Introduction to HPC – MPI

Design of parallel program and MPI

Jérôme Lelong, Christophe Picard, Laurence Viry

Université Grenoble Alpes

CEMRACS 2017

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Choice criterion

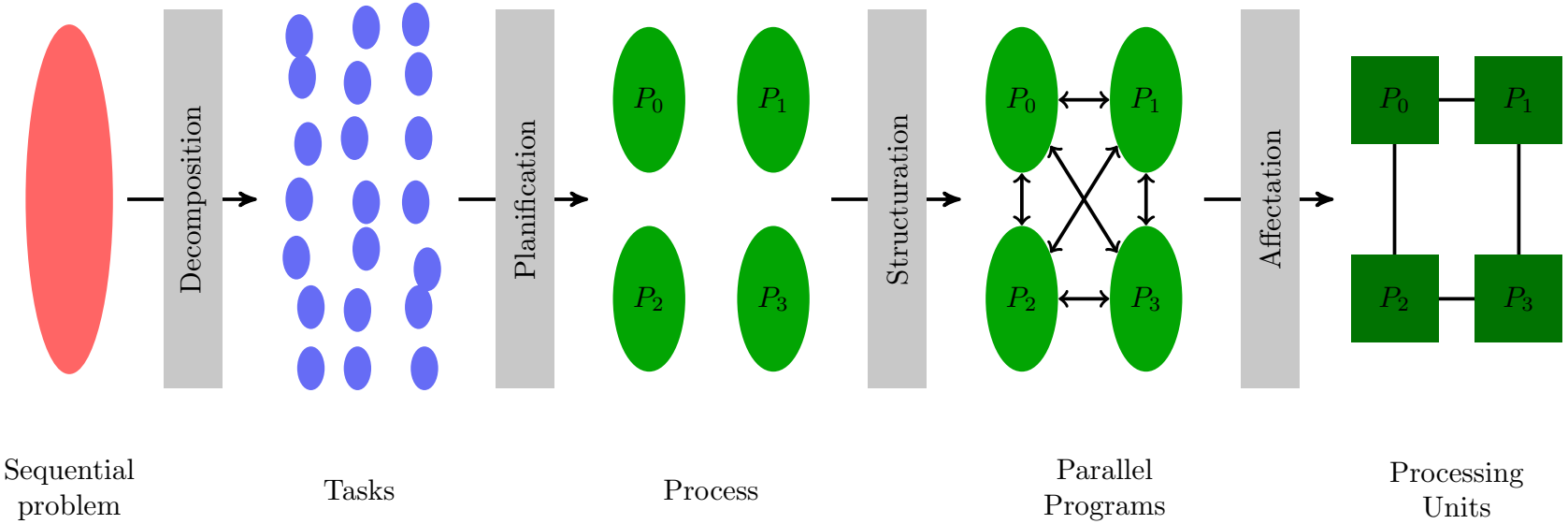
Three parameters influence the choice of the type of parallelism to use

- Flexibility: support of different programming constraints. Should adapt to different architectures.
- Efficiency: better scalability.
- Simplicity: allows to solve complex problem but with a low maintenance cost.

Dependency Analysis

- The analysis is made on three elements
  - Grouping the data
    - time dependency
    - collection of data
    - independence
  - Scheduling
    - Identify which data are required for executing a specific task.
    - Identify the tasks creating the different data.
  - Sharing the data
    - Identify the data shared between tasks.
    - Manage access to data.

Foster design



Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Decomposition I

- Identify the elements allowing parallel processing and determine the granularity of the decomposition.
- Break up computation into tasks to be divided among processes
  - tasks may become available dynamically.
  - number of tasks may vary with time.
- Enough tasks to keep processors busy: the number of tasks available at a given time is an upper bound on achievable speedup.

Decomposition II

How to decompose the code in order to achieve maximum parallelism?

- Focus on data: domain decomposition
  partition data first into elementary blocks of independent data then associate computation tasks with data.
- Focus on computation: functional decomposition
  partition computation first then associate data to tasks.
- Often, we use a combination of this decomposition.

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Communication

Describe the flow of information between the tasks.

- Structure: relation between producers and consumers.
- Content: volume of data to exchange.

We should

- Limit the number of communication operations.
- Distribute communications among tasks.
- Organize communication in such a way that they are concurrent to computations.

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Agglomeration

After partitioning and communication steps, we have a large number of tasks and a large amount of communication. Need to combine into large blocks

- Increase granularity: reduce communication costs.
- Maintain flexibility: improve scalability.
- Tasks are load balanced in terms of computations and communications.

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Mapping

Where to execute each tasks?

- Tasks which execute concurrently are placed on different processing units: increase concurrency.
- Tasks which communicate often are placed on the same processing units: increase locality.

Mapping: strategies

- Static mapping: equal–sized tasks, structured communication.
- Dynamic load balancing: variable number of computation and communication per task.
- Task scheduling: short tasks.
- In the last two cases, you can use a master/slave approach. Beware of bottlenecks.
- Probabilistic approaches.

