7. MPI: Data types and communicators

- Introduction
- Datatypes in MPI
- Constructing and using datatypes
- Groups and communicators
- Communicators with special topology
- Examples

Our running example: Conway's Game of Life

- Life is played on a grid of cells
- Each cell Is dead or alive
- Next generation
  - A dead cell turns alive, if there are exactly three neighbors
  - A living cell stays alive, if there are two or three neighbors
  - All other cells die (or stay dead)
- Why life?
  - Communication pattern is very common (8-stencil)
  - Example: heat equation
- We will play life on a 2D torus

Life: Parallelization strategy

- How to decompose the domain?
- Two basic approaches come to mind
  - Stripes
  - Patches
- Communication cost?
  - O(pn) versus O(sqrt(p)n)
  - We will use the latter

Life: Implementation in MPI

- We will attack a couple of issues, one by one
- How to send the data?
  - Assume each processor has a patch with border lines and columns contiguously in memory
  - After each iteration we have to update the border cells of each processor
- How to determine who sends to whom?
- What, if we want one process to visualize progress?
- What, if we want to implement checkpointing?
- How could we detect termination (steady state) efficiently?

Part 1

Datatypes

1 – Datatypes and packing

- Using basic datatypes messages have to be homogeneous and continuous in memory
- To avoid multiple messages for other types of data, MPI offers:
  - Sending messages of an user-defined *datatype*
  - Explicit *packing* of data
- *Datatypes* define the layout and the component types of data
  - The latter is called the type's *signature*
- *Packing* copies the data from and to a buffers

1 – Datatypes

- `MPI_Datatype` represent MPI datatypes
- MPI provides basic datatypes like `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`
  - They reflect the basic datatypes of the host language
- Type constructors allow to create complex types from elementar ones

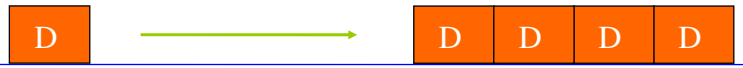
1 – Type constructors - Overview

```
MPI_Type_contiguous(int count,
  MPI_Datatype oldtype, MPI_Datatype *newtype);

MPI_Type_vector(int count,
  int blocklength, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype);

MPI_Type_indexed(int count,
  int *blocklenghts, int *displacements,
  MPI_Datatype oldtype, MPI_Datatype *newtype);

MPI_Type_struct(int count,
  int *blocklenghts, int *displacements, MPI_Datatype
  *oldtypes, MPI_Datatype *newtype);
```

1 – Datatype constructors

- `MPI_Type_contiguous` is the continuous concatenation of a base datatype

```
/* Array with 10 floating point values */
float values[10];

/* corresponding datatype in MPI */
MPI_Datatype FloatArray;
MPI_Type_contiguous(10, MPI_FLOAT, &FloatArray);
```



1 – Datatype constructors

- `MPI_type_vector` extends `MPI_Type_contiguous` for vectors of arbitrary stride.

```
/* Array of 1000 floating point values */
float Werte[1000];

/* We want to communicato only values 0,1 10,11 20,21 etc. */
MPI_Datatype FloatArray;
MPI_Type_vector(100, 2, 10, MPI_FLOAT, &FloatArray);
```


1 – Datatype constructors

- `MPI_Type_indexed` is even more general: Elements of the *same datatype* can be combined at arbitrary offset.

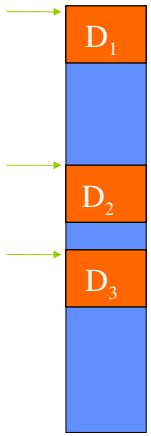
1 – Datatype constructors

- `MPI_Type_struct` is the most general constructor: Any number of different datatypes at arbitrary offsets can be combined
  - Corresponds naturally to struct and class types in C and C++

```
struct S {
  int i[2];
  double d[6];
  char c[100];
};
struct S foo;

MPI_Datatype   structure;
MPI_Datatype structure_type[3] =
   { MPI_INT, MPI_DOUBLE, MPI_CHAR };
```

