# MPI
# (Message Passing Interface)

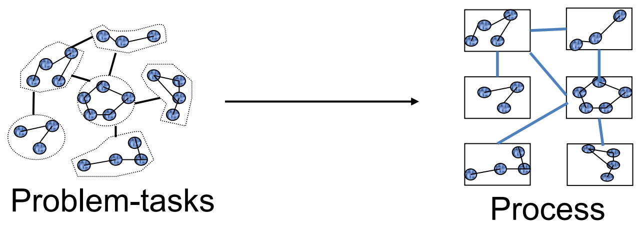
Part I

Jian-Jin LI

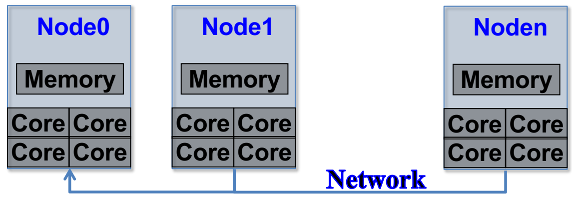


## MPI Programming Model

- Group(s) of process to solve together a problem

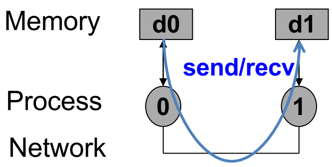


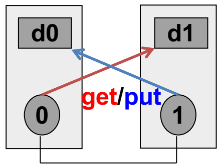

- Distributed memory API



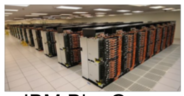
# MPI Programming Model

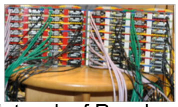
- Communication model
  - Message passing (two side operation)
  - Remote memory access (one side operation, MPI-2)




- Suitable architectures
  - Distributed memory MIMD
  - Cluster
  - Network of workstations

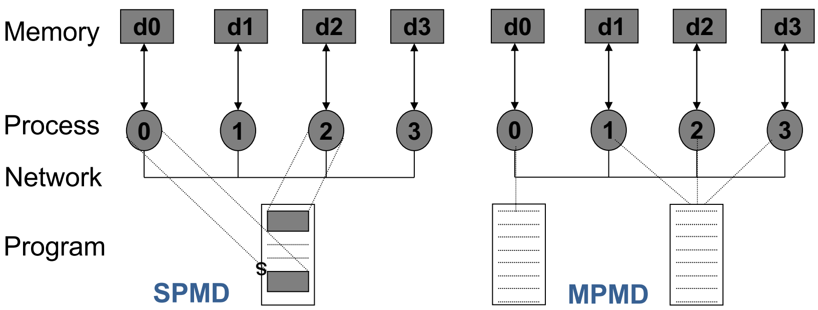

IBM BlueGene


Network of Raspberry



# MPI Programming Model

- SPMD / MPMD
  - Single / Multiple Program Multiple Data
  - Data distributed over process
  - Communication between process via message passing



# MPI Characteristics

- Message passing library, include
  - Environment management routines
  - Point to point communication
  - Datatypes management
  - Collective communications
  - Groups of process and communicators
  - Process topologies
  - Parallel I/O, RMA, dynamic process (MPI-2)
  - Non-blocking collective communication, RMA improvement, parallel programming environment (MPI-3)

to use with C, C++, Fortran

- Participants
  - Vendors: IBM, Intel, Meiko, Cray, ...
  - Libraries: PVM, Zipcode, Express, Linda, …
  - Universities: San Francisco, Santa Barbara, ...



