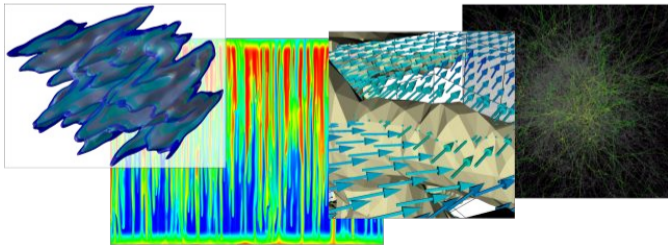


## DUNE PDELab Tutorial 06

### Parallel Computing



Speaker:

Olaf Ippisch  
Institut für Mathematik  
TU Clausthal

# Contents

---

**Introduction**

Domain Decomposition

Parallel Grids

Communicating Data with Dune

Parallel PDELab

# Why Parallel Computing?

---

- ▶ The speed of individual computer cores is not increasing essentially since nearly 20 years due to
  - ▶ Power wall
  - ▶ Memory wall
  - ▶ Instruction level parallelism (ILP) wall
- ▶ However, the number of cores is increasing. Quad-cores are the rule, up to 260-core processors are available
- ▶ Several multi-core processors can be used on one mainboard (e.g. two 32-core processors)
- ▶ Computer cluster with several multi-core multi-processor servers are affordable even for small companies

# Why Parallel Computing?

The worlds four fastest computers (November 2020) have

- ▶ Fugaku, Kōbe, Japan: 442.0 PFlop/s  
7'630'848 cores:
  - ▶ 158'976 A64FX48-core processors
- ▶ Summit, Oak Ridge National Laboratory, U.S.A.: 148.6 PFlop/s  
202'752 CPU cores + 2'211'840 GPU cores. 4'608 nodes with
  - ▶ 2 IBM Power9 22-core processors each
  - ▶ 6 Tesla-Volta 100 80-core accelerator cards each
- ▶ Sierra, Lawrence Livermore National Laboratory, U.S.A.: 94.6 PFlop/s  
190'080 CPU cores + 1'382'400 GPU cores. 4'320 nodes with
  - ▶ 2 IBM Power9 22-core processors
  - ▶ 4 Tesla-Volta 100 80-core accelerator cards
- ▶ Sunway TaihuLight, Wuxi, China: 93.0 PFlop/s  
10'649'600 cores:
  - ▶ 40'960 SW26010 260-core processors

# Architectures of Parallel Computers

---

- ▶ Shared-memory computing: all cores have access to the whole memory
  - ▶ Uniform memory access architecture (UMA):  
access to every memory location from every process takes the same amount of time (few multi-core CPUs)
  - ▶ Non-uniform memory access architecture (NUMA):  
memory is associated with a processor or a group of processor cores but address space is global. Local memory can be accessed faster than memory attached to other processors or other parts of the same processor (most multi-core CPUs, multi-processor servers)
- ▶ Message passing architecture (MP):  
each process can only access local memory, information is exchanged between processes with messages send over a network (computer clusters, super computer)

# Comparison of Architectures by Example

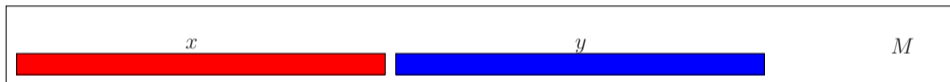
- ▶ Given vectors  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ , compute scalar product  $s = \sum_{i=0}^{N-1} x_i y_i$ :

- (1) Subdivide index set into  $P$  pieces.

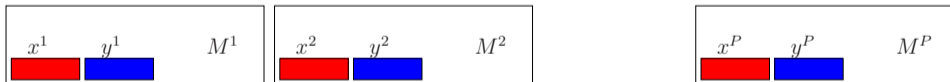
- (2) Compute  $s_p = \sum_{i=pN/P}^{(p+1)N/P-1} x_i y_i$  in parallel.

- (3) Compute  $s = \sum_{i=0}^{P-1} s_i$ .

- ▶ *Uniform memory access architecture*: Store vectors as in sequential program:



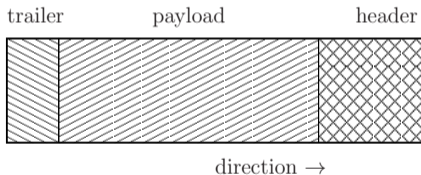
- ▶ *Nonuniform memory access architecture*: Distribute data to the local memories:



- ▶ *Message passing architecture*: Same as for NUMA!
- ▶ Parallelisation effort for NUMA and MP is almost the same.
- ▶ Distributing data structures is hard and not automatic in general.

