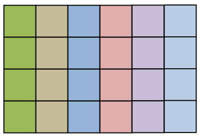
Derived types of matrix

- Column of a matrix (in C): a derived type for data of a same type, evenly spaced

```
MPI_Type_vector( int blocnumber, int bloclength, int stride,
                 MPI_Datatype oldtype, MPI_Datatype *newtype );
```

- Example Number of lines Number of columns

```
MPI_Type_vector( 4, 1, 6, MPI_DOUBLE, &type_column );
MPI_Type_Commit( &type_column )  ;
```



```
MPI_Type_free( &type_column );
```

93

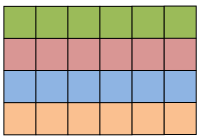
Derived types of matrix

- Line of a matrix (in C): a derived type for contiguous data of a same type

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

- Example Number of columns

```
MPI_Type_contiguous( 6, MPI_DOUBLE, &type_line );
MPI_Type_Commit( &type_line )  ;
```



```
MPI_Type_free( &type_line );
```

94

MPI_Type_indexed

- Derived type for data of a same type, not evenly spaced

```
MPI_Type_indexed( int blocnumber, int *bloclengths,
     int *displacements,
     MPI_Datatype oldtype, MPI_Datatype *newtype);
```

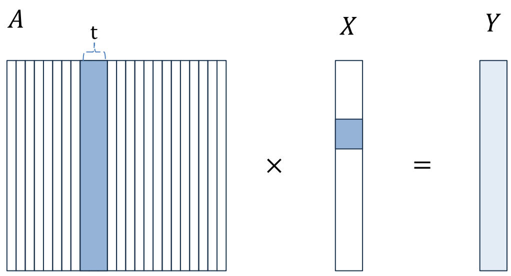
- Example

```
int bloclengths[]={1,2,3};
int displ[]={0,3,9}; /* relative to the beginning */

MPI_Type_indexed( 3, bloclengths, displ,
                     MPI_DOUBLE, &type0_3_9 );
MPI_Type_Commit( &type0_3_9 )  ;
```

```
MPI_Type_free( &type0_3_9 );
```


Derived types – Application

- Matrix–Vector multiplication
  - Parallel algorithm: column-version



$$Y_i = \sum_{j=t}^{t*q+t-1} a_{ij}X_j, i = 0, \dots, n-1$$

partial results of all Y_i are in process q

reduction of all partial results of Y_i in a process: ex. process 0

Derived types – Application

- Matrix–Vector multiplication – column version

```
/* --------------- 2- plusieurs blocs de 1 colonne -------------- */
/* type colonne pour la matrice globale (matrice de TMAT colonnes)  */
MPI_Type_vector(TMAT, 1, TMAT, MPI_DOUBLE, &col);
MPI_Type_commit(&col);
MPI_Type_create_resized(col, (MPI_Aint)0, (MPI_Aint)(1*sizeof(double)), &type_col);
MPI_Type_commit(&type_col);

 /* types colonne pour les processus (matrice de mesNbcols colonnes) */
MPI_Type_vector(TMAT, 1, mesNbcols, MPI_DOUBLE, &colProc);
MPI_Type_commit(&colProc);
MPI_Type_create_resized(colProc, (MPI_Aint)0, (MPI_Aint)(1*sizeof(double)), &type_colProc);
MPI_Type_commit(&type_colProc);

/* Distribution de mesNbcols a chaque processus */
MPI_Scatter(mat, mesNbcols, type_col,
            mesColonnes, mesNbcols, type_colProc, root, MPI_COMM_WORLD);
```

```
MPI_Type_free(&col);
MPI_Type_free(&type_col);
MPI_Type_free(&colProc);
MPI_Type_free(&type_colProc);
```

97

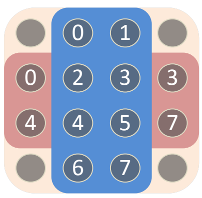
98

MPI Communicators

- What is it?
  - Provide a separate communication space to subset of processes
  - System-defined object
  - Provide safe communications
  - Examples: `MPI_COMM_WORLD`, `MPI_COMM_SELF`
- Types of communicators
  - Intra-communicators
    - A group of processes, each can send message to all other
    - Ease organization of task groups
    - Allow collective communication in a subset of processes
  - Inter-communicator
    - For sending message between processes belong to disjoint intra-communicators

