INSTITUT DU
DÉVELOPPEMENT ET DES
RESSOURCES EN
INFORMATIQUE
SCIENTIFIQUE

MPI

Dimitri Lecas - Rémi Lacroix - Serge Van Criekingen - Myriam Peyrounette

CNRS — IDRIS

v5.2 September 22th 2022

Plan I

Plan II

Plan III

Plan IV

Introduction

Introduction

Availability and updating

This document is likely to be updated regularly. The most recent version is available on the Web server of IDRIS : http://www.idris.fr/formations/mpi/

- IDRIS
  Institut for Development and Resources in Intensive Scientific Computing
  Rue John Von Neumann
  Bâtiment 506
  BP 167
  91403 ORSAY CEDEX
  France
  http://www.idris.fr
- Translated with the help of Cynthia TAUPIN.

Introduction

Parallelism

The goal of parallel programming is to :

- Reduce elapsed time.
- Do larger computations.
- Exploit parallelism of modern processor architectures (multicore, multithreading).

For group work, coordination is required. MPI is a library which allows process coordination by using a message-passing paradigm.

Introduction

Sequential progamming model

- The program is executed by one and only one process.
- All the variables and constants of the program are allocated in the memory of the process.
- A process is executed on a physical processor of the machine.

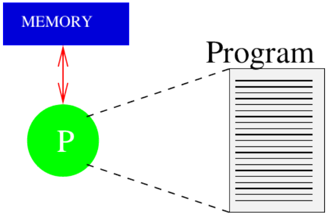


Figure 1 – Sequential programming model

Introduction

Message passing programming model

- The program is written in a classic language (Fortran, C, C++, etc.).
- All the program variables are private and reside in the local memory of each process.
- Each process has the possibility of executing different parts of a program.
- A variable is exchanged between two or several processes via a programmed call to specific subroutines.

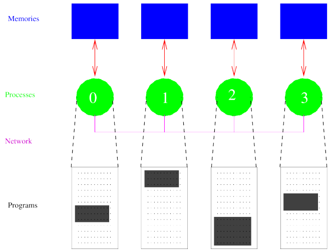


Figure 2 – Message Passing Programming Model

Introduction

Message Passing concepts

If a message is sent to a process, the process must receive it.

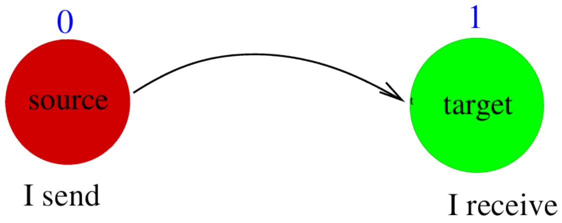


Figure 3 – Message Passing

Introduction

Message content

- A message consists of data chunks passing from the sending process to the receiving process/pocesses.
- In addition to the data (scalar variables, arrays, etc.) to be sent, a message must contain the following information :
    - The identifier of the sending process
    - The datatype
    - The length
    - The identifier of the receiving process

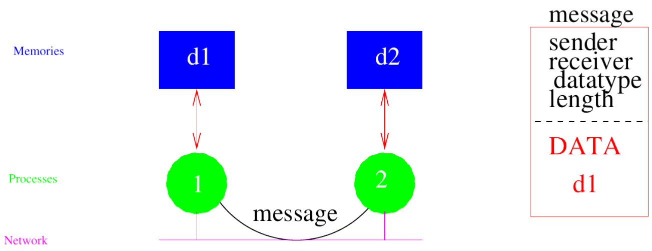


Figure 4 – Message Construction

Introduction

Environment

- The exchanged messages are interpreted and managed by an environment comparable to telephony, e-mail, postal mail, etc.
- The message is sent to a specified address.
- The receiving process must be able to classify and interpret the messages which are sent to it.
- The environment in question is MPI (Message Passing Interface). An MPI application is a group of autonomous processes, each executing its own code and communicating via calls to MPI library subroutines.

Introduction

Supercomputer architecture

Most supercomputers are distributed-memory computers. They are made up of many nodes and memory is shared within each node.

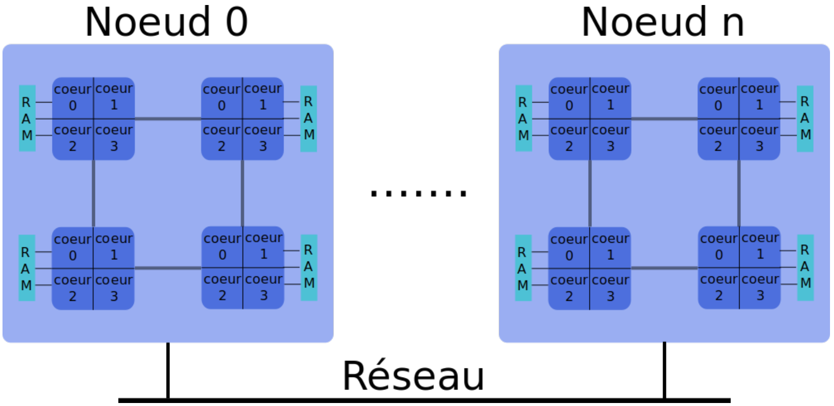


Figure 5 – Supercomputor architecture

Introduction

Jean Zay

- 2 140 nodes
- 2 Intel Cascade Lake processor (20 cores), 2,5 Ghz by node
- 4 GPU Nvidia V100 by node (on 612 nodes)
- 85 600 cores
- 410 TB (192 GB by node)
- 26 Pflop/s peak
- 15,6 Pflop/s (linpack)

Introduction

MPI vs OpenMP

OpenMP uses a shared memory paradigm, while MPI uses a distributed memory paradigm.

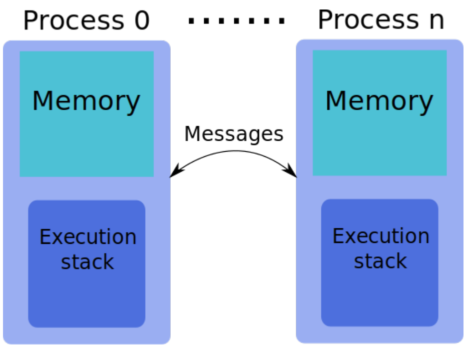


Figure 6 – MPI scheme

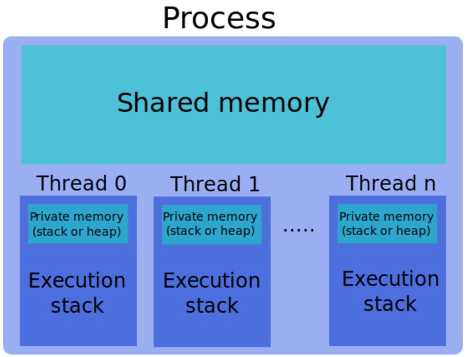


Figure 7 – OpenMP scheme

Introduction

Domain decomposition

A schema that we often see with MPI is domain decomposition. Each process controls a part of the global domain and mainly communicates with its neighbouring processes.

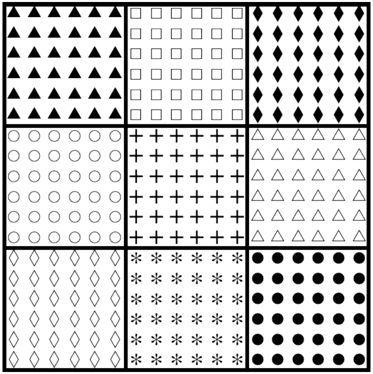


Figure 8 – Decomposition in subdomains

Introduction

History

- Version 1.0 : June 1994, the MPI (Message Passing Interface) Forum, with the participation of about forty organisations, developed the definition of a set of subroutines concerning the `MPI` library.
- Version 1.1 : June 1995, only minor changes.
- Version 1.2 : 1997, minor changes for more consistency in the names of some subroutines.
- Version 1.3 : September 2008, with clarifications of the MPI 1.2 version which are consistent with clarifications made by MPI-2.1.
- Version 2.0 : Released in July 1997, important additions which were intentionally not included in MPI 1.0 (process dynamic management, one-sided communications, parallel I/O, etc.).
- Version 2.1 : June 2008, with clarifications of the MPI 2.0 version but without any changes.
- Version 2.2 : September 2009, with only "small" additions.

Introduction

MPI 3.0

- Version 3.0 : September 2012 Changes and important additions compared to version 2.2 ;
  - Nonblocking collective communications
  - Revised implementation of one-sided communications
  - Fortran (2003-2008) bindings
  - C++ bindings removed
  - Interfacing of external tools (for debugging and performance measurements)
  - etc.
- Version 3.1 : June 2015
  - Correction to the Fortran (2003-2008) bindings ;
  - New nonblocking collective I/O routines ;

MPI 4.0

Version 4.0 : June 2021

- Large count
- Partitioned communication
- MPI Session

Introduction

Library

- Message Passing Interface Forum. *MPI : A Message-Passing Interface Standard, Version 3.1*. High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, 2015. https://fs.hlrs.de/projects/par/mpi/mpi31/
- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI, third edition Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 2014.
- William Gropp, Torsten Hoefler, Rajeev Thakur and Erwing Lusk : *Using Advanced MPI Modern Features of the Message-Passing Interface*, MIT Press, 2014.
- Additional references : http://www.mpi-forum.org/docs/ http://www.mcs.anl.gov/research/projects/mpi/learning.html

Introduction

Open source MPI implementations

These can be installed on a large number of architectures but their performance results are generally inferior to the implementations of the constructors.

- MPICH : `http://www.mpich.org`
- Open MPI : `http://www.open-mpi.org`

Introduction

Tools

- Debuggers
  - Totalview
    https://totalview.io
  - DDT
    https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt
- Performance measurement
  - FPMPI : *FPMPI*
    http://www.mcs.anl.gov/research/projects/fpmpi/WWW/
  - Scalasca : *Scalable Performance Analysis of Large-Scale Applications*
    http://www.scalasca.org/
  - MUST : *MPI Runtime Correctness Analysis*
    https://itc.rwth-aachen.de/must/

Introduction

Open source parallel scientific libraries

- ScaLAPACK : Linear algebra problem solvers using direct methods.
  `http://www.netlib.org/scalapack/`
- PETSc : Linear and non-linear algebra problem solvers using iterative methods.
  `https://www.mcs.anl.gov/petsc/`
- PaStiX : Parallel sparse direct Solvers.
  `http://pastix.gforge.inria.fr/files/README-txt.html`
- FFTW : Fast Fourier Transform.
  `http://www.fftw.org`

Environment

Environment

Description

- Every program unit calling MPI routines has to include the header file `mpi.h`.
- The `MPI_Init()` subroutine initializes the MPI environment :

```
int MPI_Init(int *argc, char **arv)
```

- The `MPI_Finalize()` subroutine disables this environment :

```
int MPI_Finalize(void)
```

Environment

Communicators

- All the MPI operations occur in a defined set of processes, called communicator. The default communicator is `MPI_COMM_WORLD`, which includes all the active processes.

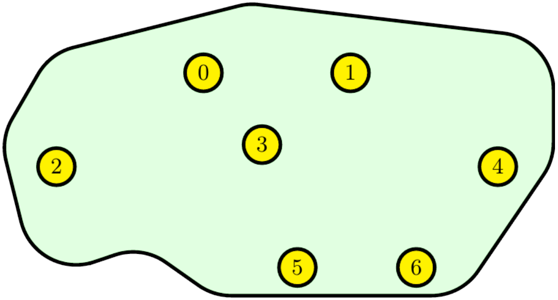


Figure 9 – MPI_COMM_WORLD Communicator

Environment

Termination of a program

Sometimes, a program encounters some issue during its execution and has to stop prematurely. For example, we want the execution to stop if one of the processes cannot allocate the memory needed for its calculation. In this case, we call the `MPI_Abort()` subroutine instead of the Fortran instruction *stop* (Or *exit* in C).

```
int MPI_Abort(MPI_Comm comm, int error)
```

- `comm` : the communicator of which all the processes will be stopped ; it is advised to use `MPI_COMM_WORLD` in general ;
- `error` : the error number returned to the UNIX environment.

Code

It is not necessary to check the `code` value (return value in C) after calling MPI routines. By default, when MPI encounters a problem, the program is automatically stopped as in an implicit call to `MPI_Abort()` subroutine.

Environment

Rank and size

- At any moment, we have access to the number of processes managed by a given communicator by calling the `MPI_Comm_size()` subroutine :

```
int MPI_Comm_size(MPI_Comm comm,int *nb_procs)
```

- Similarly, the `MPI_Comm_rank()` subroutine allows us to obtain the rank of an active process (i.e. its instance number, between 0 and `MPI_Comm_size()` – 1) :

```
int MPI_Comm_rank(MPI_Comm comm,int *rank)
```

Environment

Example

```c
/* who_am_i */
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  int nb_procs,rank;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  printf("I am the process %d among %d\n",rank,nb_procs);

  MPI_Finalize();
}
```

```
> mpiexec -n 7 who_am_I

I am process 3 among 7
I am process 0 among 7
I am process 4 among 7
I am process 1 among 7
I am process 5 among 7
I am process 2 among 7
I am process 6 among 7
```

Environnement

Compilation and execution of an MPI code

- To compile an MPI code, we use a compiler *wrapper*, which makes the link with the chosen MPI library.
- This *wrapper* is different depending on the programming language, the compiler and the MPI library. For example : `mpif90`, `mpifort`, `mpicc`, ...

```
> mpif90 <options> -c source.f90
> mpif90 source.o -o my_executable_file
```

- To execute an MPI code, we use an MPI launcher, which runs the execution on a given number of processes.
- The `mpiexec` launcher is defined by the MPI standard. There are also non-standard launchers, such as `mpirun`.

```
> mpiexec -n <number of processes> my_executable_file
```

MPI Hands-On – Exercise 1 : MPI Environment

- Write an MPI program in such a way that each process prints a message, which indicates whether its rank is odd or even. For example :

```
> mpiexec -n 4 ./even_odd
I am process 0, my rank is even
I am process 2, my rank is even
I am process 3, my rank is odd
I am process 1, my rank is odd
```

- To test whether the rank is odd or even, the Fortran intrinsic function corresponding to the *modulo* operation is `mod` :

```
mod(a,b)
```

  (use `%` symbol in C : `a%b`)

- To compile your program, use the command `make`
- To execute your program, use the command `make exe`

- For the program to be recognized by the Makefile, it must be named `even_odd.f90` (or `even_odd.c`)

Point-to-point Communications

Point-to-point Communications

General Concepts

A point-to-point communication occurs between two processes : the sender process and the receiver process.

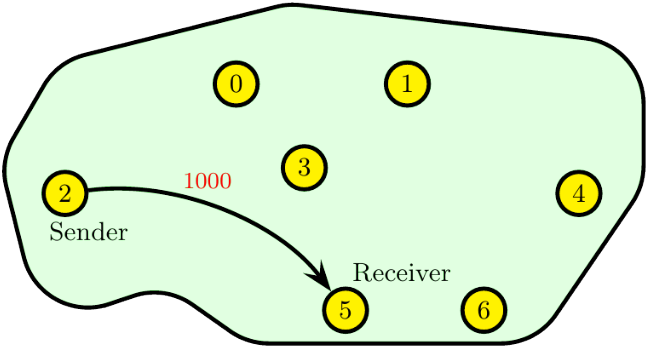


Figure 10 – Point-to-point communication

Point-to-point Communications

General Concepts

- The sender and the receiver are identified by their ranks in the communicator.
- The object communicated from one process to another is called message.
- A message is defined by its envelope, which is composed of :
  - the rank of the sender process
  - the rank of the receiver process
  - the message tag
  - the communicator in which the transfer occurs
- The exchanged data has a datatype (integer, real, etc, or individual derived datatypes).
- There are several transfer modes, which use different protocols.

Point-to-point Communications

Blocking Send MPI_Send

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

Sending, from the address buf, a message of count elements of type datatype, tagged tag, to the process of rank dest in the communicator comm.

Remark :
This call is blocking : the execution remains blocked until the message can be re-written without risk of overwriting the value to be sent. In other words, the execution is blocked as long as the message has not been received.

Point-to-point Communications

Blocking Receive MPI_Recv

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

Receiving, at the address buf, a message of count elements of type datatype, tagged tag, from the process of rank source in the communicator comm.

Remarks :

- status_msg stores the state of a receive operation : source, tag, code, ...
- An `MPI_Recv` can only be associated to an `MPI_Send` if these two calls have the same envelope (source, dest, tag, comm).
- This call is blocking : the execution remains blocked until the message content corresponds to the received message.

Point-to-point Communications

Example (see Fig. 10)

```c
/* point_to_point */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,value;
  int tag=100;
  MPI_Status statut_msg;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank == 2) {
    value = 1000;
    MPI_Send(&value,1,MPI_INT,5,tag,MPI_COMM_WORLD);
  } else if ( rang == 5){
    MPI_Recv(&value,1,MPI_INT,2,tag,MPI_COMM_WORLD,&statut_msg);
    printf("I, process 5, I received %d from the process 2.\n", value);
  }

  MPI_Finalize();
}
```

```
> mpiexec -n 7 point_to_point

I, process 5, I received 1000 from the process 2
```

Point-to-point Communications

C MPI Datatypes

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

Point-to-point Communications

Other possibilities

- When receiving a message, the rank of the sender process and the tag can be replaced by « *jokers* » : `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively.
- A communication involving the dummy process of rank `MPI_PROC_NULL` has no effect.
- `MPI_STATUS_IGNORE` is a predefined constant, which can be used instead of the status variable.
- It is possible to send more complex data structures by creating derived datatypes.
- There are other operations, which carry out both send and receive operations simultaneously : `MPI_Sendrecv()` and `MPI_Sendrecv_replace()`.

Point-to-point Communications

Simultaneous send and receive MPI_Sendrecv

```
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_msg)
```

- Sending, from the address sendbuf, a message of sendcount elements of type sendtype, tagged sendtag, to the process dest in the communicator comm ;
- Receiving, at the address recvbuf, a message of recvcount elements of type recvtype, tagged recvtag, from the process source in the communicator comm.

Remark :

- Here, the receiving zone recvbuf must be different from the sending zone sendbuf.

Point-to-point Communications

Simultaneous send and receive MPI_Sendrecv

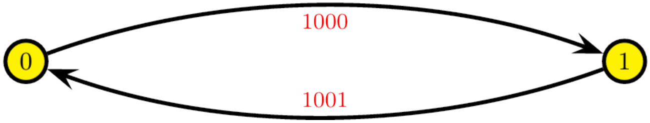


Figure 11 – `sendrecv` Communication between the Processes 0 and 1

Point-to-point Communications

Example (see Fig. 11)

```c
/* sendrecv */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,value,num_proc,message;
  int tag=110;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rang);

  num_proc=(rank+1)%2;
  message = rank+1000;
  MPI_Sendrecv(&message,1,MPI_INT,num_proc,tag,&value,1,MPI_INT,
               num_proc,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

  printf("I, process %d, I received %d from process %d.\n", rank,value,num_proc);

  MPI_Finalize();
}
```

```
> mpiexec -n 2 sendrecv

I, process 1, I received 1000 from process 0
I, process 0, I received 1001 from process 1
```

Point-to-point Communications

Be careful !

In the case of a synchronous implementation of the `MPI_Send()` subroutine, if we replace the `MPI_Sendrecv()` subroutine in the example above by `MPI_Send()` followed by `MPI_Recv()`, the code will deadlock. Indeed, each of the two processes will wait for a receipt confirmation, which will never come because the two sending operations would stay suspended.

```
val = rank+1000;
MPI_Send(&val,1,MPI_INT,num_proc,tag,MPI_COMM_WORLD);
MPI_Recv(value,1,MPI_INT,num_proc,tag,MPI_COMM_WORLD,&statut);
```

Point-to-point Communications

Simultaneous send and receive MPI_Sendrecv_replace

```
int MPI_Sendrecv_replace(void * buf,int count, MPI_Datatype datatype,
                         int dest, int sendtag,
                         int source, int recvtag, MPI_Comm comm,
                         MPI_Status  *statut_msg)
```

- Sending, from the address buf, a message of count elements of type datatype, tagged sendtag, to the process dest in the communicator comm ;
- Receiving a message at the same address, with same count elements and same datatype, tagged recvtag, from the process source in the communicator comm.

Remark :

- Contrary to the usage of MPI_Sendrecv, the receiving zone is the same here as the sending zone buf.

Point-to-point Communications

Example

```c
/* wildcard */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int nb_procs,rank;
  int m=4,tag=11;
  int A[m][m];
  MPI_Status statut_msg;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank == 0) {
    A[0][0]=1;A[0][1]=2;A[0][2]=3;A[0][3]=4;A[1][0]=5;A[1][1]=6;A[1][2]=7;
    A[1][3]=8;A[2][0]=9;A[2][1]=10;A[2][2]=11;A[2][3]=12;A[3][0]=13;
    A[3][1]=14;A[3][2]=15;A[3][3]=16;
    MPI_Send(A,3,MPI_INT,1,tag,MPI_COMM_WORLD);
  } else {
    MPI_Recv(&(A[0][1]),3,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
             MPI_COMM_WORLD,&statut_msg);
    printf("I process %d, I received 3 elements from the process %d with tag"
           "%d the elements are %d %d %d.\n",
            rank,statut_msg.MPI_SOURCE,statut_msg.MPI_TAG, A[0][1],A[0][2],A[0][3]);
  }
  MPI_Finalize();
}
```

Point-to-point Communications

```
> mpiexec -n 2 wildcard
I, process         1, I have received 3 elements from the process         0
 with tag         11 the elements are         1           2           3
```

MPI Hands-On – Exercise 2 : Ping-pong

- Point to point communications : *Ping-Pong* between two processes

- This exercice is composed of 3 steps :

  1. *Ping* : complete the script `ping_pong_1.c` in such a way that the process 0 sends a message containing 1000 random reals to process 1.

  2. *Ping-Pong* : complete the script `ping_pong_2.c` in such a way that the process 1 sends back the message to the process 0, and measure the communication duration with the `MPI_Wtime()` function.

  3. *Ping-Pong match* : complete the script `ping_pong_3.c` in such a way that processes 0 and 1 perform 9 *Ping-Pong*, while varying the message size, and measure the communication duration each time. The corresponding bandwidths will be printed.