# Message Passing

- ▶ Users view: Copy (contiguous) memory block from one address space to another.
- ▶ During transmission the message is subdivided into individual packets.
- ▶ Network is packet-switched.
- ▶ A packet consists of an envelope and the data:



- ▶ Header: Destination, size and kind of data.
- ▶ Payload: Size ranges from some bytes to kilobytes.
- ▶ Trailer: e.g. checksum for error detection.

# The Message Passing Interface (MPI)

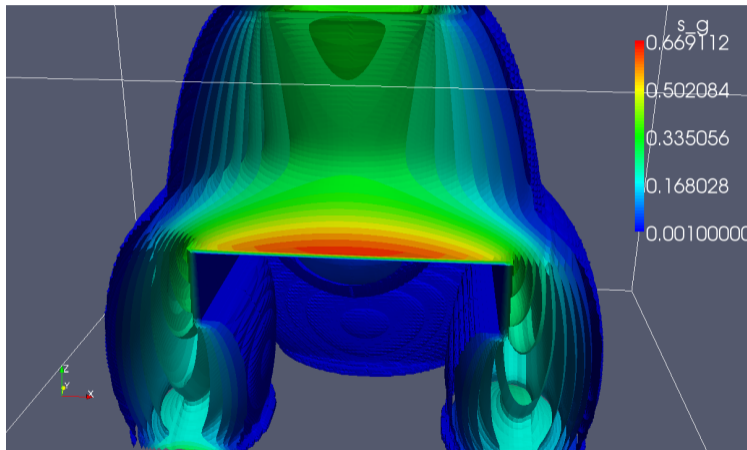
- ▶ Portable Library with functions for message exchange between processes
- ▶ Developed 1993-94 by a international board
- ▶ Available on nearly all computer platforms
- ▶ Free Implementations for Linux Clusters: **MPICH** and **OpenMPI** <sup>1</sup>
- ▶ Properties of MPI:
  - ▶ library with C- and Fortran bindings (no language extension)
  - ▶ large variety of point-to-point communication functions
  - ▶ global communication
  - ▶ data conversion for heterogeneous systems
  - ▶ subset formation and topologies possible
  - ▶ parallel I/O, one-sided communication and dynamic process management (MPI-2)
  - ▶ hybrid-parallelism (MPI-3)

---

<sup>1</sup><http://www-unix.mcs.anl.gov/mpi/mpich> and <http://www.open-mpi.org/>



# Strong Scalability of 3D Parallel Computation



3D DNAPL Infiltration

## Strong Scalability of 3D Parallel Computation

Simulation of a DNAPL infiltration with a coarse lense on a grid with  $160 \times 160 \times 96$  unknowns on a server with  $4 \times 12$  AMD Magny Cours, 2.1 GHz,  $12 \times 0.5$ MB L2, 12MB L3 processors.

Computation time for one time step with BiCGStab + AMG prec.:

P	#IT(max)	$T_{it}$	S	$T_{asm}$	S	$T_{total}$	S
1	6.5	4.60	-	43.7	-	713.8	-
4	10	1.85	2.5	17.5	2.5	295.9	2.4
8	9	0.63	7.3	8.4	5.2	127.1	5.6
16	9.5	0.40	11.5	4.1	10.7	73.1	9.8
32	15	0.27	17.0	1.9	23.0	43.5	16.4

# Contents

---

Introduction

**Domain Decomposition**

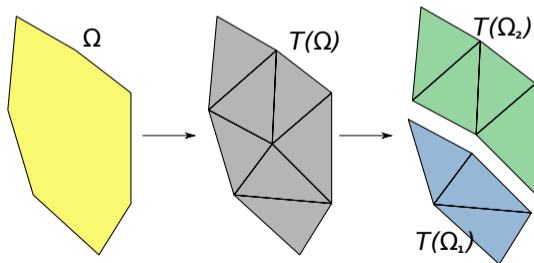
Parallel Grids

Communicating Data with Dune

Parallel PDELab

# Domain Decomposition

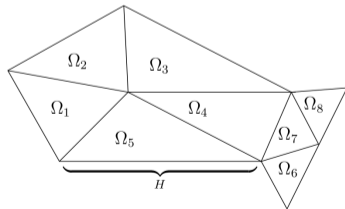
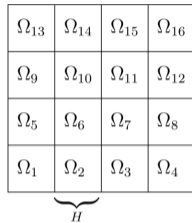
- ▶ partition a problem by splitting the domain into smaller subdomains
- ▶ each part is solved by a different process
- ▶ goes back to an idea of H.A. Schwarz who in 1890 presented a method to prove the existence of solutions of the Laplace equation on “complicated” domains.
- ▶ Different variants:
  - ▶ overlapping domain decomposition
  - ▶ non-overlapping domain decomposition



# Nonoverlapping Domain Decomposition

- ▶ Given a domain  $\Omega \subseteq \mathbb{R}^d$
- ▶ partition  $\Omega$  into *non-overlapping* sub-domains:

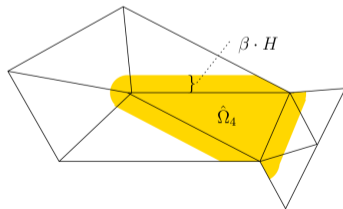
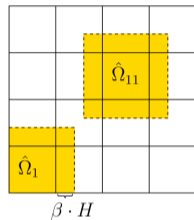
$$\Omega_i: \quad \bigcup_{i=1}^p \bar{\Omega}_i = \bar{\Omega}, \quad \Omega_i \cap \Omega_j = \emptyset \quad \forall i \neq j.$$



# Overlapping Domain Decomposition

- ▶ Extend each  $\Omega_i$  by an overlap  $\hat{\Omega}_i$  of width  $\beta \cdot H$ :

$$\hat{\Omega}_i = \{x \in \Omega \mid \text{dist}(x, \Omega_i) < \beta \cdot H\}$$



# Contents

---

Introduction

Domain Decomposition

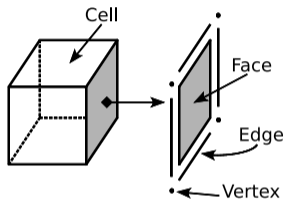
**Parallel Grids**

Communicating Data with Dune

Parallel PDELab

# Recapitulation: The Grid: A Container of Entities

In the DUNE sense a *Grid* is a container of entities:

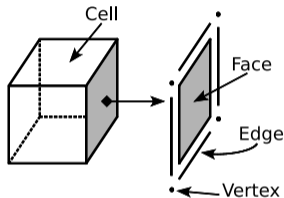


- ▶ vertices ,
- ▶ edges ,
- ▶ faces ,
- ▶ cells , ...



# Recapitulation: The Grid: A Container of Entities

In the DUNE sense a *Grid* is a container of entities:

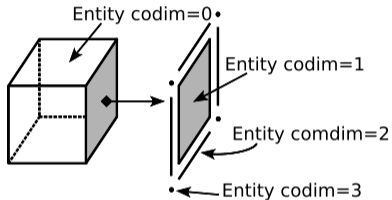


- ▶ vertices ,
- ▶ edges ,
- ▶ faces ,
- ▶ cells , ...

In order to do dimension independent programming, we need a dimension independent naming for different entities.

# Recapitulation: The Grid: A Container of Entities

In the DUNE sense a *Grid* is a container of entities:



- ▶ vertices (*Entity codim* =  $d$ ),
- ▶ edges (*Entity codim* =  $d - 1$ ),
- ▶ faces (*Entity codim* = 1),
- ▶ cells (*Entity codim* = 0), ...

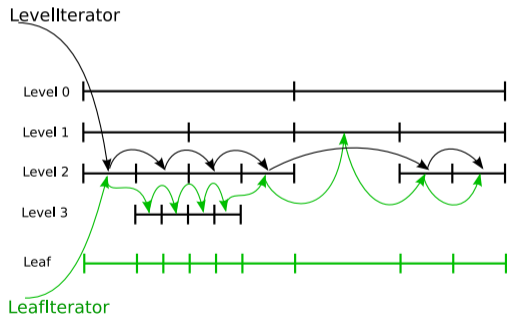
In order to do dimension independent programming, we need a dimension independent naming for different entities.

We distinguish entities according to their codimension.

Entities of  $\text{codim} = c$  contain subentities of  $\text{codim} = c + 1$ . This gives a recursive construction down to  $\text{codim} = d$ .

# Recapitulation: Iterators

Access to the entities of a grid is given by iterators provided by a GridView. DUNE provides appropriate iterators for both LeafGridView and LevelGridView.



`GridView::Codim<c>::Iterator` iterates over codimension  $c$  entities on a given view.

# Recapitulation: Entities

Iterating over a grid view, we get access to the entities.

```
template<class GridView>
void do_something(const GridView &gv) {
    // iterate over the grid
    for (auto entity : entities(gv, DUNE::Codim<0>)) {
        ...
    }
}
```

- ▶ Entities cannot be modified.
- ▶ Entities can be copied and stored (but copies may be expensive).
- ▶ Entities provide topological and geometrical information.

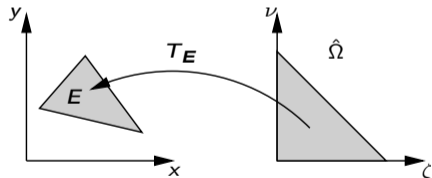
# Recapitulation: Entities

An Entity  $E$  provides both topological information

- ▶ Type of the entity (triangle, quadrilateral, etc.).
- ▶ Relations to other entities.

and geometrical information

- ▶ Position of the entity in the grid.