Intra-Communicators

- An intra-communicator is composed of
  - *A group* of `p` processes: identified by a unique rank (`0, …, p-1`)
  - Predefined groups: `MPI_GROUP_EMPTY`
  - *A context*: a system-defined object, each context is exclusive
  - Attributes: *topology*
  - A minimal intra-communicator = a group + a context
- Group / Communicator
  - Groups and communicators are associated.
  - Groups/Communicators are dynamics.
  - A process may be in several groups/communicators. It has a unique rank within each group/communicator.
  - Creation from existing groups/ communicators



Intra-Communicators

❑ Some defined functions

- get the processes group of a given communicator

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

- create un new group from a subset processes from an existing group, processes are reordered

```
int MPI_Group_incl( MPI_Group old_grp, int n,
                    const int ranks[], MPI_Group *new_grp );
```

- create un new communicator from a group of processes, collective operation

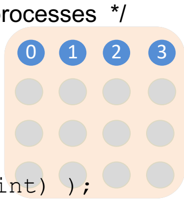
```
int MPI_Comm_create( MPI_Comm old_comm,
                    MPI_Group group, MPI_Comm *newcomm );
```

- `MPI_Comm_free(&comm); MPI_Group_free(&group);`

101

Group / Intra-Communicator - Example

/* Building of a communicator with $row\_size = \sqrt{nbProcs}$ processes */




```
MPI_Group MPI_GROUP_WORLD, row1_grp;
MPI_Comm  row1_comm;
int       row_size, p, *proc_ranks;

proc_ranks =(int *)malloc( row_size*sizeof(int) );
for ( p=0; p<row_size; p++ ) proc_ranks[p]=p;
MPI_Comm_group( MPI_COMM_WORLD, &MPI_GROUP_WORLD );
MPI_Group_incl( MPI_GROUP_WORLD, row_size,
                 proc_ranks, &row1_grp );
MPI_Comm_create(MPI_COMM_WORLD, row1_grp, &row1_comm);
......

MPI_Group_free(&row1_group);
MPI_Comm_free(&row1_comm);
```

102

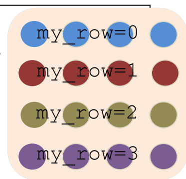
Intra-Communicators

- Other constructors
  - duplicate an existing communicator and it's context: `MPI_Comm_dup`
  - partition of a given communicator into disjoint sub-communicators. A sub-communicator is composed by processes with the same color.

```
int MPI_Comm_split( MPI_Comm old_comm, int color,
                    int key, MPI_Comm *new_comm);
```

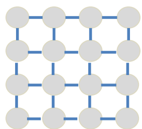
`processes with the color are in the same communicator`

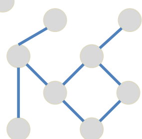
```
MPI_Comm my_row_comm;
int       my_row, row_size; // row_size = √nbProcs

my_row=my_rank/row_size;
MPI_Comm_split(MPI_COMM_WORLD, my_row,
               my_rank, &my_row_comm);
```



MPI Topologies

- Characteristics
  - Topologies define different addressing scheme of processes
  - Fit the communication pattern of parallel application to processes connection
  - Topology is virtual in MPI (machine independent)
  - Can match the physical network for performance (in theory)
- Types of MPI topologies
  - Cartesian topology (grid/torus)
    - For Cartesian communication pattern
  - Graph: general purpose case defined by
    - List of nodes-processes
    - Neighbours number
    - List of edges-connection between processes





MPI Topologies

- Information for grid/torus creation
  - Number of dimensions
  - Order of dimensions
  - Wrap on (*torus*) or no (*grid*)
- Predefined functions
  - Grid constructor

```
int MPI_Cart_create( MPI_Comm old_com, int ndims,
          int *dims_size, int *periods,
          int reorder,   MPI_Comm *cart_comm );
```

  - Transformation rank <-> coordinates

maxdims: length of coords

```
int MPI_Cartdim_get( MPI_Comm comm, int *ndims );
int MPI_Cart_coords( MPI_Comm comm, int rank,
    rank → coords    int maxdims, int *coords );
int MPI_Cart_rank( MPI_Comm, int *coords, int *rank );
    coords → rank
```

105

MPI Topologies

- Predefined functions

```
int MPI_Cart_create( MPI_Comm old_com, int ndims,
          int *dims_size, int *periods,
          int reorder,   MPI_Comm *cart_comm );
```