MPI Hands-On – Exercise 2 : Ping-pong

Remarks :

- To compil the first step : `make ping_pong_1`
- To execute the first step : `make exe1`
- To compil the second step : `make ping_pong_2`
- To execute the second step : `make exe2`
- To compil the last step : `make ping_pong_3`
- To execute the last step : `make exe3`

- The generation of random numbers uniformly distributed in the range [0,1[is made by calling the C `rand` subroutine :

```
rand() / (RAND_MAX+1.);
```

- The time duration measurements can be done like this :

```
time_begin=MPI_Wtime();
..................................................
time_end=MPI_Wtime();
printf("... en %f secondes.\n",temps_fin-temps_debut);
```

Collective communications

Collective communications

General concepts

- Collective communications allow making a series of point-to-point communications in one single call.
- A collective communication always concerns all the processes of the indicated communicator.
- For each process, the call ends when its participation in the collective call is completed, in the sense of point-to-point communications (therefore, when the concerned memory area can be changed).
- The management of tags in these communications is transparent and system-dependent. Therefore, they are never explicitly defined during calls to subroutines. An advantage of this is that collective communications never interfere with point-to-point communications.

Collective communications

Types of collective communications

There are three types of subroutines :

1. One which ensures global synchronizations : `MPI_Barrier()`
2. Ones which only transfer data :
   - Global distribution of data : `MPI_Bcast()`
   - Selective distribution of data : `MPI_Scatter()`
   - Collection of distributed data : `MPI_Gather()`
   - Collection of distributed data by all the processes : `MPI_Allgather()`
   - Collection and selective distribution by all the processes of distributed data : `MPI_Alltoall()`
3. Ones which, in addition to the communications management, carry out operations on the transferred data :
   - Reduction operations (sum, product, maximum, minimum, etc.), whether of a predefined or personal type : `MPI_Reduce()`
   - Reduction operations with distributing of the result (this is in fact equivalent to an `MPI_Reduce()` followed by an `MPI_Bcast()`) : `MPI_Allreduce()`

Collective communications

Global synchronization : MPI_Barrier()

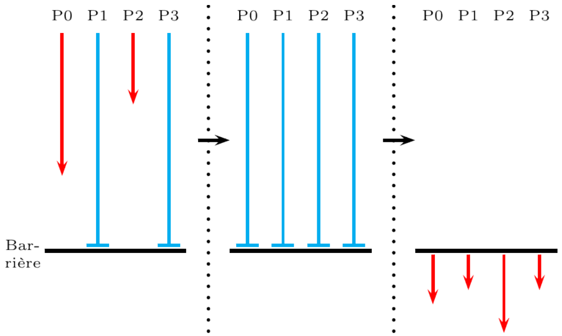


Figure 12 – Global Synchronization : MPI_Barrier()

```
int MPI_Barrier(MPI_Comm MPI_COMM_WORLD)
```

Collective communications

Global distribution : `MPI_Bcast()`

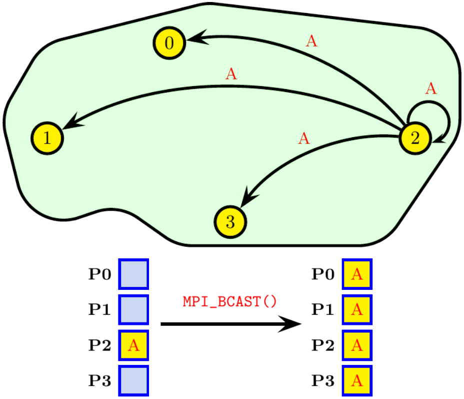


Figure 13 – Global distribution : `MPI_Bcast()`

Collective communications

Global distribution : MPI_Bcast()

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

1. Send, starting at position buffer, a message of count element of type datatype, by the root process, to all the members of communicator comm.
2. Receive this message at position buffer for all the processes other than the root.

Collective communications

Example of MPI_Bcast()

```c
/* bcast */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,value;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank == 2) value = rank+1000;

  MPI_Bcast(&value,1,MPI_INT,2,MPI_COMM_WORLD);

  printf("I, process %d, received %d of process 2\n",
          rank,value);

  MPI_Finalize();
}
```

```
> mpiexec -n 4 bcast

I, process 2, received 1002 of process 2
I, process 0, received 1002 of process 2
I, process 1, received 1002 of process 2
I, process 3, received 1002 of process 2
```

Collective communications

Selective distribution : `MPI_Scatter()`

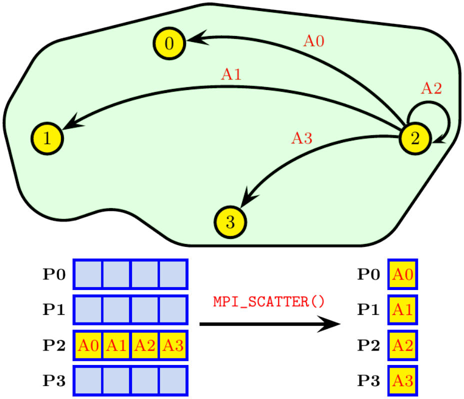


Figure 14 – Selected distribution : `MPI_Scatter()`

Collective communications

Selective distribution : 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)
```

1. Scatter by process root, starting at position sendbuf, message sendcount element of type sendtype, to all the processes of communicator comm.
2. Receive this message at position recvbuf, of recvcount element of type recvtype for all processes of communicator comm.

Remarks :

- The couples (sendcount, sendtype) and (recvcount, recvtype) must represent the same quantity of data.
- Data are scattered in chunks of same size ; a chunk consists of sendcount elements of type sendtype.
- The i-th chunk is sent to the i-th process.

Collective communications

Example of MPI_Scatter()

```c
/* scatter */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
  int nb_values=8,rank,nb_procs,block_length,i;
  float *values, *recvdata;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  block_length = nb_values/nb_procs;
  recvdata = (float *) malloc(block_length*sizeof(float));
  if (rank == 2) {
    values = (float *) malloc(nb_values*sizeof(float));
    for (i=0; i<nb_values;i++) values[i]=1000.+i;
    printf("I, process %d send my values array : ",rank);
    for (i=0; i<nb_values;i++) {printf("%f ",values[i]);} printf("\n"); }
  MPI_Scatter(values,block_length,MPI_FLOAT,
              recvdata,block_length,MPI_FLOAT,2,MPI_COMM_WORLD);
  printf("I, process %d, received ",rank);
  for (i=0;i<block_length;i++) printf("%f ", recvdata[i]);
  printf("of process 2\n");
  MPI_Finalize(); }
```

```
> mpiexec -n 4 scatter
I, process 2 send my values array :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.

I, process 0, received 1001. 1002. of processus 2
I, process 1, received 1003. 1004. of processus 2
I, process 3, received 1007. 1008. of processus 2
I, process 2, received 1005. 1006. of processus 2
```

Collective communications

Collection : `MPI_Gather()`

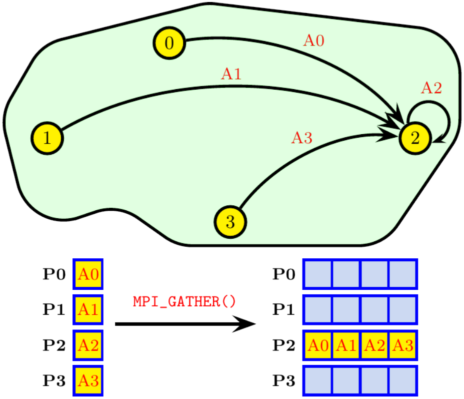


Figure 15 – Collection : `MPI_Gather()`

Collective communications

Collection : MPI_Gather()

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

1. Send for each process of communicator comm, a message starting at position sendbuf, of sendcount element type sendtype.
2. Collect all these messages by the root process at position recvbuf, recvcount element of type recvtype.

Remarks :

- The couples (sendcount, sendtype) and (recvcount, recvtype) must represent the same size of data.
- The data are collected in the order of the process ranks.

Collective communications

Collection : MPI_Gather()

```c
/* Gather */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
  int nb_values=8,rank,nb_procs,block_length,i;
  float recvdata[nb_values],*values;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  block_length = nb_values/nb_procs;
  values = (float *) malloc(block_length*sizeof(float));
  for (i=0;i<block_length;i++) values[i]=1000.+rank*block_length+i;
  printf("I, process %d sent my values array : ",rank);
  for (i=0; i<block_length;i++) {printf("%f ",values[i]);} printf("\n");
  MPI_Gather(values,block_length,MPI_FLOAT,
             recvdata,block_length,MPI_FLOAT,2,MPI_COMM_WORLD);
  if (rank==2) {
    printf("I, process %d, received ",rang);
    for (i=0;i<nb_values;i++) { printf("%f ", recvdata[i]); } printf("\n"); }
  MPI_Finalize(); }
```

```
> mpiexec -n 4 gather
I, process 1 sent my values array :1003. 1004.
I, process 0 sent my values array :1001. 1002.
I, process 2 sent my values array :1005. 1006.
I, process 3 sent my values array :1007. 1008.

I, process 2, received 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
```

Collective communications

Gather-to-all : `MPI_Allgather()`

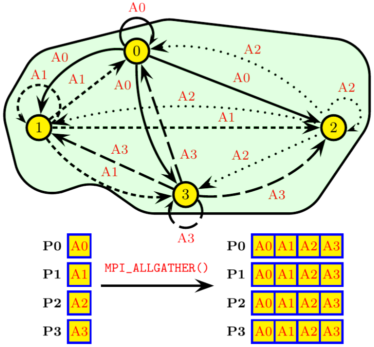


Figure 16 – Gather-to-all : `MPI_Allgather()`

Collective communications

Gather-to-all : MPI_Allgather()

Corresponds to an MPI_Gather() followed by an MPI_Bcast() :

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

1. Send by each process of communicator comm, a message starting at position sendbuf, of sendcount element, type sendtype.
2. Collect all these messages, by all the processes, at position recvbuf of recvcount element type recvtype.

Remarks :

- The couples (sendcount, sendtype) and (recvcount, recvtype) must represent the same data size.
- The data are gathered in the order of the process ranks.

Collective communications

Example of `MPI_Allgather()`

```c
/* allgather */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
  int nb_values=8,rank,nb_procs,block_length,i;
  float recvdata[nb_values],*values;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  block_length = nb_values/nb_procs;
  values = (float *) malloc(block_length*sizeof(float));
  for (i=0;i<block_length;i++) values[i]=1000.+rank*block_length+i;
  MPI_Allgather(values,block_length,MPI_FLOAT,
                recvdata,block_length,MPI_FLOAT,MPI_COMM_WORLD);
  printf("I, process %d, received ",rank);
  for (i=0;i<nb_values;i++) {printf("%f ", recvdata[i]);} printf("\n");
  MPI_Finalize(); }
```

```
> mpiexec -n 4 allgather

I, process 1, received 1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.
I, process 3, received 1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.
I, process 2, received 1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.
I, process 0, received 1001. 1002.  1003. 1004.  1005. 1006.  1007. 1008.
```

Collective communications

Extended gather : `MPI_Gatherv()`

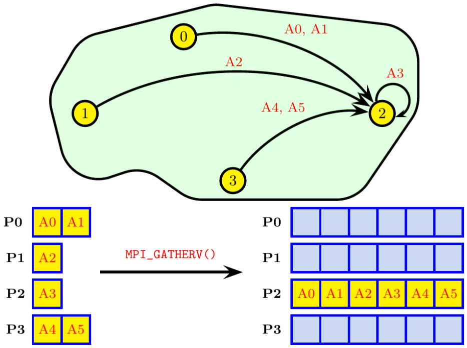


Figure 17 – Extended gather : `MPI_Gatherv()`

Collective communications

Extended Gather : MPI_Gatherv()

This is an MPI_Gather() where the size of messages can be different among processes :

```
int MPI_Gatherv(const void *sendbuf,int sendcount,MPI_Datatype sendtype,
                void *recvbuf,const int *recvcounts,const int *displs,
                MPI_Datatype recvtype,root,MPI_Comm comm)
```

The i-th process of the communicator comm sends to process root, a message starting at position sendbuf, of sendcount element of type sendtype, and receives at position recvbuf, of recvcounts(i) element of type recvtype, with a displacement of displs(i).

Remarks :

- The couples (sendcount,sendtype) of the i-th process and (recvcounts(i), recvtype) of process root must be such that the data size sent and received is the same.

Collective communications

Example of `MPI_Gatherv()`

```c
/* gatherv */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int nb_values=8,rank,nb_procs,block_length,remainder,i;
  float recvdata[nb_values];
  float *values;
  int *nb_elements_received,*displacement;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  block_length = nb_values/nb_procs;
  remainder = nb_values%nb_procs;
  if (rank < remainder) block_length = block_length+1;
  values = (float *) malloc(block_length*sizeof(float));
  for (i=0;i<block_length;i++)
   values[i]=1000.+rank*block_length+(rank<remainder?rank:remainder)+i;
  printf("I, process %d send my values array : ",rank);
  for (i=0; i<block_length;i++) {printf("%f ",values[i]); }printf("\n");
  if (rank == 2) {
    nb_elements_received = (int *) malloc(nb_procs*sizeof(int));
    displacement = (int *) malloc(nb_procs*sizeof(int));
    nb_elements_received[0] = nb_values/nb_procs;
    if (remainder > 0) nb_elements_received[0] = nb_elements_received[0]+1;
    displacement[0] = 0;
    for (i=1;i<nb_procs;i++) {
      displacement[i] = displacement[i-1]+nb_elements_received[i-1];
      nb_elements_received[i] = nb_values/nb_procs;
      if (i < remainder) nb_elements_received[i] = nb_elements_received[i]+1;
  } }
```

Collective communications

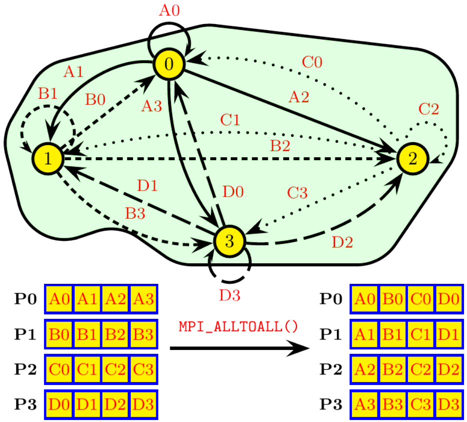
Example of MPI_Gatherv()

```
  MPI_Gatherv(values,block_length,MPI_FLOAT,
              recvdata,nb_elements_received,displacement,MPI_FLOAT,2,MPI_COMM_WORLD);
  if (rank==2) {
    printf("I, process %d, received ",rang);
    for (i=0;i<nb_values;i++) printf("%f ", recvdata[i]);
    printf("\n"); }
  MPI_Finalize();
}
```

```
> mpiexec -n 4 gatherv

 I, process  0 sent my values array :  1001. 1002. 1003.
 I, process  2 sent my values array :  1007. 1008.
 I, process  3 sent my values array :  1009. 1010.
 I, process  1 sent my values array :  1004. 1005. 1006.

 I, process 2 receives  1001. 1002. 1003.  1004. 1005. 1006.  1007. 1008.  1009. 1010.
```

Collective communications

Collection and distribution : `MPI_Alltoall()`



P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

P0	A0	B0	C0	D0
P1	A1	B1	C1	D1
P2	A2	B2	C2	D2
P3	A3	B3	C3	D3

Figure 18 – Collection and distribution : : `MPI_Alltoall()`

Collective communications

Collection and distribution : MPI_Alltoall()

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

Here, the i-th process sends its j-th chunk to the j-th process which places it in its i-th chunk.

Remark :

- The couples (sendcount, sendtype) and (recvcount, recvtype) must be such that they represent equal data sizes.

Collective communications

Example of MPI_Alltoall()

```
/* alltoall */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int nb_values=8;
  int rank,nb_procs,block_length,i;
  float recvdata[nb_values],values[nb_values];


  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  for (i=0;i<nb_values;i++) values[i]=1000.+rank*nb_values+i;
  block_length = nb_values/nb_procs;

  printf("I, process %d sent my values array : ",rank);
  for (i=0; i<nb_values;i++) printf("%f ",values[i]);
  printf("\n");

  MPI_Alltoall(values,block_length,MPI_FLOAT,
               recvdata,block_length,MPI_FLOAT,MPI_COMM_WORLD);

  printf("I, process %d, received ",rank);
  for (i=0;i<nb_values;i++) printf("%f ", recvdata[i]);
  printf("\n");
  MPI_Finalize();
}
```

Collective communications

Example of MPI_Alltoall()

```
> mpiexec -n 4 alltoall
I, process 1 sent my values array :
1009. 1010. 1011. 1012. 1013. 1014. 1015. 1016.
I, processus 0 sent my values array :
1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
I, processus 2 sent my values array :
1017. 1018. 1019. 1020. 1021. 1022. 1023. 1024.
I, processus 3 sent my values array :
1025. 1026. 1027. 1028. 1029. 1030. 1031. 1032.

I, process 0, received 1001. 1002.  1009. 1010.  1017. 1018.  1025. 1026.
I, process 2, received 1005. 1006.  1013. 1014.  1021. 1022.  1029. 1030.
I, process 1, received 1003. 1004.  1011. 1012.  1019. 1020.  1027. 1028.
I, process 3, received 1007. 1008.  1015. 1016.  1023. 1024.  1031. 1032.
```

Collective communications

Global reduction

- A reduction is an operation applied to a set of elements in order to obtain one single value. Typical examples are the sum of the elements of a vector (`SUM(A(:))`) or the search for the maximum value element in a vector (`MAX(V(:))`).
- MPI proposes high-level subroutines in order to operate reductions on data distributed on a group of processes. The result is obtained on only one process (`MPI_Reduce()`) or on all the processes (`MPI_Allreduce()`, which is in fact equivalent to an `MPI_Reduce()` followed by an `MPI_Bcast()`).
- If several elements are implied by process, the reduction function is applied to each one of them (for instance to each element of a vector).

Collective communications

Distributed reduction : MPI_Reduce

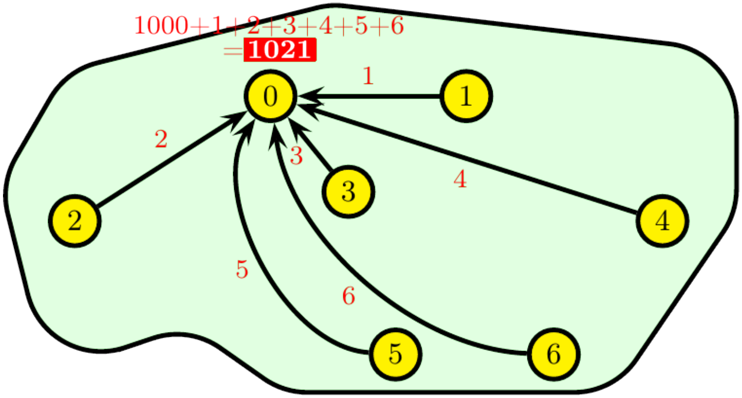


Figure 19 – Distributed reduction (sum)

Collective communications

Operations

Name	Operation
`MPI_SUM`	Sum of elements
`MPI_PROD`	Product of elements
`MPI_MAX`	Maximum of elements
`MPI_MIN`	Minimum of elements
`MPI_MAXLOC`	Maximum of elements and location
`MPI_MINLOC`	Minimum of elements and location
`MPI_LAND`	Logical AND
`MPI_LOR`	Logical OR
`MPI_LXOR`	Logical exclusive OR

Collective communications

Global reduction : MPI_Reduce()

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

1. Distributed reduction of count elements of type datatype, starting at position sendbuf, with the operation op from each process of the communicator comm,
2. Return the result at position recvbuf in the process root.

Collective communications

Example of MPI_Reduce()

```c
/* reduce */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,nb_procs,value,sum,i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank == 0)
    value = 1000;
  else
    value = rank;

  MPI_Reduce(&value,&sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

  if (rank == 0)
    printf("I, process 0, have the global sum value %d\n",sum);

  MPI_Finalize();
}
```

```
> mpiexec -n 7 reduce

I, process 0, have the global sum value 1021
```

Collective communications

Distributed reduction with distribution of the result : `MPI_Allreduce()`

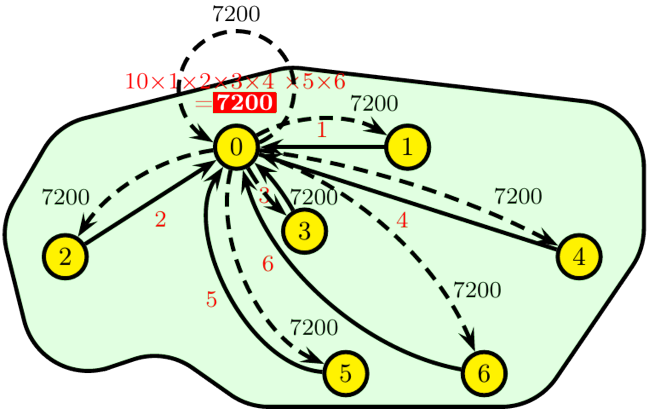


Figure 20 – Distributed reduction (product) with distribution of the result

Collective communications

Global all-reduction : MPI_Allreduce()

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

1. Distributed reduction of count elements of type datatype starting at position sendbuf, with the operation op from each process of the communicator comm,
2. Write the result at position recvbuf for all the processes of the communicator comm.

Collective communications

Example of MPI_Allreduce()

```c
/* allreduce */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,nb_procs,value,product,i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank == 0)
    value = 10;
  else
    value = rank;

  MPI_Allreduce(&value,&product,1,MPI_INT,MPI_PROD,MPI_COMM_WORLD);

  printf("I, process %d, received the value of the global product %d\n",
         rank,product);

  MPI_Finalize();
}
```

Collective communications

Example of MPI_Allreduce()

