SPD : Systèmes et Programmation Distribués

Nicolas Szczepanski

20 février 2018

szczepanski.nicolas@gmail.com

Partie 3 : Message Passing Interface

Baselines

MPI : Message Passing Interface

- A standard for message passing library
- Efficient, Portable, Scalable, Vendor Independent
- Follow to use different Parallel Programming Model by adding other libraries :
  - Distributed Memory
  - Shared Memory
  - Hybrid Memory
- Several implementations : OpenMPI, MPICH2, . . .
- Implementations support different versions and functionalities of the standard
- Dedicated to Hight Performance Computing (HPC)

MPI : Hello World

```
#include <iostream>
#include <mpi.h>
int main(int argc,char** argv){
  MPI_Init(&argc,&argv);
  std::cout << "Hello World!" << std::endl;
  MPI_Finalize();
}
```

```
COPTION = -std=c++11 -Wall -Wextra -O3
LOPTION = -lpthread
main : main.o
       mpic++ -o hello main.o $(LOPTION)

main.o : main.cc
         mpic++ $(COPTION) -c main.cc
```

```
//Run four times the program hello
mpirun -n 4 hello
//Run two times the program hello on the computer A
//And run two times this program on the computer B
mpirun -n 2 --host hostnameA hello : -n 2 --host hostnameB hello
//Run one server and four clients
mpirun -n 1 server : -n 4 client
```

MPI : `MPI_Init`

```
void MPI_Init(int argc, char** argv)
void MPI_Init()
```

- Initialize resources for MPI according to hardwares, nodes, ...
- Must be called before most other MPI routines are called
- MPI can be initialized at most once
- Subsequent calls to `MPI_Init` are erroneous
- `mpiexec` or `mpirun` can also append extra arguments
- `MPI_Init(argc, argv)` is the only way to clean `argc` and `argv`
- Use `MPI_Init(argc, argv)` to have no MPI arguments
- Use `MPI_Init` to have MPI arguments

MPI : `MPI_Finalize()`

```
int MPI_Finalize()
```

- Cleans up all MPI states
- Once this routine is called, no MPI routine may be called
- All processes must call this routine before exiting

Communicators and Ranks

Definition :

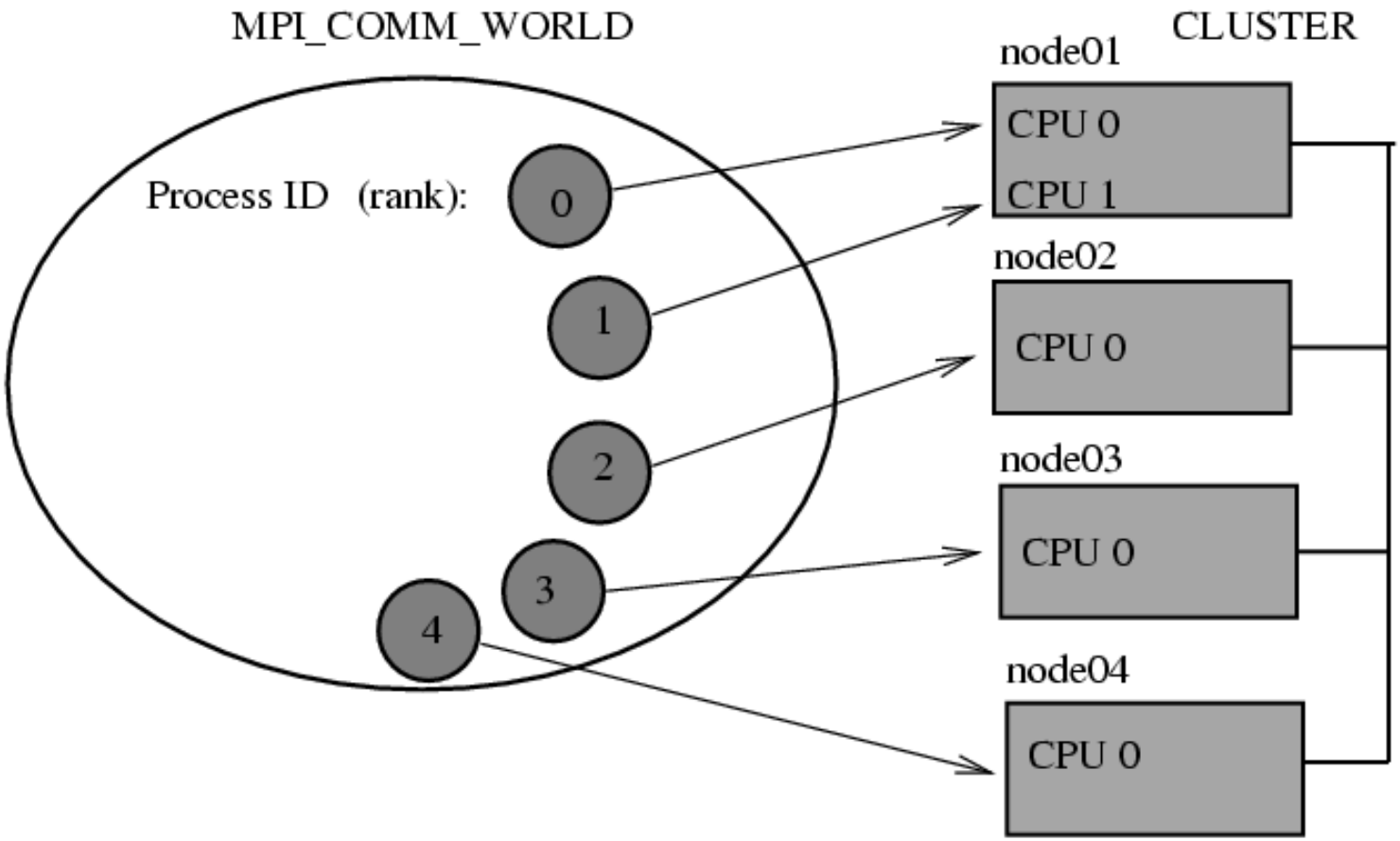
MPI uses a **communicator** object to identify a set of processes which communicate only within their set. `MPI_COMM_WORLD` is defined in the MPI include file as all processes (ranks) of your job. You can create subsets of `MPI_COMM_WORLD`.

