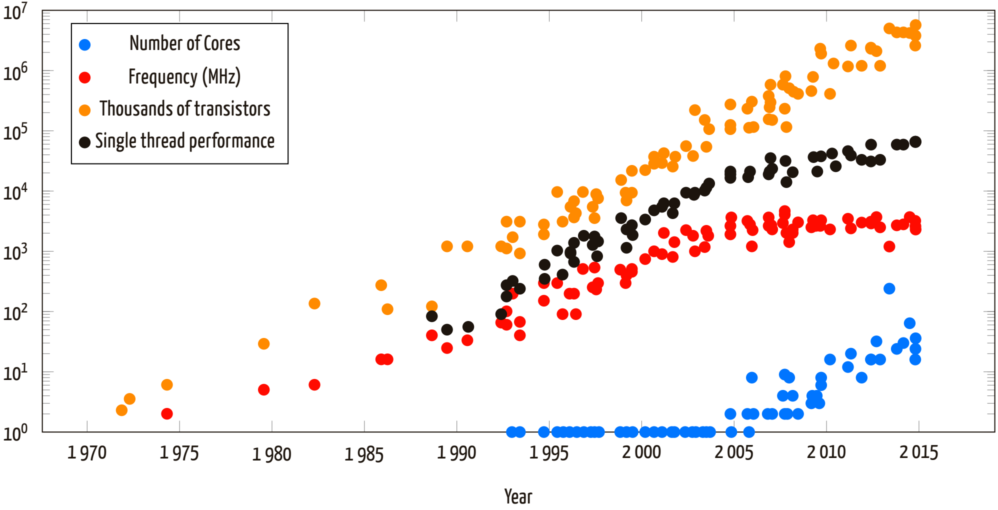
Message Passing Interface
Distributed-Memory Parallel Programming

Orian Louant

orian.louant@uliege.be

October 28, 2020

Motivations for Parallel Computing



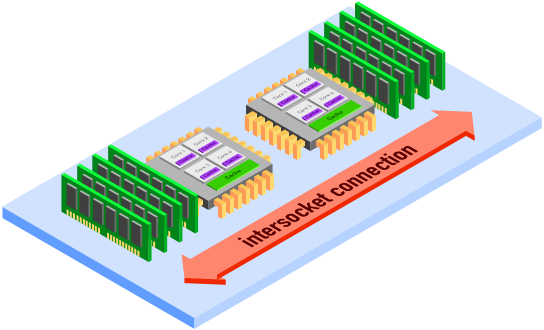
Motivations for Parallel Computing

- 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.

The parallel programming model became necessary.

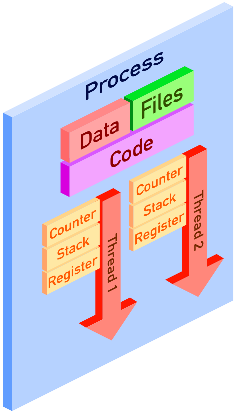
- The MPI standard was introduced by the MPI Forum in May, 1994 and updated in June, 1995.
- The OpenMP standard was introduced in 1997 (Fortran) and 1998 (C/C++).

Shared-Memory



- At least one multi-core CPU
- All CPUs can access a single memory address space
- Systems memory may be physically distributed, but logically shared

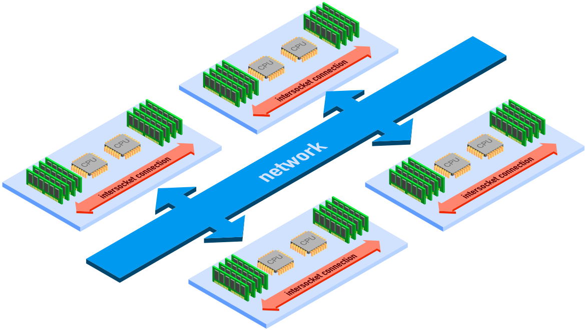
Threads



- A thread is an independent stream of instructions that can be scheduled to run by the operating system.
- Multiple threads can exist within one process, and they share the memory
- A thread only the owns the bare essential resources to exist as executable code: execution counter, stack pointer, registers and thread-specific data

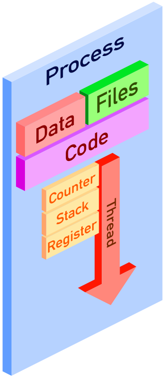
In scientific computing the dominant paradigm for thread parallelism is OpenMP

Distributed-Memory



- Multiple nodes
- Interconnected by a high-speed network
- Nodes consist of (a) processor(s) and local memory
- Communication is done via message passing

Process



- A process is an instance of an application
- A process is executed by at least one thread
- A process is a container describing the state of an application: code, memory mapping, shared libraries, ...

In scientific computing, the dominant paradigm for process parallelism is the single program multiple data model with MPI.

What is MPI?

- MPI which stands for Message Passing Interface, is a communication protocol for programming parallel computers
- 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 is not

It is not a language.

- MPI standard defines a library by specifying the names and results of functions
- MPI programs are compiled with ordinary compilers and linked with the MPI library

What MPI is not

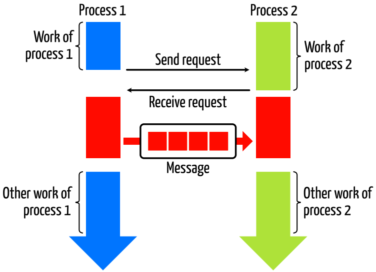
It is not a particular implementation.

- Vendors provide an MPI implementation for their machines and there is free, open source, implementations available as well
- An MPI program should be able to run on all MPI implementations

Principle of MPI

- Most program using MPI are based on the Single Program Multiple Data model (SPMD)
- Multiple parallel processes run the same program, 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

Principle of MPI



- Multiple copies of the same program (processes) are started. They do their work until communication is needed
- When one process is ready to send a message, it signals it to the other processes which signal that they are ready to receive
- Communication occurs as a collective undertaking
- The processes continue with their respective work

The Six-Functions MPI

- `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

MPI Basics

Initializing and Finalizing the MPI Environment

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

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

Initialize the MPI environment. It must be called by each MPI process, once and before any other MPI function call.

```c
MPI_Finalize()
```

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

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

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

`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

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.

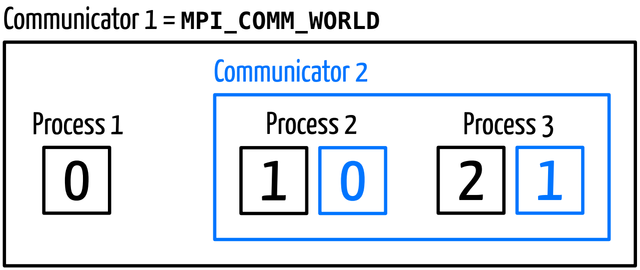
```
double start = MPI_Wtime();
// [...]
double end = MPI_Wtime();

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 and Rank

A process can have multiple ranks depending on the communicator. Here processes 2 and 3 have ranks 1 and 2 for communicator 1 while in communicator 2 their ranks are 0 and 1 respectively.



The MPI specification provides a default communicator: `MPI_COMM_WORLD`. It groups all the processes.

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)
integer, intent(in)  :: communicator
integer, intent(out) :: size, 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.

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