Mapping from  $\hat{\Omega}$  into global coordinates.

`GridView::Codim<c>::Entity` implements the entity concept.

Entity  $E$  is defined by...

- ▶ Reference Element  $\hat{\Omega}$
- ▶ Transformation  $T_E$

# Partition Types

- ▶ Each grid entity can be present on one or more processes.
- ▶ Each entity on one process has a partition type, which can be determined by the method

```
entity.partitionType()
```

- ▶ The possible partition types are:

***interior*** Entity is owned by the process

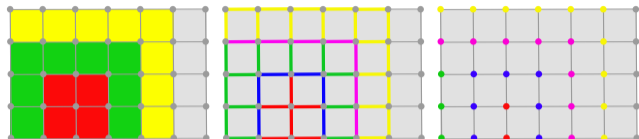
***overlap*** Entity is owned by a different process, but a full copy exists

***ghost*** Entity is owned by a different process, but a partial copy exists

***border*** Boundary of interior. (only exists for entities with  $\text{codimension} > 0$ )

***front*** Boundary of interior+overlap if not ***border*** (only exists for entities with  $\text{codimension} > 0$ )

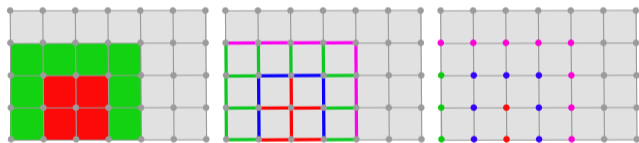
# Partition Types Example



$c = 0$

$c = 1$

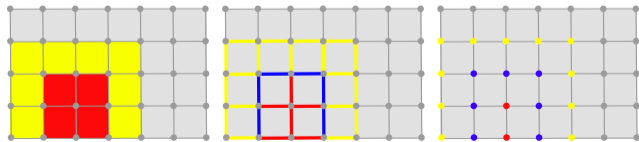
$c = 2$



$c = 0$

$c = 1$

$c = 2$



$c = 0$

$c = 1$

$c = 2$

**First row:** with overlap and ghosts  
**Second row:** with overlap only  
**Third row:** with ghosts only

interior
overlap
ghost
border
front
not stored

# Parallel Grids in Dune

---

- ▶ YaspGrid
  - ▶ structured
  - ▶ n-dimensional
  - ▶ arbitrary overlap



# Parallel Grids in Dune

---

- ▶ YaspGrid
  - ▶ structured
  - ▶ n-dimensional
  - ▶ arbitrary overlap
- ▶ Unstructured 2D/3D grids with full load-balancing:
  - ▶ UGGrid
    - ▶ multi-element (e.g. tetrahedrons, pyramids, prisms and hexahedrons simultaneously in 3D)
    - ▶ non-conforming/conforming refinement
    - ▶ ghost cells

# Parallel Grids in Dune

- ▶ YaspGrid
  - ▶ structured
  - ▶ n-dimensional
  - ▶ arbitrary overlap
- ▶ Unstructured 2D/3D grids with full load-balancing:
  - ▶ UGGrid
    - ▶ multi-element (e.g. tetrahedrons, pyramids, prisms and hexahedrons simultaneously in 3D)
    - ▶ non-conforming/conforming refinement
    - ▶ ghost cells
  - ▶ ALUGrid
    - ▶ either tetrahedral or hexahedral elements
    - ▶ non-conforming refinement, conforming bisection refinement for 3D tetrahedral grid
    - ▶ ghost cells (non-conforming grids only)

# Iterators in a parallel Grid

Dune offers Iterators which only iterate over elements with certain partition types. The partition type can be specified as additional parameter in the range based for loop, e.g.

```
for (const auto &cell : elements(gv, Dune::Partitions::Interior) {  
    ...  
}
```

`Dune::Partitions` contains the following partitions:

Interior	interior entities only
Border	border entities only
Overlap	overlap entities only
Front	front entities only
InteriorBorder	interior entities plus border (identical to Interior for entities of codimension==0)
InteriorBorderOverlap	interior, border and overlap entities
InteriorBorderOverlapFront	interior, border, overlap and front entities
Ghost	ghost entities only
All	all entities available to the process

## Example

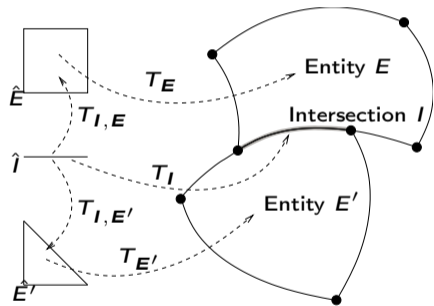
```
#include <dune/grid/somegrid.hh>

void iterate_the_grid() {
    // we have a grid
    Dune::SomeGrid grid(parameters);

    // iterate over the interior Border Partition of level 2
    auto levelGV = grid.levelGridView(2);
    for (const auto &face : facets(levelGV, Dune::Partitions::InteriorBorder) {
        ...
    }

    // iterate over all partition of the leaf
    auto leafGV = grid.leafGridView();
    for (const auto &node : entities(leafGV, Dune::Codim<DIM>, Dune::Partitions::All) {
        ...
    }
}
```