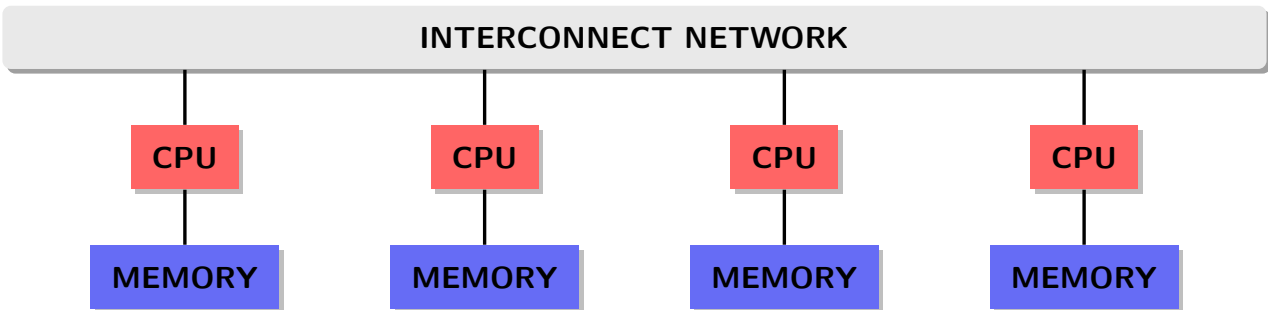
Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Distributed memory



Why?

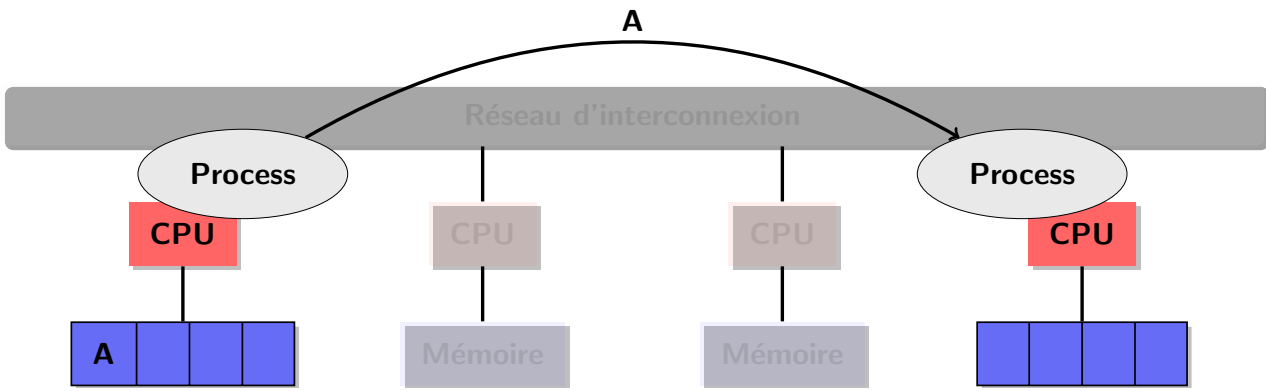
- OpenMP targets shared memory architectures, but the number of processing units is limited.
- On the different architectures with distributed memory, the main bottleneck is the connection between the components.
- We wish to use the different computing units available on a network.
- In order to simplify the communication protocols, a common library is needed.
- MPI standard allows to manage heterogeneous systems: clusters of PC or high performances systems with millions of cores.
- It allows hybrid programming for different systems.

The developer still need to handle communications and how the data are shared between the various resources.

Message Passing Communication

- A program execute several processes simultaneously. Some small part of the code may differ between the processes.
- Each process has its own data.
- Data from other processes cannot be accessed directly.
- Locally stored data are exchanged between the processes using specialized communication protocols.

Exchange of message



MPI standard

- MPI: message passing interface
- The first draft of the standard was proposed at Supercomputing 1993.
- Standardized, portable, efficient and flexible
  - can be used in C, C++, Fortran
  - avoids explicit use of network protocols and allows communications and computations simultaneously.
  - supported by a large number of manufacturers.
  - interface close to already existing protocols (PVM,. . .)
  - independence of the semantic with respect to the programming language.
  - thread–safe

Message passing

- Relies on exchanging message between processes to transfer data, synchronization of processes and global operations.
- MPI provides a complete infrastructure for managing the communications..
- Relies heavily on single program multiple data.
- Each process has its own data without being able to access others.
- The sharing of data is left to the programmer.
- Exchanges are done within a global space: a communicator.
- Each process is identified by a rank (int) within the communicator or sub-communicator.

Organization of MPI

1. environment
2. point to point communications
3. global communications
4. derived types
5. communicators
6. I/O

Compiling – Executing

- Include the file `"mpi.h"`
- Collection of wrappers for `gcc`: compile everything with `mpicc`, `mpic++` or `mpif90`
- Wrappers allow to use the compiler with the right options.
- To run an MPI program, we should specify the communication protocol between the processes: `export RSHCOMMAND=ssh`
- With the library OpenMPI
  `mpirun -machinefile file -np X executable [options]`
- `X` is the number of cores to be used.
- `file` specifies the list of processors and how to use them.
  `host1.exemple.com [slots=X1 max_slots=Y1]`
  `host2.exemple.com [slots=X2 max_slots=Y2]`
  `X` number of core/CPU on the node.
  `Y` max number of processes that MPI can use on this node.