`ndims=2; dims_size[0]=3; dims_size[1]=2;`

	-1,0 (4)	-1,1 (5)	
0,-1 (-1)	0,0 (0)	0,1 (1)	0,2 (-1)
1,-1 (-1)	1,0 (2)	1,1 (3)	1,2 (-1)
2,-1 (-1)	2,0 (4)	2,1 (5)	2,2 (-1)
	3,0 (0)	3,1 (1)	

`periods[0]=1; periods[1]=0;`

	-1,0 (-1)	-1,1 (-1)	
0,-1 (1)	0, 0 (0)	0, 1 (1)	0, 2 (0)
1,-1 (3)	1, 0 (2)	1, 1 (3)	1, 2 (2)
2,-1 (5)	2, 0 (4)	2, 1 (5)	2, 2 (4)
	3,0 (-1)	3,1 (-1)	

`periods[0]=0; periods[1]=1;`

-1 ↔ MPI_PROC_NULL

106

107

Topology and intra-communicator

- Torus creation
  - Data structure

```
typedef struct {
   MPI_Comm tor_comm; // communicator of torus
   int      nb_procs;  // processes number of tor_comm
   int      my_rank;   // rank of process in tor_comm

   MPI_Comm row_comm;  // communicator of row
   MPI_Comm col_comm;  // communicator of column
   int      order;     // order of grid
   int      coords[2]; // process' coordinates 2D
} torus2d_info_type;
```

107

108

Topology and intra-communicator

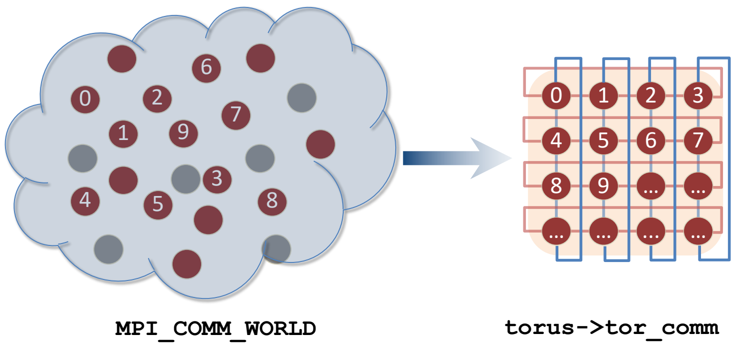
- Torus creation

```
void Setup_torus( torus2d_info_type *torus )
{
   int dims[2], periods[2];
   int coordinates[2], varying_coords[2];

    /* Informations of default communicator */
   MPI_Comm_size( MPI_COMM_WORLD, &(torus->nb_procs) );

    /* Creation of torus communicator */
   torus->order = (int) sqrt( (double)torus->nb_procs );
   dims[0] = dims[1] = torus->order ;
   periods[0] = periods[1] = 1;
   MPI_Cart_create( MPI_Comm_WORLD, 2, dims,
                    periods, 1, &(torus->tor_comm) );
```

108

Topology and intra-communicator

- Torus creation – torus 2D communicator



109

Topology and intra-communicator

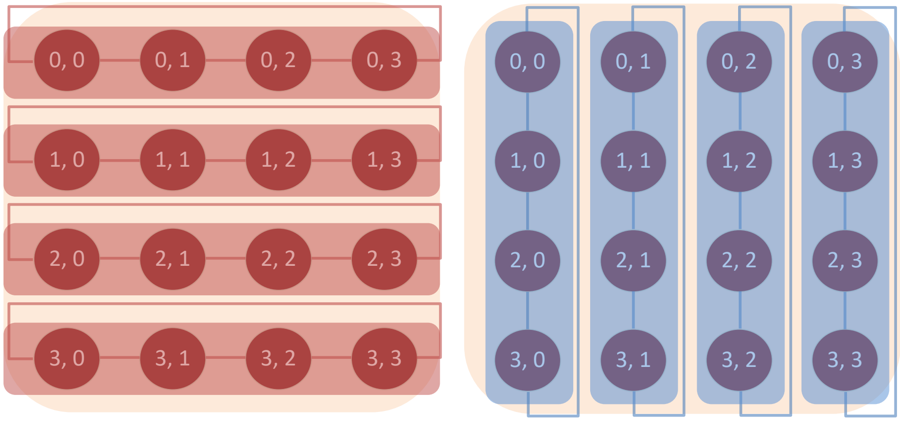
- Torus creation

```
    if (torus->tor_comm != MPI_COMM_NULL) {
        /* Informations in the new communicator */
        MPI_Comm_rank(torus->tor_comm, &(torus->my_rank) );
        MPI_Cart_coords(torus->tor_comm, torus->my_rank, 2,
                     torus->coords );

        /* row communicators creation */
        varying_coords[0] = 0; varying_coords[1] = 1;
        MPI_Cart_sub( torus->tor_com, varying_coords,
                     &(torus->row_comm) );

        /* column communicators creation */
        varying_coords[0] = 1; varying_coords[1] = 0;
        MPI_Cart_sub( torus->tor_com, varying_coords,
                     &(torus->col_comm) );
    }
}
```