Definition :

A **rank** is a unique process ID within a communicator

- Ranks of the communicator `MPI_COMM_WORLD` are assigned by MPI by `MPI_Init`
- Ranks of a communicator with n processes are assigned numbers 0 to n-1
- Ranks ares used to specify sources and destinations of messages and some other operations

Communicators and Ranks



MPI_Comm_Size() and MPI_Comm_Rank()

```
//comm <=> the communicator
//size <=> the size of this communicator
int MPI_Comm_size(MPI_Comm comm, int *size)
```

- Reports the number of processes of a communicator

```
//comm <=> the communicator
//rank <=> the rank of the process in this communicator
int MPI_Comm_rank(MPI_Comm comm,int* rank)
```

- Determines the rank of the calling process in the communicator

```
#include <iostream>
#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);
  std::cout << "Hello World! rank "
            << rank << " on " << size << std::endl;
  MPI_Finalize();
}
```

Elementary MPI datatypes

Communication functions utilize MPI Datatypes as a means to specify the structure of a message at a higher level

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Blocking communication : `MPI_Send()`

Blocking communications = do not return (block)
until the communication is not finished

```
int MPI_Send(const void *buf
             ,int count
             ,MPI_Datatype datatype
             ,int dest
             ,int tag
             ,MPI_Comm comm)
//buf <=> data to send
//count <=> the number of data to send
//datatype <=> type of data to send
//dest <=> the rank of the receiver process
//tag <=> used to differentiate messages
//comm <=> the communicator of processes used in the communication
```

- Function blocks until message buffer is not freed
- The message buffer is freed when :
  - Message copied to system buffer, or
  - Message transmitted

Point to Point Communications

Blocking communication : `MPI_Recv()`

Blocking communications = do not return (block) until the communication is not finished

```
int MPI_Recv(void *buf
             ,int count
             ,MPI_Datatype datatype
             ,int source
             ,int tag
             ,MPI_Comm comm
             ,MPI_Status *status)
//buf <=> data to receive
//count <=> the number of data to receive
//datatype <=> type of data to receive
//source <=> the rank of the sender process
//tag <=> used to differentiate messages
//comm <=> the communicator of processes used in the communication
//status <=> to have some information on the communication
```

- Function blocks until message is in the receiver buffer
- If message never arrives, function never returns !

MPI_Send() and MPI_Recv() example

```
#include <iostream>
#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);
  int number;
  if(!rank){
    number=666;
    MPI_Send(&number,1,MPI_INT,1,MPI_ANY_TAG,MPI_COMM_WORLD);
  }else if(rank == 1){
    MPI_Recv(&number,1,MPI_INT,0,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    std::cout << "Process 1 received number"
              << number << "from process 0" << std::endl;
  }
  MPI_Finalize();
}
```

Exercise : MPI ping pong program

Exercise : MPI ping pong program

```
#include <iostream>
#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);
  int pingPong=0;
  const int limit=100;
  int partnerRank=(rank+1)%2;
  while(pingPong < limit){
    if(rank == pingPong%2){
      pingPong++;
      MPI_Send(&pingPong,1,MPI_INT,partnerRank,0,MPI_COMM_WORLD);
    }else{
      MPI_Recv(&pingPong,1,MPI_INT,partnerRank,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
  }
  MPI_Finalize();
}
```

Deadlock

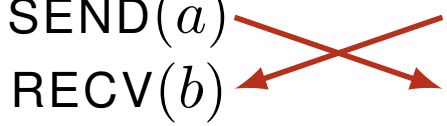
Définition :

Un **interblocage** (en anglais, *deadlock*) est un état dans lequel chaque membre d'un groupe est en train d'attendre indéfiniment qu'un ou plusieurs autres membres fassent une action comme la récupération d'un message ou le blocage/déblocage d'un mutex. Les processus bloqués dans cet état le sont définitivement, il s'agit donc d'une situation catastrophique. Nous pouvons avoir des interblocages provenant d'un modèle de programmation :

- de la mémoire partagée (attente d'un mutex (d'une ressource))
- de la mémoire distribuée (attente d'un message)

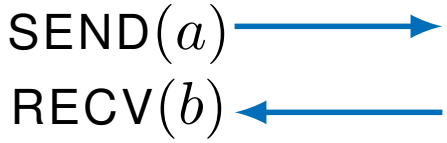
Distributed Memory Deadlock

PROCESSUS 1	PROCESSUS 2
SEND(a)	SEND(b)
RECV(b)	RECV(a)



Deadlock !