# References


- Wiliam Gropp, Ewing Lusk and Anthony Skjellum, Using MPI, 2 volume set, The MIT Press, 01/2000, ISBN : 0-262-57134-X .
- Peter Pachero, Parallel Programming with MPI, Morgan Kaufmann, 01/1997, ISBN : 1-55860-339-5
- http://www-unix.mcs.anl.gov/mpi
- http://www.mpi-forum.org/docs/docs.html
- http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html

# MPI – Free Implementations

- MPICH
  - Leader: Argonne National Laboratory
  - http://www.mpich.org
- Open MPI
  - ⇧ LAM (Ohio Supercomputer Center)
  - http://www.open-mpi.org

Ohio Supercomputer Center
An OH·TECH Consortium Member



# Environment Management Routines

- hello.c

```
#include "mpi.h"   MPI's header file
#include <stdio.h>

int main(int argc, char **argv)
{
    int myrank, nbprocs;

    MPI_Init( &argc, &argv );   Execution environment initialization
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    MPI_Comm_size( MPI_COMM_WORLD, &nbprocs);

    printf( " Hello from proc. %d of %d\n ",
            myrank, nbprocs);

    MPI_Finalize();   End of MPI execution
    return 0;
}
```

# Compiling and Running MPI applications

- Application implementation
  - using C, C++ or Fortran and MPI library for process and communication management
- Compiling
  - `mpicc hello.c –o hello`
- Execution
  - 8 process on local node
    `mpiexec –np 8 ./hello`
  - 16 process on nodes h1-4, 1 mpd run on each node
    `mpiexec –hosts h1,h2,h3,h4 –np 4 ./hello`



# MPI's world

- Group of process and Communicator
  - MPI process are enrolled into groups
  - Group + context = Communicator
  - Default communicator: `MPI_COMM_WORLD`
  - Process identification (`rank`): `0, 1, …, size-1`
  - `MPI_Comm_rank( MPI_COMM_WORLD, &rank );`
  - `MPI_Comm_size( MPI_COMM_WORLD, &size );`

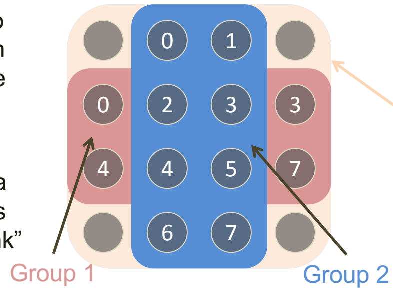

`MPI_COMM_WORLD`

# Communicators

```
mpiexec -n 16 ./test
```

Communicators do not need to contain all processes in the system

Every process in a communicator has an ID called as “rank”




When you start an MPI program, there is one predefined communicator `MPI_COMM_WORLD`

Can make copies of this communicator (same group of processes, but different “aliases”)

The same process might have different ranks in different communicators



# Environment management routines

- MPI_Abort
  - `int MPI_Abort(MPI_Comm comm, int errorcode);`
  - terminates all process of communicator if exception: ex. malloc
- MPI_Get_processor_name
  - `int MPI_Get_processor_name(char *name, int resultlength);`
  - return the processor name and its length
  - name buffer size: `MPI_MAX_PROCESSOR_NAME`
- MPI_Wtime / MPI_Wtick
  - `double MPI_Wtime(void); double MPI_Wtick();`
  - return an ellapsed wall clock time in seconds / the number of seconds between successive clock ticks.

# Point to Point Communication

- p2p.c

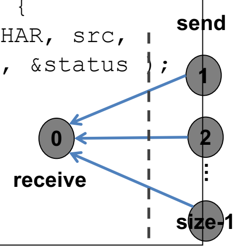
```
#include "mpi.h "
#include <stdio.h>

int main(int argc, char **argv)
{
   int        myrank,nbprocs,src,tag=50,nameLength;
   MPI_Status status;
   char       message[100],
              procName[MPI_MAX_PROCESSOR_NAME];

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
   MPI_Comm_size( MPI_COMM_WORLD, &nbprocs);
   MPI_Get_processor_name(procName, &nameLength);
```



# Point to Point Communication

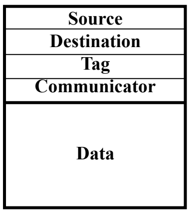
- p2p.c (continue)