110

Topology and intra-communicator

- Torus creation – row/column communicators



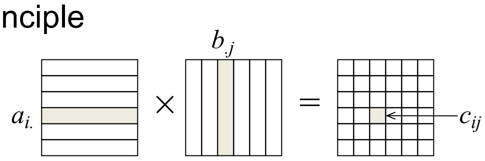
row communicators **column communicators**

111

Matrix-Matrix Multiplication

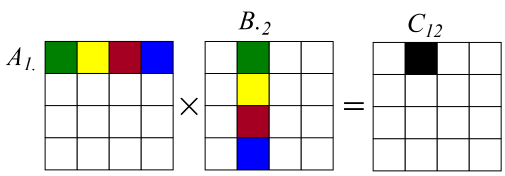
- Problem
  - Compute $C=A\times B$, A, B: matrix nxn on a torus of qxq process
- Parallel algorithm: principle

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$



Process *(i,j)* calculates:

$$C_{ij} = \sum_{k=0}^{q-1} A_{ik} B_{kj}$$



112

Matrix-Matrix Multiplication – Algorithm of Fox

❑ Assumptions

- `A`, `B`: `nxn` matrices
- `N`: the number of processes and `N=q`^{2}, `n'=n/q`
- Input data of Process `(i,j)`:

$$A_{ij} = \begin{pmatrix} a_{i*n',j*n'} & \dots & a_{i*n',(j+1)*n'-1} \\ \vdots & \dots & \vdots \\ a_{(i+1)*n'-1,j*n'} & \dots & a_{(i+1)*n'-1,(j+1)*n'-1} \end{pmatrix}$$

$$B_{ij} = \begin{pmatrix} b_{i*n',j*n'} & \dots & b_{i*n',(j+1)*n'-1} \\ \vdots & \dots & \vdots \\ b_{(i+1)*n'-1,j*n'} & \dots & b_{(i+1)*n'-1,(j+1)*n'-1} \end{pmatrix}$$

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

Data distribution with `q=4`

Principle: A_{ij} remain in `Process(i,j)`,
B_{ij} move in the column communicateur

113

Matrix-Matrix Multiplication – Algorithm of Fox

❑ Parallel algorithm

/* Process `(i,j)` computes $C_{ij} = \sum_{k=0}^{q-1} A_{ik}B_{kj}$ */
for (k=0; k<q; k++) {
1. Select a block of `A` for each row of the grid
2. Broadcast of the chosen block in each line for each process
3. Multiply the block of `A` with the block of `B` in each process
4. Send the block of `B` to his upper row neighbor
5. Receive a block of `B` from his lower row neighbor

} /* block of `A` to broadcast at each step:
A_{iu} avec `u = (i+k) mod q` for the row `i` */
/* example: bleu, rouge, vert, orange if `q=4` */

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

114

Matrix-Matrix Multiplication – Algorithm of Fox

```
void Fox( grid2d_info_t *grid,
             LOCAL_MATRIX_TYPE *lA,
             LOCAL_MATRIX_TYPE *lB,
             LOCAL_MATRIX_TYPE *lC,  int ln )
{
    LOCAL_MATRIX_TYPE *temp_A;
    int         step, bcast_root;
    int         source, dest, tag = 43;
    MPI_Status status;

    Set_to_zero( lC, ln );

    source = (grid->my_row+1)%grid->order;
    dest = (grid->my_row-1+grid->order)%grid->order;
    temp_A = Local_matrix_allocate(ln);
```

local matrices

order of local matrices

115

Matrix-Matrix Multiplication – Algorithm of Fox

```
    for ( step=0; step<grid->order; step++ ) {
        bcast_root = (grid->my_row+step)%grid->order;
        if ( bcast_root == grid->my_col ) {
            MPI_Bcast(lA, 1, DERIVED_LOCAL_MATRIX,
                          bcast_root, grid->row_comm );
            Local_matrix_multiply( lA, lB, lC, ln );
        }
        else {
            MPI_Bcast(temp_A, 1, DERIVED_LOCAL_MATRIX,
                     bcast_root, grid->row_comm );
            Local_matrix_multiply( temp_A, lB, lC, ln );
        }
        MPI_Send( lB, 1, DERIVED_LOCAL_MATRIX,
                 dest, tag, grid->col_comm );
        MPI_Recv( lB, 1, DERIVED_LOCAL_MATRIX,
                  source, tag, grid->col_comm, &status );
    }
}
```

Safe program?