```fortran
MPI_Comm_rank(communicator, rank, ierror)
integer, intent(in)               :: communicator
integer, intent(out)              :: rank, ierror
```

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

MPI Hello Worlds

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

int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);

  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

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

  MPI_Finalize();

  return 0;
}
```

```fortran
    program hello_world
      use mpi

      integer rank, size, ierror

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      print 100, rank, size
100   format('Hello world from rank ', i0, &
&              ' out of ', i0, '.')

      call MPI_Finalize(ierror)
   end
```

Compile an MPI Program

To have access to the MPI tools, first load the `OpenMPI` module. On Lemaitre3:

```
$ module load OpenMPI/3.1.4-GCC-8.3.0
```

Then you can compile your program using the `mpicc (mpif90)` compiler wrapper

```
$ mpicc -o hello_world hello_world.c
$ mpif90 -o hello_world hello_world.f90
```

`mpicc (mpif90)` 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 Program 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 Lemaitre3
#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
#SBATCH --partition=batch
#SBATCH --output=mpi_hello.out

module load OpenMPI/3.1.4-GCC-8.3.0
cd $SLURM_SUBMIT_DIR

mpirun -np $SLURM_NTASKS ./hello_world
```

Running an MPI Program on a CÉCI Cluster

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

```
 $ sbatch submit_mpi.job
Submitted batch job <jobid> on cluster lemaitre3

[...]

 $ cat mpi_hello.out
Hello world from rank 0 out of 4.
Hello world from rank 1 out of 4.
Hello world from rank 2 out of 4.
Hello world from rank 3 out of 4.
```

MPI Programming on the CÉCI Cluster

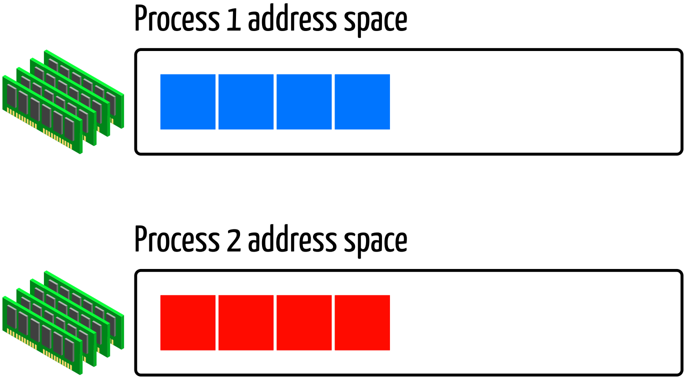
MPI implementations are available on all the CÉCI. For example, the combination of the OpenMP 3.1 with GCC 8 as the host compiler is available using the modules:

	Module	Execution
Hercules2	`module load 2019b` `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
Lemaitre3	`module load OpenMPI/3.1.4-GCC-8.3.0`	multi-node

Point to Point Communication

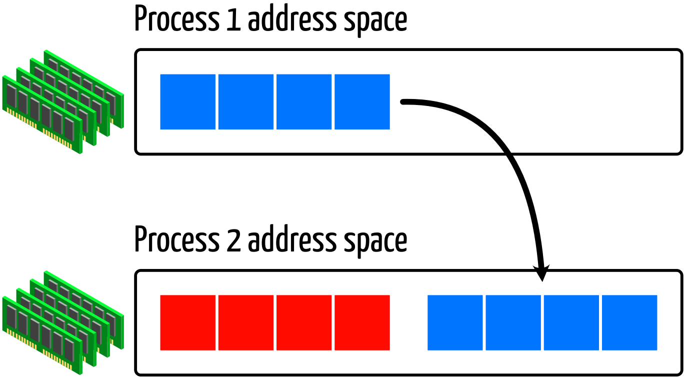
Point to Point Communication

Multiple processes are spawn to run in parallel and in the message-passing model, the processes have separate address spaces.



Point to Point Communication

Communication is needed if we want to copy a portion of one process's address space is copied into another process's address space.



Sending And Receiving a Message

Sending a Message

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

```
MPI_Send(void* address,             = 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 destination,           = the recipient of the message (rank)
         int tag,                   = identify the type of the message
         MPI_Comm communicator)     = the communicator used for this message
```

```
MPI_Send(address, count, datatype, destination, tag, communicator, ierror)
         type(*), dimension(..), intent(in) :: address
         integer, intent(in)                :: count, datatype, destination
         integer, intent(in)                :: tag, comm
         integer, intent(out)               :: ierror
```

Receiving a Message

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

```
MPI_Recv(void* address,                = where to receive the data
         int count,                    = number of elements to send
         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 communicator)        = the communicator used for this message
         MPI_Status* status)           = informations about the message
```

```
MPI_Recv(address, count, datatype, source, tag, communicator, status, ierror)
         type(*), dimension(..)         :: address
         integer, intent(in)            :: count, datatype, source
         integer, intent(in)            :: tag, communicator
         integer, intent(out)           :: status(MPI_STATUS_SIZE)
         integer, 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_CHAR`	`character`
`MPI_INTEGER`	`integer`
`MPI_INTEGER8`	`integer*8`
`MPI_REAL`	`real`
`MPI_DOUBLE`	`double`

In addition you can create your own custom data type.

The Receive Buffer

For the communication to succeed, the receive buffer passed to `MPI_Recv` function must be

- large enough to hold the message. If it is not, behaviour is undefined.
- however, the buffer may be longer than the data received.

The `count` argument 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 less than this.

The MPI Status Structure

Informations about the message can be obtained through the `MPI_Status` structure. In Fotran, `MPI_Status` is an array of integers of size `MPI_STATUS_SIZE`.

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

In Fortran, these fields are defined as the indexes of the values in the `MPI_Status` array.

```
src = status(MPI_SOURCE)
```

The MPI Status Structure

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)
        integer, intent(int) :: status(MPI_STATUS_SIZE)
        integer, intent(int) :: datatype
        integer, intent(out) :: count, 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`.

Simple Send and Receive Example

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

int main(int argc, char* argv[]) {
  char msg[20];
  int rank, recv_count, tag = 99;
  MPI_Status status;

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

  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);
  }

  MPI_Finalize();

  return 0;
}
```

```fortran
    program main
      use mpi

      implicit none

      character(len = 20) :: msg
      integer(4) :: rank, recv_count, ierror, status(
             MPI_STATUS_SIZE)
      integer(4) :: tag = 99;

      call MPI_Init(ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      if (rank .eq. 0) then
        msg = 'Hello mate!'
        call MPI_Send(msg, 20, MPI_CHAR, 1, tag, &
&                    MPI_COMM_WORLD, ierror)
        print 100, rank, msg
100     format('Process ', i0,' send: ', a);
      else if (rank .eq. 1) then
        call MPI_Recv(msg, 20, MPI_CHAR, 0, tag, &
&                    MPI_COMM_WORLD, status, ierror)
        call MPI_Get_count(status, MPI_CHAR, recv_count, ierror)

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

      call MPI_Finalize(ierror)
    end
```

Simple Send and Receive Example

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

In this example we set the maximun allowed lenght 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 lenth of the message using the **`MPI_Get_count`** function.

Get the Size of a Message

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 communicator)    = the communicator used for this message
          MPI_Status* status)       = informations about the message
```

```
MPI_Probe(source, tag, communicator, status, ierror)
          integer, intent(in)  :: source, tag, communicator
          integer, intent(out) :: status(MPI_STATUS_SIZE)
          integer, intent(out) :: ierror
```

Get the Size of a Message 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(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      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, ierror)
      else if (rank .eq. 1) then
        call MPI_Probe(0, 10, MPI_COMM_WORLD, status, ierror)
        call MPI_Get_count(status, MPI_INTEGER, count, ierror)

        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, ierror)

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

You Know Nothing...