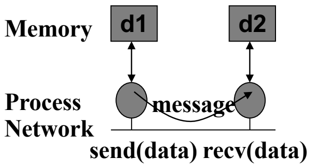
```
   if ( myrank != 0 ) {
      sprintf( message, "Hello from %d on %s!" ,
               myrank, procName );
      MPI_Send( message, strlen(message)+1, MPI_CHAR,
                0, tag, MPI_COMM_WORLD ); }
   else
      for ( src=1; src<nbprocs; src++ ) {
          MPI_Recv( message, 100, MPI_CHAR, src,
                    tag, MPI_COMM_WORLD, &status );
          printf( "%s\n", message );
      }
   MPI_Finalize();
   return 0;
}
```



# Point to Point Communication

- Communication between two process
  - Source and Destination
- `Message = header + data`
- Data conversion if necessary
- Transmission mechanism






# Point to Point Communication

- Blocking communication
  - Blocking send and receive: `MPI_Send, MPI_Recv`
- Parameters of `MPI_Send` and `MPI_Recv`
  - Data address
  - Elements number of data
  - Type of data elements: `MPI_Datatype`
  - Source or Destination of the message (`MPI_ANY_SOURCE`)
  - Tag of message (`MPI_ANY_TAG`), may be used to indicate different type of message
  - Communicator (`MPI_COMM_WORLD`): `MPI_Comm`
  - Status: `MPI_Status` (`MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`)

# MPI Datatypes

- Similar to C, examples:

| MPI datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_INT | signed int |
| MPI_UNSIGNED | unsigned int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_BYTE | |

- Complex datatypes
  - MPI_PACKED
  - Derived types: structure, colonne of matrix …



# Point to Point Communication

- Two side operation
  - a *Send* must be matched by a *Recv*
- Safe program in blocking communication

```
MPI_Comm_rank (comm, rank);
if (rank == 0) {
   MPI_Send( sendbuf, count, MPI_FLOAT, 1, tag, comm );
   MPI_Recv( recvbuf, count, MPI_FLOAT, 1,
             tag, comm,  &status );
}
else if (rank == 1) {
   MPI_Recv( recvbuf, count, MPI_FLOAT, 0,
             tag, comm, &status );                Dead lock
   MPI_Send( sendbuf, count, MPI_FLOAT, 0, tag, comm );
}
```

# Blocking Communication

- Features
  - Completion of `MPI_Send` means send variable can be reused
  - Completion of `MPI_Recv` mean receive variable can be read
  - Cause synchronization -> Increase communication time
  - Affect the performance of parallel program
- Solution
  - Non-blocking communication



# Non-blocking Communication

- Operation in 2 steps
  - Request: `MPI_Isend`, `MPI_Irecv`

```
MPI_Isend(&buf, count, datatype, dest,
                tag, comm, &request);
MPI_Irecv(&buf, count, datatype, src,
                tag, comm, &request);
```

  - Completion: MPI_WAIT(&request, &status);
  - Test of completion:

```
MPI_TEST(&request, &flag, &status);
```

  - Avoid dead lock (ex. T.16 with non-blocking send/recv)
  - Allow communication / computation overlapping
  - Persistent request can be used if many communication

# Non-blocking Communication

- Example

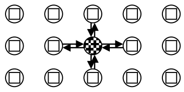
```
MPI_Comm_rank (comm, rank);
if (rank == 0) {
   MPI_Isend( sendbuf, count, MPI_REAL, 1,
                 tag, comm, &request0 );
   MPI_Recv( recvbuf, count, MPI_REAL, 1,
               tag, comm,  &status );
    /* or MPI_Irecv + MPI_Wait */
}
else if (rank == 1) {
    MPI_Isend( sendbuf, count, MPI_REAL, 0,
                 tag, comm, &request1 );
    MPI_Recv( recvbuf, count, MPI_REAL, 0,
               tag, comm, &status );
    /* or MPI_Irecv + MPI_Wait */
}
```