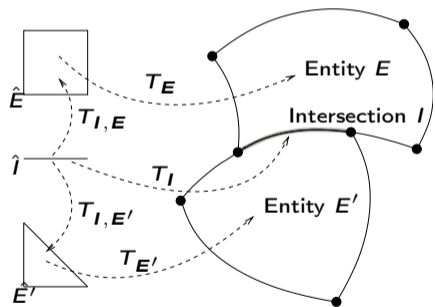
# Repetition: Intersections



- ▶ Grids may be non conforming.
- ▶ Entities can intersect with neighbours and boundary.
- ▶ IntersectionIterators give access to intersections of an Entity in a given view.
- ▶ Intersections hold topological and geometrical information.
- ▶ Intersections depend on the view:
- ▶ **Note:** Intersections are always of codimension 1!

# Intersection Interface

Iterating over intersections in entity  $E$  yields an Intersection to  $E'$  with the methods:



Method name	Result
boundary	Boolean
neighbor	Boolean
inside	Entity $E$
outside	Entity $E'$
geometry	Geometry $T_I$
geometryInInside	Geometry $T_{I,E}$
geometryInOutside	Geometry $T_{I,E'}$
unitOuterNormal	outer normal $n$ , $ n  = 1$
centerUnitOuterNormal	outer normal at <code>geometry().center()</code>

# Intersections and Domain Decomposition

---

- ▶ On each intersection there exists a method `neighbor()`. This method returns `true` if there is a neighbor available on the same process (even if it is a ghost).
- ▶ The method `boundary()` only returns `true` at the domain boundary (even if the grid is periodic at this boundary) not at a process boundary.
- ▶ If there is no neighbor but also no domain boundary, there is a process boundary.

## Example

```
for (const auto &cell : elements(gv, Dune::Partitions::InteriorBorder))
  for(const auto &is : intersections(gv, cell)) {
    // evaluate fluxes
    Dune::FieldVector<ctype, dim> center = is.geometry().center();
    // neighbor intersection
    if (is.neighbor()) {
      // mean flux
      flux = ( myshapefkt.gradient(center)
              - nbshapefkt.gradient(center) )
            * is.centerUnitOuterNormal()
            * is.geometry().volume();
    }
    // boundary intersection
    else if (is.boundary()) {
      // neumann boundary condition
      flux = j(center);
    }
  }
}
```



# Load-Balancing

---

- ▶ parallelization only scales well if all processes have the same work load  
⇒ well balanced grids necessary
- ▶ adaptation leads to unbalanced work load
- ▶ UGGrid and 3D ALUGrid provide methods to re-balance the work load after grid adaptation

`loadBalance(DataHandle &data)` :

re-balances a parallel grid, optionally sends also user data

`DataHandle` :

works like the data handle for the communicate methods

# Grid-Distribution with YaspGrid

With YaspGrid you can determine how the grid is partitioned (adaptive grid refinement and load-balancing are not possible as it is a structured grid) by writing a class derived from `Dune::YLoadBalance<dim>`

```
template<int dim>
class YaspPartition : public Dune::YLoadBalance<dim> {
public:
    using iTuple = std::array<int, dim>;
    void loadbalance (const iTuple& size, int P, iTuple& dims) const {
        for (auto& x : dims)
            x = 1;
        dims.back() = P;
    }
};
```

# Grid-Distribution with YaspGrid

---

Now you can pass the object to the constructor during grid creation

```
const int dim=2;
Dune::FieldVector<double,dim> upper(1.0);
auto n = Dune::filledArray<dim,int>(10);
std::bitset<dim> periodic(false);
int overlap = 1;
YaspPartition<2> yp;
YaspGrid<2> grid(upper,n,periodic,overlap,helper.getCommunicator(),&yp);
```

# Contents

---

Introduction

Domain Decomposition

Parallel Grids

**Communicating Data with Dune**

Parallel PDELab

## Communicating Data with Dune

---

- ▶ Data is associated with grid entities using an `IndexSet`.
- ▶ The index set provides indices for all entities stored by the process (i.e. the `Dune::Partitions::All`)
- ▶ Data is stored locally.
- ▶ Algorithms may require data exchange e.g. for synchronization or the calculation of updates
- ▶ Dune provides methods for the communication of data and methods for collective communication

# DUNE Lowlevel Communication API

GridView provides a method for the communication between processes

```
template<class DHImp, class DataType>
void communicate (CommDataHandleIF<DHImp, DataType> &datahandle,
                 InterfaceType interface,
                 CommunicationDirection dir) const;
```

where

- ▶ `CommDataHandleIF`  
is a user defined class describing what data should be communicated. The class has to provide methods to assemble the data on the source process and distribute the data on the target process (see exercises).

