DUNE PDELab Tutorial 06

Paralleles Rechnen

Olaf Ippisch

Institut für Mathematik, TU Clausthal
Erzstr. 1, D-38678 Clausthal-Zellerfeld

August 31, 2021

Why Parallel Computing?

- The speed of individual computer cores is not increasing essentially since some 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 10-core processors)
- Computer cluster with several multi-core multi-processor servers are affordable even for small companies

Why Parallel Computing?

The worlds three fastest computers have

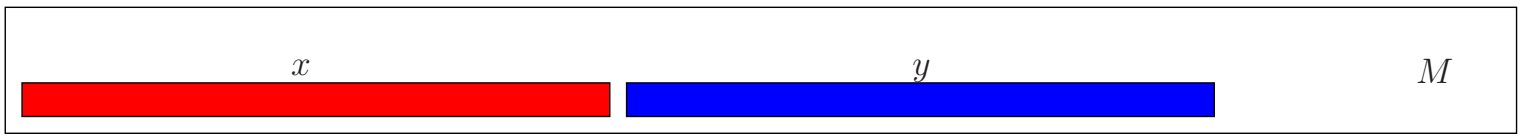
- Sunway TaihuLight, Wuxi, China: 93.0 PFlop/s
  10'649'600 cores:
  - 40'960 SW26010 processors, 260 cores
- Tianhe-2, Guangzhou, China: 33.8 PFlop/s
  3'120'000 cores:
  - 32'000 Intel Xeon 12-core processors
  - 48'000 Intel Phi 57-core accelerator cards
- Titan, Oak Ridge National Laboratory, U.S.A.: 17.6 PFlop/s
  299'008 cores:
  - 18'688 AMD Opteron 16-core processors
  - 18'688 NVIDIA Kepler K20X GPU accelerator cards

Architectures of Parallel Computers

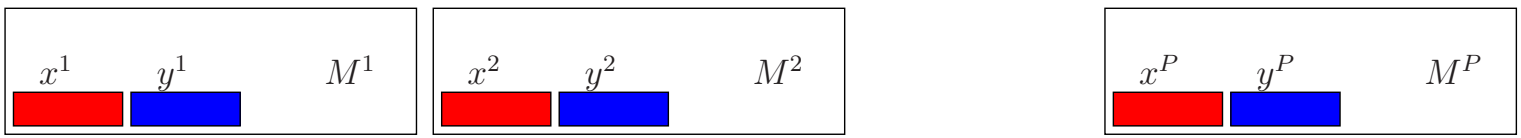
- Shared-memory computing: all cores have access to the whole memory
  - Uniform memory access architecture (UMA):
    access to every memory location from every process takes the same amount of time (some multi-core CPUs)
  - Non-uniform memory access architecture (NUMA):
    memory is associated with a processor or a group of processor cores but address space is global. Local memory can be accessed faster than memory attached to other processes (some 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 $s = \sum_{i=0}^{N-1} \mathbf{x}_i \, \mathbf{y}_i$:
  (1) Subdivide index set into P pieces.
  (2) Compute $s_p = \sum_{i=pN/P}^{(p+1)N/P-1} \mathbf{x}_i \, \mathbf{y}_i$ in parallel.
  (3) Compute $s = \sum_{i=0}^{P-1} s_i$.
- *Uniform memory access architecture*: Store vectors as in sequential program:



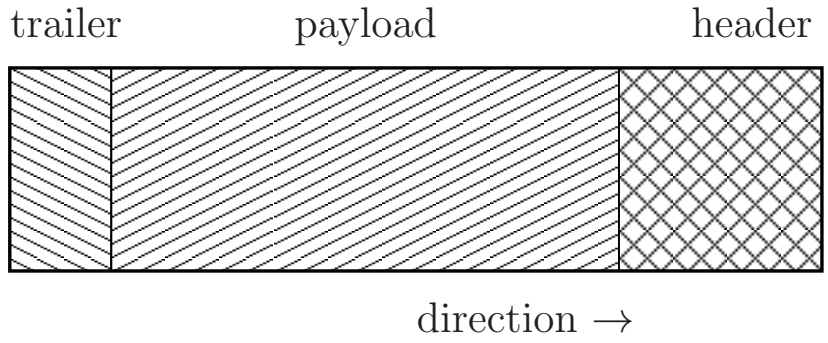
- *Nonuniform memory access architecture*: Distribute data to the local memories:



- *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 the other.
- During transmission the message is subdivided into individual packets.
- Network is packet-switched.
- A packet consists of an envelope and the data:



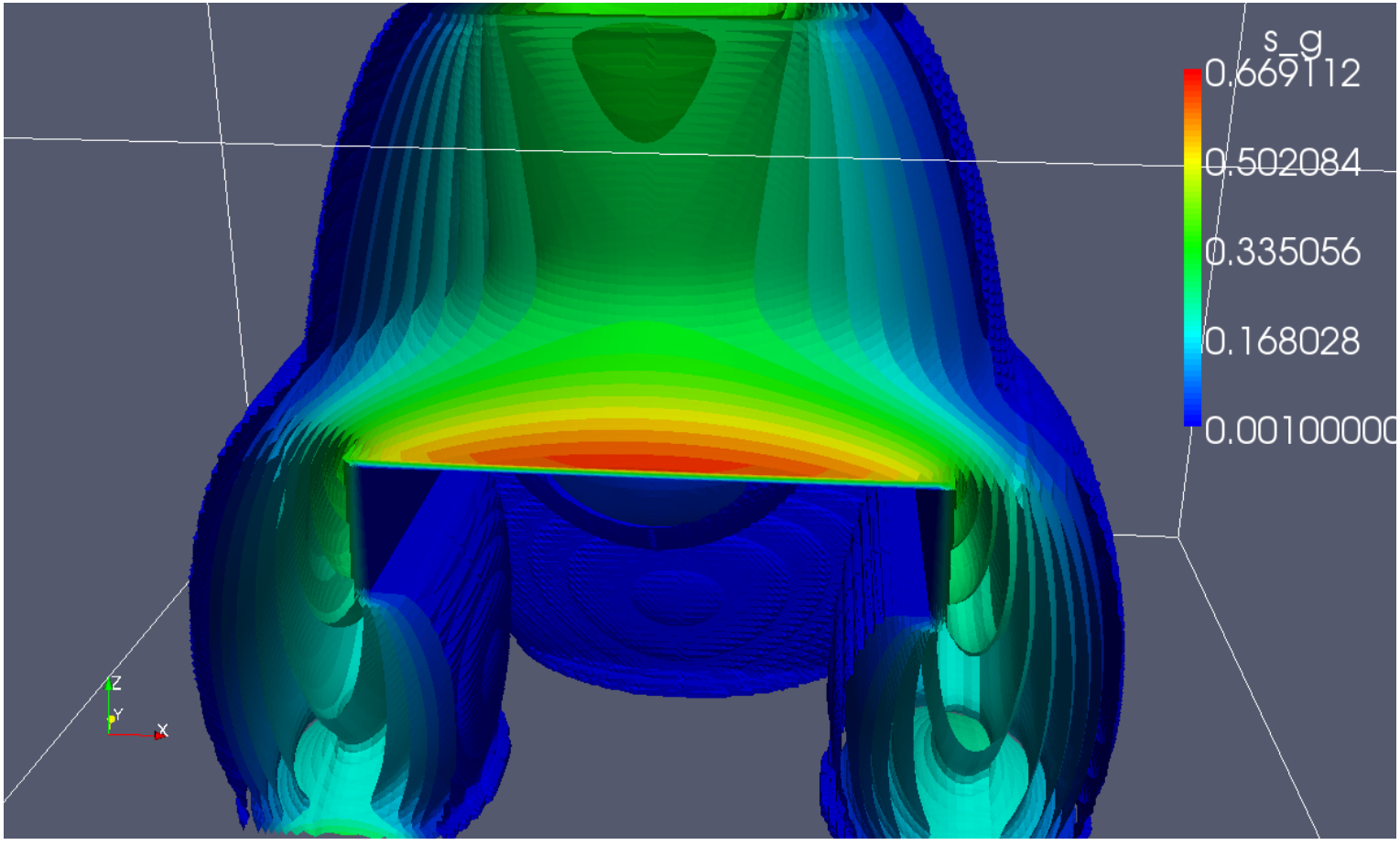
- Header: Destination, size and kind of data.
- Payload: Size ranges from some bytes to kilobytes.
- Trailer: e.g. checksum for error detection.