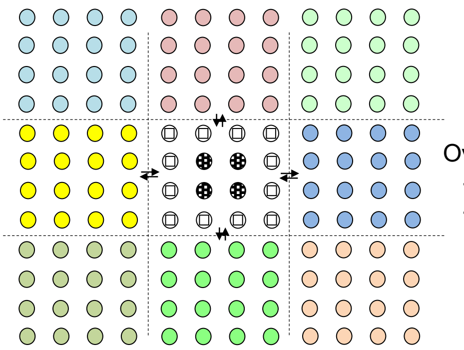


# Overlapping Communication/Computation

- 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

# Message test routines

- MPI_Probe/ MPI_Iprobe
  - Availability test of message
  - Where is it from ?
  - What is its length ?

```
MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG,
            MPI_COMM_WORLD, &status);
if ( status.MPI_SOURCE == 0 )
   MPI_Recv( buf, 1, MPI_INT, 0, 0, &status );
else {
   MPI_Get_count(&status, MPI_BYTE, &incoming_msg_size);
   buf2=(float *)malloc(incoming_msg_size*sizeof(float));
   MPI_Recv( buf2, incoming_msg_size, MPI_FLOAT,
             status.MPI_SOURCE, 0, &status );
}
```



# Design of Parallel Program

- Those who have to think
  - Launch of program with various number of process
  - Load balancing
  - Change of data over the time
  - Prefer local communication than distant one
  - Performance of parallel program, yet ?
- Another solutions
  - Collective communication
  - Decrease the communication number
    - Data grouping for communication

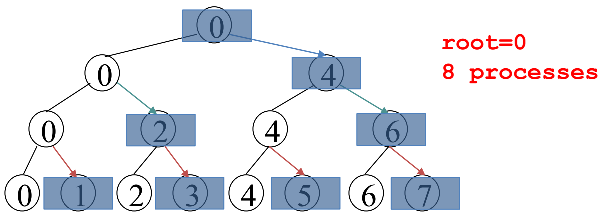
# Collective Communication

- What is it?
  - Communication involving all processes of a group
- Objective
  - Increase the performance of parallel program
- How?
  - By reduce of idle processes ⟹ decrease the communication time
- Use cases
  - When I/O
  - Parallel algorithms need collective communication



# Collective Communication

- Broadcast
  - A process (`root`) has a message to send to others
  - Possible implementation:




  - MPI routine:

```
int MPI_Bcast(void *msg, int count,
     MPI_Datatype datatype, int root, MIP_Comm comm);
```

# Collective Communication

- Example - Broadcast of the dimension of a image

**Point to point communication:**
Steps - `O(size)`

```
int myrank, size, dims[2];
int i, tag=30;

MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
MPI_Comm_size( MPI_COMM_WORLD, &size );

if (myrank == 0){
   /* Fill dims */
   for ( i=1; i<size; i++)
       MPI_Send( dims, 2, MPI_INT, i, tag, MPI_COMM_WORLD);
}
else
   MPI_Recv(dims, 2, MPI_INT, 0, tag, MPI_COMM_WORLD);
```

```
MPI_Bcast(dims, 2, MPI_INT, 0, MPI_COMM_WORLD);
```

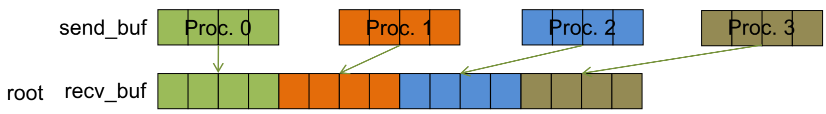
**Collective communication:**
Steps – $O(\log_2(size))$ with binary tree



# Collective Communication

- Gather / Reduce
  - The processes of a group have data to merge together
  - MPI routines:

```
int MPI_Gather( void *sendbuf, int sendcnt,
       MPI_Datatype sendtype, void *recvbuf,
       int recvcnt, MPI_Datatype recvtype,
       int root, MPI_Comm comm );
```