1 – Datatype constructors

```
int num_elements[3] = {2,6,100};
MPI_Aint Offset[3];

MPI_Address(foo.i, &Offset[0]);
MPI_Address(foo.d, &Offset[1]);
MPI_Address(foo.c, &Offset[2]);

Offset[2]-=Offset[0];
Offset[1]-=Offset[0];
Offset[0]=0;

MPI_Type_struct(3, num_elements, Offset,
          structure_type, &structure);
```



1 – Usage of datatypes

- Before use in message the user has to „activate“ the datatype by calling `MPI_Type_commit`
- If not needed any more, a call to `MPI_Type_free` releases internal resources

1 – Usage of datatypes

```
MPI_Type_commit(MPI_Datatype *datatype);
MPI_Typ_free(MPI_Datatype *datatype);
```

```
MPI_Datatype FloatArray;
... /* initialize the datatype */

MPI_Type_commit(&FloatArray);
... /* use the datatype */
MPI_Type_free(&FloatArray);
```


1 – Usage of datatypes

- `MPI_Get_elements` returns the number of primitive elements in a datatype
- `MPI_Type_extent` returns the size of a datatype in memory
- `MPI_Type_size` returns the size of a message of the datatype
- Differences are due to alignment restrictions or holes in memory

1 – Usage of datatypes

```
MPI_Get_elements(MPI_Status *status,
                 MPI_Datatype type, int *count);
MPI_Type_extent(MPI_Datatype type,
                MPI_Aint *extent);
MPI_Type_size(MPI_Datatype type, int *size);
```

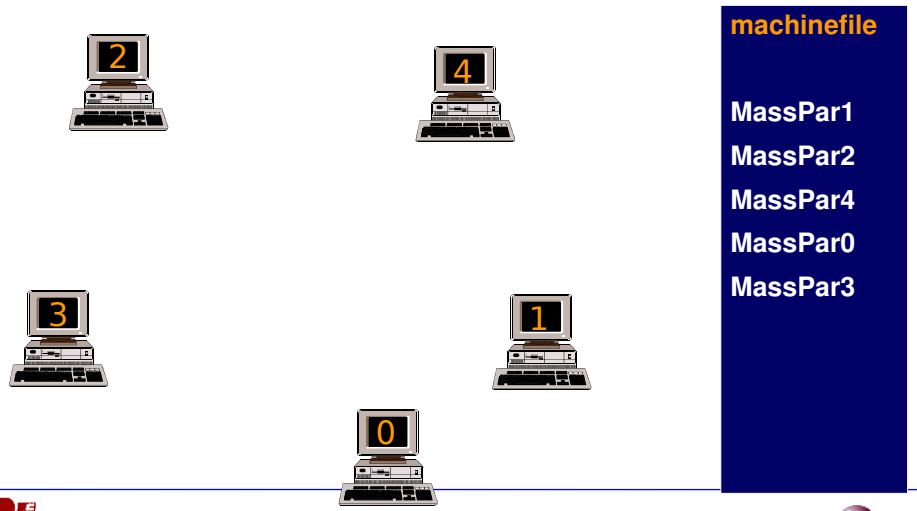
```
int size;
MPI_Datatype FloatArray;
...
MPI_Type_size(FLoatArray, &size);
```