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 also for Linux Clusters: **MPICH** and **OpenMPI** [1]
- 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

[1] `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 4×12 AMD Magny Cours, 2.1 GHz, 12×0.5MB L2, 12MB L3 processors. Computation time for one time step with BiCGStab + AMG prec.:

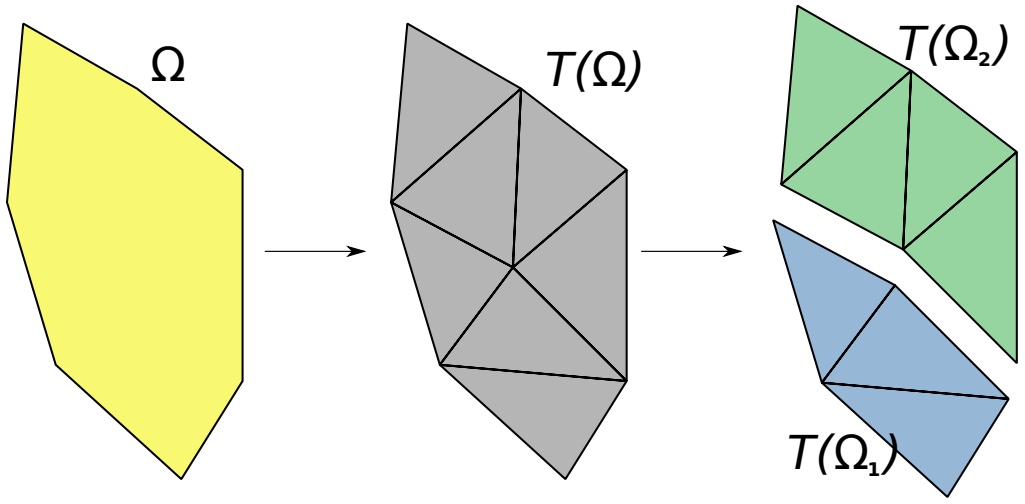
P	#IT(max)	T_{it}	S	T_{asm}	S	T_{total}	S
1	6.5	4.60	-	43.7	-	713.8	-
4	10	1.85	2.5	17.5	2.5	295.9	2.4
8	9	0.63	7.3	8.4	5.2	127.1	5.6
16	9.5	0.40	11.5	4.1	10.7	73.1	9.8
32	15	0.27	17.0	1.9	23.0	43.5	16.4

Comparison with T3E from 1999

Machine	Cells	Time steps	Newton steps	T_{total}
256 T3E	2621440	50	264	14719
16 Cores AMD	2457600	50	231	2500

Domain Decomposition

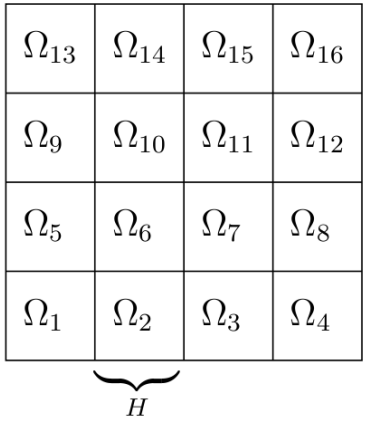
- partition a problem by splitting the domain into smaller subdomains
- each part is solved by a different process
- goes back to an idea of H.A. Schwarz who in 1890 presented a method to prove the existence of solutions of the Laplace equation on “complicated” domains.
- Different variants:
  - overlapping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