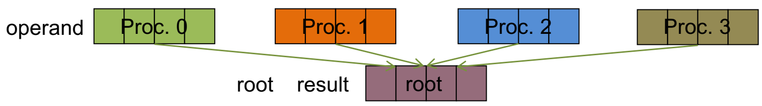

- The length of `recvbuf` must be the one of `sendbuf * number_of_processes`, data in `recvbuf` are stored in the order of processes

# Collective Communication

- Reduce

```
int MPI_Reduce( void *operand, void *result,
      int count, MPI_Datatype datatype, MIP_Op
op,
      int root, MPI_Comm comm );
```

| Operation | Meaning | Operation | Meaning |
|---|---|---|---|
| MPI_MAX | Maximum | MPI_LOR | logical OR |
| MPI_MIN | Minimum | MPI_BOR | bitwise OR |
| MPI_SUM | Sum | MPI_LXOR | XOR |
| MPI_PROD | Product | MPI_BXOR | |
| MPI_LAND | Logical AND | MPI_MAXLOC | max or min + |
| MPI_BAND | bitwise AND | MPI_MINLOC | index |




✧`operand` and `result` have the same length



# Collective Communication

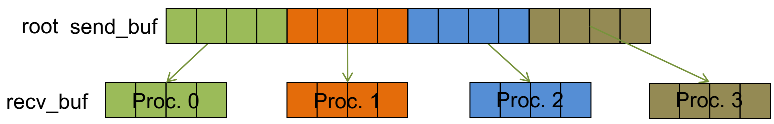
- Example

```
int myrank, size;
double myresult, globalresult, *resultvec;

MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
MPI_Comm_size( MPI_COMM_WORLD, &size );

/* computation of myresult */

resultvect = (double *)malloc(size*sizeof(double));
MPI_Gather( &myresult, 1, MPI_DOUBLE, resultvec, 1,
            MPI_DOUBLE, 0, MPI_COMM_WORLD );

MPI_Reduce( &myresult, &globalresult, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD );
```

# Data distribution

- MPI_Scatter
  - The root has data to distribute to the processes of the group

```
int MPI_Scatter( void *sendbuf, int sendcnt,
      MPI_Datatype sendtype, void *recvbuf,
      int recvcnt, MPI_Datatype recvtype,
      int root, MPI_Comm comm );
```




- The length of `sendbuf` must be the one of `recvbuf * number_of_processes.`

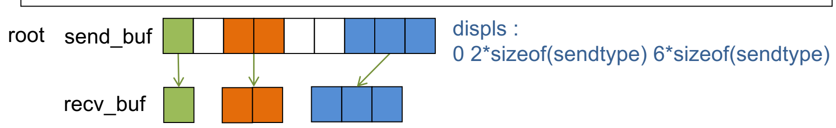


# Data distribution

- MPI_Gatherv / MPI_Scatterv
  - More flexibility in message length of processes and message position in root memory

```
int MPI_Gatherv( void *sendbuf, int sendcnt,
     MPI_Datatype sendtype, void *recvbuf,
     int *recvcnts, int *displs, MPI_Datatype recvtype,
     int root, MPI_Comm comm );
```

```
                                          1 2 3
int MPI_Scatterv( void *sendbuf, int *sendcnts,
     int *displs, MPI_Datatype sendtype,
                               1/2/3
     void *recvbuf, int recvcnt, MPI_Datatype recvtype,
     int root, MPI_Comm comm );
```



# Collective Communication

- Barrier synchronization
  - Make a appointment for all processes

```
int MPI_Barrier( MIP_Comm comm );
```

  - Use case: time measurement
    - Time measurement for each process

```
double start, proc_exectime;
MPI_Barrier(comm);
start = MPI_Wtime();
……
proc_exectime= MPI_Wtime()- start;
```