You can receive a message with no prior infomation 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.

You Know Nothing...

```c
char sendbuf[20];
char *recvbuf;

int msgsize, size, rank;
MPI_Status status;

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

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

MPI_Finalize();
```

```fortran
      character(len = 20) :: sendbuf
      character(len = 20) :: recvbuf = ''

      integer :: msgsize, size, rank, ierror
      integer :: status(MPI_STATUS_SIZE)

      call MPI_Init(ierror);
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      if(rank .eq. 0) then
        sendbuf = 'Hello Mate!'
        call MPI_Send(sendbuf, LEN_TRIM(sendbuf), MPI_CHAR, 1, 10,
                &
&                    MPI_COMM_WORLD, ierror)
      else
        call MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, &
&                     MPI_COMM_WORLD, status, ierror)
        call MPI_Get_count(status, MPI_CHAR, msgsize, ierror)
        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, ierror)
        print 200, recvbuf
200     format('Received message: ', a)
      end if

      call MPI_Finalize(ierror)
```

SPMD: Sum of Integers

Now, as an illustration of the concept of Single Program Multiple Data, we will consider a program that sums the integers between 1 and N. We will start from the serial code

```c
#include <stdio.h>

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

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

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

  return 0;
}
```

```fortran
    program main
      implicit none

      integer, parameter :: N = 10000000

      integer*8 :: sum = 0
      integer   :: i

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

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

SPMD: Sum of Integers

What are the tasks ahead in order to parallelize this simple program with MPI?

- Divide the work among the processes, i.e. each process will compute the sum on a range of integers
- Once the sum for each process is computed, we send the partial result to the process with rank 0
- We compute the final sum

SPMD: Sum of Integers

The first step is to divide the work among the processes, i.e. each process will compute the sum on a range of integers. To do so,

- We use the size of the communicator (number of processes) to divide the work
- We use the rank to determine the range of data on which process need to perform the work

$$\text{start} = \frac{N \cdot \text{rank}}{\text{commsize}}$$

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

SPMD: Sum of Integers

Once we have a way to divide the work among the processes, we can compute the partial sum for each of them.

```c
int startidx = (N * rank / world_size) + 1;
int   endidx = N * (rank+1) / world_size;

unsigned int proc_sum = 0;
for (int i = startidx; i <= endidx; ++i)
    proc_sum += i;
```

```fortran
startidx = (N * rank / world_size) + 1
  endidx = N * (rank+1) / world_size

do i = startidx, endidx
  proc_sum = proc_sum + i
end do
```

The next step is to compute the final sum. This may be done by sending all the partial sum to one of the processes which will compute the final value.

SPMD: Sum of Integers

In order to compute the final sum, we send all the partial sums to the process with rank 0 that will compute the final value.

```c
if (rank > 0) {
  MPI_Send(&proc_sum, 1, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD);
} else {
  unsigned int remote_sum;
  for(int src = 1; src < world_size; ++src) {
    MPI_Recv(&remote_sum, 1, MPI_UNSIGNED, src, 1,
             MPI_COMM_WORLD, &status);
    proc_sum += remote_sum;
  }
}
```

```fortran
if (rank .gt. 0) then
  call MPI_Send(proc_sum, 1, MPI_INTEGER8, 0, 1, MPI_COMM_WORLD,
         ierror)
else
  do src = 1, world_size-1
    call MPI_Recv(remote_sum, 1, MPI_INTEGER8, src, 1, &
              & MPI_COMM_WORLD, status, ierror)
    proc_sum = proc_sum + remote_sum
  end do
end if
```

Example: Sum of Integers (part. 1)

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

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);

  if(argc != 2) {
    printf("usage: isum <size>\n");
    exit(1);
  }

  int N = atoi(argv[1]);

  int rank, world_size, nthreads;
  MPI_Status status;

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

  if (rank == 0) {
    printf("Sum of integer from 1 to %d running on %d.\n",
            N, world_size);
  }
```

```fortran
program main
  use mpi

  implicit none

  integer, parameter :: N = 10000000

  integer*8 :: proc_sum, remote_sum
  integer   :: rank, world_size, ierror
  integer   :: endidx, endidx, src, i
  integer   :: status(MPI_STATUS_SIZE)

  call MPI_Init(ierror)

  call MPI_Comm_size(MPI_COMM_WORLD, world_size, ierror)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

  if (rank .eq. 0) then
    print 100, N, world_size
100 format('Sum of integer from 1 to ', i0, &
         & ' running on ', i0 ' processes.')
  end if
```

Example: Sum of Integers (final, part. 2)

```c
  int startidx = (N * rank / world_size) + 1;
  int   endidx = N * (rank+1) / world_size;

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

  if (rank > 0) {
    MPI_Send(&proc_sum, 1, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD)
          ;
  } else {
    unsigned int remote_sum;
    for(int src = 1; src < world_size; ++src) {
      MPI_Recv(&remote_sum, 1, MPI_UNSIGNED, src, 1,
               MPI_COMM_WORLD, &status);
      proc_sum += remote_sum;
    }
  }

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

  MPI_Finalize();

  return 0;
}
```

```fortran
      startidx = (N * rank / world_size) + 1
        endidx = N * (rank+1) / world_size - 1

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

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

      if (rank .gt. 0) then
        call MPI_Send(proc_sum, 1, MPI_INTEGER8, 0, 1,
               MPI_COMM_WORLD, ierror)
      else
        do src = 1, world_size-1
          call MPI_Recv(remote_sum, 1, MPI_INTEGER8, src, 1, &
                    & MPI_COMM_WORLD, status, ierror)
          proc_sum = proc_sum + remote_sum
        end do
      end if

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

      call MPI_Finalize(ierror)
    end
```

Domain decomposition

In the previous (very) simple example, we demonstrated a basic strategy to divide the work among processes. In doing so, we have highlighted the most common strategy for parallelization with MPI: subdivision the computational domain.

- sub-domains can have equal or variable size
- try to have the same amount of computational work
- minimize the amount of data that needs to be communicated

Domain decomposition comes in many shapes and sizes: cuboids, binary trees, quad-trees, oct-trees, pencils, slabs, ...

Communication Performance and Mode

Communication Performance

There is a cost to communication: nothing is free.

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

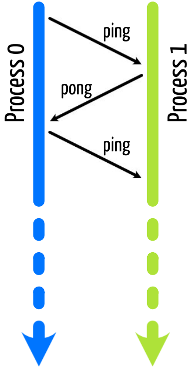
- $T_{latency}$: inherent cost of communication (in s)
- n: number of bytes to transfer
- B_{peak}: asymptotic bandwidth of the network (in bytes/s)

From this we can compute the effective bandwith (in bytes/s), i.e. the transfert 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 program.



- 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)

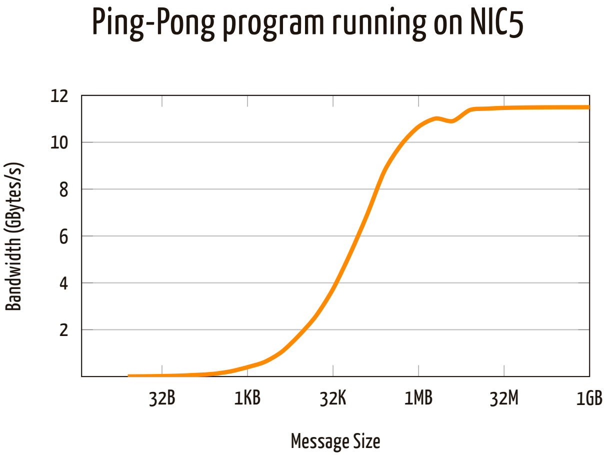
We repeat this ping-pong 50 times and measure the time to determine the transfer time of one message with increasing message size.