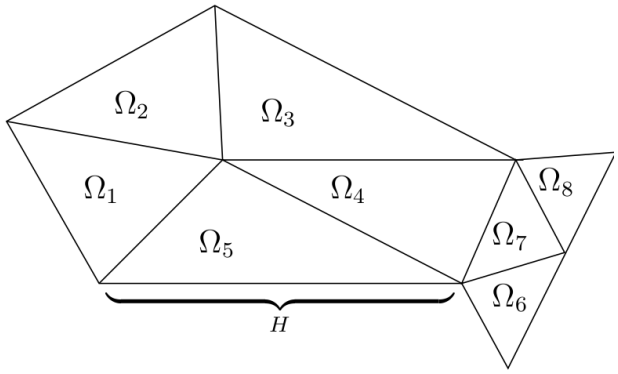


Nonoverlapping Domain Decomposition

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

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

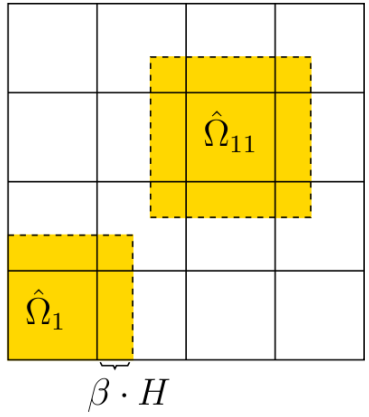


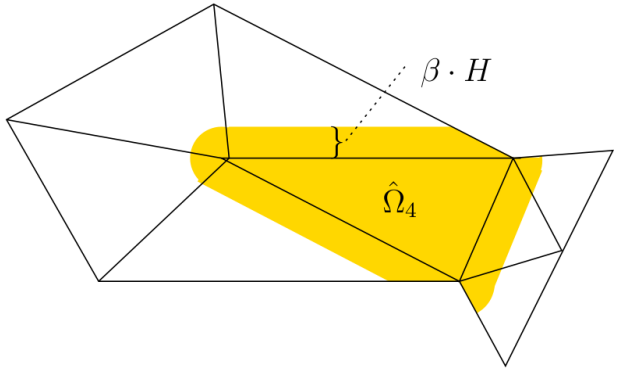


Overlapping Domain Decomposition

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

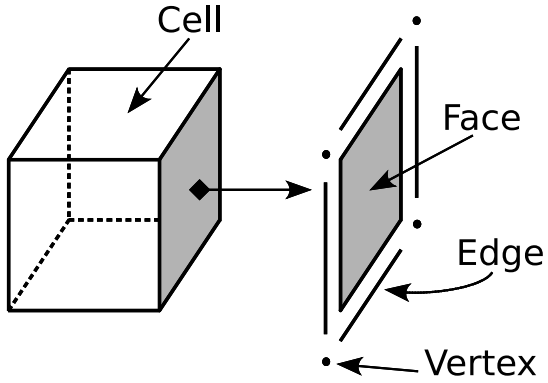
$$\hat{\Omega}_i = \{x \in \Omega \mid \text{dist}(x, \Omega_i) < \beta \cdot H\}$$





Recapitulation: The Grid: A Container of Entities

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

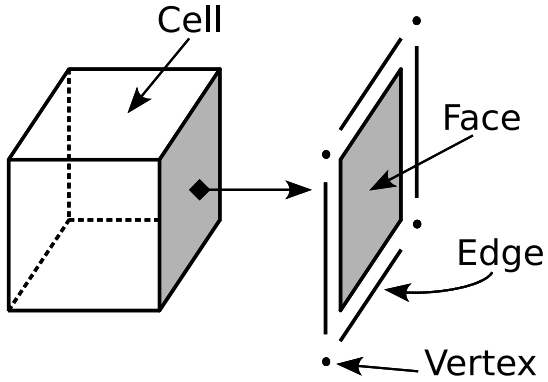


- vertices ,
- edges ,
- faces ,
- cells , . . .

In order to do dimension independent programming, we need a dimension independent naming for different entities.
We distinguish entities according to their codimension.
Entities of codim $= c$ contain subentities of codim $= c + 1$. This gives a recursive construction down to codim $= d$.

Recapitulation: The Grid: A Container of Entities

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



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

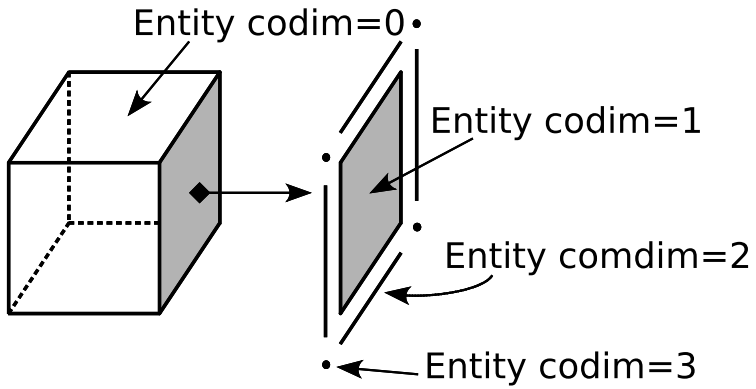
In order to do dimension independent programming, we need a dimension independent naming for different entities.