In main function

    - Execution time of program: the one of the slowest process



# Data grouping

- Need to send heterogeneous data
- Methods
  - `MPI_Pack` / `MPI_Unpack`
    - Use of a buffer of bytes to send arbitrary data
  - Derived types
    - Corresponding MPI datatype for structure
- Choice of methods
  - `MPI_Pack` / `MPI_Unpack`
    - Heterogeneous data to send a few times
    - Communication of data of variable length
  - Derived types
    - Heterogeneous data to send repeatedly
    - Non-contiguous data of the same type ⇒ `MPI_Type_vector, MPI_Type_indexed`

# Pack / Unpack

- Objective
  - Put the heterogeneous non-contiguous data together to be sent at one time
- Example

```
int getdata ( int rank, int root, float *a, int *n)
{
   char buffer[100];
   int    position;

   if ( rank == root ) {
      position = 0;
      MPI_Pack( a, 1, MPI_FLOAT, buffer, 100,
                &position, MPI_COMM_WORLD );
      MPI_Pack( n, 1, MPI_INT, buffer, 100,
                &position, MPI_COMM_WORLD );
   }
```



# Pack / Unpack

- Example (continue)

```
   MPI_Bcast( buffer, 100, MPI_PACKED,
              root, MPI_COMM_WORLD );

   if (rank != root) {
      position = 0;
      MPI_Unpack( a, 1, MPI_FLOAT, buffer, 100,
                  &position, MPI_COMM_WORLD );
      MPI_Unpack( n, 1, MPI_INT, buffer, 100,
                  &position, MPI_COMM_WORLD );
   }
}
```

# Derived type

- Objective
  - Define heterogeneous data access
- General method
  - Build
  - Validate
  - Destroy

```
typedef struct {
    float a, b;
    int   n;
} indata_type;

void Build_derived_type(indata_type *indata,
        MPI_Datatype *msg_type_ptr)
{
    int             blok_lengths[3];
    MPI_Aint        displacements[3], addresses[4];
    MPI_Datatype typelist[3];
```



# Derived type

- General method (continue)

```
    typelist[0] = typelist[1] = MPI_FLOAT; typelist[2] = MPI_INT;
    block_lengths[0]= block_lengths[1]= block_lengths[2]=1;
    MPI_Address(indata, &addresses[0]);
    MPI_Address(&(indata->a), &addresses[1]);
    MPI_Address(&(indata->b), &addresses[2]);
    MPI_Address(&(indata->n), &addresses[3]);
    displacements[0] = addresses[1] - addresses[0];
    displacements[1] = addresses[2] - addresses[0];
    displacements[2] = addresses[3] - addresses[0];

    MPI_Type_struct( 3, block_lengths, displacements,
                     typelist, msg_type_ptr );
    MPI_Type_commit( msg_type_ptr );
}                   MPI_Type_free( msg_type_ptr );
```

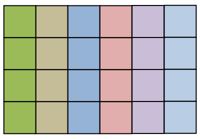
# Derived types of matrix

- 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 )  ;
```



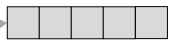
```
MPI_Type_free( &type_column );
```



# Distribution of square matrix

- MPI_Scatter – 1 column per process
  - Condition: processes number = order of matrix

```
/* Definition du type colonne */
MPI_Type_vector(order_mat, 1, order_mat, 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);

lA = (double *) malloc( 1 * order_mat * sizeof(double) );
if ( lA==NULL ) MPI_Abort(MPI_COMM_WORLD, 1);

/* Distribution de colonnes */
MPI_Scatter( A, 1, type_col,
    lA, order_mat, MPI_DOUBLE, 0, MPI_COMM_WORLD );
```



# Distribution of square matrix

- MPI_Scatterv – 1 column per process

Fait pas !

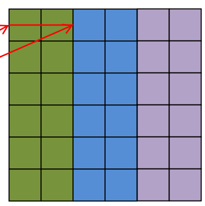
```
/* Definition du type colonne */
MPI_Type_vector(order_mat, 1, order_mat, MPI_DOUBLE, &col);
MPI_Type_commit(&col);
MPI_Type_create_resized(col, (MPI_Aint)0,
    (MPI_Aint)(1*sizeof(double)), &type_col);
MPI_Type_commit(&type_col);