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Environment

- Initialization at the begin of the program
  `int MPI_Init( int argc, char **argv)`
- `MPI` creates a communicator that groups active processes. It will manage communication among them.
- The default communicator is `MPI_COMM_WORLD`
- The program must finish with a call to
  `int MPI_Finalize(void)`
- To get the total number of processes managed by the communicator
  `int MPI_Comm_size( MPI_Comm comm, int *size )`
- To identify the rank of the process that is currently used
  `int MPI_Comm_rank( MPI_Comm comm, int *rank )`
- All `MPI` programs must contain these functions call.

A first example

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

int main (int argc, char **argv)
{
  int size, rank;
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD , &size);
  MPI_Comm_rank (MPI_COMM_WORLD , &rank);
  printf ("I am process %d/%d\n", rank, size-1);

  MPI_Finalize ();
  return 0;
}
```

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

General elements

- Communication between 2 processes: a transmitter and a receiver.
- There are two complementary operations: sending and receiving.
- The message contains at least the communicator, the identifier of the transmitter, the identifier of the receiver and a tag.
- Sent data are always typed. A message contains the address of the data to send, its type and its size.
- Basic send procedure:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
      int dest, int tag, MPI_Comm comm)
```

- Basic receive procedure:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Function send details

- `MPI_Send(buf, count, datatype, dest, tag, comm)`
  - `buf`: address of the buffer of the data to send
  - `count`: number of elements to send
  - `datatype`: type of each element
  - `dest`: rank of the process that will receive the message
  - `tag`: id (int) of the message
  - `comm`: communicator in which the communication takes place

Function receive details

- `MPI_Recv(buf, count, datatype, dest, tag, comm,status)`
  - `buf`: address of the buffer in which the message will be received
  - `count`: number of elements to receive
  - `datatype`: type of each element
  - `dest`: rank of the process emitting the message
  - `tag`: id (int) of the message
  - `comm`: communicator in which the communication takes place
  - `status`: `MPI` object containing information about the message

Example

```c
#include <stdio.h>
#include <string.h>
#include "mpi.h"
#define LENGTH 10

int main (int argc, char **argv) {
  int size, rank;
  char message[LENGTH];
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD , &size);
  MPI_Comm_rank (MPI_COMM_WORLD , &rank);
  MPI_Status status;

  if (rank == 0) {
    for (int i = 1; i < size; i++) {
      sprintf(message,"Hello%d}",i);
      MPI_Send(message,strlen(message),MPI_CHARACTER,i,99,MPI_COMM_WORLD);
    }
  } else {
    MPI_Recv(message,LENGTH,MPI_CHARACTER,0,99,MPI_COMM_WORLD,&status);
    printf("Process %d received %s\n",rank, message);
  }
  MPI_Finalize();
  return 0;
}
```

Some explanations

- Dispatch and receive a data to a destination and wait for the communication to end before continuing the execution of the program.
- The message is received only if the description matches.
- The data is sent or received before it is used.
- The execution of the program stops until the end of the communication before continuing.
- This communication is blocking.

Datatype in MPI

Type MPI	Type C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Communications

We can

- receive a message from any source: `MPI_ANY_SOURCE`
- receive a message with any tag: `MPI_ANY_TAG`
- get information about the source or the tag by looking at the variable `status` that contains the fields `MPI_TAG` and `MPI_SOURCE`.
- send a message containing derived types.
- send and receive a message at the same time: `MPI_Sendrecv`
- swap data using only one communication function: `MPI_Sendrecv_replace`

Optimization of point to point communications

- Optimization of communications $\Rightarrow$ improve the performances
- We wish to minimize the time spent in communicating and maximize the time spent in computing.
- Different complementary strategies:
  - overlap computation with communications.
  - avoid duplicating a message in a buffer.
  - Prefer few communications but with large data.

Standard blocking communications

- `MPI_Send(buf,count,datatype,dest,tag,comm)`
- `MPI_Recv(buf,count,datatype,source,tag,comm,&status)`
- The reception is finished only when the message is received: this is a blocking communication.
- It is guaranteed for the send either
  - using a buffer: the message is duplicated in a buffer.
  - using synchronization: before computation can advance, we wait for the reception to be initiated.
- This is the situation with `MPI_Send` and `MPI_Recv` functions.
- The function `MPI_Send` becomes blocking if the message is too large.

Standard non-blocking communications

- The program may continue its execution before its corresponding reception is initiated: it allows to overlap communication with computation.
- `MPI_ISend( buf,count,datatype, dest, tag, request)`
- `MPI_IRecv( buf,count,datatype,source, tag, request)`
- The field `request` is of type `MPI_Request`. It allows to identify the corresponding request.
- Once a communication has been initiated, it can be necessary to check the flow of the program.