```
> mpiexec -n 7 allreduce

I, process 6, received the value of the global product 7200
I, process 2, received the value of the global product 7200
I, process 0, received the value of the global product 7200
I, process 4, received the value of the global product 7200
I, process 5, received the value of the global product 7200
I, process 3, received the value of the global product 7200
I, process 1, received the value of the global product 7200
```

Collective communications

Additions

- The `MPI_Scan()` subroutine allows making partial reductions by considering, for each process, the previous processes of the communicator and itself. `MPI_Exscan()` is the *exclusive* version of `MPI_Scan()`, which is *inclusive*.
- The `MPI_Op_create()` and `MPI_Op_free()` subroutines allow personal reduction operations.
- For each reduction operation, the keyword `MPI_IN_PLACE` can be used in order to keep the result in the same place as the sending buffer (but only for the rank(s) that will receive results). Example :
  `MPI_Allreduce(MPI_IN_PLACE,sendrecvbuf,...);`

Collective communications

Additions

- Similarly to what we have seen for `MPI_Gatherv()` with repect to `MPI_Gather()`, the `MPI_Scatterv()`, `MPI_Allgatherv()` and `MPI_Alltoallv()` subroutines extend `MPI_Scatter()`, `MPI_Allgather()` and `MPI_Alltoall()` to the cases where the processes have different numbers of elements to transmit or gather.
- `MPI_Alltoallw()` is the version of `MPI_Alltoallv()` which enables to deal with heterogeneous elements (by expressing the displacements in bytes and not in elements).

MPI Hands-On – Exercise 3 : Collective communications and reductions

- The aim of this exercice is to compute *pi* by numerical integration. $\pi = \int_0^1 \frac{4}{1+x^2} dx$.
- We use the rectangle method (mean point).
- Let $f(x) = \frac{4}{1+x^2}$ be the function to integrate.
- *nbblock* is the number of points of discretization.
- $width = \frac{1}{nbblock}$ the length of discretization and the width of all rectangles.
- Sequential version is available in the `pi.c` source file.
- You have to do the parallel version with MPI in this file.

Communication Modes

Communication Modes

Point-to-Point Send Modes

Mode	Blocking	Non-blocking
Standard send	`MPI_Send()`	`MPI_Isend()`
Synchronous send	`MPI_Ssend()`	`MPI_Issend()`
Buffered send	`MPI_Bsend()`	`MPI_Ibsend()`
Receive	`MPI_Recv()`	`MPI_Irecv()`

Communication Modes

Blocking call

- A call is blocking if the memory space used for the communication can be reused immediately after the exit of the call.
- The data sent can be modified after the call.
- The data received can be read after the call.

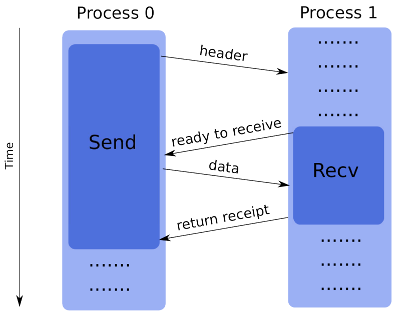
Communication Modes

Synchronous sends

A synchronous send involves a synchronization between the involved processes. A send cannot start until its receive is posted. There can be no communication before the two processes are ready to communicate.

Rendezvous Protocol

The rendezvous protocol is generally the protocol used for synchronous sends (implementation-dependent). The return receipt is optional.



Communication Modes

Interface of MPI_Ssend()

```
int MPI_Ssend(const void* values,int count, MPI_Datatype msgtype,
              int dest,int tag, MPI_Comm comm)
```

Advantages of synchronous mode

- Low resource consumption (no buffer)
- Rapid if the receiver is ready (no copying in a buffer)
- Knowledge of receipt through synchronization

Disadvantages of synchronous mode

- Waiting time if the receiver is not there/not ready
- Risk of deadlocks

Communication Modes

Deadlock example

In the following example, there is a deadlock because we are in synchronous mode. The two processes are blocked on the `MPI_Ssend()` call because they are waiting for the `MPI_Recv()` of the other process. However, the `MPI_Recv()` call can only be made after the unblocking of the `MPI_Ssend()` call.

```c
/* ssendrecv */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,num_proc,tmp,value;
  int tag=110;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  /* On suppose avoir exactement 2 processus */
  num_proc = (rank+1)%2;

  tmp = rank+1000;
  MPI_Ssend(&tmp,1,MPI_INT,num_proc,tag,MPI_COMM_WORLD);
  MPI_Recv(&value,1,MPI_INT,num_proc,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

  printf("I, process %d received %d from process %d\n",rank,value,num_proc);

  MPI_Finalize();
}
```

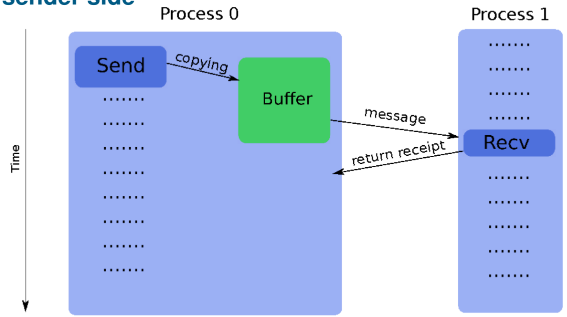
Communication Modes

Buffered sends

A buffered send implies the copying of data into an intermediate memory space. There is then no coupling between the two processes of communication. Therefore, the return of this type of send does not mean that the receive has occurred.

Protocol with user buffer on the sender side

In this approach, the buffer is on the sender side and is managed explicitly by the application. A buffer managed by MPI can exist on the receiver side. Many variants are possible. The return receipt is optional.



Communication Modes

Buffered sends

The buffers have to be managed manually (with calls to MPI_Buffer_attach() and MPI_Buffer_detach()). Message header size needs to be taken into account when allocating buffers (by adding the constant MPI_BSEND_OVERHEAD() for each message occurrence).

Interfaces

```
int MPI_Buffer_attach(void *buf, int typesize)
int MPI_Bsend(const void *values,int count, MPI_Datatype msgtype,
              int dest, int tag, MPI_Comm comm)
int MPI_Buffer_detach(void *buf, int typesize)
```

Communication Modes

Advantages of buffered mode

- No need to wait for the receiver (copying in a buffer)
- No risk of deadlocks

Disadvantages of buffered mode

- Uses more resources (memory use by buffers with saturation risk)
- The send buffers in the `MPI_Bsend()` or `MPI_Ibsend()` calls have to be managed manually (often difficult to choose a suitable size)
- Slightly slower than the synchronous sends if the receiver is ready
- No knowledge of receipt (send-receive decoupling)
- Risk of wasted memory space if buffers are too oversized
- Application crashes if buffer is too small
- There are often hidden buffers managed by the MPI implementation on the sender side and/or on the receiver side (and consuming memory resources)

Communication Modes

No deadlocks

In the following example, we don't have a deadlock because we are in buffered mode. After the copy is made in the *buffer*, the `MPI_Bsend()` call returns and then the `MPI_Recv()` call is made.

```c
/* bsendrecv */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,num_proc,tmp,value,bufsize,overhead,typesize;
  int tag=110,nb_elt=1,nb_msg=1;
  int * buffer;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Type_size(MPI_INT,&typesize);
  /* Convertir taille MPI_BSEND_OVERHEAD (octets) en nombre d'integer */
  overhead = (int) (1+(MPI_BSEND_OVERHEAD*1.)/typesize);
  buffer = (int *) malloc(nb_msg*(nb_elt+overhead)*sizeof(int));
  bufsize = typesize*nb_msg*(nb_elt+overhead);
  MPI_Buffer_attach(buffer,bufsize);
  /* On suppose avoir exactement 2 processus */
  num_proc = (rank+1)%2;
  tmp = rank+1000;
  MPI_Bsend(&tmp,nb_elt,MPI_INT,num_proc,tag,MPI_COMM_WORLD);
  MPI_Recv(&value,nb_elt,MPI_INT,num_proc,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  printf("I, process %d received %d from process %d\n",rank,value,num_proc);
  MPI_Buffer_detach(buffer,&bufsize);
  MPI_Finalize(); }
```

Communication Modes

Standard sends

A standard send is made by calling the MPI_Send() subroutine. In most implementations, the mode is buffered (*eager*) for small messages but is synchronous for larger messages.

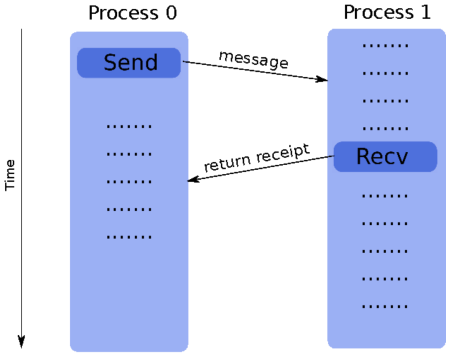
Interfaces

```
int MPI_Send(const void *values, int count, MPI_Datatype msgtype,
             int dest, int tag, MPI_Comm comm)
```

Communication Modes

The eager protocol

The eager protocol is often used for standard sends of small-size messages. It can also be used for sends with `MPI_Bsend()` for small messages (implementation-dependent) and by bypassing the user buffer on the sender side. In this approach, the buffer is on the receiver side. The return receipt is optional.



Communication Modes

Advantages of standard mode

- Often the most efficient (because the constructor chose the best parameters and algorithms)

Disadvantages of standard mode

- Little control over the mode actually used (often accessible via environment variables)
- Risk of deadlocks depending on the mode used
- Behavior can vary according to the architecture and problem size

Communication Modes

Presentation

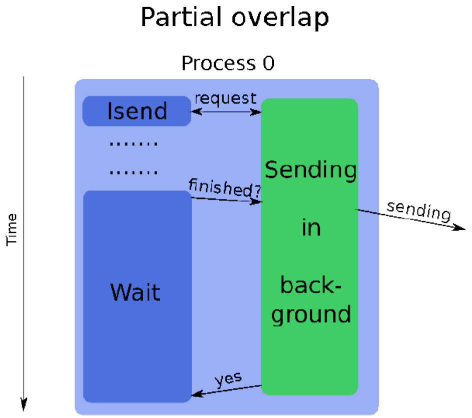
The overlap of communications by computations is a method which allows executing communications operations in the background while the program continues to operate. On Jean Zay, the latency of a communication internode is 1.5 μs, or 2500 processor cycles.

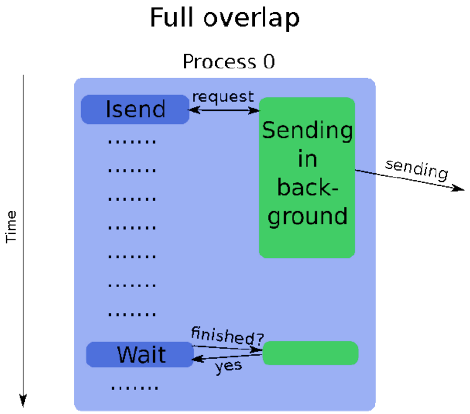
- It is thus possible, if the hardware and software architecture allows it, to hide all or part of communications costs.
- The computation-communication overlap can be seen as an additional level of parallelism.
- This approach is used in MPI by using nonblocking subroutines (i.e. `MPI_Isend()`, `MPI_Irecv()` and `MPI_Wait()`).

Non blocking communication

A nonblocking call returns very quickly but it does not authorize the immediate re-use of the memory space which was used in the communication. It is necessary to make sure that the communication is fully completed (with `MPI_Wait()`, for example) before using it again.

Communication Modes





Communication Modes

Advantages of non blocking call

- Possibility of hiding all or part of communications costs (if the architecture allows it)
- No risk of deadlock

Disadvantages of non blocking call

- Greater additional costs (several calls for one single send or receive, request management)
- Higher complexity and more complicated maintenance
- Less efficient on some machines (for example with transfer starting only at the `MPI_Wait()` call)
- Risk of performance loss on the computational kernels (for example, differentiated management between the area near the border of a domain and the interior area, resulting in less efficient use of memory caches)
- Limited to point-to-point communications (it is extended to collective communications in MPI 3.0)

Communication Modes

Interfaces

MPI_Isend() MPI_Issend() and MPI_Ibsend() for nonblocking send

```
int MPI_Isend(const void*values, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, MPI_Request *req)
int MPI_Issend(const void*values, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *req)
int MPI_Ibsend(const void*values, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm, MPI_Request *req)
```

MPI_Irecv() for nonblocking receive.

```
int MPI_Irecv(void *values, int count, MPI_Datatype msgtype, int source,
              int tag, MPI_Comm comm, MPI_Request *req)
```

Communication Modes

Interfaces

MPI_Wait() wait for the end of a communication, MPI_Test() is the nonblocking version.

```
int MPI_Wait(MPI_Request *req, MPI_Status *statut)
int MPI_Test(MPI_Request *req, int *flag, MPI_Status *statut)
```

MPI_Waitall() (MPI_Testall()) await the end of all communications.

```
int MPI_Waitall(int count, MPI_Request reqs[],MPI_Status statuts[])
int MPI_Testall(int count, MPI_Request reqs[],int *flag, MPI_Status statuts[])
```

Communication Modes

Interfaces

`MPI_Waitany()` wait for the end of one communication, `MPI_Testany()` is the nonblocking version.

```
int MPI_Waitany(int count,MPI_Request reqs[],int *indice,MPI_Status *statut)
int MPI_Testany(int count,MPI_Request reqs[],int *indice,int *flag,MPI_Status *statut)
```

`MPI_Waitsome()` wait for the end of at least one communication, `MPI_Testsome()` is the nonblocking version.

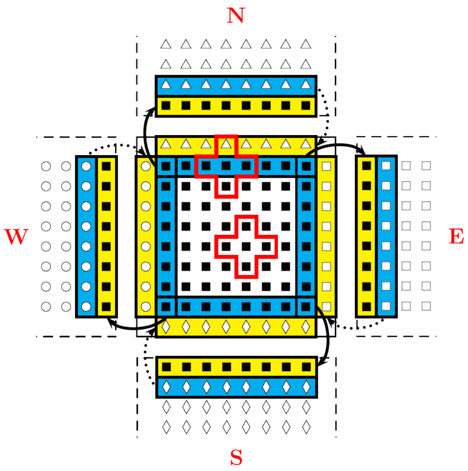
```
int MPI_Waitsome(int count,MPI_Request reqs[],int *outcount,int *indices,MPI_Status *statuses)
int MPI_Testsome(int count,MPI_Request reqs[],int *outcount,int *indices,MPI_Status *statuses)
```

Communication Modes

Request management

- After a call to a blocking wait function (`MPI_Wait()`, `MPI_Waitall()`,...), the request argument is set to `MPI_REQUEST_NULL`.
- The same for a nonblocking wait when the *flag* is set to true.
- A wait call with a `MPI_REQUEST_NULL` request does nothing.

Communication Modes



Communication Modes

```
void start_communication(double *u){
  /* Send to the North and receive form the South */
  MPI_Irecv(&(u[]),1,rowtype,neighbor[S],tag,comm2d,&(request[0]));
  MPI_Isend(&(u[]),1,rowtype,neighbor[N],tag,comm2d,&(request[1]));

  /* Send to the South and receive from the North */
  MPI_Irecv(&(u[]),1,rowtype,neighbor[N],tag,comm2d,&(request[2]));
  MPI_Isend(&(u[]),1,rowtype,neighbor[S],tag,comm2d,&(request[3]));

  /* Send to the West and receive from the East */
  MPI_Irecv(&(u[]),1,columntype,neighbor[E],tag,comm2d,&(request[4]));
  MPI_Isend(&(u[]),1,columntype,neighbor[W],tag,comm2d,&(request[5]));

  /* Send to the East and receive from the West */
  MPI_Irecv(&(u[]),1,columntype,neighbor[W],tag,comm2d,&(request[6]));
  MPI_Isend(&(u[]),1,columntype,neighbor[E],tag,comm2d,&(request[7]));
}
void end_communication(double *u) {
  MPI_Waitall(2*NB_NEIGHBORS,request,tab_status);
}
```

Communication Modes

```c
while(!(convergence) && (it < it_max) ) {
  it = it+1;
  /* Swap pointers */
  temp = u; u = u_new; u_new = temp;

  start_communication(u);
  calcul(u,u_new, sx+1, ex-1, sy+1, ey-1);
  end_communication(u);

  /* North */
  calcul(u,u_new,sx,sx,sy,ey);
  /* South */
  calcul(u,u_new,ex,ex,sy,ey);
  /* West */
  calcul(u,u_new,sx,ex,sy,sy);
  /* East */
  calcul(u,u_new,sx,ex,ey,ey);

  /* Compute global error */
  diffnorm = global_error (u, u_new);

  convergence = (diffnorm < eps);
}
```

Communication Modes

Overlap levels on different machines

Machine	*Level*
Zay(IntelMPI)	43%
Zay(IntelMPI) I_MPI_ASYNC_PROGRESS=yes	95%

Measurements taken by overlapping a compute kernel with a communication kernel which have the same execution times.

An overlap of 0% means that the total execution time is twice the time of a compute (or a communication) kernel.
An overlap of 100% means that the total execution time is the same as the time of a compute (or a communication) kernel.

Communication Modes

Number of received elements

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

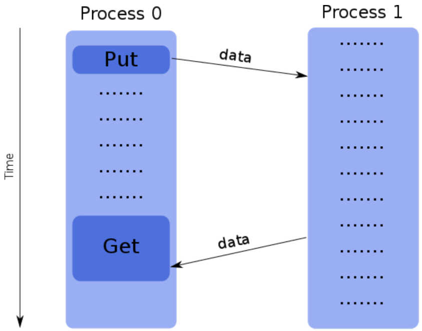
- In `MPI_Recv()` or `MPI_Irecv()` call, the count argument in the standard is the number of elements in the buffer buf.
- This number must be greater than the number of elements to be received.
- When it is possible, for increased clarity, it is adviced to put the number of elements to be received.
- We can obtain the number of elements received with `MPI_Get_count()` and the msgstatus argument returned by the `MPI_Recv()` or `MPI_Wait()` call.

```
int MPI_Get_count(MPI_Status *msgstatus,MPI_Datatype msgtype,int *count)
```

Communication Modes

One-Sided Communications

One-sided communications (Remote Memory Access or RMA) consists of accessing the memory of a distant process in *read* or *write* without the distant process having to manage this access explicitly. The target process does not intervene during the transfer.



Communication Modes

General approach

- Creation of a memory window with `MPI_Win_create()` to authorize RMA transfers in this zone.
- Remote access in *read* or *write* by calling `MPI_Put()`, `MPI_Get()` or `MPI_Accumulate()`.
- Free the memory window with `MPI_Win_free()`.

Synchronization methods

In order to ensure the correct functioning of the application, it is necessary to execute some synchronizations. Three methods are available :

- Active target communication with global synchronization (`MPI_Win_fence()`)
- Active target communication with synchronization by pair (`MPI_Win_start()` and `MPI_Win_complete()` for the origin process ; `MPI_Win_post()` and `MPI_Win_wait()` for the target process)
- Passive target communication without target intervention (`MPI_Win_lock()` and `MPI_Win_unlock()`)

Communication Modes

```c
/* ex_fence */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,realsize,i;
  int n=4,m=4,targetrank,nbelts;
  MPI_Aint dim_win,displacement;
  double *win_local, *tab,sum;
  MPI_Win win;
  int assert=0;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&ranK);
  MPI_Type_size(MPI_DOUBLE,&realsize);

  if (rank == 0 ) {
    n = 0;
    tab = (double *) malloc(m*sizeof(double)); }

  win_local = (double *) malloc(n*sizeof(double));
  dim_win = realsize*n;

  MPI_Win_create(win_local,dim_win,realsize,MPI_INFO_NULL,MPI_COMM_WORLD,&win);
```

Communication Modes

```c
    if (rank == 0) {
      for(i=0;i<m;i++) tab[i]=i+1;
    } else {
      for(i=0;i<n;i++) win_local[i]=0.0;
    }

    MPI_Win_fence(assert,win);
    if (rank == 0) {
      targetrank= 1; nb_elts = 2; displacement =1;
      MPI_Put(tab,nbelts,MPI_DOUBLE,targetrank,
              displacement,nbelts,MPI_DOUBLE,win);}

    MPI_Win_fence(assert,win);
    sum = 0.;
    if (rank == 0) {
      for (i=0;i<m-1;i++) sum=sum+tab[i];
      tab[m-1] = sum;
    } else {
      for (i=0;i<n-1;i++) sum=sum+win_local[i];
      win_local[n-1] = sum; }

    MPI_Win_fence(assert,win);
    if (rank == 0) {
      nbelts=1;displacement=m-1;
      MPI_Get(tab,nbelts,MPI_DOUBLE,targetrank,
              displacement,nbelts,MPI_DOUBLE,win);}
```

Communication Modes

Advantages of One-Sided Communications

- Certain algorithms can be written more easily.
- More efficient than point-to-point communications on certain machines (use of specialized hardware such as a DMA engine, coprocessor, specialized memory, ...).
- The implementation can group together several operations.

Disadvantages of One-Sided Communications

- Synchronization management is tricky.
- Complexity and high risk of error.
- For passive target synchronizations, it is mandatory to allocate the memory with `MPI_Alloc_mem()` which does not respect the Fortran standard (Cray pointers cannot be used with certain compilers).
- Less efficient than point-to-point communications on certain machines.

Derived datatypes

Derived datatypes

Introduction

- In communications, exchanged data have different datatypes : `MPI_INTEGER`, `MPI_REAL`, `MPI_COMPLEX`, etc.
- We can create more complex data structures by using subroutines such as `MPI_Type_contiguous()`, `MPI_Type_vector()`, `MPI_Type_indexed()` or `MPI_Type_create_struct()`
- Derived datatypes allow exchanging non-contiguous or non-homogenous data in the memory and limiting the number of calls to communications subroutines.

Derived datatypes

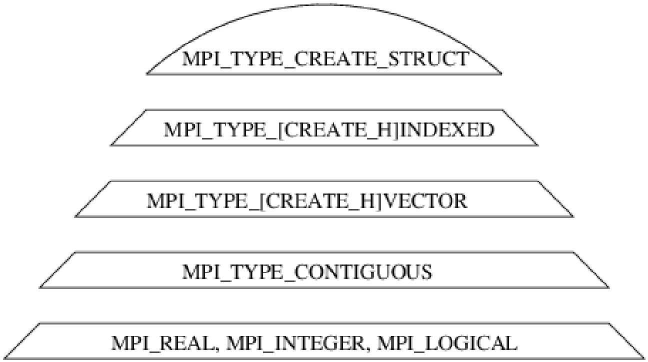

MPI_TYPE_CREATE_STRUCT
MPI_TYPE_[CREATE_H]INDEXED
MPI_TYPE_[CREATE_H]VECTOR
MPI_TYPE_CONTIGUOUS
MPI_REAL, MPI_INTEGER, MPI_LOGICAL

Figure 21 – Hierarchy of the MPI constructors

Derived datatypes

Contiguous datatypes

- `MPI_Type_contiguous()` creates a data structure from a homogenous set of existing datatypes contiguous in memory.

1.	6.	11.	16.	21.	26.
2.	7.	12.	17.	22.	27.
3.	8.	13.	18.	23.	28.
4.	9.	14.	19.	24.	29.
5.	10.	15.	20.	25.	30.

