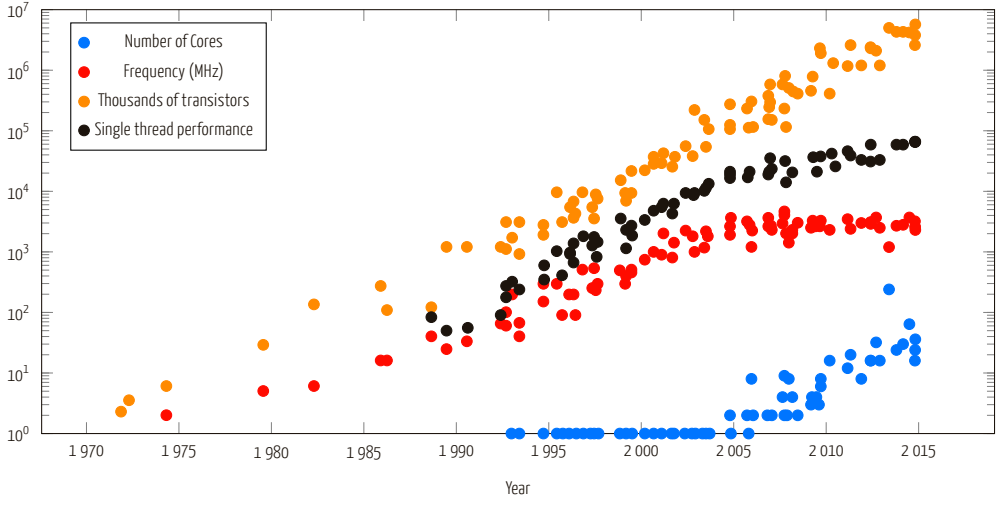
Message Passing Interface
Distributed-Memory Parallel Programming

Orian Louant
orian.louant@uliege.be

October 26, 2021

Motivations for Parallel Computing



Motivations for Parallel Computing

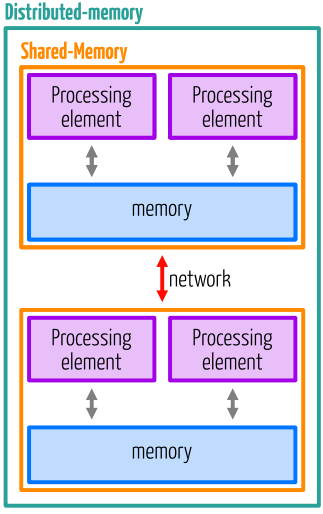
Most applications today are parallel applications

- in the years 2000's the CPU manufacturers have run out of room for boosting CPU performance
- instead of driving clock speeds and straight-line instruction throughput higher, they turn to hyperthreading and multicore architectures

In HPC in particular, it's crucial to be able to run your application in parallel

- in HPC, recent high-end CPUs have high core count: 36-64 cores
- high core count but lower clock, single core performance of an HPC CPU can be worse than your laptop CPU with turbo boost

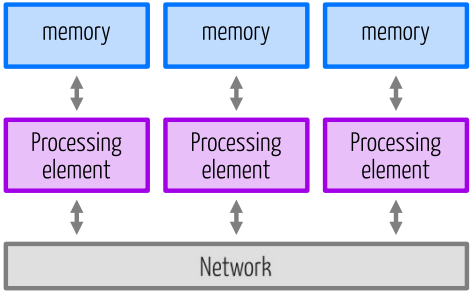
Types of Parallel Systems



- multiple compute nodes: distributed memory
- in HPC the dominant model for distributed memory programming is MPI, a standard that was introduced by the MPI Forum in May 1994 and updated in June 1995, last version 4.0 (2021).

- single compute node: shared memory
- in HPC the dominant model for shared-memory programming is OpenMP, a standard that was introduced in 1997 (Fortran) and 1998 (C/C++), last version is 5.1 (2020)

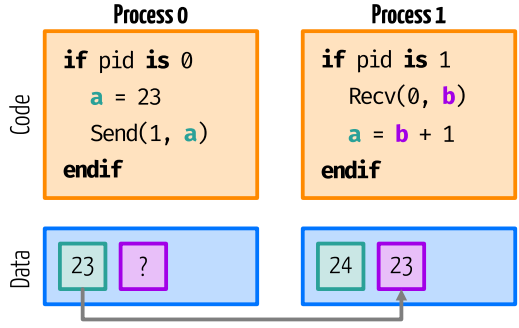
Distributed-Memory



- MPI use processes to parallelize applications for distributed-memory systems
- the system consists of processing elements (nodes, CPUs, cores) and memory
- the processing elements have their own private memory
- the processing elements cannot have a direct access to the memory of the other processing elements
- the processing elements can communicate via a network

Single Program Multiple Data

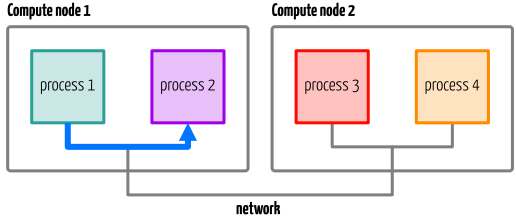
- The Single Program Multiple Data (SPMD) model is the main model for application message passing
- Multiple parallel processes run the same executable, they are identified by a unique identifier
- Each process as its own separate memory space and copy of the data
- Depending on their identifier, processes can follow different control paths



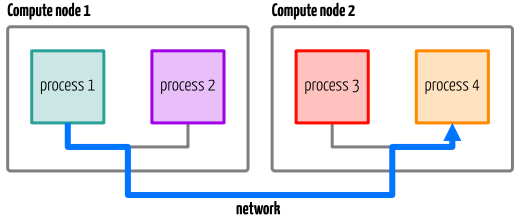
Message Passing

The message passing model allows for inter-node and intra-node communication to co-exists

- Intra-node communication can leverage interconnect loopback capabilities or make use of a shared memory region



- In the case of inter-node communication, the messages are sent through a network



What is MPI?