# DUNE Lowlevel Communication API

```
template<class DHImp, class DataType>
void communicate ( CommDataHandleIF<DHImp, DataType> &datahandle,
                  InterfaceType interface,
                  CommunicationDirection dir ) const;
```

where

- ▶ **InterfaceType**

Determines the partition type of the entities to be sent and received. With `InteriorBorder_InteriorBorder_Interface` only border entities are sent. With `All_All_Interface`, `InteriorBorder_All_Interface` and `Overlap_All_Interface` all entities, only interior and border entities or only overlap entities are sent. Only processes with common data communicate and only the entities present on both processes are included in the communication.

- ▶ **CommunicationDirection** The direction of the communication can be changed with either `ForwardDirection` or `BackwardDirection`

# Collective Communication

---

- ▶ parallel computations require global communication (e.g. `sum(defect)` or `min( $\Delta t$ )` and synchronization, e.g. a barrier needed for timing)
- ▶ You can get a collective communication object by the following method of a `GridView`:

```
const CollectiveCommunication & comm () const;
```



# Collective Communication

The class `Dune::Grid::CollectiveCommunication` provides comfortable access to a lot of MPI methods, e.g.

Method name	Description
<code>rank</code>	obtain number (rank) of this process
<code>size</code>	obtain number of processes
<code>barrier</code>	wait until all process arrived at the barrier
<code>min</code>	global min of local values
<code>max</code>	global max of local values
<code>sum</code>	global sum of local values
<code>allreduce</code>	Compute something over all processes for each component of an array and return result in every process
<code>broadcast</code>	broadcast from one process to all other processes
<code>scatter</code>	scatter individual data from root process to all other tasks
<code>gather, allgather</code>	gather data on root process (and distribute it to all other tasks)

## Example

```
// get communication object from gridview
auto comm = gridView.comm();
// get rank from communication object
int myRank = comm.rank();
// get number of processes
int numProcs = comm.size();
// calculate global sum (using MPI_Reduce)
double globalsum=comm.sum(localResult);
// calculate global maximum (using MPI_Reduce)
double globalmax=comm.max(localResult);
// broadcast result
comm.broadcast(&globalMax,1,0);
```

# MPIHelper

Dune parallel programs use a tool to help in setting up and handling the parallel communication with MPI. It also takes care that the parallel program is finished in a defined way. It is called MPIHelper. It has to be created at the very beginning of the program using the instance method of the `Dune::MPIHelper` class.

```
int main(int argc, char** argv) {
    try {
        //Maybe initialize Mpi
        Dune::MPIHelper& helper = Dune::MPIHelper::instance(argc, argv);
        if(Dune::MPIHelper::isFake)
            std::cout << "This_is_a_sequential_program." << '\n';
        else {
            if(helper.rank()==0)
                std::cout << "parallel_run_on_" << helper.size() << "_process(es)
                    " << '\n';
        }
    }
}
```

- ▶ MPIHelper provides the methods

Method name	Description
rank	obtain number (rank) of this process
size	obtain number of processes

- ▶ MPIHelper provides the static methods

Method name	Description
getCommunicator	get communicator to exchange data with all process (MPI_COMM_WORLD)
getLocalCommunicator	get communicator to exchange data with the local process only (MPI_COMM_SELF)
getCollectiveCommunication	get collective communication object for MPI_COMM_WORLD
instance	get access to the helper singleton

- ▶ MPIHelper additionally provides an `enum isFake` which is `true` if the program was compiled without MPI support

## Norms and Scalar-Products on Parallel Grids

If you have a parallel grid and for some reason want to calculate norms or scalar products of vectors associated with degrees of freedom, you cannot calculate them directly, as border entities exist on more than one process.

For an overlapping grid you need the class `OverlappingScalarProduct`. Additionally you need the auxiliary class `ISTL::ParallelHelper`.

```
std::vector<double> dataVector(gv.indexSet().size(dim));  
// obtain data from some calculations  
Dune::PDELab::ISTL::ParallelHelper<GFS> parHelper(gfs);  
Dune::PDELab::OverlappingScalarProduct<GFS, std::vector<double>> ovlpScalProd(gfs,  
    parHelper);  
// calculate norm  
norm=ovlpScalProd.norm(dataVector);
```

# Norms and Scalar-Products on Parallel Grids

For a non-overlapping grid the respective class is `NonoverlappingScalarProduct`:

```
std::vector<double> dataVector(gv.indexSet().size(dim));  
// obtain data from some calculations  
Dune::PDELab::ISTL::ParallelHelper<GFS> parHelper(gfs);  
Dune::PDELab::NonoverlappingScalarProduct<GFS, std::vector<double>> novlpScalProd(  
    gfs, parHelper);  
// calculate norm via scalar product  
double norm=sqrt(novlpScalProd.dot(dataVector, dataVector));
```

# Contents

---

Introduction

Domain Decomposition

Parallel Grids

Communicating Data with Dune

**Parallel PDELab**

Parallel computing in PDELab is very easy.