Part 2

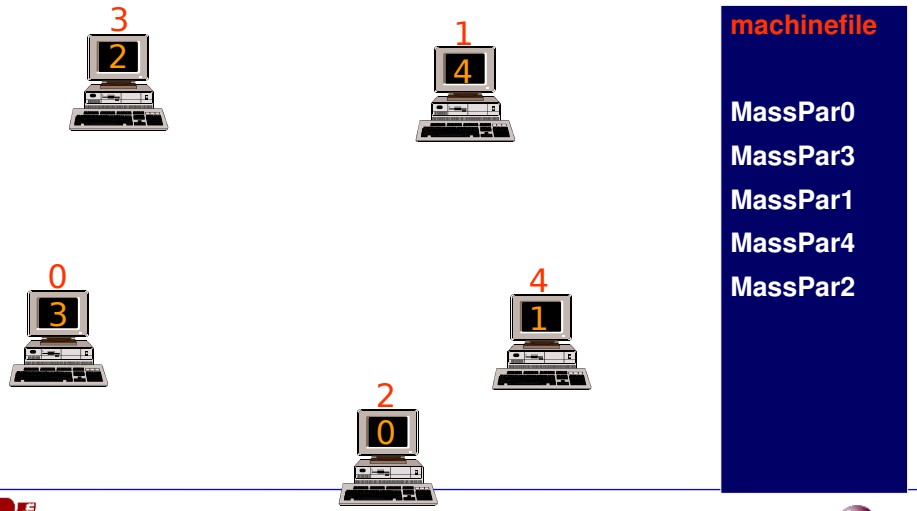
-

Groups and communicators

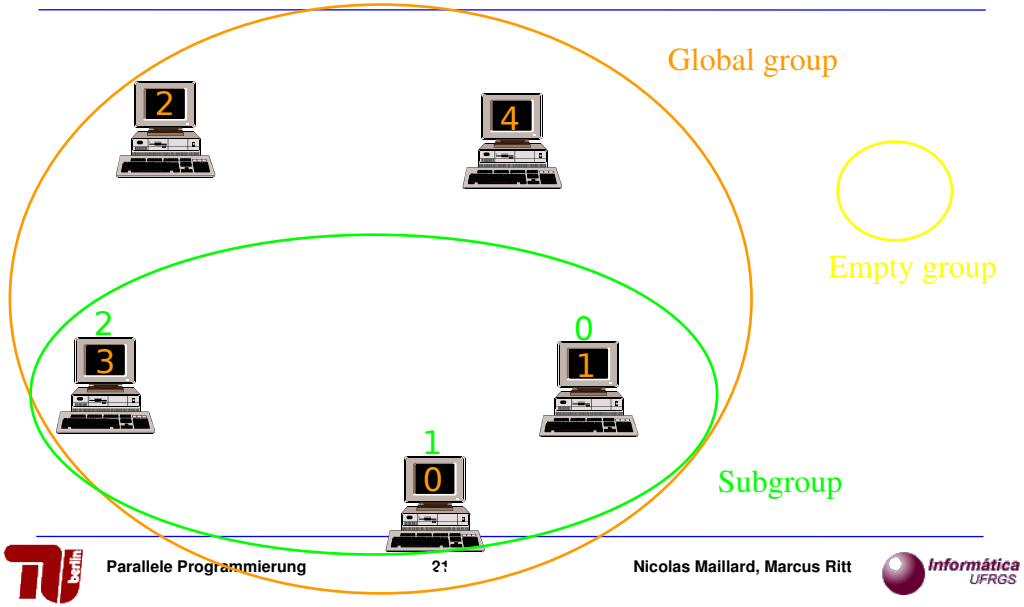
2 - Ranks



2 - Ranks



2 - Groups



2 – Groups and communicators

- A group contains all processes in a specific order
- Each process has an unique number, its *rank,* in this group
- A group is represented by `MPI_Group`
- `MPI_Group_size` determines the size of the group
- The rank can be determined calling `MPI_Group_rank`

2 – Groups and communicators

```
int MPI_Group_size(MPI_Group group, int *size);
int MPI_Group_rank(MPI_Group group, int *rank);


MPI_Group mygroup;
int size, rank;
...
MPI_Group_size(mygroup, &size);
MPI_Group_rank(mygroup, &rank);
```


2 – Groups and communicators

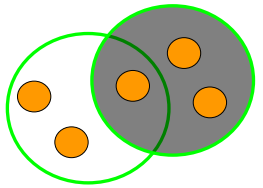
- `MPI_Comm_Group` returns the underlying group of a communicator
- Groups can be manipulated like *sets*
- `MPI_Group_Union`, `MPI_Group_intersection` und `MPI_Group_difference` bfor the union, intersection or difference between two groups
- `MPI_Group_incl` and `MPI_Group_excl` introduce or remove from single processes from a group
- Groups can be freed using `MPI_Group_free`

2 – Groups and communicators

```
MPI_Comm_group (MPI_Comm comm, MPI_Group *group)


MPI_Group_union(MPI_Group group1,
           MPI_Group group2,
           MPI_Group *newgroup);

MPI_Group_incl(MPI_Group group,
          int n, int *ranks,
          MPI_Group *newgroup);
```