In scientific computing, the dominant paradigm for process parallelism is the single program multiple data model using MPI for interprocess communication

- MPI, which stands for Message Passing Interface, is a communication protocol for programming parallel computers
- The MPI standard was introduced by the MPI Forum in May 1994 and updated in June 1995. Version 3.1 of the standard has been approved in June 2015, and the last version (4.0) in June 2021
- It is a portable standard which defines the syntax and semantics of a core of library functions allowing the user to write message-passing programs
- It allows for both point-to-point and collective communication between processes

What MPI provides

- Management: managing processes, communicators, creating datatypes, topologies, ...
- Point-to-Point: blocking and non-blocking communication based on send and receive
- Collectives: communications that involve a group or groups of processes
- One-Sided: where one process specifies all communication parameters, both for the sending side and for the receiving side (including shared memory)
- I/O: to write and read files to disk in parallel

MPI Basics

The Six-Functions MPI

The basics of MPI can be limited to six functions:

- `MPI_Init`: Initialize MPI
- `MPI_Comm_size`: Find out how many processes there are
- `MPI_Comm_rank`: Find out which process I am
- `MPI_Send`: Send a message
- `MPI_Recv`: Receive a message
- `MPI_Finalize`: Terminate MPI

These functions are accessible by including `mpi.h` (C/C++) or by using the `mpi` or `mpi_f08` module (Fortran).

Initializing and Finalizing the MPI Environment

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

```
MPI_Init(ierror)
integer, optional, intent(out) :: ierror
```

Initialize the MPI environment. It must be called by each MPI process, once and before any other MPI function call. In C you can pass **`NULL`** for both the **`argc`** and **`argv`** arguments

```
MPI_Finalize()
```

```
MPI_Finalize(ierror)
integer, optional, intent(out) :: ierror
```

Terminates MPI execution environment. Once this function has been called, no MPI function may be called afterward.

Initializing and Finalizing the MPI Environment

- the MPI standard does not say what a program can do before an `MPI_Init` or after an `MPI_Finalize`
- in general you should do as little as possible before calling `MPI_Init`
- these functions should be called by one thread only (the main thread)
- `MPI_Init` and `MPI_Finalize` should be called by the same thread

Return Value of MPI Function/Routine

All MPI routines return an error value.

- In C this is the return value of the function
- In Fortran this is returned in the last argument (optional for the Fortran 2008 binding)

`MPI_SUCCESS`	Successful return code
`MPI_ERR_BUFFER`	Invalid buffer pointer
`MPI_ERR_COUNT`	Invalid count argument
`MPI_ERR_TYPE`	Invalid datatype argument
`MPI_ERR_TAG`	Invalid tag argument
`MPI_ERR_COMM`	Invalid communicator

Rank in a Communicator

The name of the processor on which a process is running can be queried by a call to the `MPI_Get_processor_name` function. Most of the time this function return the hostname.

```
MPI_Get_processor_name(char *name, int *namelen)
```

```
MPI_Get_processor_name(name, namelen, ierror)
character(len=MPI_MAX_PROCESSOR_NAME), intent(out) :: name
integer, intent(out) :: namelen
integer, optional, intent(out) :: ierror
```

The maximum length of the name is defined by `MPI_MAX_PROCESSOR_NAME`. You should ensure that **`name`** storage space is at least this value. The actual length of the name is returned in **`namelen`**.

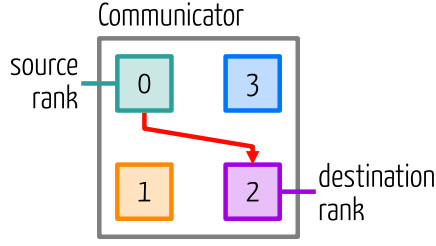
The MPI_Wtime function

An exception to the "all MPI functions return an error" is the `MPI_Wtime` function: it returns a double which represents a time stamp. The difference between two values obtained from this function allow you to compute the elapsed walltime between these two calls.

```
double start = MPI_Wtime();
// [...]
double end = MPI_Wtime();

double elapsed = end - start;
```

Communicator and Rank

- A communicator represents a group of processes that can communicate with each other
- Inside a communicator, each process is identified by an integer which is called a rank
- Each process has a unique rank inside a communicator but if a process is present in multiple communicators, it may have different ranks for each communicator.



Communicator Size

To get the size of a communicator, i.e. the number of processes, use the `MPI_Comm_size` function.

```
MPI_Comm_size(MPI_Comm communicator, int* size)

MPI_Comm_size(communicator, size, ierror)
    type(MPI_Comm), intent(in)     :: comm
    integer, intent(out)           :: size
    integer, optional, intent(out) :: ierror
```

After this function call, the size of the group associated with a communicator is stored in the variable `size`.

Rank in a Communicator

The rank of a process in a communicator is obtained using the `MPI_Comm_rank` function.

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

MPI_Comm_rank(communicator, rank, ierror)
    type(MPI_Comm), intent(in)     :: comm
    integer, intent(out)           :: rank
    integer, optional, intent(out) :: ierror
```

After this function call, the rank of the calling process inside the communicator is stored in the variable `rank`.

The World Communicator

The MPI specification provides a predefined communicator: `MPI_COMM_WORLD`

- it allows communication with all processes that are accessible after MPI initialization
- this communicator act as the default communicator as most applications use a flat name space for processes as well as a single communication context

Using the `MPI_COMM_WORLD` communicator, you can retrieve

- the total number of processes with `MPI_Comm_size`
- the global rank of a process with `MPI_Comm_rank`

MPI Hello Worlds

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

int main(int argc, char** argv) {
  int world_size, rank, name_len;
  char proc_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(proc_name, &name_len);

  printf("Hello world from rank %d "
         "out of %d on node %s.\n",
         rank, world_size, proc_name);

  MPI_Finalize();

  return 0;
}
```

```fortran
    program hello_world
      use mpi_f08

      implicit none

      integer :: rank, size, namelen, ierror
      character*(MPI_MAX_PROCESSOR_NAME) ::
           procname

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank)
      call MPI_Get_processor_name(procname,
           namelen)

      print 100, rank, size, procname(1:namelen)
100   format('Hello world from rank ', i0, &
&            ' out of ', i0,               &
&            ' on node ', a, '.')

      call MPI_Finalize(ierror)
    end
```

Compile a MPI Application

To have access to the MPI tools, first load the **OpenMPI** module. On NIC5:

```
$ module load releases/2020b
$ module load OpenMPI/4.1.0-GCC-10.2.0
```

Then you can compile your application using the **mpicc (mpifort)** compiler wrapper

```
$ mpicc -o hello_world hello_world.c
$ mpifort -o hello_world hello_world.f90
```

mpicc (mpifort) is not a compiler, it is a wrapper: it adds all the relevant compiler or linker flags and then invoke the underlying compiler or linker. In our case it invokes **gcc (gfortran)**