```
    MPI_Type_contiguous(6,MPI_FLOAT,&new_type);
```

Figure 22 – MPI_Type_contiguous subroutine

```
int MPI_Type_contiguous(int count,MPI_Datatype old_type,MPI_Datatype *new_type)
```

Derived datatypes

Constant stride

- MPI_Type_vector() creates a data structure from a homogenous set of existing datatypes separated by a constant stride in memory. The stride is given in number of elements.

1.	6.	11.	16.	21.	26.
2.	7.	12.	17.	22.	27.
3.	8.	13.	18.	23.	28.
4.	9.	14.	19.	24.	29.
5.	10.	15.	20.	25.	30.

```
MPI_Type_vector(5,1,6,MPI_FLOAT,&new_type);
```

Figure 23 – MPI_Type_vector subroutine

```
int MPI_Type_vector(int count,int block_length,int stride,
                    MPI_Datatype old_type,MPI_Datatype *new_type)
```

Derived datatypes

Constant stride

- `MPI_Type_create_hvector()` creates a data structure from a homogenous set of existing datatype separated by a constant stride in memory.
  The stride is given in bytes.
- This call is useful when the old type is no longer a base datatype (`MPI_INT`, `MPI_FLOAT`,...) but a more complex datatype constructed by using MPI subroutines, because in this case the stride can no longer be given in number of elements.

```
int MPI_Type_create_hvector(int count,int block_length,MPI_Aint stride,
                            MPI_Datatype old_type,MPI_Datatype *new_type)
```

Derived datatypes

Commit derived datatypes

- Before using a new derived datatype, it is necessary to validate it with the `MPI_Type_commit()` subroutine.

```
int MPI_Type_commit(MPI_Datatype *new_type)
```

- The freeing of a derived datatype is made by using the `MPI_Type_free()` subroutine.

```
int MPI_Type_free(MPI_Datatype *new_type)
```

Derived datatypes

```c
/* line */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,i,j;
  int nb_lines=5,nb_columns=6, tag=100;
  float a[nb_lines][nb_columns];
  MPI_Datatype type_line;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  /* Initialization of the matrix on each process */
  for(i=0;i<nb_lines;i++)
    for(j=0;j<nb_columns;j++)
      a[i][j]=rank;

  /* Definition of the type_line datatype */
  MPI_Type_contiguous(nb_colonnes,MPI_FLOAT,&type_line);

  /* Validation of the type_line datatype */
  MPI_Type_commit(&type_line);
```

Derived datatypes

```
  /* Sending of the first column */
  if (rank == 0) {
    MPI_Send(a,1,type_line,1,tag,MPI_COMM_WORLD);
  } else {
    MPI_Recv(&(a[nb_lines-1][0]),1,type_line,0,tag,
             MPI_COMM_WORLD,&statut); }

  /* Free the datatype */
  MPI_Type_free(&type_line);

  MPI_Finalize();
}
```

Derived datatypes

```c
/* column */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,i,j;
  int nb_lines=5,nb_columns=6, tag=100;
  float a[nb_lines][nb_columns];
  MPI_Datatype type_column;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  /* Initialisation of the matrix on each process */
  for(i=0;i<nb_lines;i++)
    for(j=0;j<nb_columns;j++)
      a[i][j]=rank;

  /* Definition of the datatype type_column */
  MPI_Type_vector(nb_lines,1,nb_columns,MPI_FLOAT,&type_column);

  /* Validation of the datatype type_column */
  MPI_Type_commit(&type_column);
```

Derived datatypes

```c
  /* Sending of the first column */
  if (rank == 0) {
    MPI_Send(&(a[1][0]),1,type_column,1,tag,MPI_COMM_WORLD);
  } else {
    MPI_Recv(&(a[nb_lignes-2][0]),1,type_column,0,tag,
             MPI_COMM_WORLD,&statut); }

  /* Free the datatype type_column */
  MPI_Type_free(&type_colonne);

  MPI_Finalize();
}
```

Derived datatypes

```c
/* block */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,i,j;
  int nb_lines=5,nb_columns=6, tag=100;
  int nb_lines_block=2,nb_columns_block=3;
  float a[nb_lines][nb_columns];
  MPI_Datatype type_block;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  /* Initialization of the matrix on each process */
  for(i=0;i<nb_lines;i++)
    for(j=0;j<nb_columns;j++)
      a[i][j]=rank;

  /* Creation of the datatype type_block */
  MPI_Type_vector(nb_lines_block,nb_columns_block,nb_columns,
                  MPI_FLOAT,&type_block);

  /* Validation of the datatype type_block */
  MPI_Type_commit(&type_block);
```

Derived datatypes

```c
  /* Sending of a block */
  if (rank == 0) {
    MPI_Send(a,1,type_block,1,tag,MPI_COMM_WORLD);
  } else {
    MPI_Recv(&(a[nb_lines-2][nb_columns-3]),1,type_block,0,tag,
             MPI_COMM_WORLD,&statut); }

  /* Freeing of the datatype type_block */
  MPI_Type_free(&type_block);

  MPI_Finalize();
}
```

Derived datatypes

Homogenous datatypes of variable strides

- `MPI_Type_indexed()` allows creating a data structure composed of a sequence of blocks containing a variable number of elements separated by a variable stride in memory. The stride is given in number of elements.
- `MPI_Type_create_hindexed()` has the same functionality as `MPI_Type_indexed()` except that the strides separating two data blocks are given in bytes.
  This subroutine is useful when the old datatype is not an MPI base datatype(`MPI_INT`, `MPI_FLOAT`, ...). We cannot therefore give the stride in number of elements of the old datatype.
- For `MPI_Type_create_hindexed()`, as for `MPI_Type_create_hvector()`, use `MPI_Type_size()` or `MPI_Type_get_extent()` in order to obtain in a portable way the size of the stride in bytes.

Derived datatypes

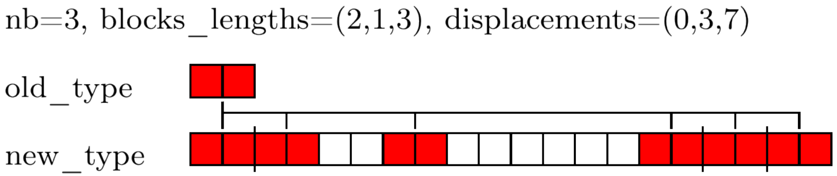


Figure 24 – The `MPI_Type_indexed` constructor

```
int MPI_Type_indexed(int nb,const int block_lengths[],const int displacements[],
                     MPI_Datatype old_type,MPI_Datatype *new_type)
```

Derived datatypes

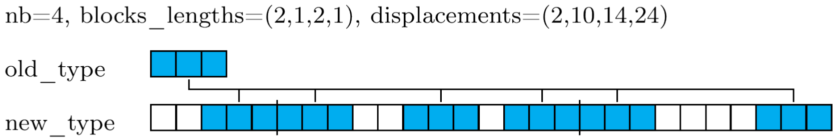


Figure 25 – The MPI_Type_create_hindexed constructor

```
int MPI_Type_create_hindexed(int nb,const int block_lengths[],
                             const MPI_Aint displacements,
                             MPI_Datatype old_type,
                             MPI_Datatype *new_type)
```

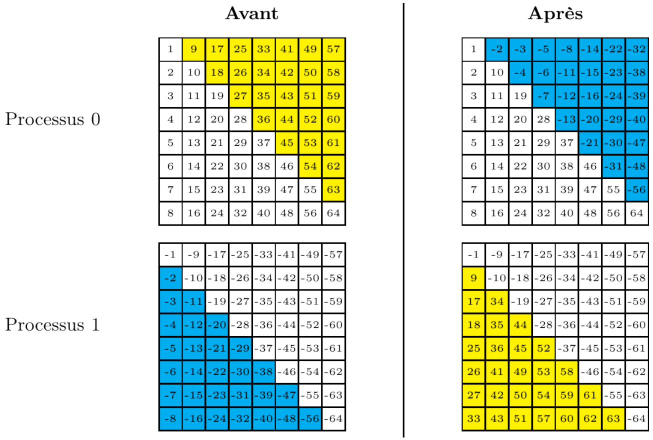
Derived datatypes

Example : triangular matrix

In the following example, each of the two processes :

1. Initializes its matrix (positive growing numbers on process 0 and negative decreasing numbers on process 1).
2. Constructs its datatype : triangular matrix (superior for the process 0 and inferior for the process 1).
3. Sends its triangular matrix to the other process and receives back a triangular matrix which it stores in the same place which was occupied by the sent matrix. This is done with the `MPI_Sendrecv_replace()` subroutine.
4. Frees its resources and exits MPI.

Derived datatypes



Processus 0 – Avant

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

Processus 0 – Après

1	-2	-3	-5	-8	-14	-22	-32
2	10	-4	-6	-11	-15	-23	-38
3	11	19	-7	-12	-16	-24	-39
4	12	20	28	-13	-20	-29	-40
5	13	21	29	37	-21	-30	-47
6	14	22	30	38	46	-31	-48
7	15	23	31	39	47	55	-56
8	16	24	32	40	48	56	64

Processus 1 – Avant

-1	-9	-17	-25	-33	-41	-49	-57
-2	-10	-18	-26	-34	-42	-50	-58
-3	-11	-19	-27	-35	-43	-51	-59
-4	-12	-20	-28	-36	-44	-52	-60
-5	-13	-21	-29	-37	-45	-53	-61
-6	-14	-22	-30	-38	-46	-54	-62
-7	-15	-23	-31	-39	-47	-55	-63
-8	-16	-24	-32	-40	-48	-56	-64

Processus 1 – Après

-1	-9	-17	-25	-33	-41	-49	-57
9	-10	-18	-26	-34	-42	-50	-58
17	34	-19	-27	-35	-43	-51	-59
18	35	44	-28	-36	-44	-52	-60
25	36	45	52	-37	-45	-53	-61
26	41	49	53	58	-46	-54	-62
27	42	50	54	59	61	-55	-63
33	43	51	57	60	62	63	-64

Figure 26 – Exchange between the two processes

Derived datatypes

```c
/* triangle */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,i,j;
  int n=8, tag=100,sign=1;
  float a[n][n];
  MPI_Datatype type_triangle;
  MPI_Status statut;
  int block_lengths[n],displacements[n];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  /* Initialisation of the matrix on each process */
  if (rank == 1) sign=-1;
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      a[i][j]=sign*(1+i+n*j);

  /* Creation of type_triangle for each process */
  if (rank == 0) {
    for(i=0;i<n;i++) block_lengths[i] = n-i-1;
    for(i=0;i<n;i++) displacements[i] = (n+1)*i+1;
  } else {
    for(i=0;i<n;i++) block_lengths[i] = i;
    for(i=0;i<n;i++) displacements[i] = n*i;
  }

  MPI_Type_indexed(n,block_lengths,displacements,MPI_FLOAT,&type_triangle);
  MPI_Type_commit(&type_triangle);

  /* Swap of matrix  */
  MPI_Sendrecv_replace(a,1,type_triangle,(rank+1)%2,tag,
                       (rank+1)%2,tag,MPI_COMM_WORLD,&statut);

  /* Free the triangle datatype */
  MPI_Type_free(&type_triangle);
  MPI_Finalize();
}
```

Derived datatypes

Size of datatype

- MPI_Type_size() returns the number of bytes needed to send a datatype. This value ignores any holes present in the datatype.

```
int MPI_Type_size(MPI_Datatype datatype,int *typesize)
```

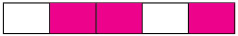
- The extent of a datatype is the memory space occupied by this datatype (in bytes). This value is used to calculate the position of the next datatype element (i.e. the stride between two successive datatype elements).

```
int MPI_Type_get_extent(MPI_Datatype datatype,MPI_Aint *lb,
                        MPI_Aint *extent)
```

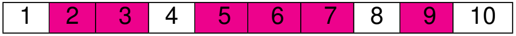
Derived datatypes

Example 1 : `MPI_Type_indexed(2,{2,1},{1,4},MPI_INT,&type)`

MPI Datatype :



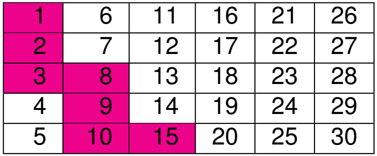
Two succesives elements :



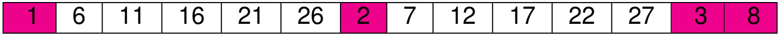
`size = 12 (3 integers); lb = 4 (1 integer); extent = 16 (4 integers)`

Example 2 : `MPI_Type_vector(3,1,nb_columns,MPI_INT,&type_half_column)`

2D View :



1D View :



`size = 12 (3 integers); lb = 0; extent = 44 (11 integers)`

Derived datatypes

Modify the extent

- The extent is a datatype parameter. By default, it's the space in memory between the first and last component of a datatype (bounds included and with alignment considerations). We can modify the extent to create a new datatype by adapting the preceding one using `MPI_Type_create_resized()`. This provides a way to choose the stride between two successive datatype elements.

```
int MPI_Type_create_resized(MPI_Datatype old,MPI_Aint lb,
                            MPI_Aint extent,MPI_Datatype *new)
```

Derived datatypes

```c
/* half_column */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,i,j,integer_size;
  int nb_lines=5, nb_columns=6, tag=100,sign=1;
  int size_half_column=nb_lines/2;
  int a[nb_lines][nb_columns];
  MPI_Datatype type_half_column1,type_half_column2;
  MPI_Status statut;
  MPI_Aint lb1,extent1,lb2,extent2;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  /* Initialization of the matrix on each process */
  if (rank == 1) sign=-1;
  for(i=0;i<nb_lines;i++)
    for(j=0;j<nb_columns;j++)
      a[i][j]=sign*(1+i+nb_lines*j);

  /* Construction of type type_half_column1 */
  MPI_Type_vector(size_half_column,1,nb_columns,MPI_INT,&type_half_column1);

  /* Know the size of  MPI_INT */
  MPI_Type_size(MPI_INT,&integer_size);

  /* Informations on the type type_half_column1 */
  MPI_Type_get_extent(type_half_column1,&lb1,&extent1);
  if (rank == 0) printf("Type_half_column1: lb=%d extent %d\n",
                        lb1,extent1);

  /* Construction of type type_half_column2 */
  lb2 = 0;
  extent2 = taille_integer;
  MPI_Type_create_resized(type_half_column1,lb2,extent2,
                          &type_half_column2);
```

Derived datatypes

```c
  /* Information on type_half_column2 */
  MPI_Type_get_extent(type_half_column2,&lb2,&extent2);
  if (rank == 0) printf("Type_half_column2: lb=%d extent %d\n",
                        lb2,extent2);

  /* Validation of type_half_column2 */
  MPI_Type_commit(&type_half_column2);

  if (rank == 0) {
    MPI_Send(a,2,type_half_column2,1,tag,MPI_COMM_WORLD);
  } else {
    MPI_Recv(&(a[nb_lines-2][1]),4,MPI_INT,0,tag,MPI_COMM_WORLD,&statut);
    printf("Matrix A on the process 1\n");
    for(i=0;i<nb_lines;i++) {
      for(j=0;j<nb_columns;j++)
        printf("%d ",a[i][j]);
      printf("\n"); } }

  MPI_Finalize();
}
```

```
> mpiexec -n 2 half_line
 type_half_column1: lb=0, extent=28
 type_half_column2: lb=0, extent=4
```

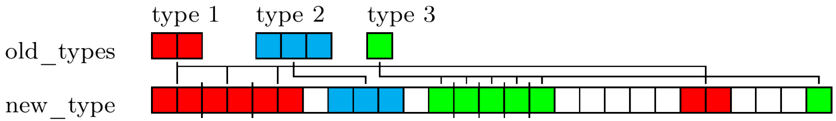
```
Matrice A sur le processus 1
 -1  -6 -11 -16 -21 -26
 -2  -7 -12 -17 -22 -27
 -3  -8 -13 -18 -23 -28
 -4  1 2 6 7 -29
 -5 -10 -15 -20 -25 -30
```

Derived datatypes

Heterogenous datatype

- MPI_Type_create_struct() call allows creating a set of data blocks indicating the type, the count and the displacement of each block.
- It is the most general datatype constructor. It further generalizes MPI_Type_indexed() by allowing a different datatype for each block.

nb=5, blocks_lengths=(3,1,5,1,1), displacements=(0,7,11,21,26),
old_types=(type1,type2,type3,type1,type3)



```
int MPI_Type_create_struct(int nb,const int blocks_lengths[],const MPI_Aint displacements[],
                           const MPI_Datatype old_types[],MPI_Datatype *new_type)
```

Derived datatypes

Compute displacements

- MPI_Type_create_struct() is useful for creating MPI datatypes corresponding to Fortran derived datatypes or to C structures.
- The memory alignment of heterogeneous data structures is different for each architecture and each compiler.
- The displacement between two components of a Fortan derived datatype (or of a C structure) can be obtained by calculating the difference between their memory addresses.
- MPI_Get_address() provides the address of a variable. It's equivalent of & operator in C.
- Warning, even in C, it is better to use this subroutine for portability reasons.
- Warning, you have to check the extent of the MPI datatypes obtaineds.

```
int MPI_Get_address(const void *variable,MPI_Aint *address_variable)
```

Derived datatypes

```c
/* Interaction_Particles */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct Particle {
  char category[5];
  int mass;
  float coords[3];
  bool class;
};

int main(int argc,char *argv[]) {
  int rank,i;
  int n=1000,tag=100;
  int blocks_length[4];
  MPI_Datatype types[4],type_particle,temp;
  MPI_Status statut;
  MPI_Aint addresses[5],displacements[5],lb,extent;
  struct Particle p[n],temp_p[n];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  /* Construction du type de donnees */
  types[0] = MPI_CHARACTER; types[1] = MPI_INT;
  types[2] = MPI_FLOAT; types[3] = MPI_LOGICAL;
  blocks_length[0]=5;blocks_length[1]=1;
  blocks_length[2]=3;blocks_length[3]=1;
```

Derived datatypes

```
  MPI_Get_address(&(p[0].category),&(addresses[0]));
  MPI_Get_address(&(p[0].mass),&(addresses[1]));
  MPI_Get_address(&(p[0].coords),&(addresses[2]));
  MPI_Get_address(&(p[0].class),&(addresses[3]));
  /* Compute displacements relative to start address */
  for (i=0;i<4;i++) displacements[i] = addresses[i]-addresses[0];
  MPI_Type_create_struct(4,blocks_length,displacements,types,&temp);
  MPI_Get_address(&(p[1].category),&(addresses[4]));
  lb=0;
  extent = addresses[4]-addresses[0];
  MPI_Type_create_resized(temp,lb,extent,&type_particle);
  /* Validation of type */
  MPI_Type_commit(&type_particle);

  /* Initialization of particles on each process */

  /* Send particles */
  if (rank == 0) {
    MPI_Send(&(p[0].category),n,type_particle,1,tag,MPI_COMM_WORLD);
  } else {
    MPI_Recv(&(temp_p[0].category),n,type_particle,0,tag,
             MPI_COMM_WORLD,&statut);
  }