Standard non-blocking communications

- Complete a non blocking operation:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- Verify if a request is completed:

```
int MPI_Test(MPI_Request *request,int *flag,
             MPI_Status *status)
```

- Test of a message arrived:

```
int MPI_Probe(int source,int tag,
              MPI_Comm comm,MPI_Status *status)
```

Synchronous communication

- Before advancing a computation after dispatching a message, we wait for the associated reception to be initialized.
- The message is not buffered $\Rightarrow$ we avoid the copy of temporary messages.
  `MPI_Ssend( buf,count,datatype, dest, tag, comm)`
- Completion is achieved only if the reception is initiated.
- The completion of `MPI_Ssend` means that the transmitted data is available for the transmitter and that the reception started.

Buffered communications

- The message is duplicated in a buffer before being dispatched.
- Allow the transmission to be initiated without having the corresponding reception ready.
- `MPI_BSend( buf,count,datatype, dest, tag, comm)`
- Allocation of buffers are managed with the functions

```
int MPI_Buffer_attach(void *buffer, int size)
int MPI_Buffer_detach(void *buffer, int size)
```

- The buffer is allocated in the memory of the transmitting process.
- The buffer can only be used for message bufferization.

Communication functions

`MPI_Send`	blocking	standard
`MPI_Isend`	non blocking	standard
`MPI_Ssend`	blocking	synchronous
`MPI_Issend`	non blocking	synchronize
`MPI_Bsend`	blocking	buffered
`MPI_Ibend`	non blocking	buffer's
`MPI_Recv`	blocking	standard
`MPI_Irecv`	non blocking	standard

Optimization

- Initiate reception before transmitting: if all communications are data exchange, put `MPI_Irecv` calls before `MPI_Send` calls.
- Overlap communications with computations: we use non blocking communications such as `MPI_Isend` and `MPI_Irecv`
- We prefer to send few large messages in place of several small messages.

Time measurement

- `double MPI_Wtime(void)` returns an elapsed time (wall time).
- Time measurement may change between executions.

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Collective communications

- A collective communication is a communication involving all the processes of the communicator provided as argument.
- Allows to perform in one operation multiple point to point communications.
- All the processes perform the same call with their own arguments.
- Similar in many ways to point to point communication.
- There in no tag.

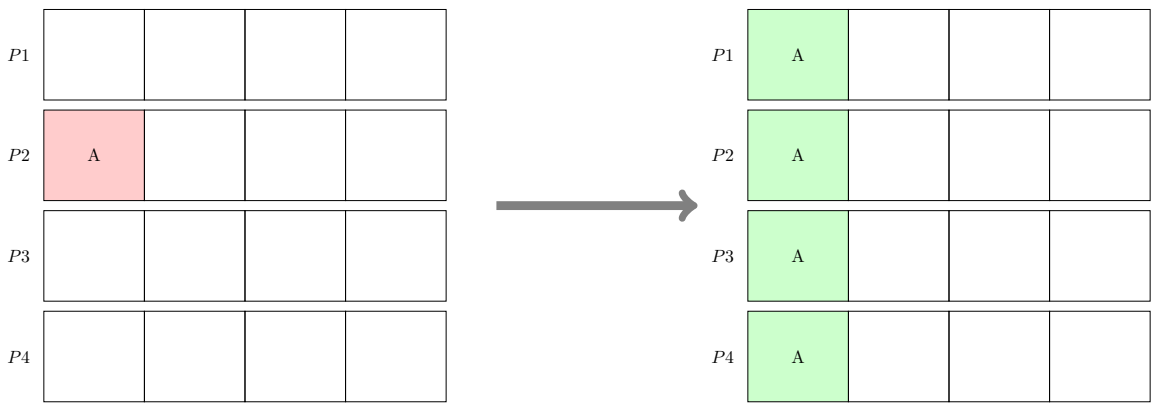
Short listing

- Global broadcast of data: `MPI_Bcast`
- Selective broadcast of data: `MPI_Scatter`
- Gather distributed data: `MPI_Gather`
- Gather distributed data by all processes: `MPI_Allgather`
- Global broadcast of distributed data: `MPI_Alltoall`
- Global reduction of data: `MPI_Reduce` and `MPI_Reduce_scatter`
- Partial reduction of data: `MPI_Scan`
- Global reduction of data with global broadcast of the result: `MPI_Allreduce`
- Synchronization of processes: `MPI_Barrier`

Global broadcast of data

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm)
```

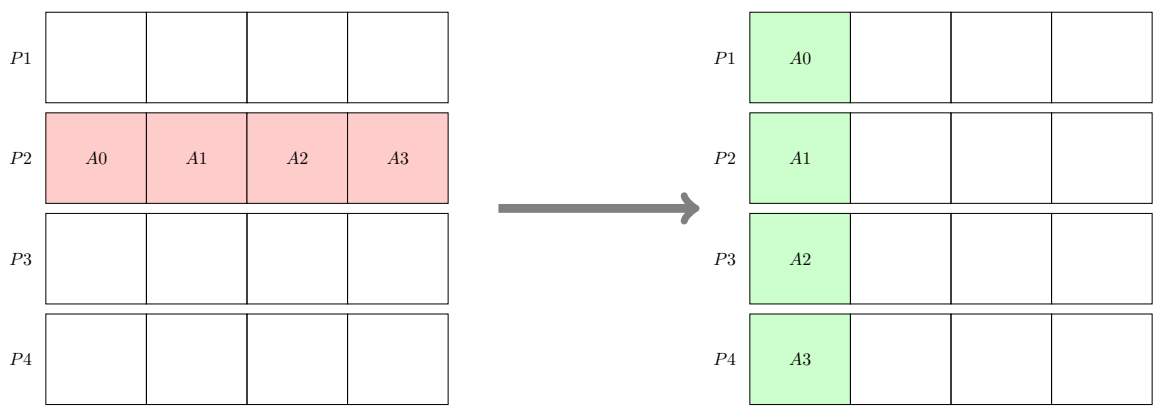
- The `root` process sends a message to all processes of the communicator. The message has length `count`, type `datatype` and is stored at the address `buf`
- Diffusion scheme



Selective broadcast of data