PROCESSUS 1	PROCESSUS 2
SEND(a)	RECV(a)
RECV(b)	SEND(b)



The solution : Serialize communications !

Distributed Memory Deadlock

Why ?

Pairwise Exchange

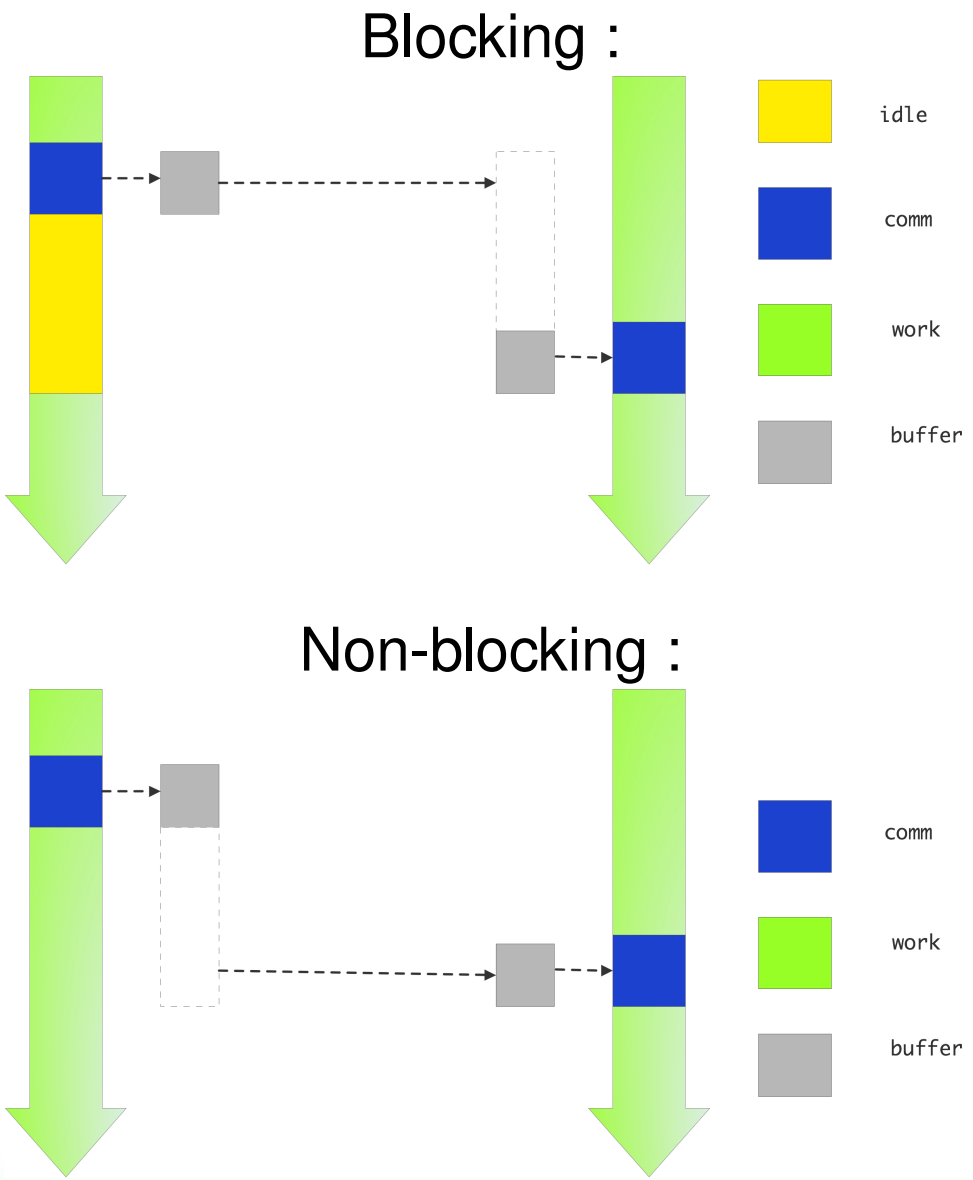
```
int MPI_Sendrecv(const void *sendbuf
                ,int sendcount
                ,MPI_Datatype sendtype
                ,int dest
                ,int sendtag
                ,void *recvbuf
                ,int recvcount
                ,MPI_Datatype recvtype
                ,int source
                ,int recvtag
                ,MPI_Comm comm
                ,MPI_Status *status)
//sendbuf <=> initial address of send buffer (choice)
//sendcount <=> number of elements in send buffer
//sendtype <=> type of elements in send buffer
//dest <=> rank of destination
//sendtag <=> send tag
//recvbuf <=> initial address of receive buffer
//recvcount <=>number of elements in receive buffer
//recvtype <=>type of elements in receive buffer
//source <=>rank of source or MPI_ANY_SOURCE
//recvtag <=> receive tag or MPI_ANY_TAG
//comm <=> communicator
//status <=> status object
```