  MPI_Type_free(&type_particle);
  MPI_Finalize();
}
```

Derived datatypes

Conclusion

- The MPI derived datatypes are powerful data description portable mechanisms.
- When they are combined with subroutines like `MPI_Sendrecv()`, they allow simplifying the writing of interprocess exchanges.
- The combination of derived datatypes and topologies (described in one of the next chapters) makes MPI the ideal tool for all domain decomposition problems with both regular or irregular meshes.

MPI Hands-On – Exercise 4 : Matrix transpose

- The goal of this exercise is to practice with the derived datatypes.
- *A* is a matrix with 5 lines and 4 columns defined on the process 0.
- Process 0 sends its *A* matrix to process 1 and transposes this matrix during the send.

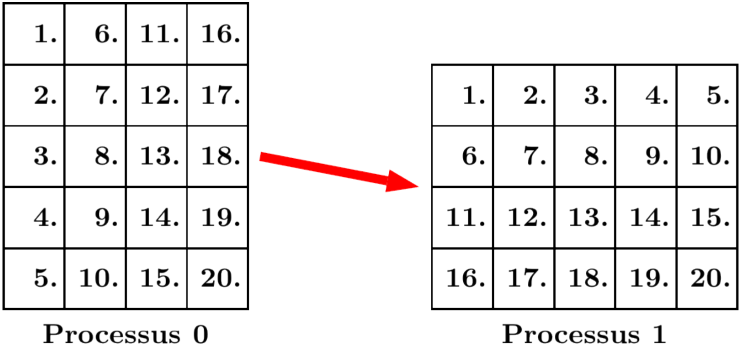


Figure 27 – Matrix transpose

- To do this, we need to create two derived datatypes, a derived datatype `type_column` and a derived datatype `type_transpose`.

MPI Hands-On – Exercise 5 : Matrix-matrix product

- Collective communications : matrix-matrix product $C = A \times B$
  - The matrixes are square and their sizes are a multiple of the number of processes.
  - The matrixes A and B are defined on process 0. Process 0 sends a horizontal slice of matrix A and a vertical slice of matrix B to each process. Each process then calculates its diagonal block of matrix C.
  - To calculate the non-diagonal blocks, each process sends to the other processes its own slice of A.
  - At the end, process 0 gathers and verifies the results.

MPI Hands-On – Exercise 5 : Matrix-matrix product

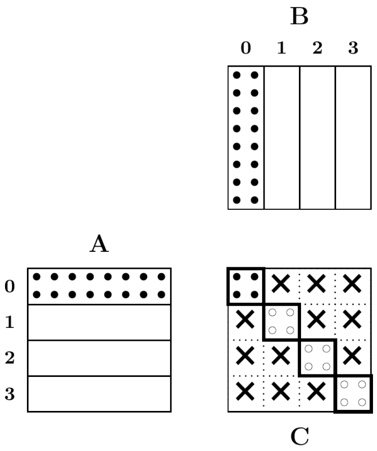


Figure 28 – Distributed matrix product

MPI Hands-On – Exercise 5 : Matrix-matrix product

- The algorithm that may seem the most immediate and the easiest to program, consisting of each process sending its slice of its matrix *A* to each of the others, does not perform well because the communication algorithm is not well-balanced. It is easy to seen this when doing performance measurements and graphically representing the collected traces.

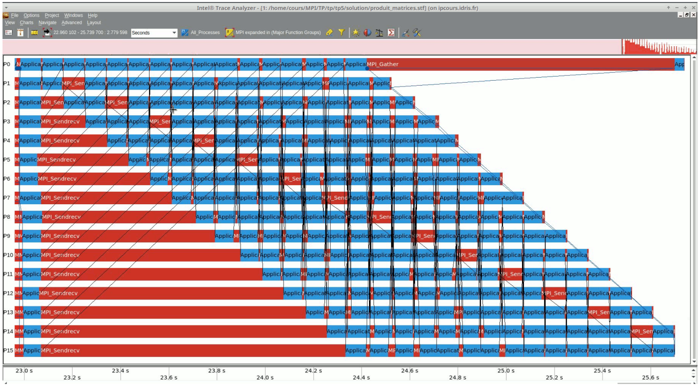
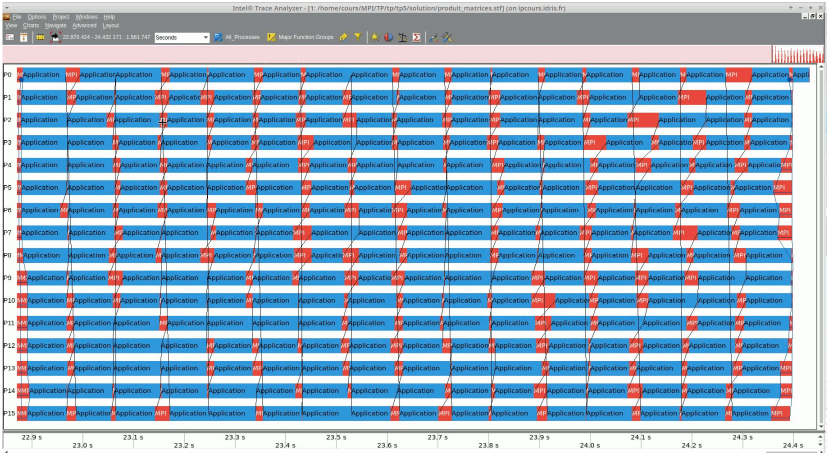

Intel® Trace Analyzer - [1: /home/cours/MPI/TP/tp/tp5/solution/produit_matrices.stf] (on ipcours.idris.fr)
File Options Project Windows Help
View Charts Navigate Advanced Layout
22.960 102 - 25.739 700 : 2.779 598
Seconds
All_Processes
MPI expanded in (Major Function Groups)
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
Application
MPI_Gather
MPI_Sendrecv
23.0 s
23.2 s
23.4 s
23.6 s
23.8 s
24.0 s
24.2 s
24.4 s
24.6 s
24.8 s
25.0 s
25.2 s
25.4 s
25.6 s

Figure 29 – Parallel matrix product on 16 processes, for a matrix size of 1024 (first algorithm)

MPI Hands-On – Exercise 5 : Matrix-matrix product

- Changing the algorithm in order to *shift* slices from process to process, we obtain a perfect balance between calculations and communications and have a speedup of 2 compared to the naive algorithm.



Intel® Trace Analyzer - [1: /home/cours/MPI/TP/tp/tp5/solution/produit_matrices.stf] (on ipcours.idris.fr)
File Options Project Windows Help
View Charts Navigate Advanced Layout
22.870 424 - 24.432 171 : 1.561 747 Seconds
All_Processes
Major Function Groups
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
Application
MPI
22.9 s
23.0 s
23.1 s
23.2 s
23.3 s
23.4 s
23.5 s
23.6 s
23.7 s
23.8 s
23.9 s
24.0 s
24.1 s
24.2 s
24.3 s
24.4 s

Figure 30 – Parallel matrix product on 16 processes, for a matrix size of 1024 (second algorithm)

Communicators

Communicators

Introduction

The purpose of communicators is to create subgroups on which we can carry out operations such as collective or point-to-point communications. Each subgroup will have its own communication space.

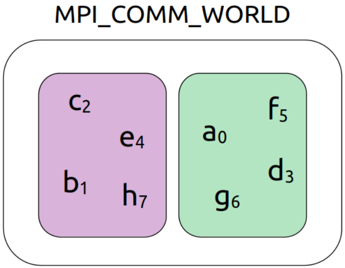


Figure 31 – Communicator partitioning

Communicators

Example

For example, we want to broadcast a collective message to even-ranked processes and another message to odd-ranked processes.

- Looping on *send/recv* can be very detrimental especially if the number of processes is high. Also a test inside the loop would be compulsory in order to know if the sending process must send the message to an even or odd process rank.
- A solution is to create a communicator containing the even-ranked processes, another containing the odd-ranked processes, and initiate the collective communications inside these groups.

Communicators

Default communicator

- A communicator can only be created from another communicator. The first one will be created from the `MPI_COMM_WORLD`.
- After the `MPI_Init()` call, a communicator is created for the duration of the program execution.
- Its identifier `MPI_COMM_WORLD` is an integer value defined in the header files.
- This communicator can only be destroyed via a call to `MPI_Finalize()`.
- By default, therefore, it sets the scope of collective and point-to-point communications to include all the processes of the application.

Communicators

Groups and communicators

- A communicator consists of :
  - A group, which is an ordered group of processes.
  - A communication context put in place by calling one of the communicator construction subroutines, which allows determination of the communication space.
- The communication contexts are managed by MPI (the programmer has no action on them : It is a hidden attribute).
- In the MPI library, the following subroutines exist for the purpose of building communicators : `MPI_Comm_create()`, `MPI_Comm_dup()`, `MPI_Comm_split()`
- The communicator constructors are collective calls.
- Communicators created by the programmer can be destroyed by using the `MPI_Comm_free()` subroutine.

Communicators

Partitioning of a communicator

In order to solve the problem example :

- Partition the communicator into odd-ranked and even-ranked processes.
- Broadcast a message inside the odd-ranked processes and another message inside the even-ranked processes.

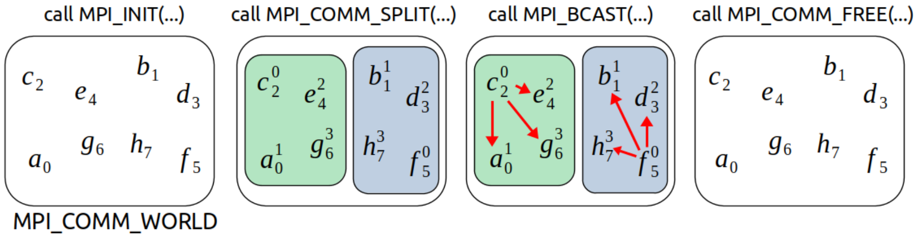


Figure 32 – Communicator creation/destruction

Communicators

Partitioning of a communicator with `MPI_Comm_split()`

The `MPI_Comm_split()` subroutine allows :

- Partitioning a given communicator into as many communicators as we want.
- Giving the same name to all these communicators : The process value will be the value of its communicator.
- Method :
  1. Define a colour value for each process, associated with its communicator number.
  2. Define a key value for ordering the processes in each communicator
  3. Create the partition where each communicator is called new_comm

```
int MPI_Comm_split(MPI_Comm comm,int color,int key,MPI_Comm *new_comm)
```

A process which assigns a color value equal to `MPI_UNDEFINED` will have the invalid communicator `MPI_COMM_NULL` for new_com.

Communicators

Example

Let's look at how to proceed in order to build the communicator which will subdivide the communication space into odd-ranked and even-ranked processes via the `MPI_Comm_split()` constructor.

process	a	b	c	d	e	f	g	h
rank_world	0	1	2	3	4	5	6	7
color	0	1	0	1	0	1	0	1
key	0	1	-1	3	4	-1	6	7
rank_even_odd	1	1	0	2	2	0	3	3

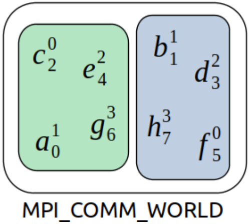
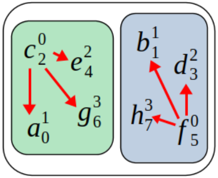


Figure 33 – Construction of the `ComEvenOdd` communicator with `MPI_Comm_split()`

Communicators

```c
/* EvenOdd */
#include <mpi.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,i,key,rank_in_world;
  int m=16;
  float a[16];
  MPI_Comm CommEvenOdd;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank_in_world);

  for(i=0;i<m;i++) a[i]=0.;
  if (rank_in_world == 2) {for(i=0;i<m;i++) a[i]=2.;}
  if (rank_in_world == 5) {for(i=0;i<m;i++) a[i]=5.;}

  key = rank_in_world;
  if ((rank_in_world == 2) || (rank_in_world == 5)) {
    key = -1; }

  /* Creation of even and odd communicators by giving them the same name */
  MPI_Comm_split(MPI_COMM_WORLD,rank_in_world%2,key,&CommEvenOdd);

  /* Broadcast of the message by the rank process 0 of each communicator to the processes
     of its group */
  MPI_Bcast(a,m,MPI_FLOAT,0,CommEvenOdd);

  /* Destruction of the communicator */
  MPI_Comm_free(&CommEvenOdd);
  MPI_Finalize();
}
```



Communicators

Communicator built from a group

- We can also build a communicator by defining a group of processes, thanks to the `MPI_Comm_create()` subroutine :

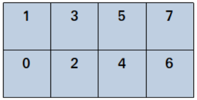
```c
/* CommCreation */
#include <mpi.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int size_new_comm=2;
  int proc_new_comm[2] = { 1,3};
  MPI_Group world_group,new_group;
  MPI_Comm new_comm;

  MPI_Init(&argc,&argv);

  /* Get processus group of the MPI_COMM_WORLD communicator */
  MPI_Comm_group(MPI_COMM_WORLD,&groupe_monde);

  /* Create a sub-group of processes */
  MPI_Group_incl(world_group,size_new_comm,proc_new_comm,&new_group);

  /* Create a new communicator based on the sub-group of processes */
  MPI_Comm_create(MPI_COMM_WORLD,new_group,&new_comm);

  MPI_Group_free(&new_group);
  MPI_Finalize();
}
```

- This process is however far more cumbersome than using `MPI_Comm_split()` whenever possible.

Communicators

Topologies

- In most applications, especially in domain decomposition methods where we match the calculation domain to the process grid, it is helpful to be able to arrange the processes according to a regular topology.
- MPI allows defining virtual cartesian or graph topologies.
  - Cartesian topologies :
    - Each process is defined in a grid.
    - Each process has a neighbour in the grid.
    - The grid can be periodic or not.
    - The processes are identified by their coordinates in the grid.
  - Graph topologies :
    - Can be used in more complex topologies.



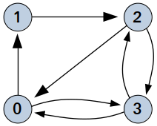


Figure 34 – A 2D Cartesian topology (left) and a Graph topology (right)

Communicators

Cartesian topologies

- A Cartesian topology is defined from a given communicator named comm_old, calling the `MPI_Cart_create()` subroutine.
- We define :
  - An integer ndims representing the number of grid dimensions.
  - An integer array dims of dimension ndims showing the number of processes in each dimension.
  - An array of ndims logicals which shows the periodicity of each dimension.
  - A logical reorder which shows if the process numbering can be changed by MPI.

```
int MPI_Cart_create(MPI_Comm comm_old,int ndims,const int *dims,const int *periods,
                    int reorder,MPI_Comm *comm_new)
```

Communicators

Example

Example on a grid having 4 domains along x and 2 along y, periodic in y.

```
int ndims=2,reorder;
int dims[ndims],periods[ndims];
MPI_Comm comm_2D;

dims[0]=4;dims[1]=2;
periods[0]=false;periods[1]=true;
reorder=false;

MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,periods,reorganisation,&comm_2D);
```

If `reorder = false` then the rank of the processes in the new communicator (comm_2D) is the same as in the old communicator (`MPI_COMM_WORLD`).
If `reorder = true`, the MPI implementation chooses the order of the processes.

Communicators

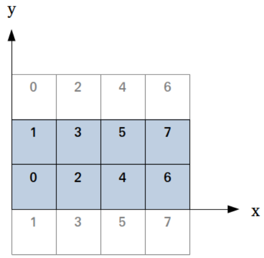


Figure 35 – A 2D periodic Cartesian topology in y

Communicators

3D Example

Example on a 3D grid having 4 domains along x, 2 along y and 2 along z, non periodic.

```
int ndims=3,reorder;
int dims[ndims],periods[ndims];
MPI_Comm comm_3D;

dims[0]=4;dims[1]=2;dims[2]=2;
periods[0]=false;periods[1]=false;periods[2]=false;
reorder=false;

MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,periods,reorder,&comm_3D);
```

Communicators

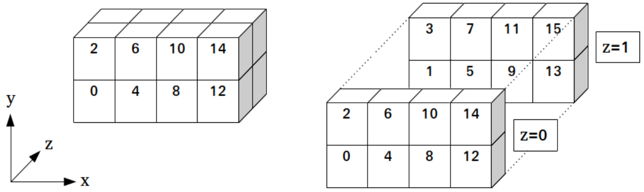


Figure 36 – A 3D non-periodic Cartesian topology

Communicators

Process distribution

The `MPI_Dims_create()` subroutine returns the number of processes in each dimension of the grid according to the total number of processes.

```
int MPI_Dims_create(int nb_procs,int ndims,int *dims)
```

Remark : If the values of dims in entry are all 0, then we leave to MPI the choice of the number of processes in each direction according to the total number of processes.

`dims` in entry	`call MPI_Dims_create`	`dims` en exit
(0,0)	(8,2,dims,code)	(4,2)
(0,0,0)	(16,3,dims,code)	(4,2,2)
(0,4,0)	(16,3,dims,code)	(2,4,2)
(0,3,0)	(16,3,dims,code)	`error`

Communicateurs

Rank and coordinates of a process

In a Cartesian topology, the rank of each process is associated with its coordinates in the grid.

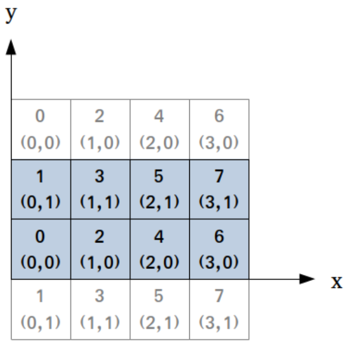


Figure 37 – A 2D periodic Cartesian topology in y

Communicators

Rank of a process

In a Cartesian topology, the `MPI_Cart_rank()` subroutine returns the rank of the associated process to the coordinates in the grid.

```
int MPI_Cart_rank(MPI_Comm comm,const int coords[],int *rank)
```

Communicators

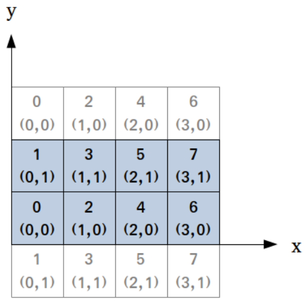


Figure 38 – A 2D periodic Cartesian topology in y

```
coords[0]=dims[0]-1;
for(i=0;i<dims[1];i++) {
  coords[1]=i;
  MPI_Cart_rank(comm_2D,coords,&(rank[i]));
}
............................................
i=0,in entry coords=[3,0],in exit rank[0]=6.
i=1,in entry coords=[3,1],in exit rank[1]=7.
```

Communicators

Coordinates of a process

In a cartesian topology, the `MPI_Cart_coords()` subroutine returns the coordinates of a process of a given rank in the grid.

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int ndims, int *coords)
```

Communicators

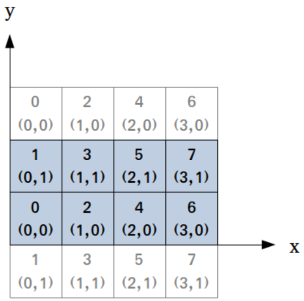


Figure 39 – A 2D periodic Cartesian topology in y

```
if (rank%2 == 0)
  MPI_Cart_coords(comm_2D,rank,2,&coords);
............................................
In entry, the rank values are : 0,2,4,6.
In exit, the coords values are :
(0,0),(1,0),(2,0),(3,0).
```

Communicators

Rank of neighbours

In a Cartesian topology, a process that calls the `MPI_Cart_shift()` subroutine can obtain the rank of a neighboring process in a given direction.

```
int MPI_Cart_shift(MPI_Comm comm,int direction, int step, int *rank_previous, int *rank_next)
```

- The direction parameter corresponds to the displacement axis (xyz).
- The step parameter corresponds to the displacement step.
- If a rank does not have a neighbor before (or after) in the requested direction, then the value of the previous (or following) rank will be `MPI_PROC_NULL`.

Communicators

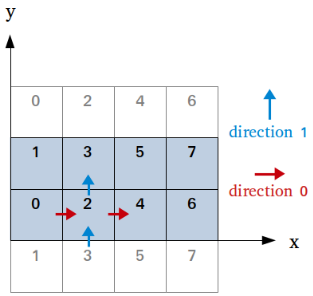


Figure 40 – Call of the MPI_Cart_shift() subroutine

```
MPI_Cart_shift(comm_2D,0,1,&rank_left,&rank_right);
............................................
For the process 2, rank_left=0, rank_right=4
```

```
MPI_Cart_shift(comm_2D,1,1,&rank_low,&rank_high);
............................................
For the process 2, rank_low=3, rank_high=3
```

Communicators

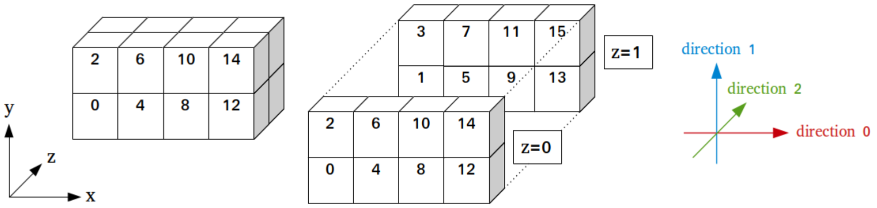


Figure 41 – Call of the MPI_Cart_shift() subroutine

```
MPI_Cart_shift(comm_3D,0,1,&rank_left,&rank_right)
............................................
For the process 0, rank_left=-1, rank_right=4
```

```
MPI_Cart_shift(comm_3D,1,1,&rank_low,&rank_high)
............................................
For the process 0, rank_low=-1, rank_high=2
```

```
MPI_Cart_shift(comm_3D,2,1,&rank_ahead,&rank_before)
............................................
For the process 0, rank_ahead=-1, rank_before=1
```

Communicators

Example

- create a 2D Cartesian grid periodic in y
- get coordinates of each process
- get neighbours ranks for each process

```c
/* decomposition */
#include <mpi.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int nb_procs,rank_in_topo;
  int ndims=2,N=1,E=2,S=3,W=4;
  int dims[ndims],coords[ndims],neighbor[4];
  int periods[ndims],reorder;
  MPI_Comm comm_2D;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&nb_procs);

  /* Know the number of processes along x and y */
  dims[0] = dims[1] = 0;

  MPI_Dims_create(nb_procs,ndims,dims);
```

Communicators

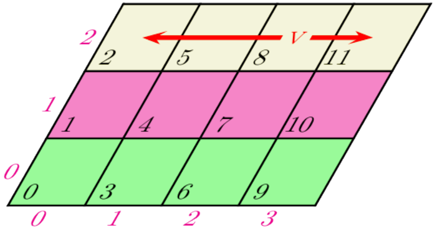
```
  /* 2D y-periodic grid creation */
  periods[0] = 0;
  periods[1] = 1;
  reorder = 0;

  MPI_Cart_create(MPI_COMM_WORLD,ndims,dims,periods,reorder,&comm_2D);

  /* Know my coordinates in the topology */
  MPI_Comm_rank(comm_2D,&rank_in_topo);
  MPI_Cart_coords(comm_2D,rank_in_topo,ndims,coords);

  /* Search of my West and East neighbors */
  MPI_Cart_shift(comm_2D,0,1,&(neighbors[W]),&(neighbors[E]));

  /* Search of my South and North neighbors */
  MPI_Cart_shift(comm_2D,1,1,&(neighbors[S]),&(neighbors[N]));

  MPI_Finalize();
}
```

Communicators

Subdividing a Cartesian topology

- The goal, by example, is to degenerate a 2D or 3D cartesian topology into, respectively, a 1D or 2D Cartesian topology.
- For MPI, degenerating a 2D Cartesian topology creates as many communicators as there are rows or columns in the initial Cartesian grid. For a 3D Cartesian topology, there will be as many communicators as there are planes.
- The major advantage is to be able to carry out collective operations limited to a subgroup of processes belonging to :
  - the same row (or column), if the initial topology is 2D ;
  - the same plane, if the initial topology is 3D.

Communicators



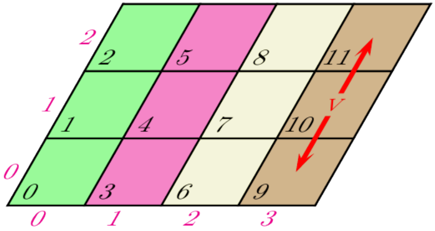


Figure 42 – Two examples of data distribution in a degenerated 2D topology

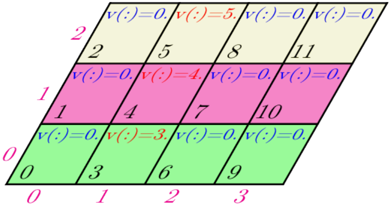
Communicators

Subdividing a Cartesian topology

There are two ways to degenerate a topology :

- By using the `MPI_Comm_split()` general subroutine
- By using the `MPI_Cart_sub()` subroutine designed for this purpose

```
int MPI_Cart_sub(MPI_Comm CommCart,const int remain_dims[],MPI_Comm *CommCartD)
```



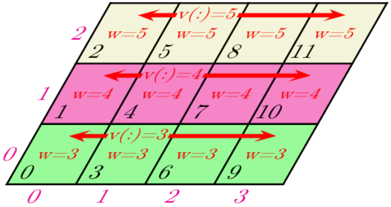


Figure 43 – Broadcast of a *V* array in the degenerated 2D grid.