```
int MPI_Scatter(void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void * recvbuf, int recvount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- The `root` process sends a message to all the processes of the communicator. The message has length `count`, type `datatype` and is stored at the address `buf`
- All the processes of the communicator will receive a message with length `recvcount` and type `recvtype`. They store it at the address `recvbuf`.
- Diffusion scheme



Selective broadcast of data: explanations

- The initial data are split in n contiguous pieces of equal size.
- Only the `root` process understands the parameters relative to the data to be sent.
- It is as if `root` performs n send calls

```
MPI_Send(sendbuf+i,sendcount, extent(sendtype),
  sendcount,sendtype,i,comm)
```

- and each process in the communicator performs a blocking receive

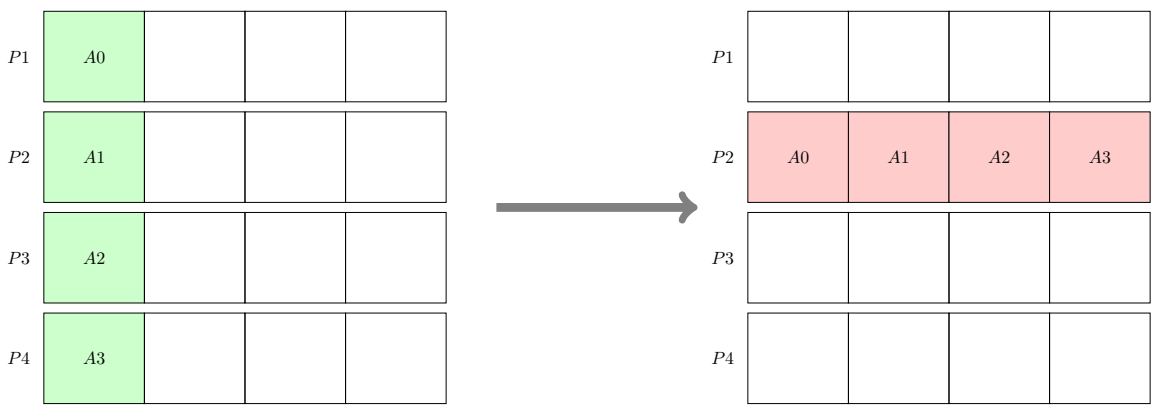
```
MPI_Recv(recvbuf+1,recvcount, recvtype ,root,comm)
```

- If the data to be received by each process have different sizes, use the function `MPI_Scatterv`.

Gather distributed data

```
int MPI_Gather(void *sendbuf, int sendcount,
      MPI_Datatype sendtype, void * recvbuf,int recvount,
      MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- Each process in the communicator sends a message with length `sendcount` and type `datatype` stored at the address `sendbuf`.
- The `root` process receives messages from all the processes in the communicator. Each message has length `recvcount`, type `recvtype` and will be stored at the address `recvbuf`
- Diffusion scheme



Gather distributed data: explanations

- Each process sends a set of compatible data.
- The `root` process calls the receive function n times

```
MPI_Recv(recvbuf+i,recvcount, extent(recvtype), recvtype
```

- Each process in the communicator performs a blocking sends