Exercise : Neighbour

Exercise : Neighbour

```
#include <iostream>
#include <mpi.h>
int main(int argc,char** argv){
  int size, rank;
  int valToSend=random();
  int valToRecv;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  right=(rank+1)%size;
  left=(!rank)?(size-1):rank-1;
  MPI_Sendrecv(&valToSend,1,MPI_INT,left,0
              ,&valToRecv,1,MPI_INT,right,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  MPI_Finalize();
}
```

Blocking Vs Non-blocking Communication



MPI_Isend() and MPI_Irecv()

```
int MPI_Isend(const void *buf
              ,int count
              ,MPI_Datatype datatype
              ,int dest
              ,int tag
              ,MPI_Comm comm
              ,MPI_Request* request)
//buf <=> data to send
//count <=> the number of data to send
//datatype <=> type of data to send
//dest <=> the rank of the receiver process
//tag <=> used to differentiate messages
//comm <=> the communicator of processes used in the communication
//request <=> communication request
```

```
int MPI_Irecv(void *buf
              ,int count
              ,MPI_Datatype datatype
              ,int source
              ,int tag
              ,MPI_Comm comm
              ,MPI_Request* request)
//buf <=> data to receive
//count <=> the number of data to receive
//datatype <=> type of data to receive
//source <=> the rank of the sender process
//tag <=> used to differentiate messages
//comm <=> the communicator of processes used in the communication
//request <=> communication request
```

Exercise : Non-blocking Neighbour

```
#include <iostream>
#include <mpi.h>
int main(int argc,char** argv){
  int size, rank;
  int valToSend=random();
  int valToRecv;
  MPI_Request request1, request2;
  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  right=(rank+1)%size;
  left=(!rank)?(size-1):rank-1;
  MPI_Irecv(&valToSend,1,MPI_INT,left,0,MPI_COMM_WORLD,&request1);
  MPI_Isend(&valToRecv,1,MPI_INT,right,0,MPI_COMM_WORLD,&request2);
  //some work
  MPI_Wait(&request1,&status);
  MPI_Wait(&request2,&status);
  MPI_Finalize();
}
```

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
//request <=> communication request
//status <=> to have some information on the communication
```

MPI_Wait(),MPI_Waitall(),...

```
for (p=0; p<nrequests ; p++) // Not efficient!
    MPI_Wait(request,&status);
```

Why ?

```
int MPI_Waitall(int count, MPI_Request requests[],MPI_Status statuses[])
//count <=> number of requests
//requests <=> array of requests
//statuses <=> array of status objects (out), may be MPI_STATUSES_IGNORE
```

```
int MPI_Waitany(int count,MPI_Request requests[],int *indx,MPI_Status *status)
//count <=> number of requests
//requests <=> array of requests
//indx <=> index of handle for operation that completed (in the range 0 to count-1)
//status <=> only one status object (out), may be MPI_STATUSES_IGNORE
```

```
for (p=0; p<nrequests; p++) {
  MPI_Waitany(nrequests,request_array,index,status);
  // operate on the message of the request number index
}
```

MPI_Test()

```
int MPI_Test(MPI_Request *request,int *flag,MPI_Status *status);
//request <=> the request to test
//flag <=> true if operation completed (a message is received) (out)
//status <=> status object (out), may be MPI_STATUS_IGNORE
```

```
#define MSG_STOP 666
void Worker::start(){
  int ManagerMessageReceived = 0;
  int ManagerFlag = 1;
  MPI_Status ManagerStatus;
  MPI_Request ManagerRequest;
  if(ManagerFlag)
  for(;;){
    if(ManagerFlag){//Init or Reinit
      ManagerFlag = 0;
      MPI_Irecv(&ManagerMessageReceived,1,MPI_INT,0
      ,TAG_MANAGER,MPI_COMM_WORLD,&ManagerRequest);
    }
    MPI_Test(&ManagerRequest, &ManagerFlag, &ManagerStatus);
    if(ManagerFlag){
      if(ManagerMessageReceived == MSG_STOP){
        return;
      }else{
        printf("Message␣received␣unknown␣from␣\%d\n",ManagerStatus.MPI_SOURCE);
        exit(0);
      }
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
  }
}
```

How to stop a `MPI_Irecv()` operation ?

Is this correct ?

```
#define MSG_STOP 666
void Worker::start(){
  int ManagerMessageReceived = 0;
  int ManagerFlag = 1;
  MPI_Status ManagerStatus;
  MPI_Request ManagerRequest;
  if(ManagerFlag)
  for(;;){
    if(IhaveToStop)return;
    if(ManagerFlag){//Init or Reinit
      ManagerFlag = 0;
      MPI_Irecv(&ManagerMessageReceived,1,MPI_INT,0
      ,TAG_MANAGER,MPI_COMM_WORLD,&ManagerRequest);
    }
    MPI_Test(&ManagerRequest, &ManagerFlag, &ManagerStatus);
    if(ManagerFlag){
      if(ManagerMessageReceived == MSG_STOP){
        return;
      }else{
        printf("Message␣received␣unknown␣from␣\%d\n",ManagerStatus.MPI_SOURCE);
        exit(0);
      }
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
  }
}
```

How to stop a `MPI_Irecv()` operation ?