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
- Transfert speed close to the theoretical performance of the network is observed for message size >10MB

NIC5 will be available late this year/begining of next year: 70 nodes with 2x32-cores AMD EPYC at 2.9 GHz, 256 GB of RAM and 100 Gbps Infiniband interconnect

Communication Mode

MPI has four communication mode. The default is the standard mode.

Synchronous	`MPI_Ssend`	Only completes when the receive has completed
Buffered	`MPI_Bsend`	Always completes (unless an error occurs), irrespective of whether the receive has completed
Standard	`MPI_Send`	Either synchronous or buffered
Ready	`MPI_Rsend`	Always completes (unless an error occurs), irrespective of whether the receive has completed. Need to have a matching receive. already listening.

Non-Blocking Communication

Deadlock

The standard **`MPI_Send`** call is blocking:

- It does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer
- Completion of a send means by definition that the send buffer can safely be re-used
- The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer

Deadlock

Consider this piece of code:

```
if (rank == 0) {
  MPI_Send(sendbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
  MPI_Recv(recvbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);

} else if(rank == 1) {
  MPI_Send(sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
  MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
}
```

Process with rank 0 is waiting for the process with rank 1 to be ready to receive data. The problem is that the process with rank 1 is also waiting for the process with rank 0. We have a deadlock: a state in which each member of a group is waiting for another member.

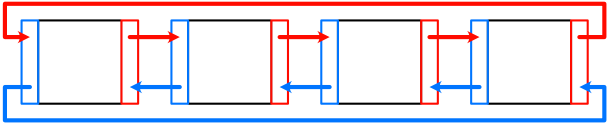
Deadlock

The solution is to reverse the order in which these MPI calls are made.

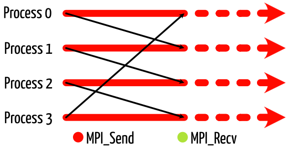
```
if (rank == 0) {
  MPI_Send(sendbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
  MPI_Recv(recvbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);

} else if(rank == 1) {
  MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
  MPI_Send(sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
}
```

Blocking Communication

The fact that send and receive operation may be blocking has practical implication. Consider the left to right (red) halo exchange with cyclic boundary conditions



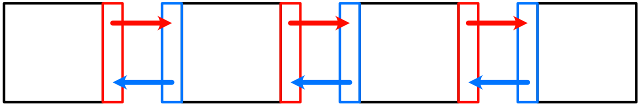
```
MPI_Send(..., right_rank, ...);
MPI_Recv(..., left_rank, ...);
```



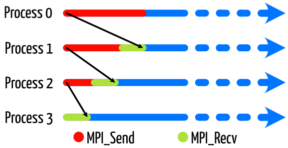
If the MPI library chooses the synchronous protocol, this leads to a deadlock, **MPI_Send** waits until **MPI_Recv** is called

Blocking Communication

And now with non-cyclic boundary conditions for the same left to right (red) halo exchange



```
if (rank < size-1) {
  MPI_Send(..., left_rank, ...);
else if(rank > 0) {
  MPI_Recv(..., right_rank, ...);
```



If the MPI library chooses the synchronous protocol, this leads to a serialization of the communications, **MPI_Send** and **MPI_Recv** are executed in sequence

Non-Blocking Communication

The other way in which these send and receive operations can be done is by using the ”I” functions. The ”I” stands for Immediate returns and allow to perform the communication in three phases:

- Initiate a non-blocking communication with **`MPI_Isend`** or **`MPI_Irecv`** functions. This function return immediately
- Do some work
- Wait for the non-blocking communication to complete

Non-Blocking Send

A non-blocking send is executed with the **`MPI_Isend`** function

```
MPI_Isend(void* address,               = 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 destination,             = the recipient of the message (rank)
          int tag,                     = identify the type of the message
          MPI_Comm communicator,       = the communicator used for this message
          MPI_Request* request)        = the handle on the non-blocking communication
```

```
MPI_Isend(address, count, datatype, destination,
          tag, communicator, request, ierror)
          Type(*), dimension(..), intent(in) :: address
          integer, intent(in)                :: count, datatype
          integer, intent(in)                :: destination, tag, communicator
          integer, intent(out).              :: request
          integer, intent(out).              :: ierror
```

Non-Blocking Receive

A non-blocking receive is executed with the **MPI_Irecv** function

```
MPI_Irecv(void* address,              = where to receive the data
          int count,                  = number of elements to send
          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 communicator)      = the communicator used for this message
          MPI_Request* request)       = the handle on the non-blocking communication
```

```
MPI_Irecv(address, count, datatype, source,
          tag, communicator, request, ierror)
          Type(*), dimension(..) :: address(*)
          integer, intent(in)    :: count, datatype
          integer, intent(in)    :: source, tag, communicator
          integer, intent(out)   :: request
          integer, intent(out)   :: ierror
```

Waiting for a Non-Blocking Communication

An important features of the non-blocking communication is the `MPI_Request` handle. The value of this handle

- is generated by the non-blocking communication function
- is used by the **`MPI_Wait`** or **`MPI_Test`** functions

When using non-blocking communication, you have to be careful and avoid to

- modify the send buffer before the send operation completes
- read the receive buffer before the receive operation completes

Waiting for a Non-Blocking Communication

The **MPI_Wait** returns when the operation identified by request is complete. This is a blocking function.

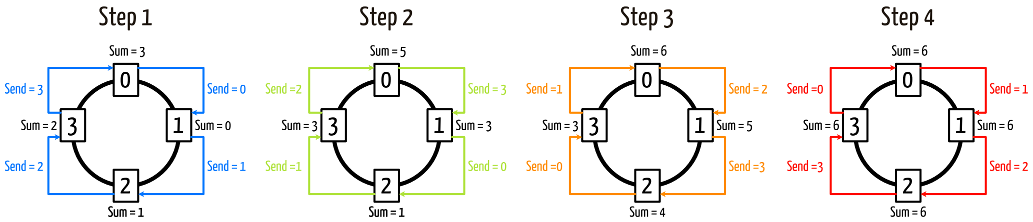
```
MPI_Wait(MPI_Request* request,    = handle of the non-blocking communication
         MPI_Status*  status)     = status of the completed communication
```

```
MPI_Wait(request, status, ierror)
         integer, intent(in)  :: request
         integer, intent(out) :: status(MPI_STATUS_SIZE)
         integer, intent(out) :: ierror
```

Example: Ring Communication

As an example, consider the rotation of an infomation inside a ring.