116

Distribution of blocks of matrix

```
void DataDistRij(grid2d_info_t *grid, int ri, int rj,
     double *A, int n, double *lA, int ln )
{
    double *tmpA = NULL;
```

square matrix
square 2d torus topology
order of matrix divisible by torus dimension

```
    /* Distribution in line comm. of root (ri, rj) */
    if ( grid->my_row == ri ) {
       tmpA = (double *) malloc( ln*n*sizeof(double) );
       MPI_Scatter( A, ln, type_col, tmpA, ln,
             type_colProc, rj, grid->row_comm );
    }
    /* Distribution in column communicators */
    MPI_Scatter( tmpA, ln*ln, MPI_DOUBLE, lA,
             ln*ln, MPI_DOUBLE, ri, grid->col_comm );

    if ( grid->my_row == ri )
        if ( tmpA ) free( tmpA );
}
```

117

Gathering of blocks of matrix

```
void DataFusionRij( grid2d_info_t *grid, int ri, int rj,
                    double *lA, int ln, double *A, int n )
{
   double *tmpA = NULL;

    if ( grid->my_row == ri )
        tmpA = (double *)malloc( ln*n*sizeof(double) );
   /* Gather in column communicators */
    MPI_Gather( lA, ln*ln, MPI_DOUBLE,
       tmpA, ln*ln, MPI_DOUBLE, ri, grid->col_comm );
   /* Gather in row communicator of root */
   if ( grid->my_row == ri ) {
      MPI_Gather( tmpA, ln, type_colProc,
         A, ln, type_col, rj, grid->row_comm );
     if ( tmpA ) free( tmpA );
   }
}
```

118

Matrix-Matrix Multiplication – Algorithm 2

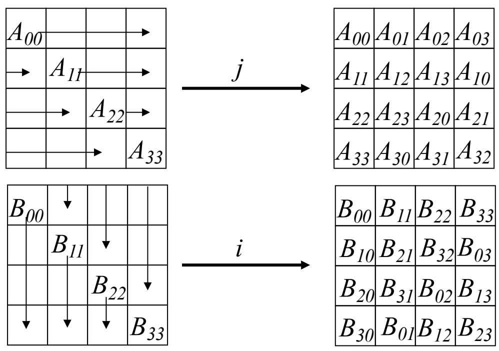
❑Parallel algorithm 1: each process computes one bloc of *C* matrix

- **First stage**
  1. Cut the matrix *A*, *B*, *C* into blocs
  2. Distribute the blocs A_{ij} and B_{ij} without replication
  3. Compute a $A_{ik}*B_{kj}$
- **Second stage**
  1. Circulating of the blocs A_{ij} and B_{ij} (For example, send the A_{ij} to the left, and the B_{ij} to the upper)
  2. Compute a $A_{ik}*B_{kj}$
  3. Repeat q-1 times ($N=q*q$)

Matrix-Matrix Multiplication – Algorithm 2

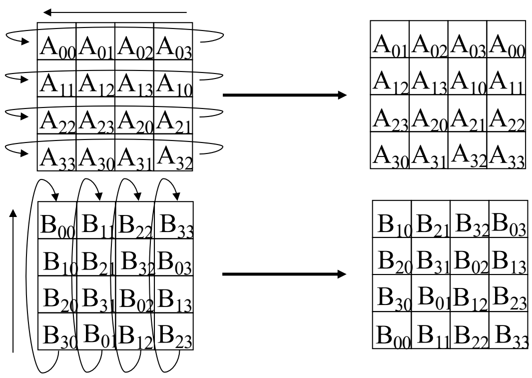
❑Parallel algorithm 1: example

- Initial distribution of A_{ij} and B_{ij}



Matrix-Matrix Multiplication – Algorithm 2

- Parallel algorithm 1: example
  - Circulating of the blocs A_{ij} and B_{ij}: p-1 times