```
int MPI_Cancel(MPI_Request *request);
//request <=> the request to test
```

Definition :

The primary expected use of `MPI_Cancel()` is in multi-buffering schemes, where speculative `MPI_Irecv()` are made. When the computation completes, some of these receive requests may remain ; using `MPI_Cancel()` allows the user to cancel these unsatisfied requests. A call to `MPI_Cancel()` marks for cancellation a pending, nonblocking communication operation (send or receive). The cancel call is local. It returns immediately, possibly before the communication is actually canceled.

Warning !

It is still necessary to complete a communication that has been marked for cancellation, using a call to `MPI_Wait()` or `MPI_TEST()` or any of the derived operations like `MPI_Request_free()`.

How to stop a `MPI_Irecv()` operation ?

```
#define MSG_STOP 666
void Worker::start(){
  int ManagerMessageReceived = 0;
  int ManagerFlag = 1;
  MPI_Status ManagerStatus;
  MPI_Request ManagerRequest;
  if(ManagerFlag)
  for(;;){
    if(IhaveToStop){
      if(!ManagerFlag){//A Irecv operation is is in progress
        MPI_Test(&ManagerRequest, &ManagerFlag, &ManagerStatus);
        if(ManagerFlag){//I have received my message
          if(ManagerMessageReceived == MSG_STOP){
            return;
          }else{
            printf("Message␣received␣unknown␣from␣\%d\n",ManagerStatus.MPI_SOURCE);
            exit(0);
          }
        }else{//I don't have received my message : cancel
          MPI_Cancel(&ManagerRequest);
          MPI_Test(&ManagerRequest, &ManagerFlag, &ManagerStatus);
        }
      }
    }
    ...

    ...
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
  }
}
```

How to stop a `MPI_Irecv()` operation ?

```
#define MSG_STOP 666
void Worker::start(){
  int ManagerMessageReceived = 0;
  int ManagerFlag = 1;
  MPI_Status ManagerStatus;
  MPI_Request ManagerRequest;
  if(ManagerFlag)
  for(;;){
    if(IhaveToStop){
      if(!ManagerFlag){//A Irecv operation is is in progress
        MPI_Test(&ManagerRequest, &ManagerFlag, &ManagerStatus);
        if(ManagerFlag){//I have received my message
          if(ManagerMessageReceived == MSG_STOP){
            return;
          }else{
            printf("Message␣received␣unknown␣from␣\%d\n",ManagerStatus.MPI_SOURCE);
            exit(0);
          }
        }else{//I don't have received my message : cancel
          MPI_Cancel(&ManagerRequest);
          MPI_Test(&ManagerRequest, &ManagerFlag, &ManagerStatus);
        }
      }
    }
    ...

    ...
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
  }
}
```

Synchronous And Asynchronous Communication

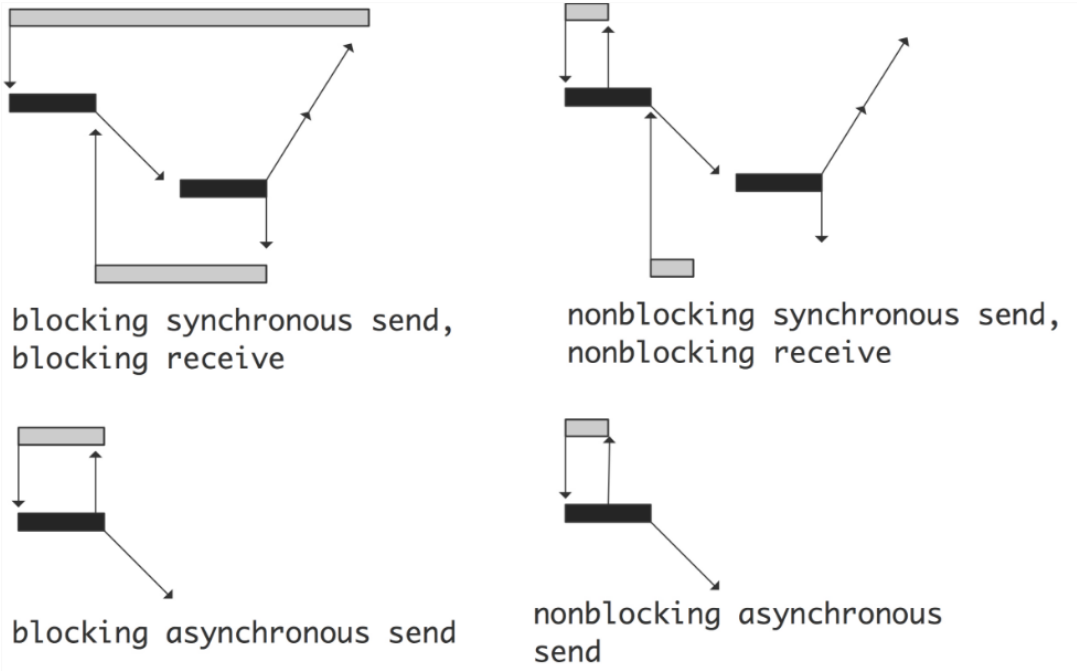
Definition :

Synchronization is a concept in itself, and we talk about **synchronous communication** if there is actual coordination going on with the other process, and **asynchronous communication** if there is not.
Blocking communications then only refers to the program waiting until the user data is safe to reuse.

- In the synchronous case a blocking call means that the data is indeed transferred
- In the asynchronous case it only means that the data has been transferred thanks to some system buffer

Communication Modes

Send Modes	MPI Function
Standard send	`MPI_Send()`
Synchronous send	`MPI_Ssend()`
Buffered send	`MPI_Bsend()`
Ready send	`MPI_Rsend()`



Standard Send : `MPI_Send()`

Definition :

In this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver. More precisely, this mode uses a first protocol for short messages, and a second protocol for long messages.

A send in standard mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. The standard mode send is **non-local** : successful completion of the send operation may depend on the occurrence of a matching receive.

Buffered send : `MPI_Bsend()`

Definition :

A buffered mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is **local**, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. An error will occur if there is insufficient buffer space. The amount of available buffer space is controlled by the user with `MPI_Buffer_attach()`),....

Buffered send : `MPI_Ssend()`

Definition :

A send that uses the synchronous mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics : a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is **non-local**.

Ready send : `MPI_Rsend()`

Definition :

A send that uses the ready communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation (faire l'operation en un seul coup) that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation ; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance.

Message Probing

MPI_Status()

Definition :

With some receive calls you know everything about the message in advance :

- its source
- its tag
- its size

Definition :

In some applications it makes sense that a message can come from one of a number of processes. In this case, it is possible to specify `MPI_ANY_SOURCE` as the source in `MPI_Recv()`, `MPI_Irecv()`, ... To find out where the message actually came from, you would use the `MPI_SOURCE` field of the status object like in this example.