We distinguish entities according to their codimension.

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

Recapitulation: The Grid: A Container of Entities

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

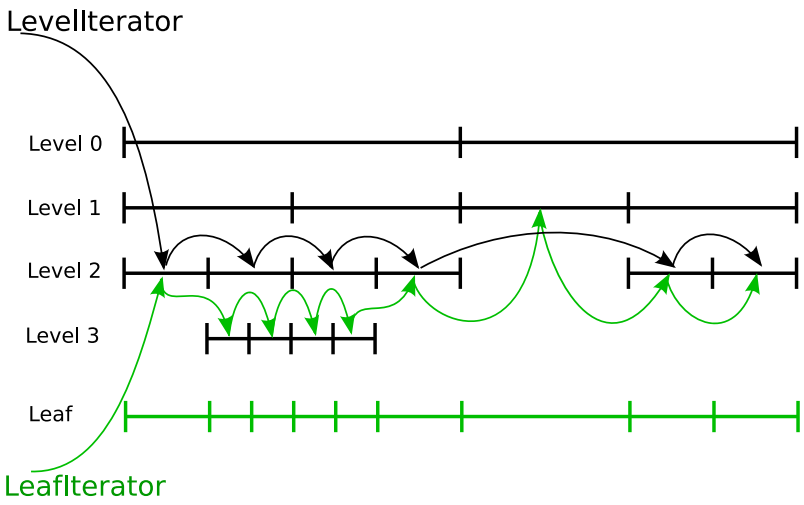


- vertices *(Entity codim* $= d$*)*,
- edges *(Entity codim* $= d - 1$*)*,
- faces *(Entity codim* $= 1$*)*,
- cells *(Entity codim* $= 0$*)*, ...

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 codim $= c$ contain subentities of codim $= c + 1$. This gives a recursive construction down to 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 &grid)
{
    // 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.

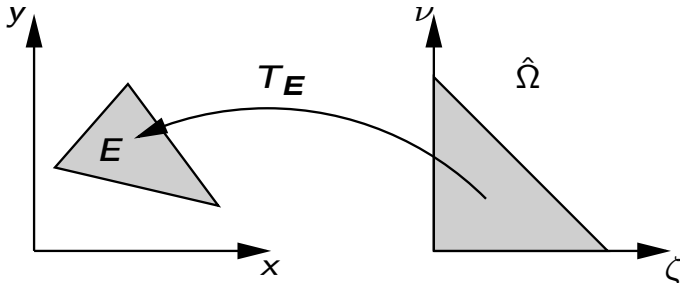
Recapitulation: Entities

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.

Entity E is defined by...