displ = (int *)malloc( order_mat * sizeof(int) );
sendcnts = (int *)malloc( order_mat * sizeof(int) );
for (i=0; i<order_mat; i++) {
    sendcnts[i] = 1;   // nb. d'elts pour chaque processus
    displ[i] = i;  // valeur liee au 3eme arg. de *_resized
}

lA = (double *) malloc( 1 * order_mat * sizeof(double) );
MPI_Scatterv( A, sendcnts, displ, type_col,
    lA, order_mat, MPI_DOUBLE, 0, MPI_COMM_WORLD );
```



# Distribution of square matrix

- MPI_Scatter – 1 bloc of columns of matrix
  - Example: 2 columns per process

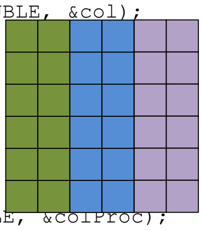
```
/* Definition du type bloc de colonnes */
nb_cols = order_mat / size;
MPI_Type_vector(order_mat, nb_cols, order_mat,
                MPI_DOUBLE, &cols);
MPI_Type_commit(&cols);
MPI_Type_create_resized(cols, (MPI_Aint)0,
    (MPI_Aint)(nb_cols*sizeof(double)),
    &type_cols);
MPI_Type_commit(&type_cols);

/* Distribution d'1 bloc de colonnes par processus */
lA = (double *) malloc( nb_cols * order_mat * sizeof(double) );
MPI_Scatter( A, 1, type_cols, lA, nb_cols * order_mat,
             MPI_DOUBLE, 0, MPI_COMM_WORLD );
```

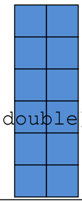


# Distribution of square matrix

- MPI_Scatter – several columns per process

```
/* Definition du type colonne (matrice globale)*/
MPI_Type_vector(order_mat, 1, order_mat, 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);

/* Definition du type colonnes des processus */
nb_cols = order_mat / size;
MPI_Type_vector(order_mat, 1, nb_cols, 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);

lA = (double *) malloc( nb_cols * order_mat * sizeof(double) );
MPI_Scatter( A, nb_cols, type_col, lA, nb_cols,
             type_colProc, 0, MPI_COMM_WORLD );
```







# Derived types of matrix

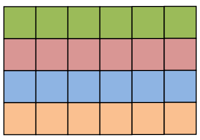
- Line of a matrix (in C): a derived type for contiguous data of a same type

```
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 );
```

# MPI_Type_indexed

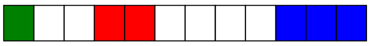
Fait pas !

- 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 );
```

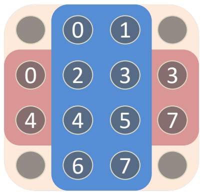


# MPI Communicators

- 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

- A 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
  - `MPI_Comm_group` : get the processes group of a given communicator

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

  - `MPI_Group_incl` : create un new group from a subset processes of a existing group, processes are reordered

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

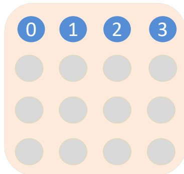
  - `MPI_Comm_create` : create un new comm. from a group of processes, collective operation

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

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

# Group / Intra-Communicator - Example

/* Building of a communicator with $row\_size = \sqrt{size}$ 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);
```

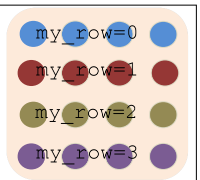


# Intra-Communicators

- Other constructors
  - `MPI_Comm_dup` : create un new group from a subset processes of a existing group
  - `MPI_Comm_split` : partition of a given communicator into disjoint sub-communicators. A sub-communicator is composed by processes with the same color.

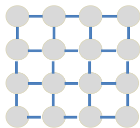
```
int MPI_Comm_split( MPI_Comm old_comm,
                    int color, MPI_Comm *new_comm);
```

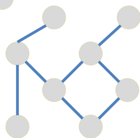
```
MPI_Comm my_row_comm;
int      my_row, row_size; // row_size = √size
my_row=my_rank/row_size;
MPI_Comm_split(MPI_COMM_WORLD, my_row,
               my_rank, &my_row_comm);
```



# 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 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

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

# Topology and intra-communicator

- Torus creation
  - Data structure