- Each process stores its rank in the send buffer
- Each process send its rank to its neighbour on the right
- Each process add the received value to the sum
- Repeat



Example: Ring Communication (Synchronous)

First we will consider the synchronous version:

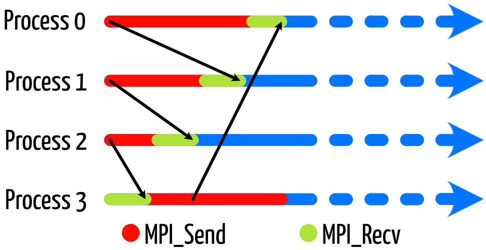
```c
sum = 0;
send_buf = rank;

for(int j = 0; j < size; ++j) {
  if (rank < size-1) {
    MPI_Ssend(&send_buf, 1, MPI_INT, right_rank, 10,
              MPI_COMM_WORLD);
    MPI_Recv(&recv_buf, 1, MPI_INT, left_rank, 10,
             MPI_COMM_WORLD, &status);
  }
  else {
    MPI_Recv(&recv_buf, 1, MPI_INT, left_rank, 10,
             MPI_COMM_WORLD, &status);
    MPI_Ssend(&send_buf, 1, MPI_INT, right_rank, 10,
              MPI_COMM_WORLD);
  }

  send_buf = recv_buf;
  sum += recv_buf;
}
```

```fortran
sum = 0
send_buf = rank

do i = 1,size
  if (rank .lt. size-1) then
    call MPI_Ssend(send_buf, 1, MPI_INTEGER, right_rank, 10, &
&                  MPI_COMM_WORLD, ierror)
    call MPI_Recv(recv_buf, 1, MPI_INTEGER, left_rank, 10,  &
&                  MPI_COMM_WORLD, status, ierror)
  else
    call MPI_Recv(recv_buf, 1, MPI_INTEGER, left_rank, 10,  &
&                  MPI_COMM_WORLD, status, ierror)
    call MPI_Ssend(send_buf, 1, MPI_INTEGER, right_rank, 10, &
&                  MPI_COMM_WORLD, ierror)
  end if

  send_buf = recv_buf
  sum = sum + recv_buf
end do
```

In order to avoid a deadlock, we have reversed the send and receive for the last process. But this solution comes with its own problem...

Example: Ring Communication (Synchronous)

In order to avoid a deadlock, we have reversed the send and receive for the last process. But this solution comes with its own problem...



Using the synchronous send operation, we end up serializing the communication in the ring. Let's give a shot to the asynchronous solution.

Example: Ring Communication (Asynchronous)

The asynchronous version:

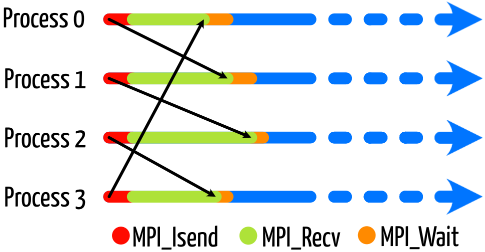
```c
right_rank = (rank+1      ) % size;
left_rank  = (rank-1+size) % size;

sum = 0;
send_buf = rank;

for(int j = 0; j < size; ++j) {
  MPI_Isend(&send_buf, 1, MPI_INT, right_rank, 10,
            MPI_COMM_WORLD, &request);
  MPI_Recv(&recv_buf, 1, MPI_INT, left_rank, 10,
           MPI_COMM_WORLD, &status);

  MPI_Wait(&request, &status);

  send_buf = recv_buf;
  sum += recv_buf;
}
```

```fortran
right_rank = mod(rank+1,      size)
left_rank  = mod(rank-1+size, size)

sum = 0
send_buf = rank

do i = 1,size
  call MPI_Isend(send_buf, 1, MPI_INTEGER, right_rank, 10, &
&                MPI_COMM_WORLD, request, ierror)
  call MPI_Recv(recv_buf, 1, MPI_INTEGER, left_rank, 10,   &
&               MPI_COMM_WORLD, status, ierror)

  call MPI_Wait(request, status, ierror)

  send_buf = recv_buf
  sum = sum + recv_buf
end do
```

We use `MPI_Isend` instead of `MPI_Ssend`. This means that the process can proceed with the receive operation before waiting for the send operation to complete.

Example: Ring Communication (Synchronous)

We use **`MPI_Isend`** instead of **`MPI_Ssend`**. This means that the process can proceed with the receive operation before waiting for the send operation to complete.



Ring Communication: Effect of Serialization

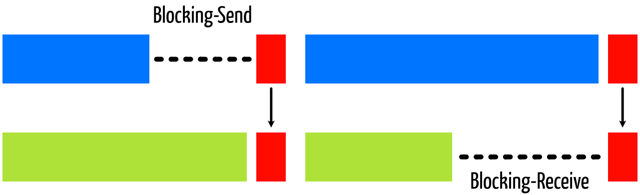
In our ring communication example, the use of synchronous send operation led to serialization of the communication. This may lead to significant performance loss in particular when we increase the number of processes.

Here are the timing for a complete ring traversal:

Number of processes	2	4	8	16	32
Ring 1GB ISend	0.089	0.177	0.353	0.705	1.420
Ring 1GB Send	0.088	0.350	1.398	5.580	22.295
Async speedup	0.98	1.97	3.96	7.91	15.7

Blocking Communication and Load Imbalance

The fact that **`MPI_Send`** and **`MPI_Recv`** are blocking can be limiting in some application. For example, when the tasks of two processes take a different amount of time to complete, i.e. when we have a load imbalance between the processes.



In this illustration, process 0 is waiting for process 1 for the first communication while, for the second communication, process 1 is waiting for process 0 to be ready. Time is wasted by processes waiting.

Better Hide this Communications

As discussed before, communication comes at a cost. You should try to hide this cost has much as possible:

- The time spend in the actual calculation should be large enough to hide communication
- Try to overlap communication and communication with asynchronous communication

Better Hide this Communications

For example, in the case of a 1D diffusion problem:

```c
for(int step = 0; step < NSTEPS; ++step) {
  MPI_Irecv(&u_prev[0] , 1, MPI_DOUBLE, left_rank, 10,
            MPI_COMM_WORLD, &requests[0]);
  MPI_Irecv(&u_prev[n+1], 1, MPI_DOUBLE, right_rank, 20,
            MPI_COMM_WORLD, &requests[1]);

  MPI_Isend(&u_prev[1] , 1, MPI_DOUBLE, left_rank, 20,
            MPI_COMM_WORLD, &requests[2]);
  MPI_Isend(&u_prev[n] , 1, MPI_DOUBLE, right_rank, 10,
            MPI_COMM_WORLD, &requests[3]);

  for (int i = 2; i < n; ++i)
    u[i] = u_prev[i] + alpha * (u_prev[i-1] - 2 * u_prev[i]
                                             + u_prev[i+1]);

  MPI_Waitall(4, requests, statuses);

  u[1] = u_prev[1] + alpha * (u_prev[0] - 2 * u_prev[1]
                                         + u_prev[2]);
  u[n] = u_prev[n] + alpha * (u_prev[n-1] - 2 * u_prev[n]
                                           + u_prev[n+1]);

  double* temp = u; u = u_prev; u_prev = temp;
}
```

```fortran
do step = 1, nsteps
  call MPI_Irecv(u_prev(1), 1, MPI_DOUBLE, left_rank, 10,    &
&                MPI_COMM_WORLD, requests(1), ierror)
  call MPI_Irecv(u_prev(n+2), 1, MPI_DOUBLE, right_rank, 20, &
&                MPI_COMM_WORLD, requests(2), ierror)

  call MPI_Isend(u_prev(2), 1, MPI_DOUBLE, left_rank, 20,    &
&                MPI_COMM_WORLD, requests(3), ierror)
  call MPI_Isend(u_prev(n+1), 1, MPI_DOUBLE, right_rank, 10, &
&                MPI_COMM_WORLD, requests(4), ierror)

  do i = 3, n
    u(i) = u_prev(i) + alpha * (u_prev(i-1) - 2 * u_prev(i) &
&                                           + u_prev(i+1))
  end do

  call MPI_Waitall(4, requests, statuses, ierror)

    u(2) = u_prev(2) + alpha * (u_prev(1) - 2 * u_prev(2)     &
&                                         + u_prev(3))
  u(n+1) = u_prev(n+1) + alpha * (u_prev(n) - 2 * u_prev(n+1) &
&                                           + u_prev(n+2))

  u_prev = u
end do
```