Running an MPI Application on a CÉCI Cluster

Create your job submission script. `ntasks` indicates the number of processes that we want to run.

```
#!/bin/bash
# Submission script for NIC5
#SBATCH --job-name=mpi-job
#SBATCH --time=00:01:00 # hh:mm:ss
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1024 # megabytes

module load releases/2020b
module load OpenMPI/4.1.0-GCC-10.2.0

cd $SLURM_SUBMIT_DIR

mpirun -np $SLURM_NTASKS ./hello_world
```

Running an MPI application on a CÉCI Cluster

Save your job submission script and run it with **sbatch**

```
 $ sbatch submit_mpi.job
Submitted batch job <jobid>

[...]

 $ cat mpi_hello.out
Hello world from rank 0 out of 4 on node nic5-w030.
Hello world from rank 2 out of 4 on node nic5-w035.
Hello world from rank 3 out of 4 on node nic5-w037.
Hello world from rank 1 out of 4 on node nic5-w034.
```

MPI Programming on the CÉCI Cluster

MPI implementations are available on all the CÉCI. For example, for OpenMPI:

	Module	Execution
Nic5	`module load 2020b` `module load OpenMPI/4.1.0-GCC-10.2.0`	multi-node
Lemaitre3	`module load OpenMPI/4.0.5-GCC-10.2.0`	multi-node
Hercules2	`module load OpenMPI/3.1.4-GCC-8.3.0`	Restricted to one node
Dragon2	`module load OpenMPI/3.1.3-GCC-8.2.0-2.31.1`	Restricted to one node

Other implementations are available and can be used by loading the appropriate environment module: `impi` (Intel MPI) and `MPICH`

Point to Point Communication

Point to Point Communication

So far, we didn't exchange messages between the processes that we spawned. The simplest type of communication is called point-to-point communication where

- the sender calls the send function and specifies the data to be sent
- the receiver calls the receive function and specify the data to be received
- the data is transferred as well as the associated metadata

In addition, we have to specify

- the communicator to use
- on the send side, the rank of the receiver in the communicator
- on the receive side, the rank of the sender in the communicator
- a tag to identify the message

Sending a Message

Sending a message with MPI is done with the `MPI_Send` function.

```
MPI_Send(void* buf,                 = address of the data you want to send
         int count,                 = number of elements to send
         MPI_Datatype datatype,     = the type of data we want to send
         int dest,                  = the recipient of the message (rank)
         int tag,                   = identify the type of the message
         MPI_Comm comm)             = the communicator used for this message
```

```
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
```

Receiving a Message

Receiving a message with MPI is done with the `MPI_Recv` function.

```
MPI_Recv(void* buf,                 = where to receive the data
         int count,                 = the receive buffer capacity
         MPI_Datatype datatype,     = the type of data we want to receive
         int source,                = the sender of the message (rank)
         int tag,                   = identify the type of the message
         MPI_Comm comm,             = the communicator used for this message
         MPI_Status* status)        = informations about the message
```

```
MPI_Recv(buf, count, datatype, source, tag, comm, status, ierror)
    type(*), dimension(..)          :: buf
    integer, intent(in)             :: count, source, tag
    type(MPI_Datatype), intent(in)  :: datatype
    type(MPI_Comm), intent(in)      :: comm
    type(MPI_Status)                :: status
    integer, optional, intent(out)  :: ierror
```

MPI Data types (C/C++)

MPI has a number of elementary data types, corresponding to the simple data types of the C programming language.

MPI Data Type	C Data Type	MPI Data Type	C Data Type
MPI_CHAR	char	MPI_UNSIGNED_CHAR	unsigned char
MPI_INT	int	MPI_UNSIGNED	unsigned int
MPI_LONG	long int	MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float	MPI_BYTE	unsigned char
MPI_DOUBLE	double		

In addition you can create your own custom data type.

MPI Data types (Fortran)

MPI has a number of elementary data types, corresponding to the simple data types of the Fortran programming language.

MPI Data Type	Fortran Data Type	MPI Data Type	Fortran Data Type
MPI_INTEGER	integer	MPI_REAL4	real*4
MPI_REAL	real	MPI_REAL8	real*8
MPI_DOUBLE_PRECISION	double precision	MPI_INTEGER4	integer*4
MPI_COMPLEX	complex	MPI_INTEGER8	integer*8
MPI_LOGICAL	logical	MPI_DOUBLE_COMPLEX	double complex
MPI_CHARACTER	character(1)		

In addition you can create your own custom data type.

Simple Send and Receive Example

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

#define BUFSZ 5

int main(int argc, char* argv[]) {
  double buf[BUFSZ];

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

  if (rank == 0) {
    for(unsigned int i = 0; i < BUFSZ; i++) buf[i] = (double)i;
    MPI_Send(buf, BUFSZ, MPI_DOUBLE, 1, 99, MPI_COMM_WORLD);
  } else if(rank == 1) {
    MPI_Recv(buf, BUFSZ, MPI_DOUBLE, 0, 99, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  }

  printf("Process with rank %d: ", rank);
  for(unsigned int i = 0; i < BUFSZ; i++)
    printf("%5.1lf", buf[i]);
  printf("\n");

  MPI_Finalize();

  return 0;
}
```

```f90
program main
  use mpi_f08

  implicit none

  integer, parameter :: bufsz = 5

  integer :: rank, i
  real    :: buf(bufsz)

  call MPI_Init()
  call MPI_Comm_rank(MPI_COMM_WORLD, rank)

  if (rank .eq. 0) then
    buf = (/ (real(i), i=1,bufsz) /)
    call MPI_Send(buf, bufsz, MPI_REAL, 1, 99, MPI_COMM_WORLD)
  else if (rank .eq. 1) then
    call MPI_Recv(buf, bufsz, MPI_REAL, 0, 99, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE)
  end if

  print 100, rank, buf
100 format('Process with rank ', i0,': ', *(f5.1));

  call MPI_Finalize()
end
```

Simple Send and Receive Example

```
 $ mpicc -o send_recv_array send_recv_array.c
 $ mpirun -np 2 ./send_recv_array
Process with rank 0:   0.0  1.0  2.0  3.0  4.0
Process with rank 1:   0.0  1.0  2.0  3.0  4.0
```

```
 $ mpifort -o send_recv_array send_recv_array.f90
 $ mpirun -np 2 ./send_recv_array
Process with rank 0:   0.0  1.0  2.0  3.0  4.0
Process with rank 1:   0.0  1.0  2.0  3.0  4.0
```

About Communication Parameters

For the communication to succeed

- the source and destination ranks must be valid and use the same communicator
- the tags must match
- the receive buffer should be large enough to hold the message but the buffer may be larger than the data received

The `count` argument passed to the `MPI_Recv` function is the maximum number of elements of a certain MPI datatype that the buffer can contain. The number of data elements actually received may be lower

The MPI Status Structure

Information (metadata) about the message can be obtained through the `MPI_Status` structure which contains the following fields

```
int MPI_SOURCE;     = source of the message
int MPI_TAG;        = tag of the message
int MPI_ERROR;      = error associated with the message
```

In Fortran, the `MPI_Status` type can be an integer array or a derived datatype depending on the binding you choose to use

```
use mpi
integer :: status(MPI_STATUS_SIZE)
...
src = status(MPI_SOURCE)
```

```
use mpi_f08
type(MPI_Status) :: status
...
src = status % MPI_SOURCE
```

Get the Size of a message

The `MPI_Get_count` function gets the actual number elements received.