```
MPI_Recv(&buffer,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
const int sender = status.MPI_SOURCE;
```

MPI_Get_count()

```
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count)
//status <=> a status object
//datatype <=> the type of the message (MPI_INT, ...)
//count <=> the number of data of the message
```

```
MPI_Recv(&buffer,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
const int sender = status.MPI_SOURCE;
const int tag = status.MPI_TAG;
int nbData;
MPI_Get_count(&status,MPI_INT,&nbData);
```

How to know the size of a message before its reception ?

MPI_Probe()

Definition :

MPI receive calls specify a receive buffer, and its size has to be enough for any data sent. In case you really have no idea how much data is being sent, and you don't want to overallocate the receive buffer, you can use `MPI_Probe()`.

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
//source <=> source rank, or MPI_ANY_SOURCE
//tag <=> tag value or MPI_ANY_TAG
//comm <=> the communicator
//status <=> information obtained by the probe (out)
```

```
#define MYTAG 666
if(rank==receiver){
  MPI_Status status;
  MPI_Probe(sender,MYTAG,MPI_COMM_WORLD,&status);
  int count;
  MPI_Get_count(&status,MPI_FLOAT,&count);
  float recvBuffer[count];
  MPI_Recv(recvBuffer,count,MPI_FLOAT,sender,MYTAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}else if(rank==sender){
  int bufferSize = 10;
  float buffer[bufferSize];
  MPI_Send(buffer,bufferSize,MPI_FLOAT,receiver,MYTAG,MPI_COMM_WORLD);
}
```

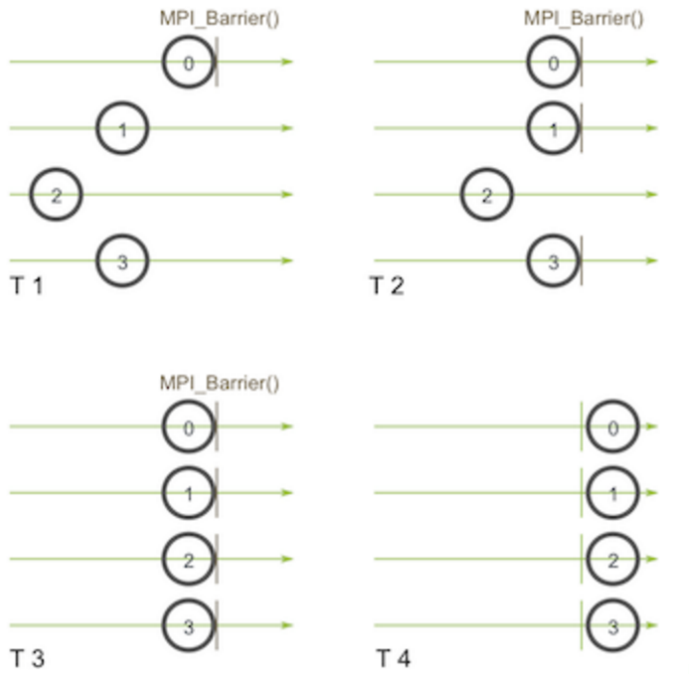
Collective Communications

MPI_Barrier())

```
int MPI_Barrier(MPI_Comm comm)
//comm <=> the communicator
```

Definition :

Blocks the caller until all processes in the communicator have called it ; that is, the call returns at any process only after all members of the communicator have entered the call.