- ▶ Go parallel by choosing
  1. a suitable parallel grid,
  2. the correct constraints for the discretization of the PDE (either `OverlappingConformingDirichletConstraints` or `NonoverlappingConformingDirichletConstraints<GV>`), and
  3. a suitable and matching parallel solver backend of the PDELab backend.



## Parallel Solver Backends

---

- ▶ ISTL solvers need to be provided a Preconditioner (like Jacobi, SSOR or ILU), a LinearOperator (providing a matrix-vector product) and a ScalarProduct. The versions of these components have to fit together.
- ▶ Parallel solver backends make sure that the correct implementations of Preconditioner, LinearOperator and ScalarProduct are chosen matching the type of domain decomposition.
- ▶ Different solver backends are provided for overlapping and nonoverlapping domain decomposition.
- ▶ The parallel solver backends can be found in the headers `dune/pdelab/backend/istl/ovlpistlsolverbackend.hh` and `dune/pdelab/backend/istl/novlpistlsolverbackend.hh`, which are automatically included by `istl.hh`.

# Parallel Preconditioners

- ▶ To run in parallel Conjugate Gradients (CG) and BiCGStab solvers have to be able to compute parallel matrix vector products and scalar products.
- ▶ As parallel preconditioners to the CG and BiCGStab solvers additive Schwarz methods can be used:
  - ▶ In this schemes a local subproblem on each process is solved where the values of the last iteration are used as Dirichlet constraints at the process boundary.
  - ▶ Different solvers can be chosen for the local problems (e.g. a direct solver like SuperLU or some steps of an iterative solver like SSOR).
  - ▶ In an overlapping decomposition the corrections are computed for the overlap at more than one process. The sum of the corrections multiplied with a relaxation coefficient is applied.
  - ▶ With an overlapping Schwarz method the convergence is better the larger the overlap.
- ▶ There exists also an algebraic multigrid preconditioner for overlapping as well as non-overlapping domain decomposition.

## Parallel Solver Backends for Overlapping DD

The linear solvers in this table are preconditioned with an overlapping domain decomposition using the respective smoother or with a parallel algebraic multigrid scheme with an SSOR smoother (AMG).

solver backend	smoother	linear solver
ISTLBackend_OVLP_CG_SSORk<GFS,CC>	SSOR	CG
ISTLBackend_OVLP_CG_SuperLU<GFS,CC>	SuperLU	CG
ISTLBackend_OVLP_CG_UMFPack<GFS,CC>	UMFPack	CG
ISTLBackend_CG_AMG_SSOR<GO>	AMG	CG
ISTLBackend_OVLP_BCGS_SSORk<GFS,CC>	SSOR	BiCGStab
ISTLBackend_OVLP_BCGS_ILU0<GFS,CC>	ILU0	BiCGStab
ISTLBackend_OVLP_BCGS_SuperLU<GFS,CC>	SuperLU	BiCGStab
ISTLBackend_BCGS_AMG_SSOR<GO>	AMG	BiCGStab

ISTLBackend\_OVLP\_ExplicitDiagonal<GFS> is a solver for explicit time-steppers with (block-) diagonal mass matrix.

The template parameter GFS is the grid function space, GO is the grid operator, CC is the type of the constraints container (usually `OverlappingConformingDirichletConstraints`).

# Overlapping Example

```
// 1. Create an overlapping grid
Dune::FieldVector<double,2> L(1.0);
auto N = Dune::filledArray<2, int>(16);
std::bitset<2> periodic (false);
int overlap=2;
Dune::YaspGrid<2> grid(L,N,periodic,overlap);
using GV = Dune::YaspGrid<2>::LeafGridView;
const GV& gv=grid.leafView();

// 2. Create correctly constrained grid function space
using FEM = Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim>;
FEM fem;
using CON = Dune::PDELab::OverlappingConformingDirichletConstraints;
using VBE = Dune::PDELab::ISTLVectorBackend<>;
using GFS = Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE>;
GFS gfs(gv,fem);
```