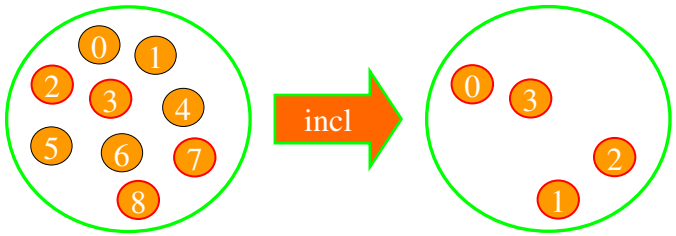

2 – Groups and communicators

```
MPI_Group mygroup, mygroup1, mygroup2;

MPI_Comm_group(comm1, &mygroup1);
MPI_Comm_group(comm2, &mygroup2);

MPI_Group_intersection(mygroup1,mygroup2,&mygroup);

MPI_Group_free(&mygroup);
```

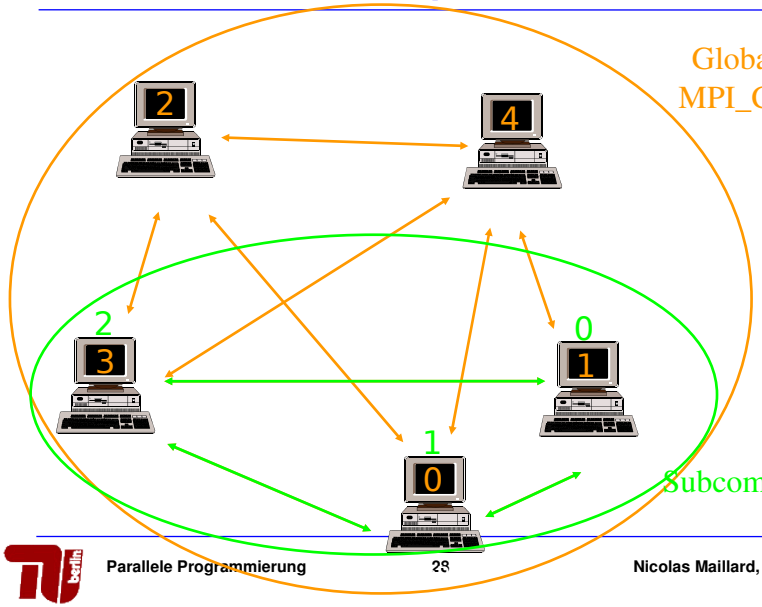


2 – Groups and communicators

```
MPI_Group oldgroup, newgroup;
int ranks[4]= { 2,8,7,3 };

MPI_Group_incl(oldgroup, 4, ranks, &newgroup);
```



Communicators = Groups + Channels



2 – Groups and communicators

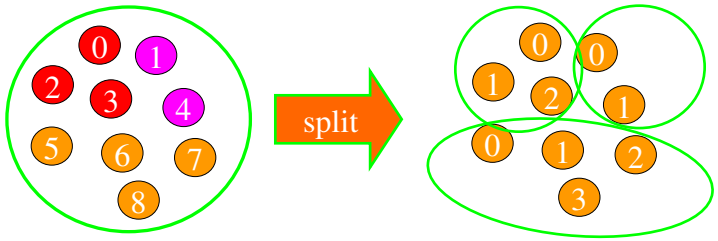
- Communicators build on groups and enable communication between group members
- Each communication call has to specify a communicator
- Besides the group, communicators have other attributes
  - One example is the underlying communication structure
- `MPI_Comm_size` and `MPI_Comm_rank` determine rank and size of the communicator's group
- `MPI_Comm_create` creates a new communicator
- `MPI_Comm_split` splits up a communicator

2 – Groups and communicators

```
MPI_Comm_size(MPI_Comm comm, int *size);
MPI_Comm_rank(MPI_Comm comm, int *rank);
MPI_Comm_create(MPI_Comm comm, MPI_Group group,
  MPI_Comm *newcomm);

int size, rank;
MPI_Comm Comm1, Comm2;
MPI_Group Gruppe;

MPI_Comm_size(Comm1, &size);
MPI_Comm_rank(Comm1, &rank);
MPI_Comm_create(Comm1, Gruppe, &Comm2);
```