```
MPI_Send(sendbuf,sendcount, sendtype,root,comm)
```

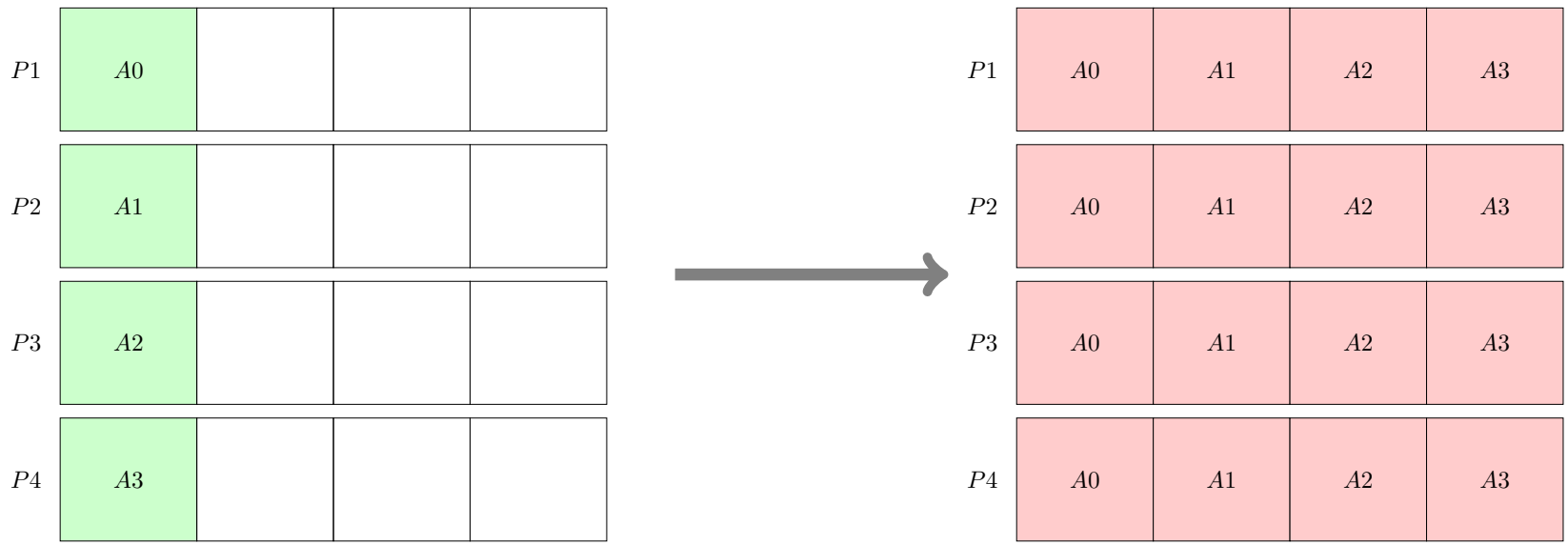
- If the data on each process have different sizes, use the function `MPI_GATHERV`.

Gather distributed data by all processes

```
int MPI_Allgather(void *sendbuf, int sendcount,
      MPI_Datatype sendtype, void * recvbuf,int recvount,
      MPI_Datatype recvtype, MPI_Comm comm)
```

- Each process in the communicator sends a message with length `sendcount` and type `datatype` stored at the address `sendbuf`.
- All the processes of the communicator receive all the messages with length `recvcount` and type `recvtype` wand store it at address `recvbuf`.
- This function can be seen as the combination of the following calls
  - All the processes call to `MPI_Gather`.
  - All the processes call to `MPI_Bcast`.
- If the data have different sizes, use the function `MPI_Allgatherv`.

Distribution scheme

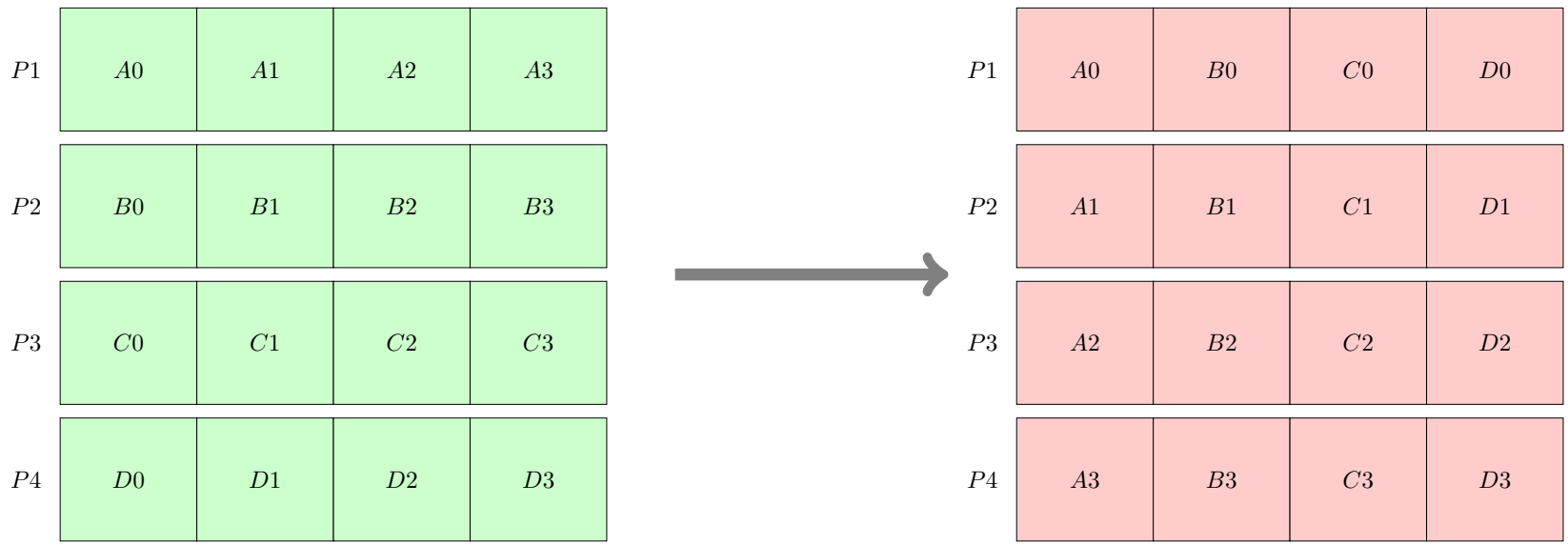


Global broadcast of distributed data