on Wait Multiple Requests

In the last example, we wait for multiple requests in one call with the **MPI_Waitall** function.

```
MPI_Waitall(int count,                    = number of request handlers to wait on
            MPI_Request* requests,        = request handlers to wait on
            MPI_Status* statuses)         = array in which write the statuses
```

```
MPI_Waitall(count, requests, statuses, ierror)
            integer, intent(in)    :: count
            integer, intent(inout) :: requests(*)
            integer, intent(out)   :: statuses(MPI_STATUS_SIZE,*)
            integer, intent(out)   :: ierror
```

Collective Communication

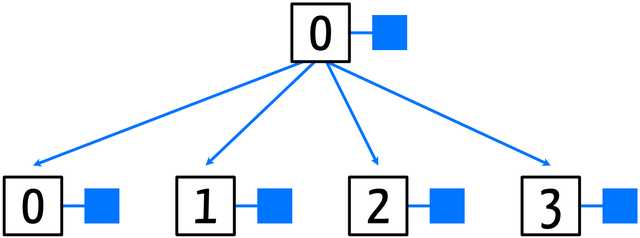
Collective Communication

So far, we have covered the topic of point-to-point communication: with a message that is exchanged between a sender and a receiver. However, in a lot of applications, collective communication may be required.

- Broadcast: Send data to all the processes
- Scatter: Distribute data between the processes
- Gather: Collect data from multiple processes to one process
- Reduce: Perform a reduction

Broadcast

During a broadcast, one process (the root) sends the same data to all processes in a communicator.



Here, the root of the broadcast is the process with rank 0 which send data to the three other processes in a communicator. At the end of the broadcast, the four processes have the same piece of data.

Broadcast

Broadcasting with MPI is done using the **MPI_Bcast** function.

```
MPI_Bcast(void* address,             = address of the data you want to broadcast
          int count,                 = number of elements to broadcast
          MPI_Datatype datatype,     = the type of data we want to broadcast
          int root,                  = rank of the broadcast root
          MPI_Comm communicator)     = the communicator used for this broadcast
```

```
MPI_Bcast(address, count, datatype, root, communicator, ierror)
          type(*), dimension(..) :: address
          integer, intent(int)   :: count, datatype, root, communicator
          integer, intent(out) :: ierror
```

Broadcast Example

```c
MPI_Init(&argc, &argv);

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int bcast_root = 0;

int value;
if(rank == bcast_root) {
    value = 12345;
    printf("I am the broadcast root with rank %d "
           "and I send value %d.\n", rank, value);
}

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

if(rank != bcast_root) {
    printf("I am a broadcast receiver with rank %d "
           "and I obtained value %d.\n", rank, value);
}

MPI_Finalize();
```

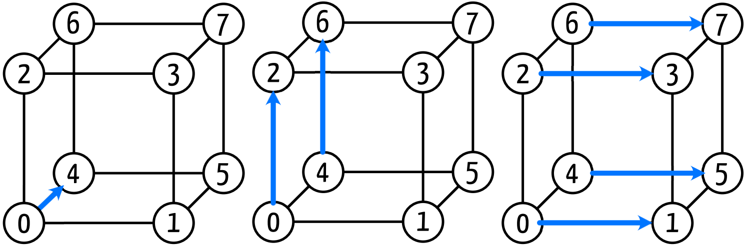
```fortran
      integer :: i, value, ierror
      integer :: rank, bcast_root

      call MPI_Init(ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      bcast_root = 0

      if (rank .eq. bcast_root) then
        value = 12345
        print 100, rank, value
100     format('I am the broadcast root with rank ', i0, &
&              ' and I send value ', i0)
      end if

      call MPI_Bcast(value, 1, MPI_INT, bcast_root, &
&                    MPI_COMM_WORLD, ierror)

      if (rank .ne. bcast_root) then
        print 200, rank, value
200     format('I am a broadcast receiver with rank ', i0, &
&              ' and I obtained value ', i0)
      end if

      call MPI_Finalize(ierror)
```

Broadcast Example

```
 $ mpicc -o broadcast broadcast.c
 $ mpirun -np 4 ./broadcast
I am the broadcast root with rank 0 and I send value 12345.
I am a broadcast receiver with rank 2 and I obtained value 12345.
I am a broadcast receiver with rank 1 and I obtained value 12345.
I am a broadcast receiver with rank 3 and I obtained value 12345.
```

Broadcast hypercube

A one to all broadcast can be visualized on a hypercube of d dimensions with $d = \log_2 p$.

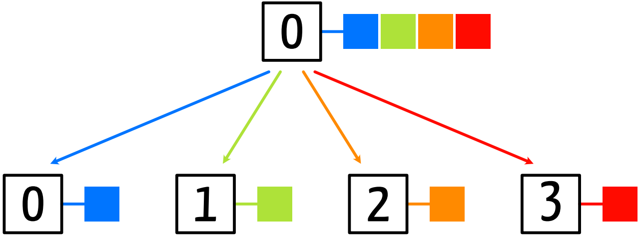


The broadcast procedure involves $\log_2 p$ point to point simple message transfers.

$$T_{broad} = \left(T_{latency} + \frac{n}{B_{peak}} \right) \log_2 p$$

MPI Scatter

During a scatter, the elements of an array are distributed in the order of process rank.



Here, the root of the scatter is the process with rank 0 which send data to the three other processes in a communicator. At the end of the scatter, the four processes have one element of the array.

MPI Scatter

Scattering with MPI is done using the **`MPI_Scatter`** function.

```
MPI_Scatter(void* saddress,           = address of the data you want to scatter
            int scount,               = number of elements sent to each process
            MPI_Datatype sdatatype,   = the type of data we want to scatter
            void* raddress,           = where to receive the data
            int rcount,               = number of elements to receive
            MPI_Datatype rdatatype,   = the type of data we want to receive
            int root,                 = rank of the scatter root
            MPI_Comm communicator)    = the communicator used for this scatter
```

```
MPI_Scatter(saddress, scount, sdatatype, raddress, rcount, rdatatype, &
&           root, communicator, ierror)
            type(*), dimension(..), intent(in) :: saddress
            type(*), dimension(..)             :: raddress
            integer, intent(in)                :: scount, sdatatype, rdatatype
            integer, intent(in)                :: rcount, root, communicator
            integer, intent(out)               :: ierror
```

Scatter Example

```c
MPI_Init(&argc, &argv);

int size, rank, value, scatt_root = 0;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int* data = NULL;
if(rank == scatt_root) {
    data = (int*)malloc(sizeof(int)*size);

    printf("Values to scatter from process %d:", rank);
    for (int i = 0; i < size; i++) {
        data[i] = 100 * i;
        printf(" %d", data[i]);
    }
    printf("\n");
}

MPI_Scatter(data, 1, MPI_INT, &value, 1, MPI_INT,
            scatt_root, MPI_COMM_WORLD);
printf("Process %d received value %d.\n", rank, value);

if(rank == scatt_root) free(data);

MPI_Finalize();
```