```
MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int* count)
```

```
MPI_Get_count(status, datatype, count, ierror)
    type(MPI_Status), intent(in)    :: status
    type(MPI_Datatype), intent(in) :: datatype
    integer, intent(out)            :: count
    integer, optional, intent(out) :: ierror
```

- The datatype argument should match the argument provided by the receive call that set the `status` variable.
- If the number of elements received exceeded the limits of the `count` parameter, then `MPI_Get_count` sets the value of count to `MPI_UNDEFINED`.

Send and Receive with Get Count

```c
char msg[20];
int recv_count;
MPI_Status status;

if (rank == 0) {
  strcpy(msg, "Hello mate!");
  MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag,
           MPI_COMM_WORLD);

  printf("Process %d send: %s\n", rank, msg);
} else if(rank == 1) {
  MPI_Recv(msg, 20, MPI_CHAR, 0, tag,
           MPI_COMM_WORLD, &status);
  MPI_Get_count(&status, MPI_CHAR, &recv_count);

  printf("Process %d received: %s (size = %d)\n",
         rank, msg, recv_count);
}
```

```fortran
    character(len = 20) :: msg
    integer :: rank, recv_count
    type(MPI_Status) :: status

    if (rank .eq. 0) then
      msg = 'Hello mate!'
      call MPI_Send(msg, 11, MPI_CHARACTER, 1, tag, &
&                   MPI_COMM_WORLD)

      print 100, rank, msg
100   format('Process ', i0,' send: ', a);
    else if (rank .eq. 1) then
      call MPI_Recv(msg, 20, MPI_CHARACTER, 0, tag, &
&                   MPI_COMM_WORLD, status)
      call MPI_Get_count(status, MPI_CHARACTER,
           recv_count)

      print 200, rank, msg(1:recv_count), recv_count
200   format('Process ', i0, ' received: ', a, &
&            ' (size = ', i0, ')')
    end if
```

Simple Send and Receive Example

```
 $ mpicc -o send_recv_get_count send_recv_get_count.c
 $ mpirun -np 2 ./send_recv_get_count
Process 0 send: Hello mate!
Process 1 received: Hello mate! (size = 12)
```

In this example we set the maximum allowed length of the message to 20. This is the value we use on the receiver side, but the value used on the sender side was the actual size of the message. At the end we retrieve the actual length of the message using the `MPI_Get_count` function.

Get the Size of a Message Before Receiving It

You can determine the size of a message before receiving it using a combination of the `MPI_Probe` and `MPI_Get_count`functions.

```
MPI_Probe(int source,              = the sender of the message (rank)
          int tag,                 = identify the type of the message
          MPI_Comm comm)           = the communicator used for this message
          MPI_Status* status)      = informations about the message
```

```
MPI_Probe(source, tag, comm, status, ierror)
    integer, intent(in)             :: source, tag
    type(MPI_Comm), intent(in)      :: comm
    type(MPI_Status)                :: status
    integer, optional, intent(out) :: ierror
```

Get the Size of a Message Before Receiving It Example

```c
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  int buffer[3] = {123, 456, 789};
  printf("Process %d: sending 3 ints: %d, %d, %d\n",
         rank, buffer[0], buffer[1], buffer[2]);
  MPI_Send(buffer, 3, MPI_INT, 1, 10, MPI_COMM_WORLD);
} else if (rank == 1) {
  MPI_Status status;
  int count;

  MPI_Probe(0, 10, MPI_COMM_WORLD, &status);
  MPI_Get_count(&status, MPI_INT, &count);
  printf("Process %d retrieved the size of the message: %d.\n",
         rank, count);

  int* buffer = (int*)malloc(sizeof(int) * count);
  MPI_Recv(buffer, count, MPI_INT, 0, 10, MPI_COMM_WORLD, &status)
      ;
  printf("Process %d received message:", rank, count);
  for(int i = 0; i < count; ++i) printf(" %d", buffer[i]);
  printf(".\n");

  free(buffer);
}
```

```fortran
      call MPI_Init()
      call MPI_Comm_size(MPI_COMM_WORLD, size)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank)

      if (rank .eq. 0) then
        allocate(buffer(count))
        buffer = (/123, 456, 789/)

        print 100, rank, buffer
100     format('Process ', i0, ': sending 3 ints:', 3(1x,i0), '.')

        call MPI_Send(buffer, 3, MPI_INTEGER, 1, 10,
     &       MPI_COMM_WORLD)
      else if (rank .eq. 1) then
        call MPI_Probe(0, 10, MPI_COMM_WORLD, status)
        call MPI_Get_count(status, MPI_INTEGER, count)

        print 200, rank, count
200     format('Process ', i0, &
     &         ' retrieved the size of the message: ', i0, '.')