```
int MPI_Alltoall(void *sendbuf, int sendcount,
                 MPI_Datatype sendtype, void * recvbuf,int recvount
                 MPI_Datatype recvtype, MPI_Comm comm)
```

- All the processes in the communicator send a message with length `sendcount` and type `datatype` stored at the address `sendbuf`.
- All the processes in the communicator receive a message with length `recvcount` and type `recvtype` and store it at address `recvbuf`.
- This function can be seen as the combination of the following calls
  - All processes call `n` times `MPI_Send`.
  - All processes call `n` times `MPI_Recv`.
- If the data are have different sizes, we may use the function `MPI_AlltoallV`

Distribution scheme



Global reduction on data

```
int MPI_Reduce(void *sendbuf,void *recvbuf,int count,
               MPI_Datatype datatype,MPI_Op operation,
               int root,MPI_Comm)
```

- All the processes in the communicator `comm` send a message with length `count` and size `datatype` starting at address `sendbuf` to perform the operation `op` on the values. The result is stored at address `recvbuf` on the `root` process.
- If the number of sent data is greater than one, the same operation is perform on all the data.
- The operation `op` is associative.

Predefined operations

- `MPI_Max`: maximum
- `MPI_Min`: minimum
- `MPI_Sum`: sum
- `MPI_Land`: logical AND
- `MPI_Lor`: logical OR
- `MPI_Prod`: product
- `MPI_Lxor`: logical exclusive OR
- `MPI_Maxloc`: maximum and position
- `MPI_Minloc`: minimum and position

Custom operators can be created with `MPI_Op_Create` or `MPI_Op_Free`.

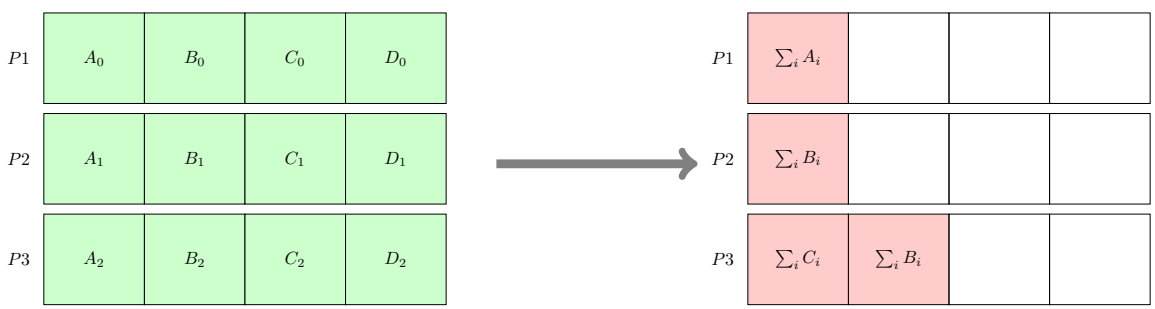
Global reduction of scatter data

```
int MPI_Reduce_Scatter(void *sendbuf,void *recbuf,
                       int revcounts, MPI_Datatype datatype,
                       MPI_Op operator, MPI_Comm comm)
```

- The operator `operator` is applied to all the data defined by `sendbuuf`, `count` and `datatype` with

$$count = \sum_i \texttt{revcounts[i]}$$

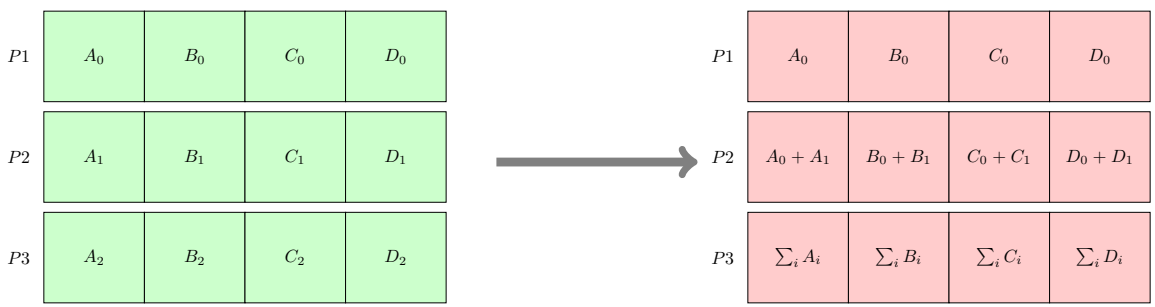
- Then, the result is shared depending on `revcounts`. Block i is sent to the process i and stored in the buffer defined by `revbuf`, `revcounts[i]` and `datatype`.



Partial reduction of data

```
int MPI_Scan(void* sendbuf,void* recvbuf, int count,
             MPI_Datatype datatype,MPI_Op operator,
             MPI_Comm comm)
```

- The operation `operator` is applied to all data depending on the rank of the process.
- Process i receives the result of the reduction performed on processes with ranks $0, \ldots, i$ in the communicator `comm`.
- The constraints are the same as for `MPI_Reduce`.



Reduction and broadcast