121

Matrix-Matrix Multiplication – Algorithm 2

- Performance evaluation
  - Computation time:
  - Communication time:

122

123

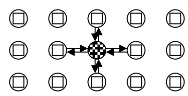
123

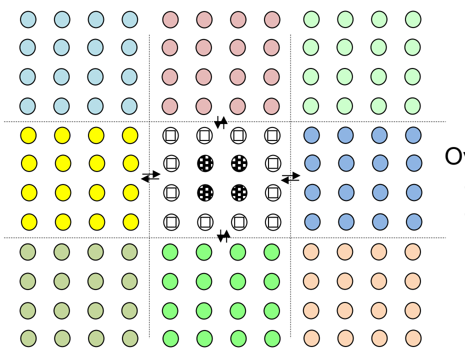
124

Domain decomposition – example

- 2D Poisson problem: Jacobi's algorithm

$$u_{i,j}^{(t+1)} = \frac{u_{i-1,j}^{(t)} + u_{i+1,j}^{(t)} + u_{i,j-1}^{(t)} + u_{i,j+1}^{(t)} - h^2 f_{i,j}}{4}$$





Overlapping:
- Communication of border cells
- Computation of central cells

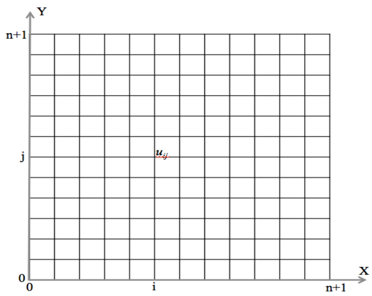
124

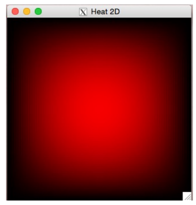
Domain decomposition – example

- Heat equation and Jacobi's algorithm

$$u_{ij}^{k+1} = u_{ij}^{k} + c_x\left(u_{i+1,j}^{k} - 2u_{i,j}^{k} + u_{i-1,j}^{k}\right) + c_y\left(u_{i,j+1}^{k} - 2u_{i,j}^{k} + u_{i,j-1}^{k}\right)$$

0	0.1	0
0.1	0.6	0.1
0	0.1	0

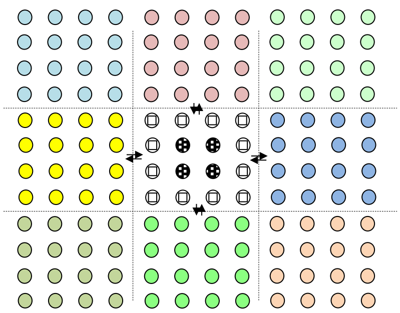




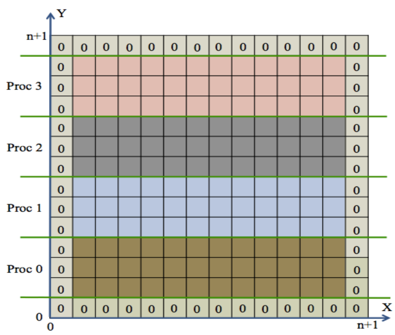
125

Domain decomposition – example

- Heat equation – Jacobi's algorithm
  - Parallelization - Performance ?



Domain decomposition – Grid



Domain decomposition – Strip of lines

126

127

Summary

- Processes group and communicator
  - A process is identified by a rank in a group
  - A communicator is composed by a group of processes and a context. It may have some attributes (ex. topology).
- Communication types
  - One-to-one
  - Collective communication
  - Blocking / non-blocking communication
- Advanced data types
  - Pack/Unpack
  - Derived data type
- Performance of parallel program
  - Collective communication
  - Non-blocking communication -> Overlapping communication-computation

128

Hybrid programming

- MPI – OpenMP

```
#include <stdio.h>
#include "mpi.h"
#include "omp.h"

int main(int argc, char *argv[])
{
    int procRank, nbProcs;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &procRank);
    MPI_Comm_size(MPI_COMM_WORLD, &nbProcs);

    // #pragma omp parallel num_threads(4)
    #pragma omp parallel
    {
        printf("Hello from P #%d of %d, T #%d of %d\n",
            procRank, nbProcs, omp_get_thread_num(), omp_get_num_threads());
    }

    MPI_Finalize();

    return 0;
}
```

```
$ mpicc -fopenmp …
$ OMP_NUM_THREADS=2 mpirun -np 3 ./mpi_openmp_prog
```