Communicators

```c
/* CommCartSub */
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int NDim2D=2,m=4;
  int Dim2D[NDim2D],Periode[NDim2D],Coord2D[NDim2D],remain_dims[NDim2D];
  int Reorder,rank,i;
  MPI_Comm Comm2D,Comm1D;
  float V[m],W;


  MPI_Init(&argc,&argv);

  /* Creation de la grille 2D initiale */
  Dim2D[0] = 4;
  Dim2D[1] = 3;
  Periode[0] = 0; Periode[1] = 0;
  Reorder = 0;
  MPI_Cart_create(MPI_COMM_WORLD,NDim2D,Dim2D,Periode,Reorder,&Comm2D);
  MPI_Comm_rank(Comm2D,&rank);
  MPI_Cart_coords(Comm2D,rank,NDim2D,Coord2D);
```

Communicators

```c
  /* Initialisation of the vector V */
  if (Coord2D[0] == 1) {for(i=0;i<m;i++) V[i]=rank; }

  /* Every row of the grid must be a 1D cartesian topology */
  remain_dims[0] = 1;
  remain_dims[1] = 0;
  /* Subdivision of the 2D cartesian grid */
  MPI_Cart_sub(Comm2D,remain_dims,&Comm1D);

  /* The process of column 2 distribute the vector V to the processes of their row */
  MPI_Scatter(V,1,MPI_FLOAT,&W,1,MPI_FLOAT,1,Comm1D);

  printf("Rank : %d ; Coordinates : ( %d,%d); W = %f\n",
         rank,Coord2D[0],Coord2D[1],W);

  MPI_Finalize();
}
```

Communicators

```
> mpiexec -n 12 CommCartSub
Rank :  0 ; Coordinates : (0,0) ; W = 3.
Rank :  1 ; Coordinates : (0,1) ; W = 4.
Rank :  3 ; Coordinates : (1,0) ; W = 3.
Rank :  8 ; Coordinates : (2,2) ; W = 5.
Rank :  4 ; Coordinates : (1,1) ; W = 4.
Rank :  5 ; Coordinates : (1,2) ; W = 5.
Rank :  6 ; Coordinates : (2,0) ; W = 3.
Rank : 10 ; Coordinates : (3,1) ; W = 4.
Rank : 11 ; Coordinates : (3,2) ; W = 5.
Rank :  9 ; Coordinates : (3,0) ; W = 3.
Rank :  2 ; Coordinates : (0,2) ; W = 5.
Rank :  7 ; Coordinates : (2,1) ; W = 4.
```

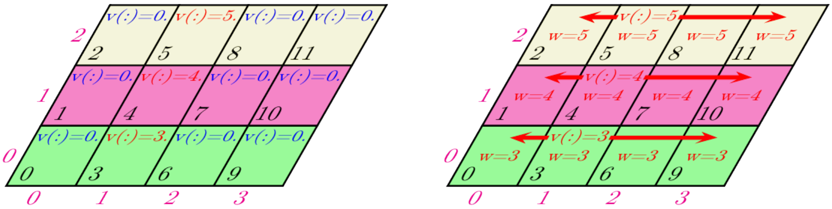


Figure 44 – Broadcast of a *V* array in the degenerated 2D grid.

MPI Hands-On – Exercise 6 : Communicators

- Using the Cartesian topology defined below, subdivide in 2 communicators following the lines by calling `MPI_Comm_split()`

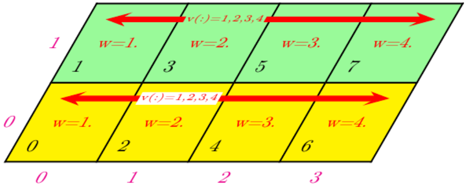


Figure 45 – Subdivision of a 2D topology and communication using the obtained 1D topology

- Constraint : define the color of each process without using the *modulo* operation.

MPI-IO

MPI-IO

Input/Output Optimisation

- Applications which perform large calculations also tend to handle large amounts of data and generate a significant number of I/O requests.
- Effective treatment of I/O can highly improve the global performances of applications.
- I/O tuning of parallel codes involves :
  - Parallelizing I/O access of the program in order to avoid serial bottlenecks and to take advantage of parallel file systems
  - Implementing efficient data access algorithms (non-blocking I/O)
  - Leveraging mechanisms implemented by the operating system (request grouping methods, I/O buffers, etc.).
- Libraries make I/O optimisations of parallel codes easier by providing ready-to-use capabilities.

MPI-IO

The MPI-IO interface

- The MPI-2 norm defines a set of functions designed to manage parallel I/O.
- The I/O functions use well-known MPI concepts. For instance, collectives and non-blocking operations on files and between MPI processes are similar. Files can also be accessed in a patterned way using the existing derived datatype functionality.
- Other concepts come from native I/O interfaces (file descriptors, attributes, . . .).

MPI-IO

Example of a sequential optimisation implemented by I/O libraries

- I/O performance suffers considerably when making many small I/O requests.
- Access on small, non-contiguous regions of data can be optimized by grouping requests and using temporary buffers.
- Such optimisation is performed automatically by MPI-IO libraries.

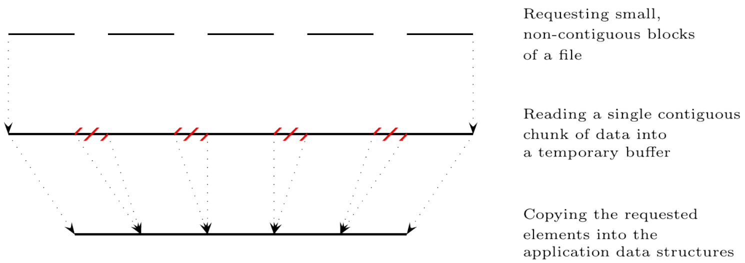

Requesting small, non-contiguous blocks of a file
Reading a single contiguous chunk of data into a temporary buffer
Copying the requested elements into the application data structures

Figure 46 – Data sieving mechanism improving I/O access on small, non-contiguous data set.

MPI-IO

Example of a parallel optimisation

Collective I/O access can be optimised by rebalancing the I/O operations in contiguous chunks and performing inter-process communications.

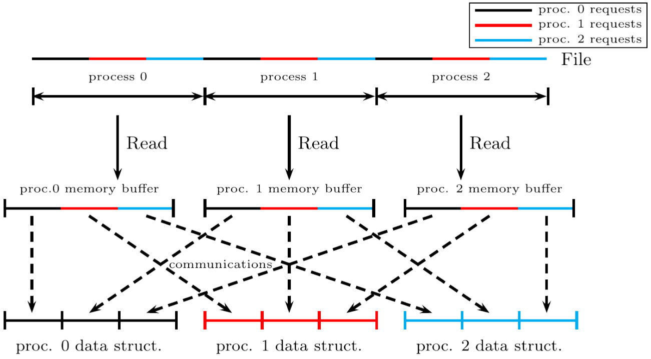


Figure 47 – Read operation performed in two steps by a group of processes

MPI-IO

Working with files

- Opening and closing files are collective operations within the scope of a communicator.
- Opening a file generates a file handle, an opaque representation of the opened file. File handles can be subsequently used to access files in MPI I/O subroutines.
- Access modes describe the opening mode, access rights, etc. Modes are specified at the opening of a file, using predefined MPI constants that can be combined together.
- All the processes of the communicator participate in subsequent collective operations.
- We are only describing here the open/close subroutines but others file management operations are available (preallocation, deletion, etc.). For instance, `MPI_File_get_info()` returns details on a file handle (information varies with implementations).

MPI-IO

```c
/* open01 */
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  MPI_File fh;
  int code,error_len;
  char error_text[MPI_MAX_ERROR_STRING];

  MPI_Init(&argc,&argv);

  code = MPI_File_open(MPI_COMM_WORLD,"file.data",
                       MPI_MODE_RDWR+MPI_MODE_CREATE,MPI_INFO_NULL,&fh);
  if (code != MPI_SUCCESS) {
    MPI_Error_string(code,error_text,&error_len);
    printf("%s\n", error_text);
    MPI_Abort(MPI_COMM_WORLD,42);
  }

  code = MPI_File_close(&fh);
  if (code != MPI_SUCCESS) {
    printf("Error in closing file\n");
    MPI_Abort(MPI_COMM_WORLD,2);
  }

  MPI_Finalize();
}
```

```
> ls -l file.data

-rw-------   1 user       grp    0 Feb 08 12:13 file.data
```

MPI-IO

Mode	Meaning
`MPI_MODE_RDONLY`	Read only
`MPI_MODE_RDWR`	Reading and writing
`MPI_MODE_WRONLY`	Write only
`MPI_MODE_CREATE`	Create the file if it does not exist
`MPI_MODE_EXCL`	Error if creating file that already exists
`MPI_MODE_UNIQUE_OPEN`	File will not be concurrently opened elsewhere
`MPI_MODE_SEQUENTIAL`	File will only be accessed sequentially
`MPI_MODE_APPEND`	Set initial position of all file pointers to end of file
`MPI_MODE_DELETE_ON_CLOSE`	Delete file on close

MPI-IO

Error handling

- The behavior concerning code argument is different for the IO part of MPI.
- It's necessary to check the value of this argument.
- It's possible to change this behaviour with `MPI_File_set_errhandler()`.
- Two error handlers are available : `MPI_ERRORS_ARE_FATAL` and `MPI_ERRORS_RETURN`.
- `MPI_Comm_set_errhandler()` provides a way to change the error handler for the communications.

```
int MPI_File_set_errhandler(MPI_File fh, MPI_Errhandler errhandler)
```

The default behaviour can be changed with `MPI_FILE_NULL` as file handler.

MPI-IO

Data access routines

- MPI-IO proposes a broad range of subroutines for transferring data between files and memory.
- Subroutines can be distinguished through several properties :
    - The position in the file can be specified using an explicit offset (ie. an absolute position relative to the beginning of the file) or using individual or shared file pointers (ie. the offset is defined by the current value of pointers).
    - Data access can be blocking or non-blocking.
    - Sending and receiving messages can be collective (in the communicator group) or noncollective.
- Different access methods may be mixed within the same program.

MPI-IO

Positioning	Synchronism	noncollective	collective
explicit offsets	blocking	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	nonblocking	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_read_at_all_begin MPI_File_read_at_all_end MPI_File_write_at_all_begin MPI_File_write_at_all_end
individual file pointers	blocking	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	nonblocking	MPI_File_iread MPI_File_iwrite	MPI_File_read_all_begin MPI_File_read_all_end MPI_File_write_all_begin MPI_File_write_all_end
shared file pointer	blocking	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	nonblocking	MPI_File_iread_shared MPI_File_iwrite_shared	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

MPI-IO

File Views

- By default, files are treated as a sequence of bytes but access patterns can also be expressed using predefined or derived MPI datatypes.
- This mechanism is called file views and is described in further detail later.
- For now, we only need to know that the views rely on an elementary data type and that the default type is `MPI_BYTE`.

MPI-IO

Explicit Offsets

- Explicit offset operations perform data access directly at the file position, given as an argument.
- The offset is expressed as a multiple of the elementary data type of the current view (therefore, the default offset unit is bytes).
- The datatype and the number of elements in the memory buffer are specified as arguments (ex : `MPI_INT`)

MPI-IO

```c
/* write_at */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,byte_in_integer,nb_values=10,code;
  int values[nb_values];
  MPI_File fh;
  MPI_Offset offset;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  for(i=0;i<nb_valeurs;i++) {values[i] = i+rank*100;}
  printf("process %d :", rank);
  for(i=0;i<nb_valeurs;i++) {printf("%d ",values[i]);}
  printf("\n");

  code = MPI_File_open(MPI_COMM_WORLD,"donnees.dat",
                       MPI_MODE_WRONLY+MPI_MODE_CREATE,MPI_INFO_NULL,&fh);
  if (code != MPI_SUCCESS) {
    printf("Error in opening file\n");
    MPI_Abort(MPI_COMM_WORLD,42); }
  MPI_Type_size(MPI_INT,&bytes_in_integer);
  offset = rank*nb_values*bytes_in_integer;

  MPI_File_set_errhandler(fh,MPI_ERRORS_ARE_FATAL);
  MPI_File_write_at(fh,offset,values,nb_values,MPI_INT,
                    &statut);
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

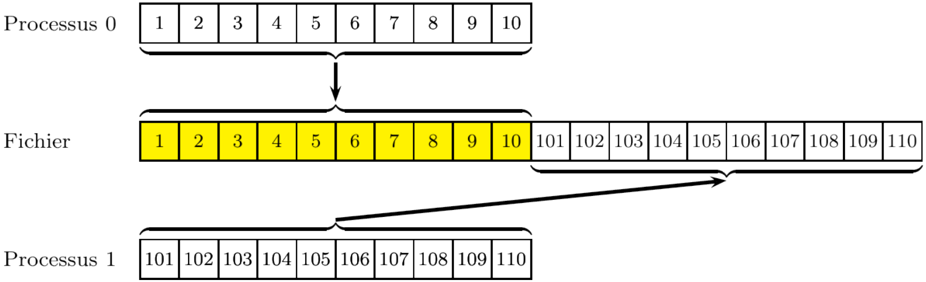


Figure 48 – `MPI_File_write_at()`

```
> mpiexec -n 2 write_at

process 0 :    1,    2,    3,    4,    5,    6,    7,    8,    9,   10
process 1 :  101,  102,  103,  104,  105,  106,  107,  108,  109,  110
```

MPI-IO

```c
/* read_at */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,bytes_in_integer,nb_values=10,code;
  int values[nb_values];
  MPI_File fh;
  MPI_Offset offset;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  code = MPI_File_open(MPI_COMM_WORLD,"data.dat",
                       MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_Type_size(MPI_INT,&bytes_in_integer);
  offset = rank*nb_values*bytes_in_integer;
  MPI_File_read_at(fh,offset,values,nb_values,MPI_INT,
                   &statut);
  printf("process %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

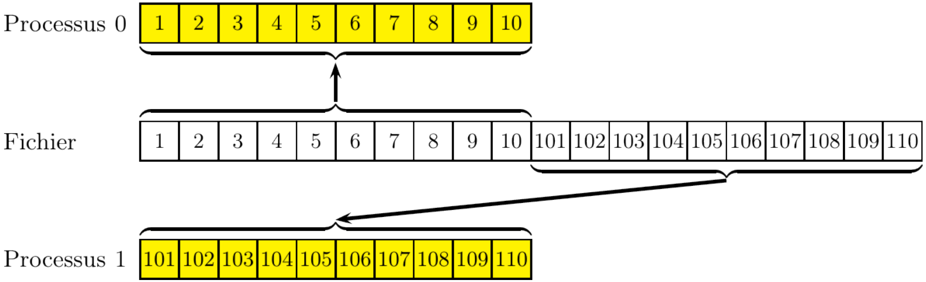


Figure 49 – `MPI_File_read_at()`

```
> mpiexec -n 2 read_at

process 0 :   1,   2,   3,   4,   5,   6,   7,   8,   9,  10
process 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

MPI-IO

Individual file pointers

- MPI maintains one individual file pointer per process per file handle.
- The current value of this pointer implicitly specifies the offset in the data access routines.
- After an individual file pointer operation is initiated, the individual file pointer is updated to point to the next data item.
- The shared file pointer is neither used nor updated.

MPI-IO

```c
/* read01 */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,nb_values=10;
  int values[nb_values];
  MPI_File fh;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
                MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_read(fh,values,6,MPI_INT,&statut);
  MPI_File_read(fh,&(values[6]),4,MPI_INT,&statut);
  printf("proces %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

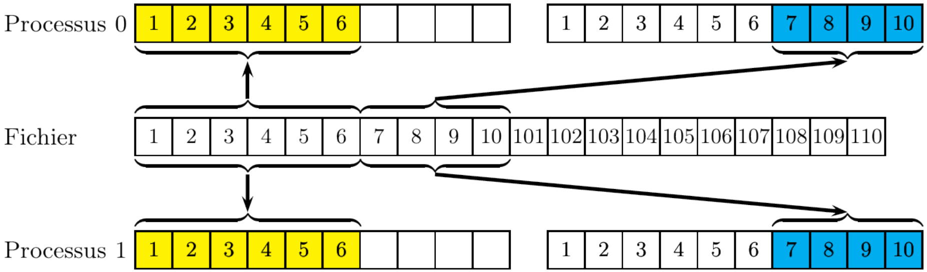


Figure 50 – Example 1 of `MPI_File_read()`

```
> mpiexec -n 2 read01

process 1 :  1,   2,   3,   4,   5,   6,   7,   8,   9,   10
process 0 :  1,   2,   3,   4,   5,   6,   7,   8,   9,   10
```

MPI-IO

```c
/* read02 */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,nb_values=10;
  int values[nb_values];
  MPI_File fh;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
               MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  if (rank == 0) {
    MPI_File_read(fh,values,6,MPI_INT,&statut);
  } else {
    MPI_File_read(fh,values,8,MPI_INT,&statut);
    MPI_File_read(fh,values,5,MPI_INT,&statut); }
  printf("process %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

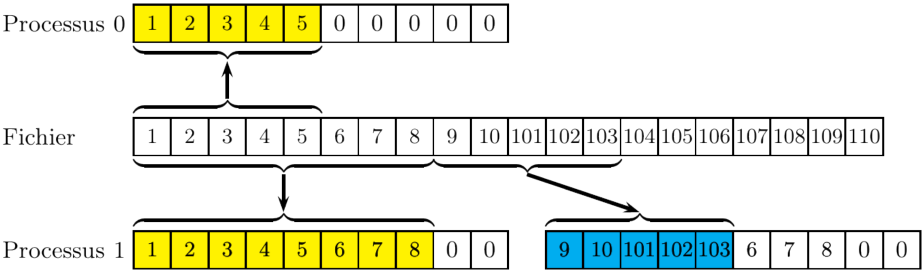


Figure 51 – Example 2 of `MPI_File_read()`

```
> mpiexec -n 2 read02

process 0 :  1,   2,   3,   4,   5,   0,   0,   0
process 1 :  9,  10, 101, 102, 103,   6,   7,   8
```

MPI-IO

Shared file pointer

- MPI maintains only one shared file pointer per collective `MPI_File_open` (shared among processes in the communicator group).
- All processes must use the same file view.
- For the noncollective shared file pointer routines, the serialisation ordering is not deterministic. To enforce a specific order, the user needs to use other synchronisation means or use collective variants.
- After a shared file pointer operation, the shared file pointer is updated to point to the next data item, that is, just after the last one accessed by the operation.
- The individual file pointers are neither used nor updated.

MPI-IO

```c
/* read_shared01 */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,nb_values=10;
  int values[nb_values];
  MPI_File fh;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
                MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_read_shared(fh,values,4,MPI_INT,&statut);
  MPI_File_read_shared(fh,&(valeurs[4]),6,MPI_INT,&statut);
  printf("process %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

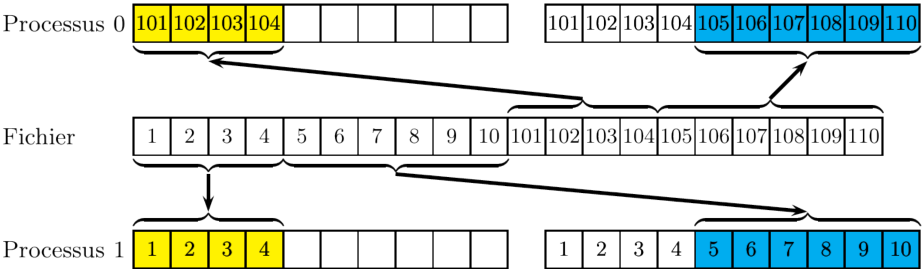


Figure 52 – Example of `MPI_File_read_shared()`

```
> mpiexec -n 2 read_shared01

process  1 :   1,   2,   3,   4,   5,   6,   7,   8,   9,  10
process  0 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

MPI-IO

Collective data access

- Collective operations require the participation of all the processes within the communicator group associated with the file handle.
- Collective operations may perform much better than their noncollective counterparts, as global data accesses have significant potential for automatic optimisation.
- For the collective shared file pointer routines, the accesses to the file will be in the order determined by the ranks of the processes within the group. The ordering is therefore deterministic.

MPI-IO

```c
/* read_at_all */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,bytes_in_integer,nb_values=10,code;
  int values[nb_values];
  MPI_File fh;
  MPI_Offset offset_file;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  code = MPI_File_open(MPI_COMM_WORLD,"data.dat",
                       MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_Type_size(MPI_INT,&bytes_in_integer);
  offset_file = rank*nb_values*bytes_in_integer;
  MPI_File_read_at_all(fh,offset_file,
                       values,nb_values,MPI_INT,&statut);
  printf("process %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

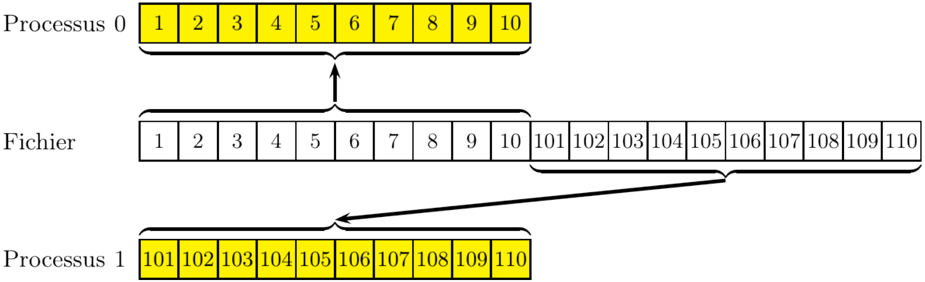


Figure 53 – Example of `MPI_File_read_at_all()`

```
> mpiexec -n 2 read_at_all

process 0 :   1,   2,   3,   4,   5,   6,   7,   8,   9,  10
process 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

MPI-IO