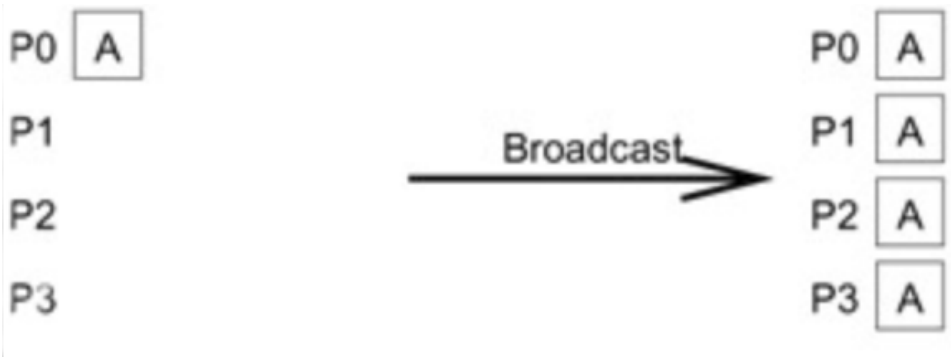


MPI_Bcast())

```
int MPI_Bcast(void *buffer,int count,MPI_Datatype datatype,int root,MPI_Comm comm)
//buffer <=> data to send
//count <=> the number of data to send
//datatype <=> type of data to send
//root <=> rank of broadcast root
//comm <=> the communicator
```

Definition :

Broadcasts a message from the process with rank ≪ root ≫ to all other processes of the communicator.



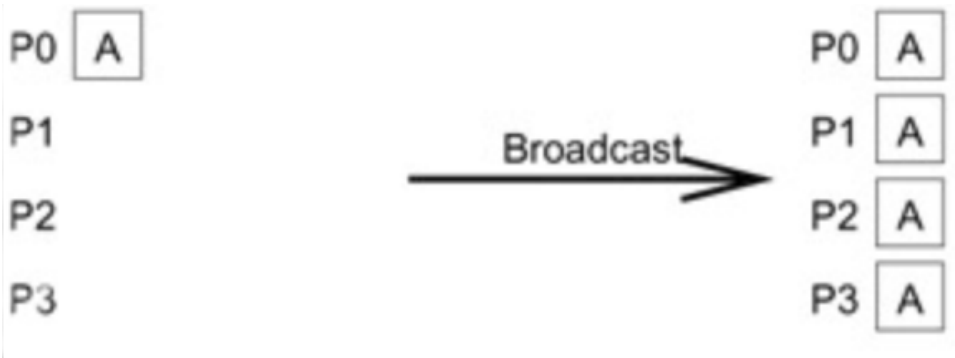
MPI_Bcast())

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  int rank;
  int buf;
  const int root=0;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if(rank == root)buf = 777;

  printf("[%d]: Before Bcast, buf is %d\n", rank, buf);
  /* everyone calls bcast, data is taken from root and ends up in everyone's buf */
  MPI_Bcast(&buf, 1, MPI_INT, root, MPI_COMM_WORLD);
  printf("[%d]: After Bcast, buf is %d\n", rank, buf);

  MPI_Finalize();
  return 0;
}
```



MPI_Scatter()

```
int MPI_Scatter(const void *sendbuf,int sendcount,MPI_Datatype sendtype,
           void *recvbuf, int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm)
//sendbuf <=> data to send
//sendcount <=> the number of data to send
//sentype <=> type of data to send
//recvbuf <=> address of receive buffer (out)
//recvcount <=> number of elements in receive buffer
//recvtype <=> data type of receive buffer elements
//root <=> rank of sending process
//comm <=> the communicator
```

Definition :

Sends data from one process (root) to all other processes in a communicator. `MPI_Scatter()` takes an array of elements and distributes the elements in the order of process rank.

MPI_Scatter()

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size(comm,&gsize);
sendbuf = (int*)malloc(gsize*100*sizeof(int));
...
MPI_Scatter(sendbuf,100,MPI_INT,rbuf,100,MPI_INT,root,comm);
```

MPI_Gather()

```
\begin{lstlisting}
  int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
           void *recvbuf, int recvcount, MPI_Datatype recvtype,
           int root, MPI_Comm comm)
  //sendbuf <=> data to send
  //sendcount <=> the number of data to send
  //sentype <=> type of data to send
  //recvbuf <=> address of receive buffer (out)
  //recvcount <=> number of elements in receive buffer
  //recvtype <=> data type of receive buffer elements
  //root <=> rank of sending process
  //comm <=> the communicator
```

Definition :

Gathers together values from a group of processes to one process (root).