        allocate(buffer(count))
        call MPI_Recv(buffer, count, MPI_INTEGER, 0, 10, &
     &                MPI_COMM_WORLD, status)

        print 300, rank, buffer
300     format('Process ', i0,' received message:', *(1x,i0), '.')
      end if
```

If You Know Nothing About The Message

You can receive a message with no prior information about the source, the tag or the size.

In order to receive such message, you can use a combination of `MPI_Probe` and `MPI_Status` as well as the `MPI_ANY_SOURCE` and `MPI_ANY_TAG` wildcards

If You Know Nothing About The Message

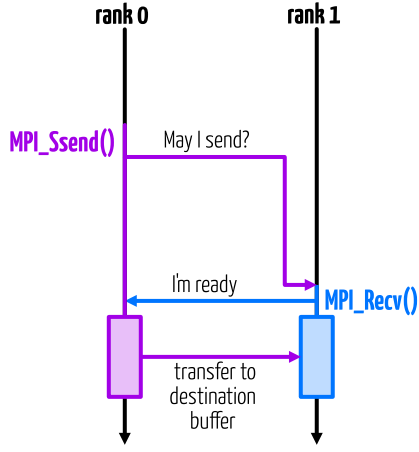
```c
if(rank == 0) {
   strcpy(sendbuf, "Hello Mate!");
   MPI_Send(sendbuf, strlen(sendbuf)+1, MPI_CHAR,
            1, 10, MPI_COMM_WORLD);
} else {
   MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
   MPI_Get_count(&status, MPI_CHAR, &msgsize);

   printf("Message incoming from process %d"
          " with tag %d and size %d\n"
          status.MPI_SOURCE, status.MPI_TAG, msgsize);

   if (msgsize != MPI_UNDEFINED)
      recvbuf = (char *)malloc(msgsize*sizeof(char));

   MPI_Recv(recvbuf, msgsize, MPI_CHAR,
            status.MPI_SOURCE,
            status.MPI_TAG,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);

   printf("Received message: %s\n", recvbuf);
}
```

```fortran
    if(rank .eq. 0) then
      sendbuf = 'Hello Mate!'
      call MPI_Send(sendbuf, LEN_TRIM(sendbuf),         &
&                   MPI_CHAR, 1, 10, MPI_COMM_WORLD)
    else
      call MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG,       &
&                    MPI_COMM_WORLD, status)
      call MPI_Get_count(status, MPI_CHAR, msgsize)

      print 100, status % MPI_SOURCE,                   &
                 status % MPI_TAG, msgsize
100   format('Message incoming from process ', i0,      &
&            ' with tag ', i0,' and size ', i0)

      call MPI_Recv(recvbuf, msgsize, MPI_CHAR,         &
&                   status % MPI_SOURCE,                &
&                   status % MPI_TAG,                   &
&                   MPI_COMM_WORLD, MPI_STATUS_IGNORE)

      print 200, recvbuf
200   format('Received message: ', a)
    end if
```

Communication mode and Message Matching

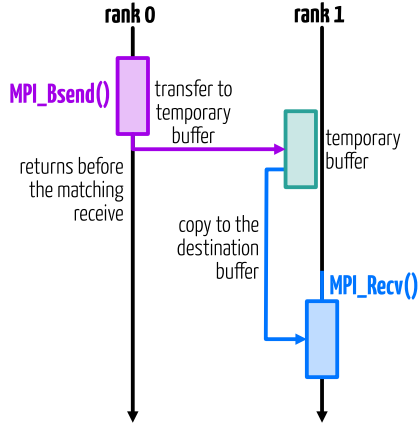
Communications Mode: Rendezvous Protocol

- Using the rendezvous Protocol, communication does not complete at either end before both processes rendezvous at the communication
- At first, the sender sends the envelope of the message and the actual transfer occurs when a matching buffer is available on the receiving side
- This communication mode is also called buffered



Communications Mode: Eager Protocol

- Using the eager protocol, data is transferred to the receiver before a matching receive is posted
- This communication mode assumes that the destination can store the data
- This communication mode is also called synchronous



Communications Mode

- `MPI_Send` is blocking, it only returns when the send buffer is safe to reuse but this does not mean that the data has reached the receive buffer
- depending on the implementation and the size of the message, `MPI_Send` use the eager (buffered) or rendezvous (synchronous) protocol

Function	Description	Mode
`MPI_Send`	Blocking send	synchronous or buffered
`MPI_Ssend`	Synchronous send	synchronous
`MPI_Bsend`	Buffered send	buffered
`MPI_Rsend`	Ready Send	

Message Matching

```
if (rank == 0) {
  MPI_Ssend(&val1, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Ssend(&val2, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD);
} else if(rank == 1) {
  MPI_Recv(&val1, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&val2, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```
 First Recv with tag 99 gets 12345
Second Recv with tag 1 gets 67890

 First Ssend with tag 99 sent 12345
Second Ssend with tag 1 sent 67890
```

Message Matching

```
if (rank == 0) {
  MPI_Ssend(&val1, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Ssend(&val2, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD);
} else if(rank == 1) {
  MPI_Recv(&val2, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&val1, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

With a synchronous send, this piece of code will produce a deadlock as there is no matching receive

- the first `MPI_Recv` is blocking until a message with **`tag2`** is received
- the first `MPI_Ssend` is blocking until its message with **`tag1`** received

Message Matching

```
if (rank == 0) {
  MPI_Bsend(&val1, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Bsend(&val2, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
} else if(rank == 1) {
  MPI_Recv(&val1, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&val2, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```
 First Bsend with tag 99 sent 12345
Second Bsend with tag 99 sent 67890

 First Recv with tag 99 gets 12345
Second Recv with tag 99 gets 67890
```

Message with the same tags are matched in order: the first message issued is received first, even in non-synchronous mode

Message Matching

```
if (rank == 0) {
  MPI_Bsend(&val1, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Bsend(&val2, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD);
} else if(rank == 1) {
  MPI_Recv(&val2, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&val1, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```
 First Bsend with tag 99 sent 12345
Second Bsend with tag 1 sent 67890

 First Recv with tag 1 gets 67890
Second Recv with tag 99 gets 12345
```

Message are matched by tags but they are not received in the same order as they were sent

Message Matching

```
if (rank == 0) {
  MPI_Bsend(&val1, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Bsend(&val2, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD);
} else if(rank == 1) {
  MPI_Recv(&val1, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&val2, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```
 First Bsend with tag 99 sent 12345
Second Bsend with tag 1 sent 67890

 First Recv gets 12345
Second Recv gets 67890
```

We did not specify a tag at the receiving end: message are match in the order they were sent

Back to the Blocking-Send

```
if (rank == 0) {
  MPI_Send(&val1, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Send(&val2, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD);
} else if(rank == 1) {
  MPI_Recv(&val2, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&val1, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

In the previous examples, we have seen that the for the piece of code above

- a deadlock will occur if the synchronous mode is used
- the message will be matched by tag at the receiving end if the buffered mode is used

Back to the Blocking-Send

The fact that the blocking send (`MPI_Send`) can change the communication protocol means that we might get different results depending on the implementation and the size of the message

```
if (rank == 0) {
  MPI_Send(&val1, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Send(&val2, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD);
} else if(rank == 1) {
  MPI_Recv(&val2, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&val1, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