```c
/* read_all01 */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,nb_values=10;
  int values[nb_values];
  MPI_File fh;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
                MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_read_all(fh,values,4,MPI_INT,&statut);
  MPI_File_read_all(fh,&(values[4]),6,MPI_INT,&statut);
  printf("process %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

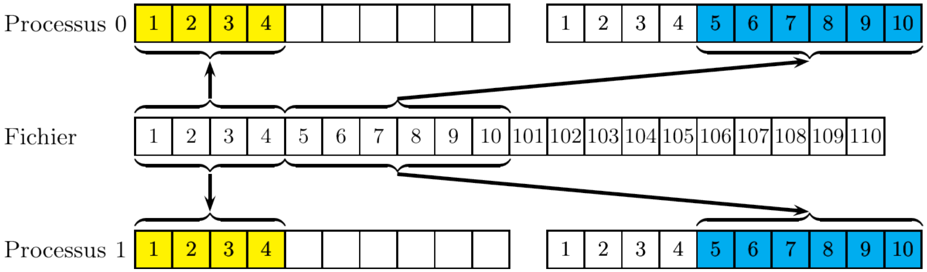


Figure 54 – Example 1 of `MPI_File_read_all()`

```
> mpiexec -n 2 read_all01

process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
process 1 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

MPI-IO

```c
/* read_all02 */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,nb_values=10,index1,index2;
  int values[nb_values];
  MPI_File fh;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
                MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  if (rank == 0) {
    index1=2;
    index2=5;
  } else {
    index1=4;
    index2=8;
  }
  MPI_File_read_all(fh,&(values[index1]),
                    index2-index1+1,MPI_INT,&statut);
  printf("process %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

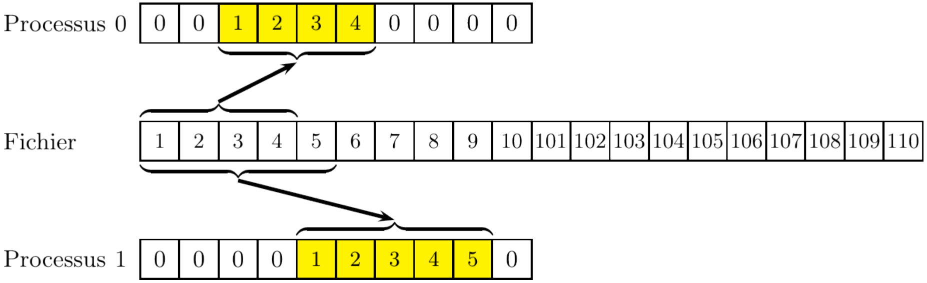


Figure 55 – Example 2 of `MPI_File_read_all()`

```
> mpiexec -n 2 read_all02

process 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
process 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```

MPI-IO

```c
/* read_all03 */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,nb_values=10,index1,index2;
  int values[nb_values];
  MPI_File fh;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
               MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  if (rank == 0) {
    MPI_File_read_all(fh,&(values[2]),4,MPI_INT,&statut);
  } else {
    MPI_File_read_all(fh,&(values[4]),5,MPI_INT,&statut);
  }
  printf("process %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

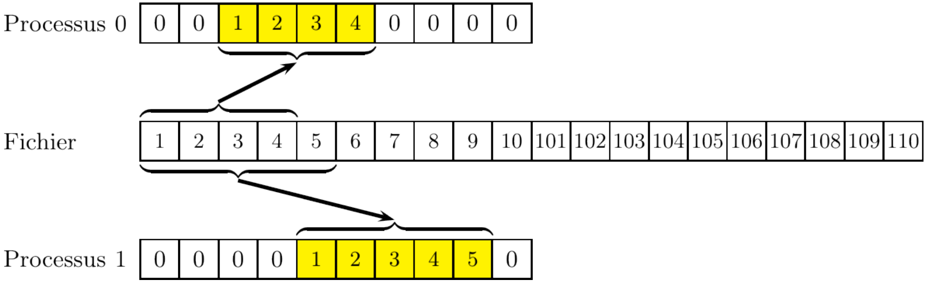


Figure 56 – Example 3 of `MPI_File_read_all()`

```
> mpiexec -n 2 read_all03

process 1 : 0, 0, 0, 0, 1, 2, 3, 4, 5, 0
process 0 : 0, 0, 1, 2, 3, 4, 0, 0, 0, 0
```

MPI-IO

```c
/* read_ordered */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,nb_values=10;
  int values[nb_values];
  MPI_File fh;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
                MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_read_ordered(fh,values,4,MPI_INT,&statut);
  MPI_File_read_ordered(fh,&(values[4]),6,MPI_INT,&statut);
  printf("process %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

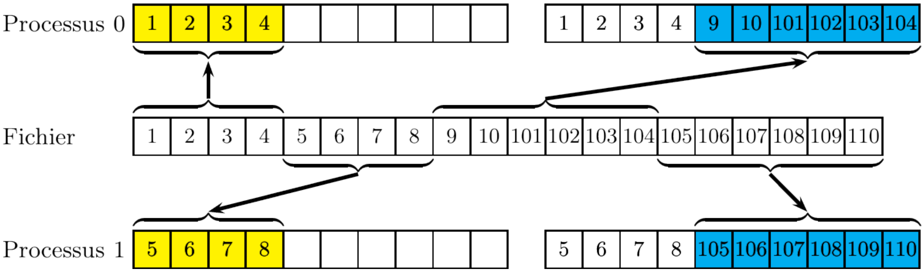


Figure 57 – Example of `MPI_File_ordered()`

```
> mpiexec -n 2 read_ordered

process 1 : 5, 6, 7, 8, 105, 106, 107, 108, 109, 110
process 0 : 1, 2, 3, 4,   9,  10, 101, 102, 103, 104
```

MPI-IO

Positioning the file pointers

- `MPI_File_get_position()` and `MPI_File_get_position_shared()` returns the current position of the individual pointers and the shared file pointer (respectively).
- `MPI_File_seek()` and `MPI_File_seek_shared()` updates the file pointer values by using the following possible modes :
  - `MPI_SEEK_SET` : The pointer is set to offset.
  - `MPI_SEEK_CUR` : The pointer is set to the current pointer position plus offset.
  - `MPI_SEEK_END` : The pointer is set to the end of file plus offset.
- With `MPI_SEEK_CUR` and `MPI_SEEK_END`, the offset can be negative, which allows seeking backwards.

MPI-IO

```c
/* seek */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,bytes_in_integer,nb_values=10;
  int values[nb_values];
  MPI_File fh;
  MPI_Status statut;
  MPI_Offset offset_file;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
              MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_read(fh,values,3,MPI_INT,&statut);
  MPI_Type_size(MPI_INT,&bytes_in_integer);
  offset_file = 8*bytes_in_integer;
  MPI_File_seek(fh,offset_file,MPI_SEEK_CUR);
  MPI_File_read(fh,&(values[3]),3,MPI_INT,&statut);
  offset_file = 4*bytes_in_integer;
  MPI_File_seek(fh,offset_file,MPI_SEEK_SET);
  MPI_File_read(fh,&(valeurs[6]),4,MPI_INT,&statut);
  printf("process %d : ",rank);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

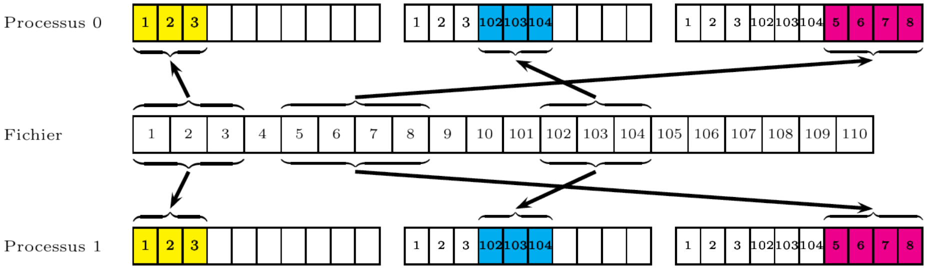


Figure 58 – Example of `MPI_File_seek()`

```
> mpiexec -n 2 seek

process 1 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
process 0 : 1, 2, 3, 102, 103, 104, 5, 6, 7, 8
```

MPI-IO

Nonblocking Data Access

- Nonblocking operations enable overlapping of I/O operations and computations.
- The semantic of nonblocking I/O calls is similar to the semantic of nonblocking communications between processes.
- A first nonblocking I/O call initiates the I/O operation and a separate request call is needed to complete the I/O requests (`MPI_Test()`, `MPI_Wait()`, etc.).

MPI-IO

```c
/* iread_at */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,bytes_in_integer,finish,nb_values=10,nb_iterations=0;
  int values[nb_values];
  MPI_File fh;
  MPI_Status statut;
  MPI_Offset offset_file;
  MPI_Request request;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

MPI-IO

```
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
                MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_Type_size(MPI_INT,&bytes_in_integer);
  offset_file = rank*nb_values*bytes_in_integer;
  MPI_File_iread_at(fh,offset_file,
                    values,nb_values,MPI_INT,&request);
  while( nb_iterations < 5000) {
    nb_iterations = nb_iterations+1;
    /* Calculs recouvrant le temps demande par l'operation de lecture */
    /* */
    MPI_Test(&request,&finish,&statut);
    if (finish) break;
  }
  if ( !finish) MPI_Wait(&request,&statut);
  printf("After %d iterations, process %d : ",nb_iterations, rang);
  for (i=0;i<nb_values;i++) {printf("%d ",values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

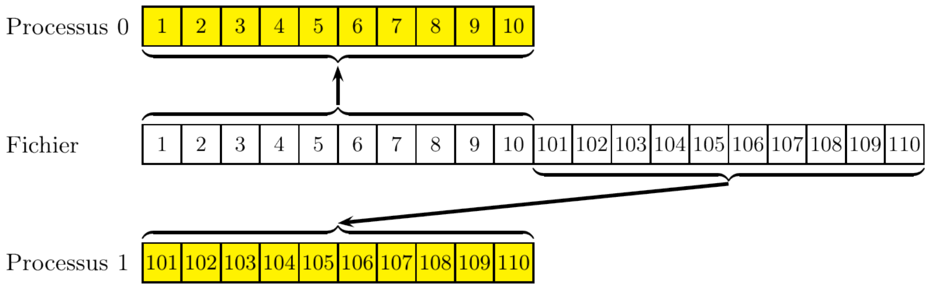


Figure 59 – Example of `MPI_File_iread_at()`

```
> mpiexec -n 2 iread_at

After 1 iterations, process 0 : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
After 1 iterations, process 1 : 101, 102, 103, 104, 105, 106, 107, 108, 109, 110
```

MPI-IO

```c
/* iwrite */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,finish,nb_values=10,nb_iterations=0;
  int values[nb_values],temp[nb_values];
  MPI_File fh;
  MPI_Request request;

  MPI_Init(&argc,&argv);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
                MPI_MODE_WRONLY+MPI_MODE_CREATE,MPI_INFO_NULL,&fh);
  for(i=0;i<nb_values;i++) temp[i]=values[i];
  MPI_File_seek(fh,offset,MPI_SEEK_SET);
  MPI_File_iwrite(fh,temp,nb_values,MPI_INT,&request);
  while( nb_iterations < 5000) {
    nb_iterations = nb_iterations+1;
    MPI_Test(&request,&finish,MPI_STATUS_IGNORE);
    if (finish) {
      for(i=0;i<nb_values;i++) temp[i]=values[i];
      MPI_File_seek(fh,offset,MPI_SEEK_SET);
      MPI_File_iwrite(fh,temp,nb_values,MPI_INT,&request); }
  }
  MPI_Wait(&request,MPI_STATUS_IGNORE);
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

Split collective data access routines

- The split collective routines support a restricted form of nonblocking operations for collective data access.
- A single collective operation is split into two parts : a begin routine and an end routine.
- On any MPI process, each file handle can only have one active split collective operation at any time.
- Collective I/O operations are not permitted concurrently with a split collective access on the same file handle (but non-collective I/O are allowed). The buffer passed to a begin routine must not be used while the routine is outstanding.

MPI-IO

```c
/* read_ordered_begin_end */
#include <mpi.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  int rank,i,finish,nb_values=10,nb_iterations=0;
  int values[nb_values],temp[nb_values];
  MPI_File fh;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_File_open(MPI_COMM_WORLD,"data.dat",
              MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);
  MPI_File_read_ordered_begin(fh,values,4,MPI_INT);
  printf("Process : %d\n",rank);
  MPI_File_read_ordered_end(fh,values,&statut);
  printf("process %d : %d %d %d %d\n",
         rank,values[0],values[1],values[2],values[3]);
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

MPI-IO

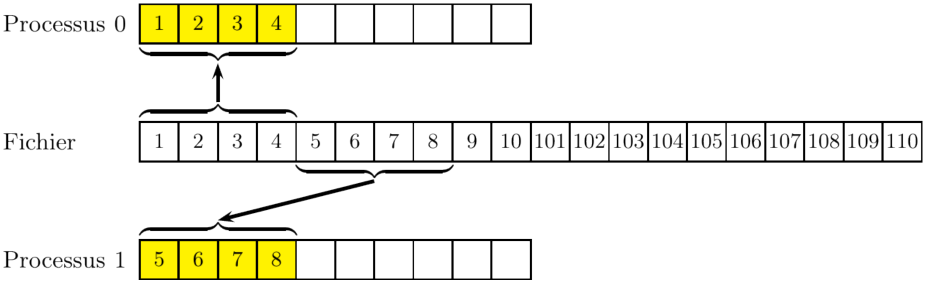


Figure 60 – Example of `MPI_File_read_ordered_begin()`

```
> mpiexec -n 2 read_ordered_begin_end

Process      : 0
process 0 : 1, 2, 3, 4
Process      : 1
process 1 : 5, 6, 7, 8
```

MPI Hands-On – Exercise 7 : Read an MPI-IO file

- We have a binary file data.dat with 484 integer values.
- With 4 processes, it consists of reading the 121 first values on process 0, the 121 next on the process 1, and so on.
- We will use 4 different methods :
  - Read via explicit offsets, in individual mode
  - Read via shared file pointers, in collective mode
  - Read via individual file pointers, in individual mode
  - Read via shared file pointers, in individual mode
- To compile use make, to execute use make exe, and to verify the results use make verification which build figure file corresponding to the four cases.

MPI 3.x

MPI 3.x

Extension

- Nonblocking collectives communications
- Neighborhood collective communications
- Fortran 2008 binding
- End of C++ bindings
- One-sided communication extension

MPI 3.x

Nonblocking collectives

- Nonblocking version of collective communications
- With an I (immediate) before : `MPI_Ireduce()`, `MPI_Ibcast()`, ...
- Wait with `MPI_Wait()`, `MPI_Test()` calls and all their variants
- No match between blocking and nonblocking
- The *status* argument retrieved by `MPI_Wait()` has an undefined value for `MPI_SOURCE` and `MPI_TAG`
- For a given communicator, the call order must be the same

```
int MPI_Ibarrier(MPI_Comm comm, MPI_Request *request)
```

MPI 3.x

Neighborhood collective communications

- `MPI_Neighbor_allgather()` and the V variation, `MPI_Neighbor_alltoall()` and the V and W variations
- Plus the nonblocking versions

```
MPI_Neighbor_allgather(u,1,MPI_INT,v,1,MPI_INT,comm2d);
```

MPI 3.x

mpi_f08 module

- Usable with the module mpi_f08
- With this module, the last argument (*code*) is *optional*
- MPI objects have a specific type and are no longer `INTEGER`
- This is the preferred module from now on

For example, for `MPI_RECV()` the interface with the classic module is :

```
<type> buf(*)
INTEGER :: count, datatype, source, tag, comm, ierror
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: msgstatus
```

With the mpi_f08 module :

```
TYPE(*), DIMENSION(..) :: buf
INTEGER                :: count, source, tag
TYPE(MPI_DATATYPE)     :: datatype
TYPE(MPI_COMM)         :: comm
TYPE(MPI_STATUS)       :: msgstatus
INTEGER, optional      :: ierror
```

MPI 3.x

mpi_f08 module

These new types are in fact INTEGER

```
TYPE, BIND(C) :: MPI_COMM
  INTEGER :: MPI_VAL
END TYPE MPI_COMM
```

falcutative functionalities in mpi_f08

- If MPI_SUBARRAYS_SUPPORTED is set to *true*, it's possible to use Fortran subarrays in nonblocking calls.
- If MPI_ASYNC_PROTECTS_NONBLOCKING is set to *true*, the send and/or receive arguments are *asynchronous* in nonblocking interfaces.

```
          call MPI_ISEND(buf,...,req)
          ...
          call MPI_WAIT(req,...)
          if (.not. MPI_ASYNC_PROTECTS_NONBLOCKING) call MPI_F_SYNC_REG(buf)
          buf = val2
```

MPI 3.x

Removal of C++ binding

Replace by either the C binding or *Boost.MPI*

MPI 3.x

One-sided communication extension

- New operation `MPI_Get_accumulate()`
- New operation `MPI_Fetch_and_op()` : an `MPI_Get_accumulate()` which works with only one element
- And the new operation `MPI_Compare_and_swap()`
- New function `MPI_Win_allocate()` for allocating and creating the window in one call
- New function `MPI_Win_allocate_shared()` for creating the window in shared memory

```
MPI_Comm_split_type(MPI_COMM_WORLD,MPI_COMM_TYPE_SHARED,key,MPI_INFO_NULL,&commnode);
MPI_Win_allocate_shared(localsize,displacement,MPI_INFO_NULL,commnode,&ptr,&win);
MPI_Win_shared_query(win,rank,&distant_size,&disp,&distantptr);
```

MPI 3.x

MPI 3.1

- New functions `MPI_Aint_add()` and `MPI_Aint_diff()` for manipulating addresses
- New functions `MPI_File_iwrite_at_all()` `MPI_File_iread_at_all()` `MPI_File_iread_all()` and `MPI_File_iwrite_all()`

MPI 4.x

MPI 4.x

Adding

- Large count
- Partitioned communication
- MPI Session
- Others

Large count

- Count parameters were in `integer` or `int`.
- MPI 4.0 add new functions with `MPI_Count` instead.
- In *C* these new functions have `_c` at the end.

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

- In *Fortran* count in `integer` can be changed in `integer(kind=MPI_COUNT_KIND)`
- Only available with the `mpi_f08` module
- No change in the name of function with polymorphism

```
MPI_Send(buf,count,datatype,dest,tag,comm,ierror)
TYPE(*), DIMENSION(..), INTENT(IN) :: buf
INTEGER, INTENT(IN)                :: count, dest, tag
TYPE(MPI_Datatype), INTENT(IN)     :: datatype
TYPE(MPI_Comm), INTENT(IN)         :: comm
INTEGER, OPTIONAL, INTENT(OUT)     :: ierror

MPI_Send(buf,count,datatype,dest,tag,comm,ierror)
TYPE(*), DIMENSION(..), INTENT(IN)          :: buf
INTEGER(KIND=MPI_COUNT_KIND), INTENT(IN)    :: count
TYPE(MPI_Datatype), INTENT(IN)              :: datatype
INTEGER, INTENT(IN)                         :: dest, tag
TYPE(MPI_Comm), INTENT(IN)                  :: comm
INTEGER, OPTIONAL, INTENT(OUT)              :: ierror
```

Partitioned communication

- Multiple contribution to a communication.
- Usefull in hybrid.
- Init with `MPI_Psend_init()` or `MPI_Precv_init()` by providing the count by partition and the number of partition.
- `MPI_Start()` to start the communication.
- `MPI_Pready()` to indicate that a partition is ready.
- Could not mix `MPI_Recv()` and `MPI_Psend_init()`.
- `MPI_Wait()` to wait for the end of communication.
- `MPI_Parrived()` to know if a partition has been received.

Session

- A way to do multiple `MPI_Init()`/`MPI_Finalize()`.
- `MPI_Session_init()` to start a session.
- `MPI_Session_finalize()` to end a session.
- No more `MPI_COMM_WORLD`.
- *Process Sets* : `mpi://WORLD` and `mpi://SELF`.
- `MPI_Group_from_session_pset()` to make a group from a *pset*.
- `MPI_Comm_create_from_group()` to make a communicator from a group.
- `MPI_Session_get_num_psets()` to known the number of *pset* available.
- `MPI_Session_get_nth_pset()` to get the name of a *pset*.

Others

- Add of `MPI_Isendrecv` and `MPI_Isendrecv_replace`.
- Add persistent collective communication.
- Add option `mpi_initial_errhandler` for *mpiexec* to specify the default errhandler.

MPI-IO Views

The View Mechanism

- File Views is a mechanism which accesses data in a high-level way. A view describes a template for accessing a file.
- The view that a given process has of an open file is defined by three components : the elementary data type, file type and an initial displacement.
- The view is determined by the repetition of the filetype pattern, beginning at the displacement.

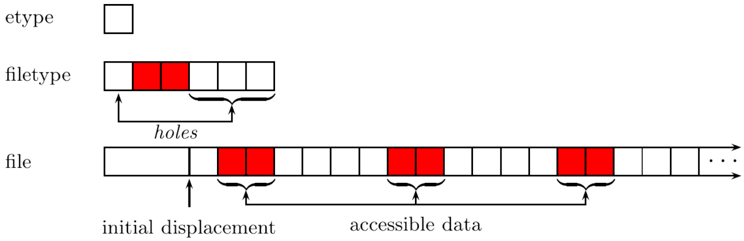


Figure 61 – Tiling a file with a filetype

The View Mechanism

- File Views are defined using MPI datatypes.
- Derived datatypes can be used to structure accesses to the file. For example, elements can be skipped during data access.
- The default view is a linear byte stream (displacement is zero, etype and filetype equal to `MPI_BYTE`).

Multiple Views

- Each process can successively use several views on the same file.
- Each process can define its own view of the file and access complementary parts of it.