MPI_Gather()

```
MPI_Comm comm;
int gsize,sendarray[100];
int root,myrank,*rbuf;
...
MPI_Comm_rank(comm,,&myrank);
if(myrank == root){
  MPI_Comm_size(comm,,&gsize);
  rbuf =(int*)malloc(gsize*100*sizeof(int));
}
MPI_Gather(sendarray,100,MPI_INT,rbuf,100, MPI_INT,root,comm);
```

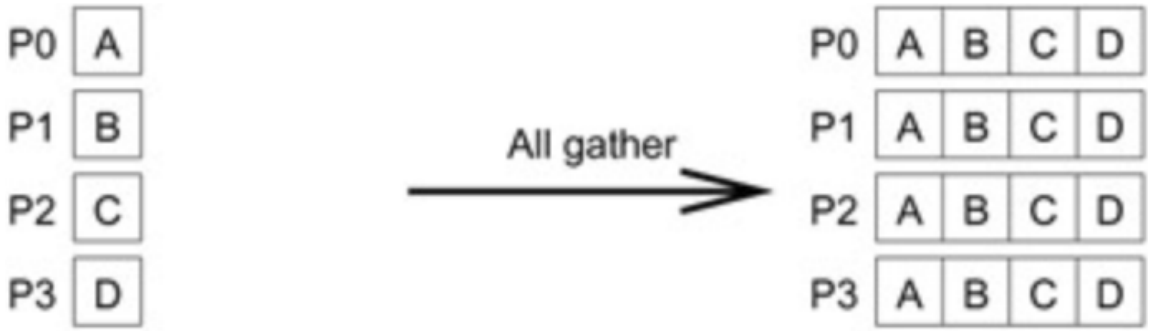
MPI_Allgather()

```
\begin{lstlisting}

  int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                    void *recvbuf, int recvcount, MPI_Datatype recvtype,
                    MPI_Comm comm)
  //sendbuf <=> data to send
  //sendcount <=> the number of data to send
  //sentype <=> type of data to send
  //recvbuf <=> address of receive buffer (out)
  //recvcount <=> number of elements in receive buffer
  //recvtype <=> data type of receive buffer elements
  //comm <=> the communicator
```

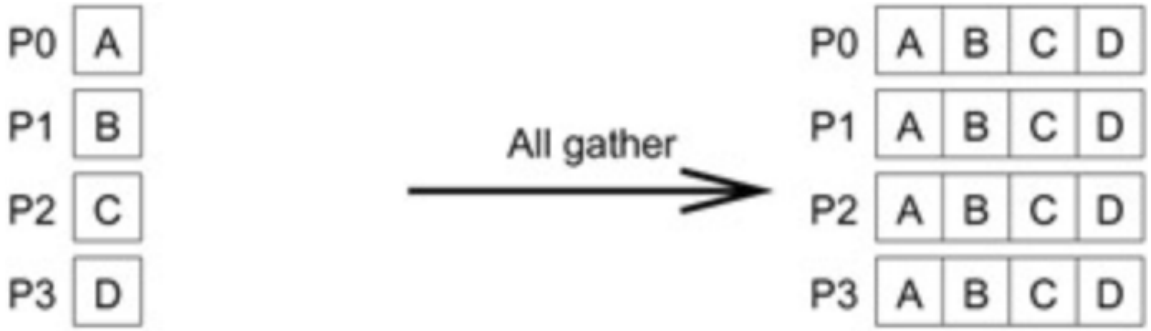
Definition :

Gathers data from all tasks and distribute the combined data to all tasks.



MPI_Allgather()

```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size(comm,&gsize);
rbuf=(int*)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray,100,MPI_INT,rbuf,100,MPI_INT,comm);
```

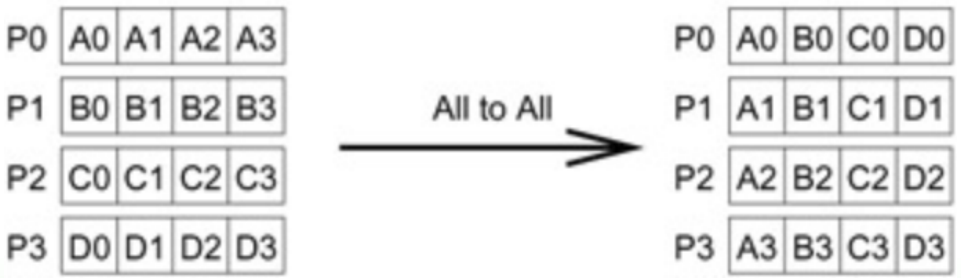


MPI_Alltoall()

```
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
          void *recvbuf, int recvcount, MPI_Datatype recvtype,MPI_Comm comm)
//sendbuf <=> data to send
//sendcount <=> the number of data to send
//sentype <=> type of data to send
//recvbuf <=> address of receive buffer (out)
//recvcount <=> number of elements in receive buffer
//recvtype <=> data type of receive buffer elements
//comm <=> the communicator
```

Definition :

`MPI_Alltoall()` works as combined `MPI_Scatter()` and `MPI_Gather()`. The send buffer in each process is split like in `MPI_Scatter()` and then each column of chunks is gathered by the respective process, whose rank matches the number of the chunk column. `MPI_Alltoall()` can also be seen as a global transposition operation, acting on chunks of data.



MPI_Alltoall()

```
sb = (int *)malloc(size*chunk*sizeof(int));
rb = (int *)malloc(size*chunk*sizeof(int));

for (int i=0 ;i < size*chunk;++i){
  sb[i] = rank + 1;
  rb[i] = 0;
}
status = MPI_Alltoall(sb, chunk, MPI_INT, rb, chunk, MPI_INT, MPI_COMM_WORLD);
```

Hybrid Programming

TODO