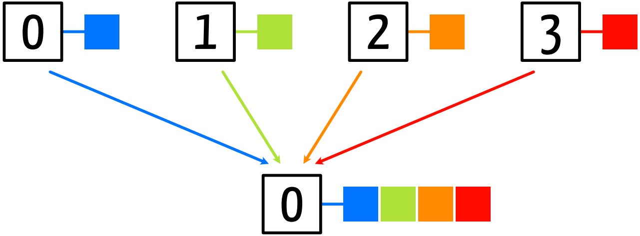
```fortran
      integer :: size, rank, ierror
      integer :: value, scatt_root, i
      integer, dimension(:), allocatable :: buffer

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      if (rank .eq. scatt_root) then
        allocate(buffer(size))
        do i = 1,size
          buffer(i) = 100 * i
        end do

        print 100, rank, data
100     format('Values to scatter from process ', i0, &
&                      ':', *(1x,i0))
      end if

      call MPI_Scatter(buffer, 1, MPI_INTEGER, value, 1,
                 MPI_INTEGER, &
&                          scatt_root, MPI_COMM_WORLD, ierror)

      print 200, rank, value
200   format('Process ', i0, ' received value ', i0)

      if (rank .eq. scatt_root) deallocate(buffer)

      call MPI_Finalize(ierror)
```

Scatter Example

```
 $ mpicc -o scatter scatter.c
 $ mpirun -np 4 ./scatter
Values to scatter from process 0: 0 100 200 300
Process 1 received value 100.
Process 2 received value 200.
Process 0 received value 0.
Process 3 received value 300.
```

MPI Gather

A gathering is taking elements from each process and gathers them to the root process.



Here we take one element of each process and gather them together is an array in the process with rank 0.

MPI Gather

Scattering with MPI is done using the **`MPI_Gather`** function.

```
MPI_Gather(void* saddress,              = address of the data you want to gather
           int scount,                  = number of elements to gather
           MPI_Datatype sdatatype,      = the type of data we want to gather
           void* raddress,              = where to receive the data
           int rcount,                  = number of elements to receive
           MPI_Datatype rdatatype,      = the type of data we want to receive
           int root,                    = rank of the gather root
           MPI_Comm communicator)       = the communicator used for this gather
```

```
MPI_Gather(saddress, scount, sdatatype, raddress, rcount, rdatatype, &
&          root, communicator, ierror)
           type(*), dimension(..), intent(in) :: saddress
           type(*), dimension(..)             :: raddress
           integer, intent(in)                :: scount, sdatatype, rcount,
           integer, intent(in)                :: rdatatype, root, communicator
           integer, intent(out)               :: ierror
```

Gather Example

```c
int gath_root = 0;

int size, rank, ierror, value;
int *buffer;

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

value = rank * 100;
printf("Process %d has value %d.\n", rank, value);

if (rank == gath_root) buffer = (int*)malloc(sizeof(int)*size);
MPI_Gather(&value, 1, MPI_INT, buffer, 1, MPI_INT,
           gath_root, MPI_COMM_WORLD);

if (rank == gath_root) {
  printf("Values collected on process %d:", rank);
  for (int i = 0; i < size; ++i) printf(" %d", buffer[i]);
  printf(".\n");

  free(buffer);
}

MPI_Finalize();
```

```fortran
      integer, parameter :: gath_root = 0

      integer :: size, rank, ierror, value
      integer, dimension(:), allocatable :: buffer

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      value = rank * 100

      print 100, rank, value
100   format('Process ', i0, ' has value ', i0, '.')

      if (rank .eq. gath_root) allocate(buffer(size))
      call MPI_Gather(value, 1, MPI_INTEGER, buffer, 1, &
&         MPI_INTEGER, gath_root, MPI_COMM_WORLD, ierror)

      if (rank .eq. gath_root) then
        print 200, rank, buffer
200     format('Values collected on process ', i0, &
&              ': ', *(1x,i0), '.')

        deallocate(buffer)
      end if

      call MPI_Finalize(ierror)
```

Gather Example

```
 $ mpicc -o gather gather.c
 $ mpirun -np 4 ./gather
Process 2 has value 200.
Process 0 has value 0.
Process 3 has value 300.
Process 1 has value 100.
Values collected on process 0: 0 100 200 300.
```

Back to the Sum of Integer

If we go back to the communication part of the sum of integer.

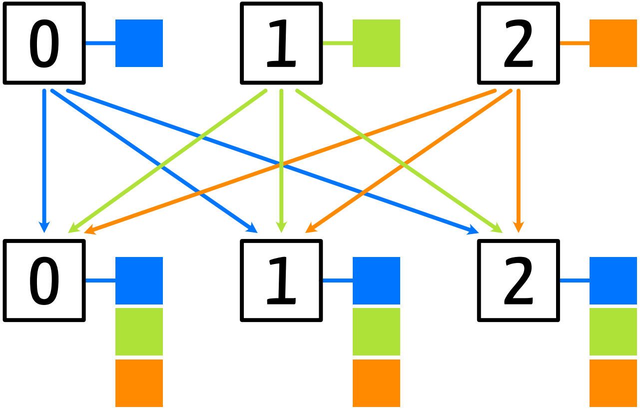
```
if (rank > 0) {
  MPI_Send(&proc_sum, 1, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD);
} else {
  unsigned int remote_sum;
  for(int src = 1; src < world_size; ++src) {
    MPI_Recv(&remote_sum, 1, MPI_UNSIGNED, src, 1, MPI_COMM_WORLD, &status);
    proc_sum += remote_sum;
  }
}
```

We can rewrite this part of the code with a gather

```
unsigned int* remote_sums;
if(rank == 0) remote_sums = (unsigned int*)malloc(sizeof(int)*world_size);

MPI_Gather(&proc_sum, 1, MPI_UNSIGNED, remote_sums, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

if(rank == 0) {
  unsigned int sum = 0;
  for(int i = 0; i < world_size; ++i)
    sum += remote_sums[i];
}
```

MPI All Gather

A process can have multiple ranks depending on the communicator. Here processes 2 and 3 have ranks 1 and 2 for communicator 1 while in communicator 2 their ranks are 0 and 1 respectively.