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

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

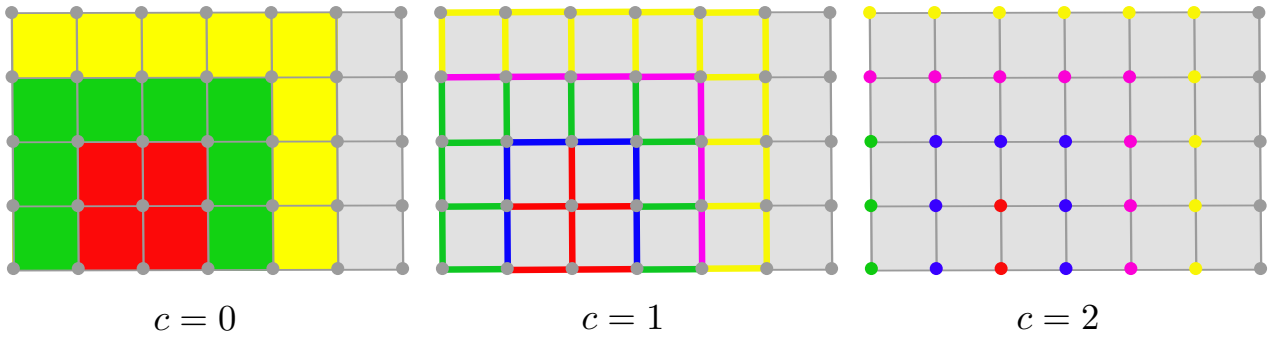
Partition Types

- 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

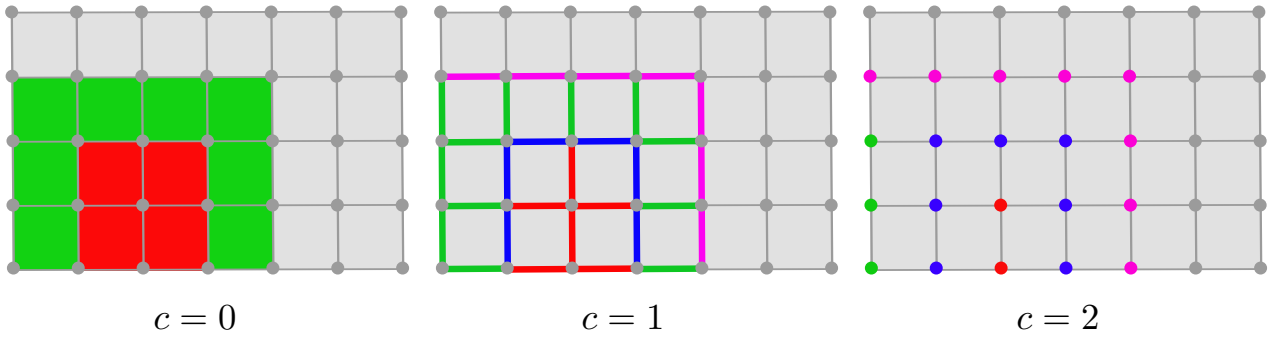
  `entity.partitionType()`
- The possible partition types are:

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

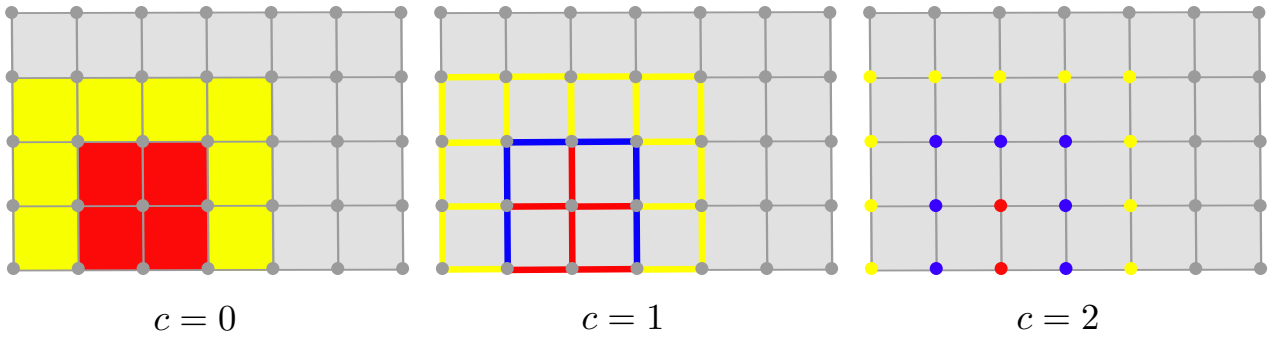
Partition Types Example



$c = 0$ $c = 1$ $c = 2$



$c = 0$ $c = 1$ $c = 2$



$c = 0$ $c = 1$ $c = 2$

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
  - nD
  - arbitrary overlap
- `UGGrid`
  - unstructured
  - 2D/3D
  - multi-element (e.g. tetrahedrons, pyramids, prisms and hexahedrons simultaneously in 3D)
  - non-conforming/conforming refinement
  - ghost cells
- `ALUGrid`
  - unstructured
  - 2D/3D
  - either tetrahedral or hexahedral elements
  - non-conforming refinement, conforming bisection refinement for 3D tetrahedral grid
  - ghost cells (non-conforming grids only)
  - full load-balancing