2 – Groups and communicators

```
MPI_Comm_split(MPI_Comm comm,
          int color, int key,
          MPI_Comm *newcomm);
```



2 – Groups and communicators

- `MPI_Comm_compare` compares groups
- The result can be

- `MPI_IDENT` if the two communicators are identical
- `MPI_CONGRUENT` if they have the same size and order
- `MPI_SIMILAR` if they are isomorphic (different order)
- `MPI_UNEQUAL` if they are different

Part 3

Process topologies

3 – Process topologies

- A linear arrangement of processor is not always optimal
- Lots of problem require naturally different topologies, for example grids
- MPI supports general topologies (graphs) and in particular simplifies cartesian (grid-like) process topologies

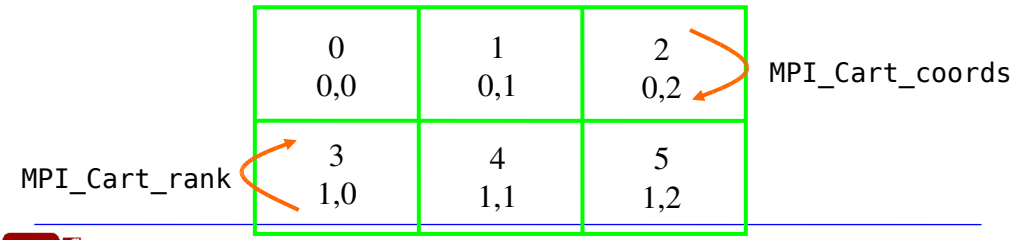
3 – Process topologies

- `MPI_Cart_create` creates a cartesian communicator of arbitrary dimension
- `MPI_Dims_create` simplifies the creation of these topologies
- `MPI_Cart_rank` maps a process' rank to its coordinates
- `MPI_Cart_coords` maps a process' coordinates to its rank

3 – Process topologies

```
MPI_Cart_create(MPI_Comm comm_old,
                int ndims, int *dims, int *periods,
                int reorder, MPI_Comm *comm_cart);
MPI_Dims_create(int nnodes,
                  int ndims, int *dims);
MPI_Cart_rank(MPI_Comm comm,
                int *coords, int *rank);
MPI_Cart_coords(MPI_Comm comm, int rank,
                  int maxdims, int *coords);
```

```
int dims[2], periods[2] = { 0, 0 };
MPI_Comm grid;
MPI_Dims_create(6, 2, dims);
MPI_Cart_create(MPI_COMM_WORLD,
          2, dims, periods, 1, grid);
```

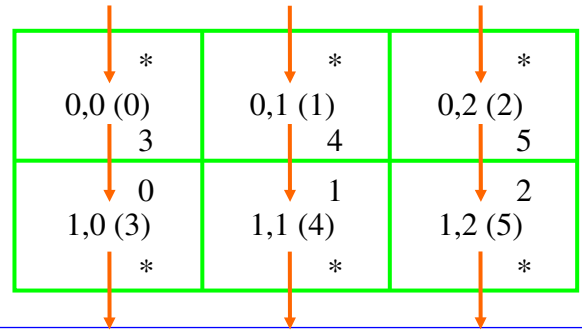


- A very common operation in grids is a shift of data
- `MPI_Cart_shift` generates the parameters for this kind of communication
- Afterwards, a `MPI_Sendrecv` can execute the shift
- For non-periodic grids `MPI_Cart_shift` generates a source or destination `MPI_PROC_NULL`
  - MPI simply discards a communication with `MPI_PROC_NULL`


```
MPI_Cart_shift(MPI_Comm comm,
          int direction,
          int disp,
          int *rank_source,
          int *rank_dest);
```



```
int source, dest;
MPI_Cart_shift(Gitter, 0, 1, &source, &dest);
```



3 – Process topologies

```
int source, dest;
MPI_Cart_shift(Gitter, 1, -1, &source, &dest);
```