## Overlapping Example Continued

```
// define problem parameters
using Param = ConvectionDiffusionProblem<GV,Real>;
Param param;
using B = Dune::PDELab::BCTypeParam_CD<Param>;
B b(gv,param);
using G = Dune::PDELab::DirichletBoundaryCondition_CD<Param>;
G g(gv,param);

// Compute constrained space
using C = typename GFS::template ConstraintsContainer<Real>::Type;
C cg;
Dune::PDELab::constraints(b,gfs,cg);
// Make grid operator
using LOP = Dune::PDELab::ConvectionDiffusion<Param>;
LOP lop(param,2);
using MBE = Dune::PDELab::ISTLMatrixBackend;
using GO = Dune::PDELab::GridOperator<GFS,GFS,LOP,MBE,double,double,double,C,C>;
GO go(gfs,cg,gfs,cg,lop);
```

## Overlapping Example Continued

---

```
// Compute affine shift
using V = typename GO::Traits::Domain;
V x(gfs,0.0);
Dune::PDELab::interpolate(g,gfs,x);
Dune::PDELab::set_nonconstrained_dofs(cg,0.0,x);

// 3. Choose a linear solver
using LS = Dune::PDELab::ISTLBackend_OVLP_BCGS_SuperLU<GFS,C>;
LS ls(gfs,cg,5000,2);
...
```

## Parallel Solver Backends for Nonoverlapping DD

The linear solvers in this table are preconditioned with a nonoverlapping domain decomposition using the respective smoother.

solver backend	smoother	linear solver
<code>ISTLBackend_NOVLP_CG_NOPREC&lt;GFS&gt;</code>	–	CG
<code>ISTLBackend_NOVLP_CG_Jacobi&lt;GFS&gt;</code>	Jacobi	CG
<code>ISTLBackend_NOVLP_CG_SSORk&lt;GO&gt;</code>	SSOR	CG
<code>ISTLBackend_NOVLP_CG_AMG_SSOR&lt;GO&gt;</code>	AMG	CG
<code>ISTLBackend_NOVLP_BCGS_NOPREC&lt;GFS&gt;</code>	–	BiCGStab
<code>ISTLBackend_NOVLP_BCGS_Jacobi&lt;GFS&gt;</code>	Jacobi	BiCGStab
<code>ISTLBackend_NOVLP_BCGS_SSORk&lt;GO&gt;</code>	SSOR	BiCGStab
<code>ISTLBackend_NOVLP_BCGS_AMG_SSOR&lt;GO&gt;</code>	AMG	BiCGStab

`ISTLBackend_NOVLP_ExplicitDiagonal` is a solver for explicit time-steppers with (block-) diagonal mass matrix.

The template parameter is either `GFS` the grid function space or `GO` the grid operator depending on the preconditioner.

# Nonoverlapping example

```
// 1. Create an overlapping grid
Dune::FieldVector<double,2> L(1.0);
auto N = Dune::filledArray<int, 2>(16);
std::bitset<2> periodic (false);
int overlap=0; // overlap 0 as overlap elements are not assembled
Dune::YaspGrid<2> grid(L,N,periodic,overlap);
using GV = Dune::YaspGrid<2>::LeafGridView;
const GV& gv=grid.leafView();

// 2. Create correctly constrained grid function space
using FEM = Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim>;
FEM fem;
using CON = Dune::PDELab::NonoverlappingConformingDirichletConstraints<GV>;
CON con(gv);
using VBE = Dune::PDELab::ISTLVectorBackend<>;
using GFS = Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE>;
GFS gfs(gv,fem,con);
con.compute_ghosts(gfs); // con stores indices of ghost dofs
using Param = ConvectionDiffusionProblem<GV,Real>;
Param param;
```



## Nonoverlapping Example Continued

```
using B = Dune::PDELab::BCTypeParam_CD<Param>;
B b(gv,param);
using G = Dune::PDELab::DirichletBoundaryCondition_CD<Param>;
G g(gv,param);
// Compute constrained space
using C = typename GFS::template ConstraintsContainer<Real>::Type;
C cg;
Dune::PDELab::constraints(b,gfs,cg);

// Make grid operator
using LOP = Dune::PDELab::ConvectionDiffusion<Param>;
LOP lop(param,2);
using MBE = Dune::PDELab::ISTLMatrixBackend;
using GO = Dune::PDELab::GridOperator<GFS,GFS,LOP,MBE,double,double,double,C,C,
    true>;
GO gos(gfs,cg,gfs,cg,lop);
```

## Nonoverlapping Example Continued

---

```
// Compute affine shift
using V = typename GO::Traits::Domain;
V x(gfs,0.0);
Dune::PDELab::interpolate(g,gfs,x);
Dune::PDELab::set_nonconstrained_dofs(cg,0.0,x);

// 3. Choose a linear solver
using LS = Dune::PDELab::ISTLBackend_NOVLP_BCGS_SSORk<GO>;
LS ls(go,5000,3,2);
...
```