Parallel Grids in Dune

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

Parallel Grids in Dune

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

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

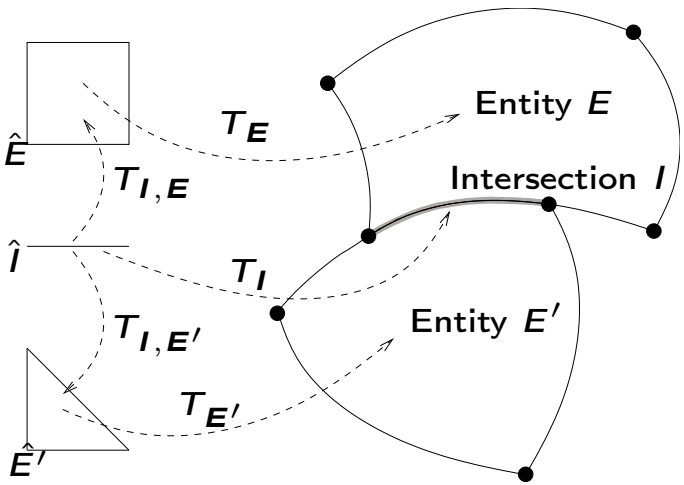
```
#include <dune/grid/somegrid.hh>

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) {
        ...
    }
}
```

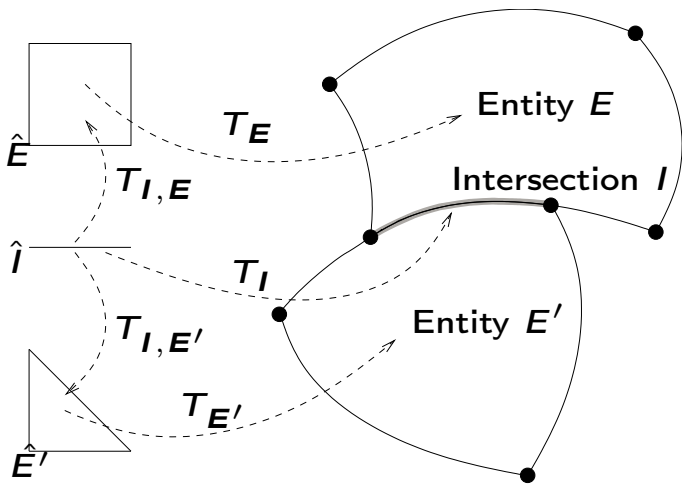
Repetition: Intersections



- 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, $\lvert n\rvert = 1$
`centerUnitOuterNormal`	outer normal at geometry().center()

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
  $\Rightarrow$ well balanced grids necessary
- adaptation leads to unbalanced work load
- only 3D ALUGrid provides a fully working method 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
- with UGGrid you can initialize a coarse grid and then call `loadBalance` before starting the computation.

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:
    typedef std::array<int, dim>  iTuple;
    void loadbalance (const iTuple& size, int P, iTuple&
        dims) const
    {
      dims[0] = 1;
      dims[1] = 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);
```

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 write 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(Δt) and synchronization (e.g. a barrier needed for a timing)
- You can get a collective communicator 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
`rank`	obtain number (rank) of this process
`size`	obtain number of processes
`barrier`	wait until all process arrived at the barrier
`min`	global min of local values
`max`	global max of local values
`sum`	global sum of local values
`allreduce`	Compute something over all processes for each component of an array and return result in every process
`broadcast`	broadcast from one process to all other processes
`scatter`	scatter individual data from root process to all other tasks
`gather, allgather`	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." << std::
          endl;
    else
      {
        if(helper.rank()==0)
          std::cout << "parallel␣run␣on␣" << helper.size()
              << "␣process(es)" << std::endl;
      }
```

MPIHelper

- 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`:

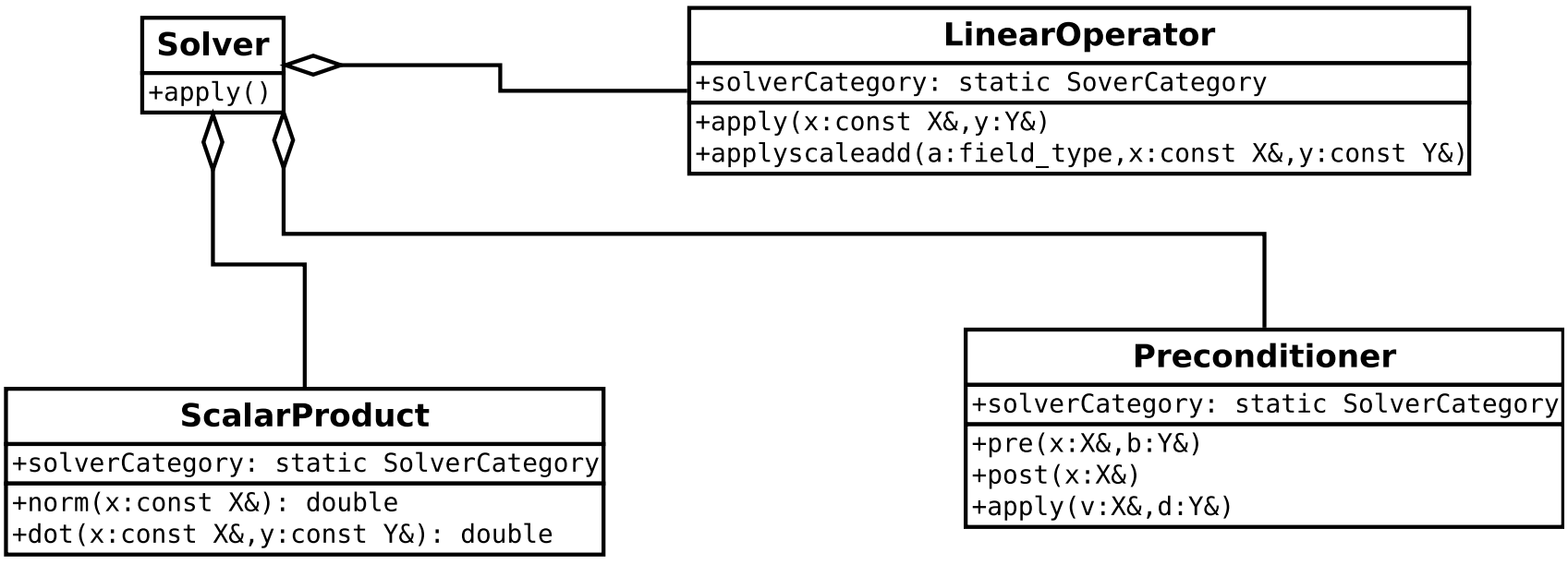
```
std::vector<double> dataVector(gv.indexSet().size(dim));
// obtain data from some calculations
Dune::PDELab::ISTL::ParallelHelper<GFS> parHelper(gfs);
Dune::PDELab::NonoverlappingScalarProduct<GFS,std::vector<
    double>> novlpScalProd(gfs,parHelper);
// calculate norm via scalar product
double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

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.

Building Blocks for Parallel Solvers



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);
typedef Dune::YaspGrid<2>::LeafGridView GV;
const GV& gv=grid.leafView();

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

Overlapping Example Continued

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

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

Overlapping Example Continued

```
//  Compute affine shift
typedef typename GO::Traits::Domain V;
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
typedef Dune::PDELab::ISTLBackend_OVLP_BCGS_SuperLU<GFS,C> LS;
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.

nnsolver backend	smoother	linear solver
`ISTLBackend_NOVLP_CG_NOPREC<GFS>`	–	CG
`ISTLBackend_NOVLP_CG_Jacobi<GFS>`	Jacobi	CG
`ISTLBackend_NOVLP_CG_SSORk<GO>`	SSOR	CG
`ISTLBackend_NOVLP_CG_AMG_SSOR<GO>`	AMG	CG
`ISTLBackend_NOVLP_BCGS_NOPREC<GFS>`	–	BiCGStab
`ISTLBackend_NOVLP_BCGS_Jacobi<GFS>`	Jacobi	BiCGStab
`ISTLBackend_NOVLP_BCGS_SSORk<GO>`	SSOR	BiCGStab
`ISTLBackend_NOVLP_BCGS_AMG_SSOR<GO>`	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);
typedef Dune::YaspGrid<2>::LeafGridView GV;
const GV& gv=grid.leafView();

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

Nonoverlapping Example Continued

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

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

Nonoverlapping Example Continued

```
// Compute affine shift
typedef typename GO::Traits::Domain V;
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
typedef Dune::PDELab::ISTLBackend_NOVLP_BCGS_SSORk<GO> LS;
LS ls(go,5000,3,2);
...
```