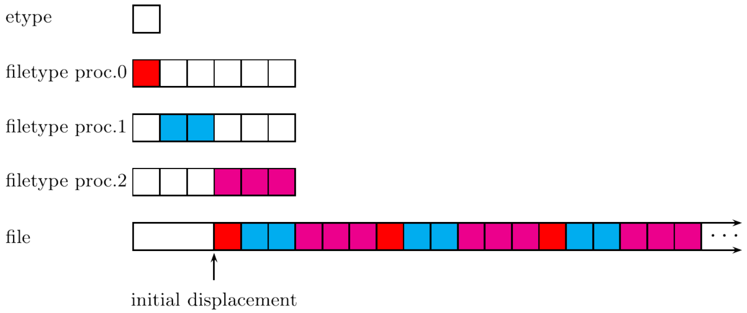


Figure 62 – Separate views, each using a different filetype, can be used to access the file

Limitations :

- Shared file pointer routines are not useable except when all the processes have the same file view.
- If the file is opened for writing, the different views may not overlap, even partially.

Changing the process's view of the data in the file : MPI_File_set_view()

```
int MPI_File_set_view(MPI_File fh, MPI_Offset displacement,MPI_Datatype etype,
                      MPI_Datatype filetype,char *mode,MPI_Info info)
```

- This operation is collective throughout the file handle.The values for the initial displacement and the filetype may vary between the processes in the group. The extents of elementary types must be identical.
- In addition, the individual file pointers and the shared file pointer are reset to zero.

Notes :

- The datatypes passed in must have been committed using the `MPI_Type_commit()` subroutine.
- MPI defines three data representations (mode) : "native", "internal" or "external32".

Subarray datatype constructor

A derived data type useful to create a filetype is the “subarray” type, that we introduce here. This type allows creating a subarray from an array and can be defined with the `MPI_Type_create_subarray()` subroutine.

The shape of an array is a vector for which each dimension equals the number of elements in each dimension. For example, the array `T(10,0:5,-10:10)` (or `T[10][6][21]`), its shape is the (10,6,21) vector.

```
int MPI_Type_create_subarray(int nb_dims,const int shape_array[],const int shape_sub_array[],
                             const int coord_start[],int order,MPI_Datatype old_type,
                             MPI_Datatype *new_type)
```

Explanation of the arguments

- nb_dims : number of dimension of the array
- shape_array : shape of the array from which a subarray will be extracted
- shape_sub_array : shape of the subarray
- coord_start : start coordinates if the indices of the array start at 0. For example, if we want the start coordinates of the subarray to be `array(2,3)`, we must have `coord_start(:)=(/ 1,2 /)`
- order : storage order of elements
  - `MPI_ORDER_FORTRAN` for the ordering used by Fortran arrays (column-major order)
  - `MPI_ORDER_C` for the ordering used by C arrays (row-major order)

Exchanges between 2 process with subarray

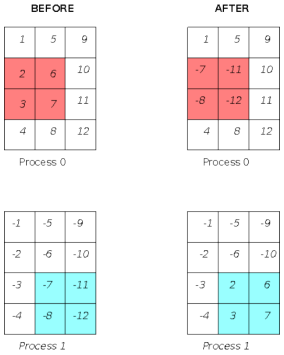


Figure 63 – Exchanges between the two processes

Exchanges between the two processes

```c
/* subarray */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,i,j;
  int nb_lines=4,nb_columns=3,sign=1,nb_dims=2,tag=1000;
  int tab[nb_lines][nb_columns],shape_array[nb_dims];
  int shape_subarray[nb_dims], coord_start[nb_dims];
  MPI_Datatype type_subarray;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  /* Initialization of the tab array on each process */
  if (rank == 1) sign=-1;
  for(i=0;i<nb_lines;i++) {
    for(j=0;j<nb_columns;j++) {
      tab[i][j] = sign*(1+i+nb_lines*j); } }
  /* Shape of the tab array from which a subarray will be extracted */
  shape_array[0] = nb_lines; shape_array[1] = nb_columns;
  /* Shape of the subarray */
  shape_subarray[0] = 2; shape_subarray[1] = 2;
  /* Start coordinates of the subarray */
  coord_start[0]= rank+1; coord_start[1]= rank;
  /* Creation of type_subarray  */
  MPI_Type_create_subarray(nb_dims,shape_array,shape_subarray,coord_start,
                           MPI_ORDER_C,MPI_INT,&type_subarray);
  MPI_Type_commit(&type_subarray);

  /* Exchange of the subarray */
  MPI_Sendrecv_replace(tab,1,type_subarray,(rank+1)%2,tag,
                       (rank+1)%2,tag,MPI_COMM_WORLD,&statut);

  MPI_Type_free(&type_subarray);
  MPI_Finalize();
}
```

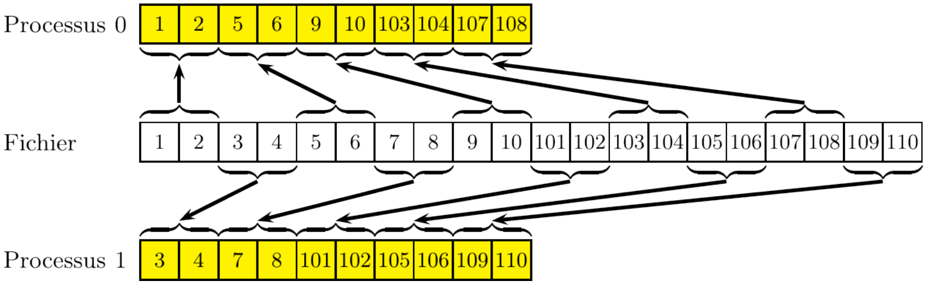


Figure 64 – Example 1 : Reading non-overlapping sequences of data segments in parallel

```
> mpiexec -n 2 read_view01

process 1 : 3, 4, 7, 8, 101, 102, 105, 106, 109, 110
process 0 : 1, 2, 5, 6,   9,  10, 103, 104, 107, 108
```

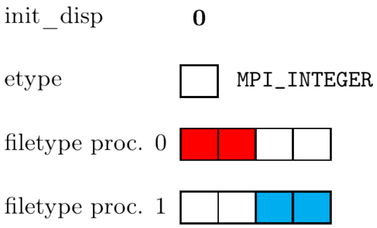


Figure 65 – Example 1 (continued)

```
shape_array[0] = 4;shape_subarray[0] = 2;
if (rank == 0) coord[0]=0;
if (rank == 1) coord[0]=2;
MPI_Type_create_subarray(1,shape_array,shape_subarray,coord,MPI_ORDER_C,MPI_INT,&filetype);
MPI_Type_commit(&filetype);
init_displacement=0
MPI_File_set_view(fh,init_displacement,MPI_INT,filetype,"native",MPI_INFO_NULL);
```

```c
/* read_view01 */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,i,coord,n1=4,n2=2,nb_values=10;
  int values[nb_values];
  MPI_Datatype filetype;
  MPI_File fh;
  MPI_Offset init_displacement;
  MPI_Status statut;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  if (rank == 0) coord=0;
  if (rank == 1) coord=2;
  MPI_Type_create_subarray(1,&n1,&n2,&coord,MPI_ORDER_C,MPI_INT,&filetype);
  MPI_Type_commit(&filetype);

  MPI_File_open(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL,
              &fh);
  init_displacement=0;
  MPI_File_set_view(fh,init_displacement,MPI_INT,filetype,
                  "native",MPI_INFO_NULL);
  MPI_File_read(fh,values,nb_values,MPI_INT,&statut);
  printf("process %d :",rank);
  for(i=0;i<nb_values;i++) {printf("%d ", values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```

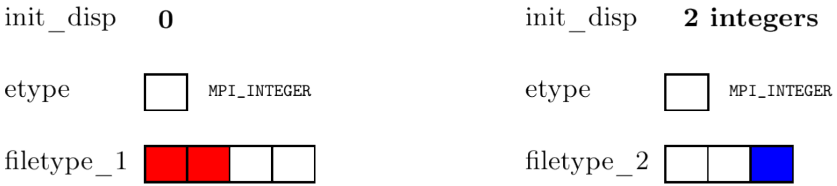
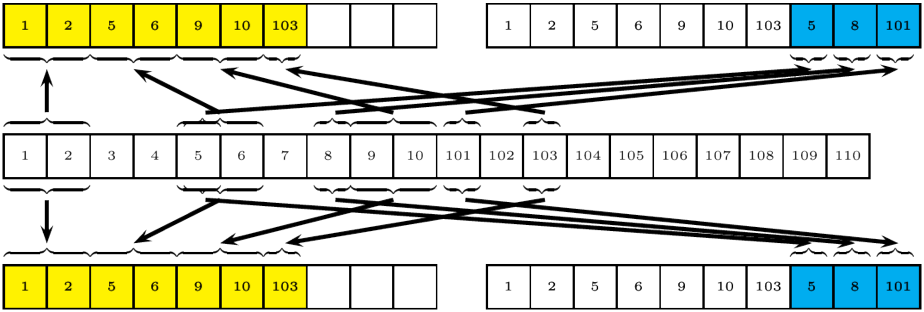


Figure 66 – Example 2 : Reading data using successive views

```c
/* read_view02 */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,n1,n2,coord,bytes_in_integer,nb_values=10,values[nb_values],i;
  MPI_Datatype filetype_1,filetype_2;
  MPI_File fh;
  MPI_Offset init_displacement;
  MPI_Status statut;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  n1=4;n2=2;coord=0;
  MPI_Type_create_subarray(1,&n1,&n2,&coord,MPI_ORDER_C,MPI_INT,&filetype_1);
  MPI_Type_commit(&filetype_1);
  n1=3;n2=1;coord=2;
  MPI_Type_create_subarray(1,&n1,&n2,&coord,MPI_ORDER_C,MPI_INT,&filetype_2);
  MPI_Type_commit(&filetype_2);

  MPI_File_open(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL,&fh);

  /* Read using the first view */
  init_displacement = 0;
  MPI_File_set_view(fh,init_displacement,MPI_INT,filetype_1,
                    "native",MPI_INFO_NULL);
  MPI_File_read(fh,values,4,MPI_INT,&statut);
  MPI_File_read(fh,&(values[4]),3,MPI_INT,&statut);

  /* Read using the second view */
  MPI_Type_size(MPI_INT,&bytes_in_integer);
  init_displacement = 2*bytes_in_integer;
  MPI_File_set_view(fh,init_displacement,MPI_INT,filetype_2,
                     "native",MPI_INFO_NULL);
  MPI_File_read(fh,&(values[7]),3,MPI_INT,&statut);
  printf("process %d :",rang);
  for(i=0;i<nb_valeurs;i++) {printf("%d ", valeurs[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```



```
> mpiexec -n 2 read_view02

process 1 : 1, 2, 5, 6, 9, 10, 103, 5, 8, 101
process 0 : 1, 2, 5, 6, 9, 10, 103, 5, 8, 101
```

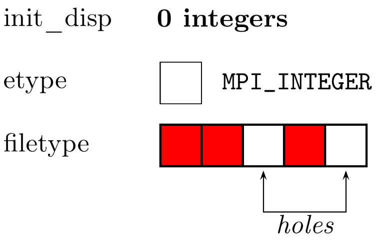
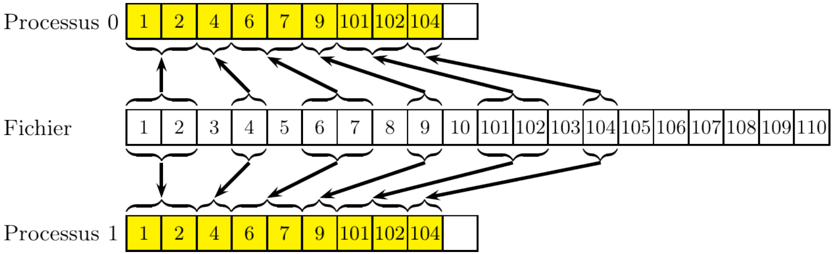


Figure 67 – Example 3 : Dealing with holes in datatypes

```c
/* read_view03_indexed */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int rank,bytes_in_integer,nb_values=9,values[nb_values],i;
  int blocklens[2],displacements[2];
  MPI_Datatype filetype_temp,filetype;
  MPI_Aint lb,extent;
  MPI_File fh;
  MPI_Offset init_displacement;
  MPI_Status statut;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  displacements[0]=0; displacements[1]=3;blocklens[0]=2;blocklens[1]=1;
  MPI_Type_indexed(2,blocklens,displacements,MPI_INT,&filetype_temp);

  /* Motif : type MPI d'etendu 5*MPI_INT */
  MPI_Type_size(MPI_INT,&bytes_in_integer);
  MPI_Type_get_extent(filetype_temp,&lb,&extent);
  extent = extent+bytes_in_integer;
  MPI_Type_create_resized(filetype_temp,lb,extent+lb,&filetype);
  MPI_Type_commit(&filetype);

  MPI_File_open(MPI_COMM_WORLD,"data.dat",MPI_MODE_RDONLY,MPI_INFO_NULL,
                &fh);

  init_displacement=0;
  MPI_File_set_view(fh,init_displacement,MPI_INT,filetype,
                    "native",MPI_INFO_NULL);
  MPI_File_read(fh,values,9,MPI_INT,&statut);

  printf("process %d :",rank);
  for(i=0;i<nb_values;i++) {printf("%d ", values[i]);} printf("\n");
  MPI_File_close(&fh);
  MPI_Finalize();
}
```



```
> mpiexec -n 2 read_view03

process 0 : 1, 2, 4, 6, 7, 9, 101, 102, 104
process 1 : 1, 2, 4, 6, 7, 9, 101, 102, 104
```

```
/* read_view03_struct */

  shape_array=3;shape_subarray=2;coord=0;
  MPI_Type_create_subarray(1,&shape_array,&shape_subarray,&coord,MPI_ORDER_C,MPI_INT,&temp_filetype1);
  shape_array=2;shape_subarray=1;
  MPI_Type_create_subarray(1,&shape_array,&shape_subarray,&coord,MPI_ORDER_C,MPI_INT,&temp_filetype2);
  MPI_Type_size(MPI_INT,&bytes_in_integer);
  displacements[0]=0;displacements[1]=3*bytes_in_integer;
  block[0]=1;block[1]=1;type[0]=temp_filetype1;type[1]=temp_filetype2;
  MPI_Type_create_struct(2,block,displacements,type,&filetype);
  MPI_Type_commit(filetype);
```

MPI-IO Views

Conclusion

MPI-IO offers a high-level interface and a very large set of functionalities. It is possible to carry out complex operations and take advantage of optimizations implemented in the library. MPI-IO also offers good portability

Advice

- The use of explicitly positioned subroutines in files should be reserved for special cases since the implicit use of individual pointers with views provides a higher level interface.
- When the operations involve all the processes (or a subset identifiable by an MPI sub-communicator), it is generally necessary to favor the collective form of the operations.
- Exactly as for the processing of messages when these represent an important part of the application, non-blocking is a privileged way of optimization to be implemented by programmers, but this should only be implemented after ensuring the correctness of behavior of the application in blocking mode.

Conclusion

Conclusion

Conclusion

- Use blocking point-to-point communications before going to nonblocking communications. It will then be necessary to try to overlap computations and communications.
- Use the blocking I/O functions before going to nonblocking I/O. Similarly, it will then be necessary to overlap I/O-computations.
- Write the communications as if the sends were synchronous (`MPI_Ssend()`).
- Avoid the synchronization barriers (`MPI_Barrier()`), especially on the blocking collective functions.
- MPI/OpenMP hybrid programming can bring gains of scalability. However, in order for this approach to function well, it is obviously necessary to have good OpenMP performance inside each MPI process. A hybrid course is given at IDRIS (`https://cours.idris.fr`).

MPI Hands-On – Exercise 8 : Poisson's equation

Resolution of the following Poisson equation :

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} & = f(x,y) \quad \text{in } [0,1]\text{x}[0,1] \\ u(x,y) & = 0. \quad \text{on the boundaries} \\ f(x,y) & = 2.\left(x^2 - x + y^2 - y\right) \end{cases}$$

We will solve this equation with a domain decomposition method :

- The equation is discretized on the domain with a finite difference method.
- The obtained system is resolved with a Jacobi solver.
- The global domain is split into sub-domains.

The exact solution is known and is $u_{exact}(x,y) = xy\,(x-1)\,(y-1)$

MPI Hands-On – Exercise 8 : Poisson's equation

To discretize the equation, we define a grid with a set of points (x_i, y_j)

$$
\begin{aligned}
x_i &= \quad i\ h_x \quad \text{for } i = 0, \ldots, ntx + 1 \\
y_j &= \quad j\ h_y \quad \text{for } j = 0, \ldots, nty + 1
\end{aligned}
$$

$$
\begin{aligned}
h_x &= \frac{1}{(ntx + 1)} \\
h_y &= \frac{1}{(nty + 1)}
\end{aligned}
$$

h_x :	x-wise step
h_y :	y-wise step
ntx :	number of x-wise interior points
nty :	number of y-wise interior points

In total, there are *ntx+2* points in the *x* direction and *nty+2* points in the *y* direction.

MPI Hands-On – Exercise 8 : Poisson's equation

- Let u_{ij} be the estimated solution at position $x_i = ih_x$ and $x_j = jh_y$.
- The Jacobi solver consist of computing :

$$u_{ij}^{n+1} = \quad c_0(c_1(u_{i+1j}^n + u_{i-1j}^n) + c_2(u_{ij+1}^n + u_{ij-1}^n) - f_{ij})$$

$$\text{with :} \quad c_0 = \frac{1}{2}\frac{h_x^2 h_y^2}{h_x^2 + h_y^2}$$

$$c_1 = \frac{1}{h_x^2}$$

$$c_2 = \frac{1}{h_y^2}$$

MPI Hands-On – Exercise 8 : Poisson's equation

- In parallel, the interface values of subdomains must be exchanged between the neighbours.
- We use ghost cells as receive buffers.

MPI Hands-On – Exercise 8 : Poisson's equation

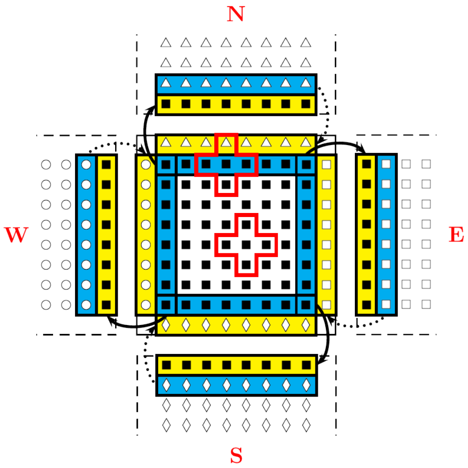


Figure 68 – Exchange points on the interfaces

MPI Hands-On – Exercise 8 : Poisson’s equation

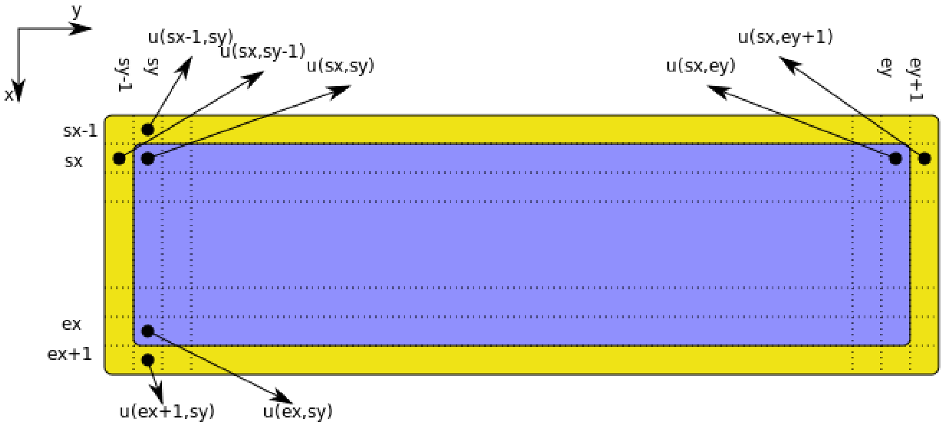


Figure 69 – Numeration of points in different sub-domains

MPI Hands-On – Exercise 8 : Poisson’s equation

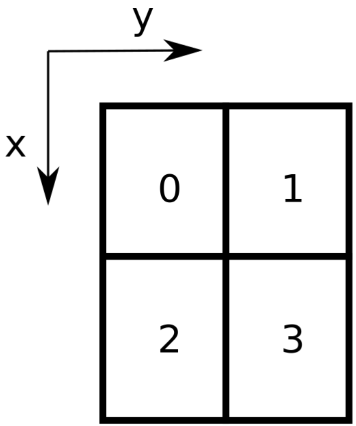


Figure 70 – Process rank numbering in the sub-domains

MPI Hands-On – Exercise 8 : Poisson's equation

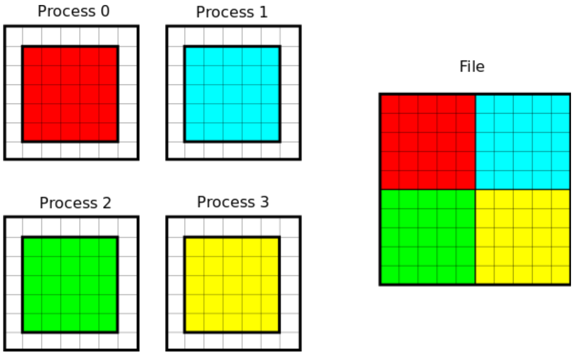


Figure 71 – Writing the global matrix u in a file

You need to :

- Define a view, to see only the owned part of the global matrix u ;
- Define a type, in order to write the local part of matrix u(without interfaces) ;
- Apply the view to the file ;
- Write using only one call.

MPI Hands-On – Exercise 8 : Poisson's equation

- Initialisation of the MPI environment.
- Creation of the 2D Cartesian topology
- Determination of the array indexes for each sub-domain.
- Determination of the 4 neighbour processes for each sub-domain.
- Creation of two derived datatypes, *type_line* and *type_column*.
- Exchange the values on the interfaces with the other sub-domains.
- Computation of the global error. When the global error is lower than a specified value (machine precision for example), we consider that we have reached the exact solution.
- Collecting of the global matrix u (the same one as we obtained in the sequential) in an MPI-IO file `data.dat`.

MPI Hands-On – Exercise 8 : Poisson's equation

- A skeleton of the parallel version is proposed : It consists of a main program (`poisson.c`) and several subroutines. All the modifications have to be done in the `parallel.c` file.
- To compile use make, to execute use make exe. To verify the results, use make verification which runs a reading program of the `data.dat` file and compares it with the sequential version.