rightarrow K}$  respectively are identical. The use of the modified definitions  $S_{L \rightarrow K}$  and  $E_{L \rightarrow K}$  enables us to compare problems with a slightly varying number of entities.

#### 4.2 Comparison between the original and the **DUNE** code

In Figure 4 we plot the average runtimes  $\eta_{2,K}, \eta_{4,K}, \eta_{5,K}$ , and  $\eta_K$  summing over all time-steps in  $[1.5, 2.0]$ . We exclude the results from the start of the simulation since at the beginning the grid is too small to reach meaningful conclusions on 32 processors. Our results confirm the observations from [1], demonstrating that the **DUNE** interface hardly reduces the efficiency of the numerical scheme.

Although the explicit finite volume scheme is very challenging for a general grid interface, the difference between the original code and the **DUNE** code in the overall runtime is small (about 9 – 12 %). Table 1 shows the relative contribution to the performance gap from each of the algorithm's substeps. It can be seen that the **DUNE** code is inferior especially in the adaptation and flux computation steps. For the adaptation step this loss in performance can be solely attributed to the disadvantage of storing the data separately from the grid, which causes a high amount of data reorganization in these explicit problems. The contribution from the flux computation is of about the same order – a time difference which is only due to the **DUNE** interface. Part of the



**Fig. 4.** Average runtime for steps 2,4,5 and the total runtime per time-step of the finite volume scheme using the original code and the using the parallel **DUNE** interface.

$K$	$\theta_{2,K}$	$\theta_{4,K}$	$\theta_{5,K}$	$\theta_K$
4	0.0772087	-0.0497788	0.0929186	0.122302
8	0.0752792	-0.0497255	0.0916843	0.117822
16	0.0685054	-0.0496885	0.0915318	0.108574
32	0.0493819	-0.0481409	0.0905287	0.090217

**Table 1.** Relative performance losses of the **DUNE** code compared to the original implementation. The relative performance loss  $\theta_{s,K}$  of a substep  $s$  is defined as  $\theta_{s,K} := \frac{\eta_{s,K}^{dune} - \eta_{s,K}^{orig}}{\eta_{s,K}^{dune}}$ ; for the total runtime we define  $\theta_K := \frac{\eta_K^{dune} - \eta_K^{orig}}{\eta_K^{dune}}$ .

arreange of the **DUNE** code can be made up in the update step, where we see a significant advantage of storing the data in a consecutive vector: no extra grid traversal is necessary, which makes this operation for the **DUNE** code about an order of magnitude faster and resulting in a speed regain of about 5 %. On 32 processors flux and update step cancel each other and performance loss occurs solely in the adaptation step.

### 4.3 Efficiency of the parallel interface

The goal of the following investigation is to quantify the additional cost of having to access the **ALUGrid** through the **DUNE** interface; in addition we also demonstrate the parallel efficiency of the code using the definition from section 4.1. We performed the forward facing step simulation on the HP C6000 Linux Cluster at the SSC Karlsruhe using  $K = 4, 8, 16,$  and  $32$  processors. Since we study a fixed size problem the parallel overhead increases with the number of processors while the cost of the numerics decreases. Hence we cannot expect

the optimal efficiency in this case. The corresponding values for the original code and the **DUNE** code are shown in Table 2(left) and Table 2(right), respectively. We observe that the efficiency is quite high (around 90%) and that the values are approximately the same for both versions of the algorithm. As already pointed out we cannot expect optimal efficiency using a fixed size problem – due to the restriction on the time-step  $\Delta t$  in the explicit finite volume scheme and due to the difficult control of the grid adaptation process, it is difficult to study problems with a fixed size per processor.

For some indication of the effectiveness of the load-balancing procedure we study the efficiency of the flux calculation (step 2). Since this step involves no communication, we expect no parallel overhead so that the runtime is only determined by the processor with the largest chunk of the grid. In Table 3 we see that for this step the efficiency is very close to 1 which demonstrates that the load-balancing procedure used for the simulations leads to a grid partitioning which is close to being optimal.

original code				<b>DUNE</b>			
$K$	$\eta_K$	$S_{4 \rightarrow K}$	$E_{4 \rightarrow K}$	$K$	$\eta_K$	$S_{4 \rightarrow K}$	$E_{4 \rightarrow K}$
4	0.00890626			4	0.0101473		
8	0.00460453	1.93424	0.967118	8	0.0052195	1.94411	0.972054
16	0.0023943	3.71978	0.929945	16	0.00268592	3.77795	0.944488
32	0.00127103	7.00712	0.87589	32	0.00139707	7.26325	0.907906

**Table 2.** Speedup and efficiency measured with respect to a run with four processors using a fixed sized problem. Left the results for the original code are shown; on the right we have the corresponding results for the **DUNE** code.

original code				<b>DUNE</b>			
$K$	$\eta_{2,K}$	$S_{4 \rightarrow K}$	$E_{4 \rightarrow K}$	$K$	$\eta_{2,K}$	$S_{4 \rightarrow K}$	$E_{4 \rightarrow K}$
4	0.00762761			4	0.00841107		
8	0.00388685	1.96241	0.981207	8	0.00427977	1.96531	0.982653
16	0.00198272	3.84705	0.961761	16	0.00216672	3.88194	0.970486
32	0.00100939	7.55666	0.944582	32	0.00107838	7.79971	0.974963

**Table 3.** Speedup and efficiency of the flux calculation measured with respect to a run with four processors using a fixed sized problem. Left the results for the original code are shown; on the right we have the corresponding results for the **DUNE** code.

## 5 Conclusions

In this paper, a parallel, adaptive grid implementation (**ALUGrid**) to the parallel grid interface of the Distributed and Unified Numerics Environment

**DUNE** was described. Using the forward facing step test case from [10], it could be shown that the implementation is efficient. Although an explicit finite volume scheme on a locally refined grid is extremely challenging for a grid interface the overhead of using **ALUGrid** through the parallel interface of **DUNE** only causes losses of less than 10 % in the total runtime.

## References

1. Bastian P, Droske M, Engwer C, Klöfkorn R, Neubauer T, Ohlberger M, Rumpf M (2004) *Towards a Unified Framework for Scientific Computing*. In Proc. of the 15th International Conference on Domain Decomposition Method.
2. Bastian P, Birken K, Johannsen K, Lang S, Neuss N, Rentz-Reichert H, Wieners C (1997) Ug - a flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science*, **1**, 27–40.
3. Dedner A, Rohde C, Schupp B, Wesenberg M (2004) A parallel, load balanced mhd code on locally adapted, unstructured grids in 3d. *Computing and Visualization in Science*, **7**, 79–96.
4. ALUGrid: <http://www.mathematik.uni-freiburg.de/IAM/Research/alugrid/>.
5. DUNE: <http://dune.uni-hd.de>.
6. METIS: <http://www-users.cs.umn.edu/~karypis/metis/>.
7. Karypis G, Kumar V (1999) Multilevel k-way partitioning scheme for irregular graphs. *SIAM Review*, **41(2)**, 278–300.
8. Schmidt A, Siebert K (2005) *Design of Adaptive Finite Element Software – The Finite Element Toolbox ALBERTA*. Springer.
9. Schupp B (1999) *Entwicklung eines effizienten Verfahrens zur Simulation kompressibler Strömungen in 3D auf Parallelrechnern..* Dissertation, Mathematische Fakultät, Universität Freiburg.
10. Woodward P, Colella P (1984) The numerical simulation of two-dimensional fluid flow with strong shocks. *Journal of Computational Physics*, **54**, 115–173.