A safer code will be

```
if (rank == 0) {
  MPI_Send(&val1, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
  MPI_Send(&val2, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD);
} else if(rank == 1) {
  MPI_Recv(&val1, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(&val2, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Communication Cost

Communication Performance

There is a cost to communication: nothing is free.

$$T_{comm} = T_{latency} + \frac{n}{B_{peak}}$$

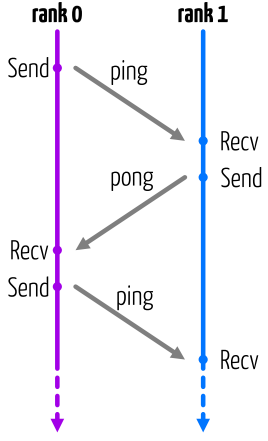
- $T_{latency}$: inherent cost of communication (in s)
- n: number of bytes to transfer
- B_{peak}: asymptotic bandwidth of the network (in bytes/s)

From this we can compute the effective bandwidth (in bytes/s), i.e. the transfer rate for a given message size.

$$B_{eff} = \frac{n}{T_{latency} + \frac{n}{B_{peak}}}$$

The Ping-Pong

We can visualize what is described theoretically in a real case: an MPI ping-pong application.



- We have two processes with respective rank 0 and 1
- process 0 sends a message to process 1 (ping)
- process 1 sends a message back to process 0 (pong)

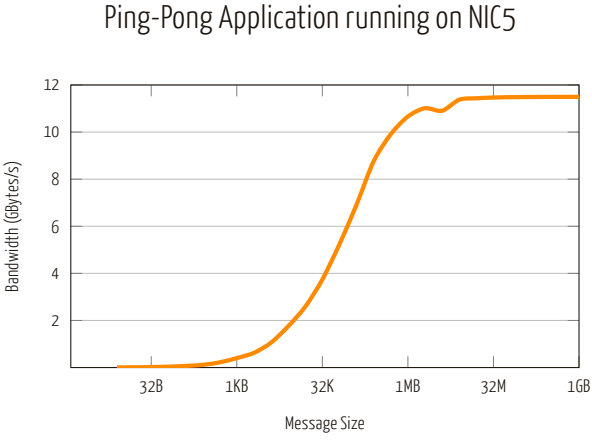
We repeat this ping-pong 50 times and measure the time to determine the transfer time of one message with increasing message size.

Ping Pong Code

```c
for (int i = 0; i <= 27; i++) {
  // Actual code has a warmup loop
  double elapsed_time = -1.0 * MPI_Wtime();
  for (int i = 1; i <= 50; ++i) {
    if (rank == 0) {
      MPI_Send(A, N, MPI_DOUBLE, 1, 10, MPI_COMM_WORLD);
      MPI_Recv(A, N, MPI_DOUBLE, 1, 20, MPI_COMM_WORLD, &status);
    } else if (rank == 1) {
      MPI_Recv(A, N, MPI_DOUBLE, 0, 10, MPI_COMM_WORLD, &status);
      MPI_Send(A, N, MPI_DOUBLE, 0, 20, MPI_COMM_WORLD);
    }
  }
  elapsed_time += MPI_Wtime();

  long int num_bytes = 8 * N;
  double num_gbytes = (double)num_bytes / (double)bytes_to_gbytes;
  double avg_time_per_transfer = elapsed_time / (2.0 * 50);

  if(rank == 0)
    printf("Transfer size (B): %10li, Transfer Time (s): %15.9f, "
           "Bandwidth (GB/s): %15.9f\n",
           num_bytes, avg_time_per_transfer,
           num_gbytes/avg_time_per_transfer );
  free(A);
}
```

```fortran
    do i = 0,27
!     Actual code has a warmup loop
      elapsed_time = -1.0 * MPI_Wtime()
      do j = 1,50
        if (rank .eq. 0) then
          call MPI_Send(buffer, length, MPI_DOUBLE, 1, 10, &
&                       MPI_COMM_WORLD, ierror)
          call MPI_Recv(buffer, length, MPI_DOUBLE, 1, 20, &
&                       MPI_COMM_WORLD, status, ierror)
        else if (rank .eq. 1) then
          call MPI_Recv(buffer, length, MPI_DOUBLE, 0, 10, &
&                       MPI_COMM_WORLD, status, ierror)
          call MPI_Send(buffer, length, MPI_DOUBLE, 0, 20, &
&                       MPI_COMM_WORLD, ierror)
        end if
      end do
      elapsed_time = elapsed_time + MPI_Wtime()
      num_bytes = 8*length
      num_gbytes = dble(num_bytes) / bytes_to_gbytes;
      avg_time_per_transfer = elapsed_time / (2.0 * 50)

      if (rank .eq. 0) then
        print 100, num_bytes, avg_time_per_transfer, &
&                  num_gbytes/avg_time_per_transfer
100     format('Transfer size (B): ', i10, &
&              ', Transfer Time (s): ', f15.9, &
&              ', Bandwidth (GB/s): ', f15.9)
      end if