```
int MPI_Allreduce(void *sendbuf,void *recvbuf,
                  int count, MPI_Datatype datatype,
                  MPI_Op op,MPI_Comm comm)
```

- From all processes in the communicator `comm`, we send a message with length `count` and type `datatype` from address `sendbuf` to perform the operation `op`. The result is stored in `recvbuf` on all processes
- Equivalent to calling `MPI_Reduce` and then `MPI_Bcast`.

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Manipulating heterogeneous data

How to group data with various types together to send a large message rather than many small ones?

- Create a derived type
- Use serialization: Pack/Unpack. This is the approach used by `Boost::MPI`.

A derived type

How to describe a C structure.

- The number of fields in the structure (int)
- The types of each field (`MPI_Datatype *`)
- The number of elements in each field (`int *`)
- The addresses of the different fields are computed relatively to the address of the first field of the structure (`MPI_Aint *`)

```
int MPI_Type_create_struct(int count,
        int *array_of_blocklengths,
        MPI_Aint *array_of_displacements,
        MPI_Datatype *array_of_types,
        MPI_Datatype *newtype)
```

Once created, a type must be recorded.

`int MPI_Type_commit(MPI_Datatype *datatype).`

An toy example

```
typedef struct { int i, j; float f; char tab[10]; } structure;
MPI_Datatype newtype,
MPI_Aint displ [3];
int longueurs [3] = {2, 1, 10} ;
MPI_Datatype types [3] = { MPI_INT, MPI_FLOAT, MPI_CHAR };

MPI_Get_address(&(structure.i), &(displ[0]));
MPI_Get_address(&(structure.f), &(displ[1]));
MPI_Get_address(structure.tab, &(displ[2]));

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

MPI_Type_create_struct(3,longueurs,displ,types,&newtype);
MPI_Type_commit(&newtype);
```

Simplified mechanisms

- Homogeneous and contiguous data: `MPI_Type_contiguous`.
  Example: a row of a matrix.
  ```
  MPI_Type_contiguous(n, MPI_DOUBLE, newtype);
  ```
- Homogeneous, non contiguous but equally spaced data:
  `MPI_Type_vector` (en stockage ligne).
  Example: a column of a row major matrix.
  ```
  MPI_Type_vector(m, 1, n, MPI_DOUBLE, newtype);
  ```
- Homogeneous, non contiguous and non equally spaced data:
  ```
  int MPI_Type_indexed(int count, int *array_of_blocklen,
        int *array_of_displ, MPI_Datatype oldtype,
        MPI_Datatype *newtype)
  ```

Packing

Enables to write heterogeneous data in a binary string, similar to serialization.

```
int MPI_Pack(void *inbuf, int incount,
             MPI_Datatype datatype, void *outbuf,
             int outsize, int *position, MPI_Comm comm)
```

- `inbuf`: address of the data to pack
- `incount`: number of elements
- `datatype`: type of each element
- `outbuf`: address where to write the binary string
- `outsize`: size of the binary string
- `position`: position at which to start writing in `outbuf`. It is automatically incremented by each call to `MPI_Pack`.

Packing

- We need to know how much memory to allocate for the binary string.

```
int MPI_Pack_size(int incount, MPI_Datatype datatype,
MPI_Comm comm, int *size)
```

- To reconstruct the data

```
int MPI_Unpack(void *inbuf, int insize, int *position,
    void *outbuf, int outcount, MPI_Datatype datatype,
    MPI_Comm comm)
```

Arguments are similar to `MPI_Pack`.

Packing: practical usage

- Call `MPI_Pack_size` for each element to be packed. Then, you know the total size required by `buffer`.
- Allocate `buffer` accordingly.
- Actually pack the data by calling `MPI_Pack` for every element to be stored.
- Send the message.
- On the destination process, call
  - `MPI_Probe`: is the message available?
  - `MPI_Get_count`: what is its size?
- On the destination process, allocate a buffer with the correct size, receive the data and unpack them.

Agenda

1. Design of parallel program
   - Decomposition
   - Communication
   - Agglomeration
   - Mapping
2. MPI
   - Presentation of MPI
   - MPI Environment
   - Point to point communication
   - Collective communications
   - Manipulating heterogeneous data
     - Derived datatype
     - Packing
   - Communicators

Construction

- A communicator contains all information required to perform communications in `MPI`.
- `MPI` provides a default communicator `MPI_COMM_WORLD` containing all the processes of the application.
- We can create communicator from
  - an other communicator.
  - a group of process.
- Communicators can be of two different kind
  - intra–communicators: for operations on a group of processes inside a communicator.
  - inter–communicators: for communications between two groups of processes.

Conclusion

- Allows to precisely manage the exchange of information between processes.
- The communication network is managed directly by `MPI`.
- Optimization are required for each problem and each architecture.
- It is possible to do hybrid programming with `MPI` and `OpenMP` (and `GPU`).

Distributed file systems — Map/Reduce

Target big data computations.

- Data are distributed over several disks, possibly in different locations.
- It enables to work with data sets too large to fit in memory.
- Do some computations on the data: *Map*.
- Gather the results: *Reduce*.

Examples: Hadoop, SPARK.