```
typedef struct {
   MPI_Comm grid_comm;/* communicator of grid      */
   int      nb_procs; /* processes number of gcomm */
   int      my_rank;  /* rank of process in gcomm  */

   MPI_Comm row_comm; /* communicator of row    */
   MPI_Comm col_comm; /* communicator of column */
   int      order;    /* order of grid          */
   int      coords[2];/* coordinates of process */
} grid2d_info_type;
```



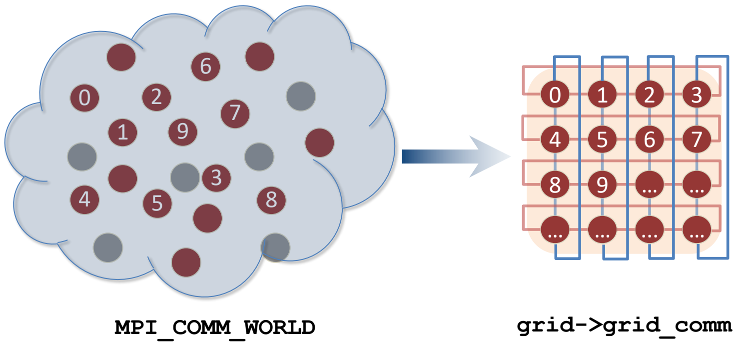
# Topology and intra-communicator

- Torus creation

```
void Setup_grid( grid2d_info_type *grid )
{
   int dims[2], periods[2];
   int coordinates[2], varying_coords[2];

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

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

# Topology and intra-communicator

- Torus creation – torus 2D communicator




P.S.: `grid->grid_comm = MPI_COMM_NULL for process not in torus`



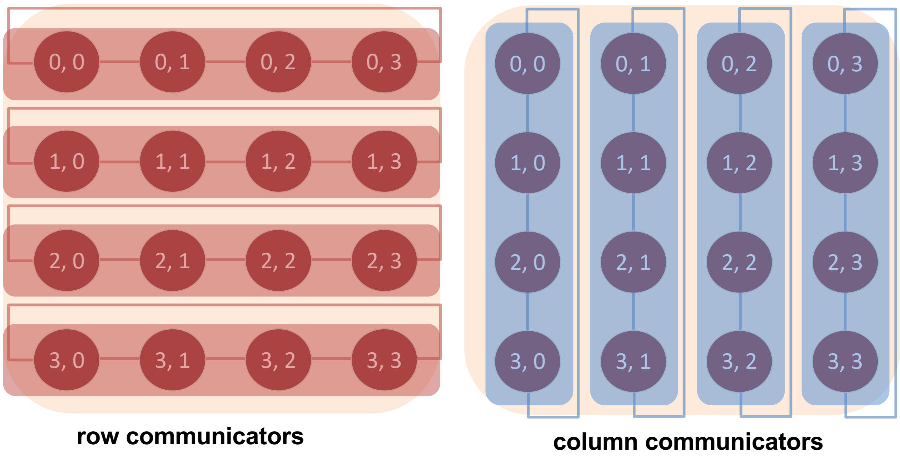
# Topology and intra-communicator

- Torus creation

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

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

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

# Topology and intra-communicator

- Torus creation – row/column communicators






# MxM – Algorithm of Fox

- Assumptions
  - `A`, `B`: `n`-by-`n` 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 each `P(i,j)`, $B_{ij}$ move

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

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

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



# MxM – 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

# MxM – 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?



# 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

# 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;

    /* 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 );
}
```

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



# 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 );
   }
}
```