      length = length * 2
    end do
```

Ping Pong Result



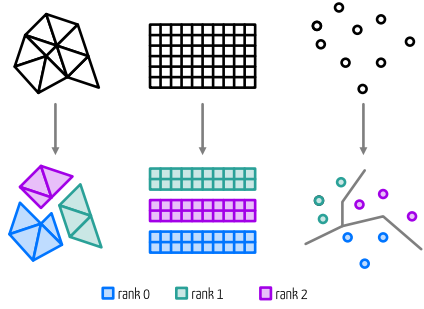
- With message size <1KB, the communication is dominated by the latency
- Transfer rate close to the theoretical performance of the network is observed for message size >10MB

Expressing Parallelism with MPI

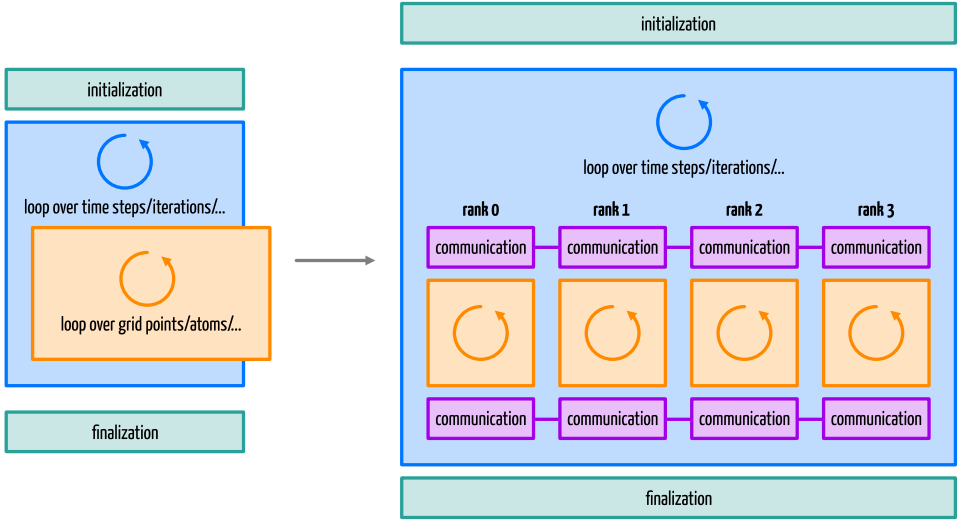
Dividing the Problem

So far, we have focused our attention on the communication between the processes (ranks) but did not really address the question the parallelization.

- divide the problem in smaller problems that are processed by different ranks
- usually this subdivision is based on the rank and world size
- MPI also includes convenience routines to create virtual topologies



Your Typical Scientific Application



Sum of Integers: Serial

As a first example, we will consider an application that sums the integer between 1 and N. The serial implementation of such an application is

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  const unsigned int N = 10000000;

  unsigned long sum = 0;
  for (unsigned int i = 1; i <= N; ++i)
    sum += i;

  printf("The sum of 1 to %d is %lu.\n", N, sum);

  return 0;
}
```

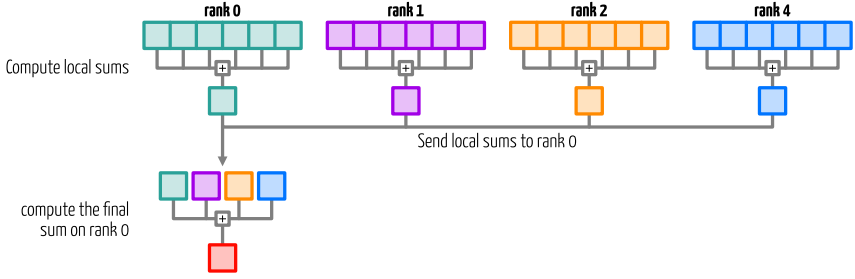
```fortran
    program main
      implicit none

      integer, parameter :: N = 10000000

      integer(kind=8) :: sum
      integer :: i

      sum = 0
      do i = 1, N
        sum = sum + i
      end do

      print 100, N, sum
100   format('The sum of 1 to ', i0, ' is ' i0 '.')

    end
```

Sum of Integers: Parallelization Strategy

The lower and upper bound for a rank can be computed using the following equations (assuming integer division):

$$\text{start}_i = \frac{N \cdot \text{rank}_i}{\text{worldsize}} + 1$$

$$\text{end}_i = \frac{N \cdot (\text{rank}_i + 1)}{\text{worldsize}}$$



Sum of Integers: Computation

```c
const unsigned int N = 10000000;

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

if (rank == 0)
  printf("Running with %d processes.\n", size);

unsigned int startidx = (N * rank / size) + 1;
unsigned int   endidx = N * (rank+1) / size;

unsigned long lsum = 0;
for (unsigned int i = startidx; i <= endidx; ++i)
  lsum += i;

printf("Process %d has local sum %lu\n", rank, lsum);
```

```fortran
      integer, parameter :: N = 10000000
      integer(kind=8) :: lsum = 0, rsum = 0
      integer :: startidx, endidx, src, i, rank, wsize

      call MPI_Init()
      call MPI_Comm_size(MPI_COMM_WORLD, wsize)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank)

      if (rank .eq. 0) then
        print 100, wsize
100     format('Running with ', i0, ' processes.')
      end if

      startidx = (N * rank / wsize) + 1
        endidx = N * (rank+1) / wsize

      do i = startidx, endidx
        lsum = lsum + i
      end do

      print 200, rank, lsum
200   format('Process ' i0, ' has local sum ', i0, '.')
```

Sum of Integers: Communication

```c
if (rank > 0) {
  MPI_Send(&lsum, 1, MPI_UNSIGNED_LONG, 0, 1,
           MPI_COMM_WORLD);
} else {
  unsigned long rsum;
  for(int src = 1; src < size; ++src) {
    MPI_Recv(&rsum, 1, MPI_UNSIGNED_LONG, src, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    lsum += rsum;
  }
}

if(rank == 0)
  printf("The sum of 1 to %d is %lu.\n", N, lsum);
```

```fortran
  if (rank .gt. 0) then
    call MPI_Send(lsum, 1, MPI_INTEGER8, 0, 1, &
&                 MPI_COMM_WORLD)
  else
    do src = 1, wsize-1
      call MPI_Recv(rsum, 1, MPI_INTEGER8, src, 1, &
&                   MPI_COMM_WORLD, MPI_STATUS_IGNORE)
      lsum = lsum + rsum
    end do
  end if

  if (rank .eq. 0) then
    print 300, N, lsum
300 format('The sum of 1 to ', i0, ' is ' i0 '.')
  end if
```

What About Interfaces?

The previous example is almost an embarrassingly parallel problem

- little is needed to separate the problem into a number of parallel tasks
- the main loop iterations (the sum) are independent of each other
- communication is only needed at the end to compute the final sum

What about problems where we have dependence on values computed by the other processes?

Diffusion Equation in 1D

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial u}{\partial x^2}$$

After discretization and by using a forward difference in time and a central difference in space, the previous equation reads

$$\frac{\partial}{\partial t} u(x_i, t_n) = \alpha \frac{\partial^2}{\partial x^2} u(x_i, t_n) \rightarrow \frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2 \cdot u_i^n + u_{i-1}^n}{\Delta x^2}$$

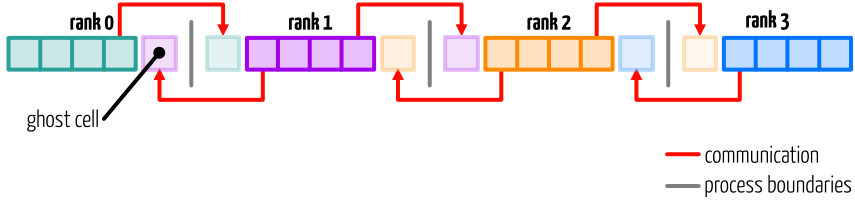
so that for each time step, u_i^{n+1} can be computed as

$$u_i^{n+1} = u_i^n + \alpha \frac{\Delta t}{\Delta x^2} \left(u_{i+1}^n - 2 \cdot u_i^n + u_{i-1}^n \right)$$

Diffusion Equation in 1D

$$u_i^{n+1} = u_i^n + \alpha \frac{\Delta t}{\Delta x^2} \left(u_{i+1}^n - 2 \cdot u_i^n + u_{i-1}^n \right)$$

Each point depends on the values on the left and right: use ghost cells that are updated (via communication) at each time step



Diffusion 1D (C)

The communication part is written so that we do the left-right communication first and then right-left.

```
const int  left_rank = my_rank > 0            ? my_rank - 1 : MPI_PROC_NULL;
const int right_rank = my_rank < world_size-1 ? my_rank + 1 : MPI_PROC_NULL;

for (unsigned int iter = 1; iter <= NITERS; iter++) {
  MPI_Send(&uold[my_size], 1, MPI_DOUBLE, right_rank, 0, /* ... */);
  MPI_Recv(&uold[0],       1, MPI_DOUBLE, left_rank,  0, /* ... */);

  MPI_Send(&uold[1],         1, MPI_DOUBLE, left_rank,  1, /* ... */);
  MPI_Recv(&uold[my_size+1], 1, MPI_DOUBLE, right_rank, 1, /* ... */);

  for (unsigned int i = 1; i < my_size+1; i++)
    unew[i] = uold[i] + DIFF_COEF * dtdx2 * (uold[i+1] - 2.0 * uold[i] + uold[i-1]);

  // ...
}
```

Diffusion 1D (Fortran)

The communication part is written so that we do the left-right communication first and then right-left.

```
 left_rank = merge(my_rank - 1, MPI_PROC_NULL, my_rank > 0)
right_rank = merge(my_rank + 1, MPI_PROC_NULL, my_rank < world_size-1)

do iter = 1, niters
  call MPI_Send(uold(my_size),  1, MPI_DOUBLE_PRECISION, right_rank, 0, ...)
  call MPI_Recv(uold(0),        1, MPI_DOUBLE_PRECISION, left_rank,  0, ...)

  call MPI_Send(uold(1),         1, MPI_DOUBLE_PRECISION, left_rank,  1, ...)
  call MPI_Recv(uold(my_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, ...)

  do i = 1, my_size
    unew(i) = uold(i) + DIFF_COEF * dtdx2 * (uold(i+1) - 2.0 * uold(i) + uold(i-1))
  end do

! ...
end do
```

Combined Sendrecv (C)

An alternative implementation could use combined send-receive

The send-receive operations combine in one operation the sending of a message to one destination and the receiving of another message, from another process

```
int MPI_Sendrecv(const void* sendbuf,      = address of the data you want to send
                 int sendcount,            = number of elements to send
                 MPI_Datatype sendtype,    = the type of data we want to send
                 int dest,                 = the recipient of the message (rank)
                 int sendtag,              = identify the type of the message
                 void* recvbuf,            = where to receive the data
                 int recvcount,            = the receive buffer capacity
                 MPI_Datatype recvtype,    = the type of data we want to receive
                 int source,               = the sender of the message (rank)
                 int recvtag,              = identify the type of the message
                 MPI_Comm comm,            = the communicator used for this message
                 MPI_Status* status);      = informations about the message
```

Combined Sendrecv (Fortran)

The send-receive operations combine in one operation the sending of a message to one destination and the receiving of another message, from another process

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
             recvcount, recvtype, source, recvtag, comm, status, ierror)
    type(*), dimension(..), intent(in) :: sendbuf
    integer, intent(in) :: sendcount, dest, sendtag, recvcount, source, recvtag
    type(MPI_Datatype), intent(in) :: sendtype, recvtype
    type(*), dimension(..) :: recvbuf
    type(MPI_Comm), intent(in) :: comm
    type(mpi_status) :: status
    integer, optional, intent(out) :: ierror
```

Diffusion 1D: Sendrecv version

The 1D diffusion communication code can be rewritten using two `MPI_Sendrecv`: one for left-right communication and a second call for the right-left communication.

```
MPI_Sendrecv(&uold[my_size], 1, MPI_DOUBLE, right_rank, 0,
             &uold[      0], 1, MPI_DOUBLE, left_rank,  0, /* ... */);

MPI_Sendrecv(&uold[        1], 1, MPI_DOUBLE, left_rank,  1,
             &uold[my_size+1], 1, MPI_DOUBLE, right_rank, 1, /* ... */);
```

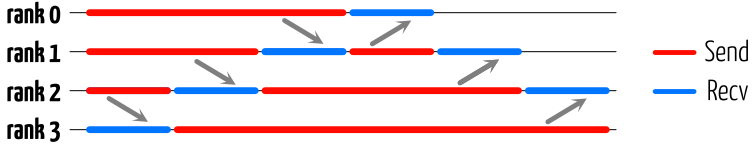
```
call MPI_Sendrecv(uold(my_size), 1, MPI_DOUBLE_PRECISION, right_rank, 0, &
&                 uold(      0), 1, MPI_DOUBLE_PRECISION, left_rank,  0, ...)

call MPI_Sendrecv(uold(        1), 1, MPI_DOUBLE_PRECISION, left_rank,  1, &
&                 uold(my_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, ...)
```

Diffusion 1D: Communication Serialization

```c
MPI_Send(&uold[my_size], 1, ..., right_rank, 0, ... );
MPI_Recv(&uold[0],       1, ..., left_rank,  0, ...);

MPI_Send(&uold[1],         1, ..., left_rank,  1, ...);
MPI_Recv(&uold[my_size+1], 1, ..., right_rank, 1, ...);
```

```fortran
call MPI_Send(uold(my_size),  1, ..., right_rank, 0, ...)
call MPI_Recv(uold(0),        1, ..., left_rank,  0, ...)

call MPI_Send(uold(1),         1, ..., left_rank,  1, ...)
call MPI_Recv(uold(my_size+1), 1, ..., right_rank, 1, ...)
```

Communication performance is poor: the communication is serial if the MPI implementation choose to use the synchronous mode



The Serialization of Communication Problem

The problem with the serialization of the communication is that

- the processes we add, the longer the communication time
- it leads to a significant impact on the parallel efficiency

The best way to improve the performance is to use **non-blocking communication** where the send and/or receive calls return immediately without waiting for the communication to be completed