MPI All Gather

An all gather with MPI is done using the **MPI_Allgather** function.

```
MPI_Allgather(void* saddress,              = address of the data you want to gather
              int scount,                  = number of elements to gather
              MPI_Datatype sdatatype,      = the type of data we want to gather
              void* raddress,              = where to receive the data
              int rcount,                  = number of elements to receive
              MPI_Datatype rdatatype,      = the type of data we want to receive
              MPI_Comm communicator)       = the communicator used for this gather
```

```
MPI_Allgather(saddress, scount, sdatatype, raddress, rcount, rdatatype, &
&             communicator, ierror)
              type(*), dimension(..), intent(in) :: saddress
              type(*), dimension(..)             :: raddress
              integer, intent(in)                :: scount, sdatatype, rcount,
              integer, intent(in)                :: rdatatype, communicator
              integer, intent(out)               :: ierror
```

All Gather Example

```c
int gath_root = 0;
int size, rank, ierror, value;
int *buffer;

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

value = rank * 100;

printf("Process %d has value %d.\n", rank, value);

buffer = (int*)malloc(sizeof(int)*size);
MPI_Allgather(&value, 1, MPI_INT, buffer, 1,
              MPI_INT, MPI_COMM_WORLD);

printf("Values collected on process %d:", rank);
for (int i = 0; i < size; ++i) printf(" %d", buffer[i]);
printf(".\n");

free(buffer);

MPI_Finalize();
```

```fortran
      integer, parameter :: gath_root = 0

      integer :: size, rank, ierror, value
      integer, dimension(:), allocatable :: buffer

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      value = rank * 100
      print 100, rank, value
100   format('Process ', i0, ' has value ', i0, '.')

      allocate(buffer(size))

      call MPI_Allgather(value, 1, MPI_INTEGER, buffer, 1, &
&                         MPI_INTEGER, MPI_COMM_WORLD, ierror)

      print 200, rank, buffer
200   format('Values collected on process ', i0, &
&            ': ', *(1x,i0), '.')

      deallocate(buffer)

      call MPI_Finalize(ierror)
```

All Gather Example

```
 $ mpicc -o allgather allgather.c
 $ mpirun -np 4 ./allgather
Process 2 has value 200.
Process 0 has value 0.
Process 3 has value 300.
Process 1 has value 100.
Values collected on process 1: 0 100 200 300.
Values collected on process 3: 0 100 200 300.
Values collected on process 0: 0 100 200 300.
Values collected on process 2: 0 100 200 300.
```

MPI All to All

All to all communication allows for data distribution to all processes.

source							destination					
10	20	30	40	50	60	0	10	20	11	21	12	22
11	21	31	41	51	61	1	30	40	31	41	32	42
12	22	32	42	52	62	2	50	60	51	61	52	62

Here each process receive two elements from each processes in the group.

MPI All to All

All to all with MPI is done using the **`MPI_Alltoall`** function.

```
MPI_Alltoall(void* saddress,            = address of the data you want to send
             int scount,                = number of elements to send
             MPI_Datatype sdatatype,    = the type of data we want to send
             void* raddress,            = where to receive the data
             int rcount,                = number of elements to receive
             MPI_Datatype rdatatype,    = the type of data we want to receive
             MPI_Comm communicator)     = the communicator used
```

```
MPI_Alltoall(saddress, scount, sdatatype, raddress, rcount, rdatatype, &
&            communicator, ierror)
    type(*), dimension(..), intent(in) :: saddress
    type(*), dimension(..)             :: raddress
    integer, intent(in)                :: scount, sdatatype, rcount
    integer, intent(in)                :: sdatatype, communicator
    integer, intent(out)               :: ierror
```

All to All Example

```c
MPI_Init(&argc, &argv);

int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int* sendbuf = (int*)malloc(sizeof(int)*size);
int* recvbuf = (int*)malloc(sizeof(int)*size);

for (int i = 0; i < size; ++i)
    sendbuf[i] = (rank * size + i) * 100;

printf("Process %d will send", rank);
for (int i = 0; i < size; ++i) printf(" %d", sendbuf[i]);
printf("\n");

MPI_Alltoall(sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT,
       MPI_COMM_WORLD);

printf("Process %d collected values", rank);
for (int i = 0; i < size; ++i) printf(" %d", recvbuf[i]);
printf("\n");

free(sendbuf); free(recvbuf);
MPI_Finalize();
```

```fortran
      integer, dimension(:), allocatable :: sendbuf, recvbuf

      integer :: rank, size, ierror
      integer :: i

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      allocate(sendbuf(size), recvbuf(size))

      do i = 1,size
        sendbuf(i) = (rank * size + i - 1) * 100
      end do

      print 100, rank, sendbuf
100   format('Process ', i0, ' will send:', *(1x,i0))

      call MPI_Alltoall(sendbuf, 1, MPI_INTEGER, recvbuf, 1, &
&                       MPI_INTEGER, MPI_COMM_WORLD, ierror)

      print 200, rank, recvbuf
200   format('Process ', i0, ' collected values', *(1x,i0))

      deallocate(sendbuf, recvbuf)
      call MPI_Finalize(ierror)
```

All to All Example

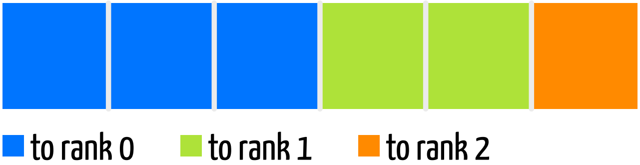
```
 $ mpicc -o alltoall alltoall.c
 $ mpirun -np 4 ./alltoall
Process 2: values = 800, 900, 1000, 1100
Process 1: values = 400, 500, 600, 700
Process 0: values = 0, 100, 200, 300
Process 3: values = 1200, 1300, 1400, 1500
Value collected on process 3: values = 300, 700, 1100, 1500
Value collected on process 1: values = 100, 500, 900, 1300
Value collected on process 0: values = 0, 400, 800, 1200
Value collected on process 2: values = 200, 600, 1000, 1400
```

Vector Variants

Ok, but what if I do not want to transfer the same number of elements from each process?

- All the functions presented previously have a ”v” variant
- These variants allow the messages received to have different lengths and be stored at arbitrary locations

counts = {3, 2, 1}   displacements = {0, 3, 5}



Vector Variants: Scatterv

For example, the vector variant of the **`MPI_Scatter`** function is **`MPI_Scatterv`**

```
MPI_Scatterv(void* saddress,            = address of the data you want to send
             int scount[],              = the number of elements to send to each process
             int displs[],              = the displacement to the message sent to each process
             MPI_Datatype sdatatype,    = the type of data we want to send
             void* raddress,            = where to receive the data
             int rcount,                = number of elements to receive
             MPI_Datatype rdatatype,    = the type of data we want to receive
             int root,                  = rank of the root proces
             MPI_Comm communicator)     = the communicator used
```

Vector Variants: Scatterv

For example, the vector variant of the `MPI_Scatter` function is `MPI_Scatterv`

```
MPI_Scatterv(saddress, scount, displs, sdatatype, raddress, rcount, rdatatype, &
&            root, communicator, ierror)

    type(*), dimension(..), intent(in) :: saddress
    type(*), dimension(..)             :: raddress
    integer, intent(in)                :: scount(*), displs(*)
    integer, intent(in)                :: sdatatype, rdatatype, count_recv
    integer, intent(in)                :: root, communicator
    integer, intent(out)               :: ierror
```

Scatterv Example

```c
int rank, size;
int *sendbuf, *recvbuf;
int *displs, *nelems;

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

if (rank == 0) {
  int n = size*(size+1)/2;
  sendbuf = (int*)malloc(sizeof(int)*n);
  displs  = (int*)malloc(sizeof(int)*size);
  nelems  = (int*)malloc(sizeof(int)*size);

  for (int i = 0; i < n; ++i) sendbuf[i] = 100*(i+1);

  for (int i = 0; i < size; ++i) {
    displs[i] = i*(i+1)/2;
    nelems[i] = i+1;
  }
}

recvbuf = (int*)malloc(sizeof(int)*(rank+1));
MPI_Scatterv(sendbuf, nelems, displs, MPI_INT, recvbuf, rank+1,
       MPI_INT, 0, MPI_COMM_WORLD);

printf("Process %d received values:", rank);
for(int i = 0; i < rank+1; i++) printf(" %d", recvbuf[i]);
printf("\n");

MPI_Finalize();
```

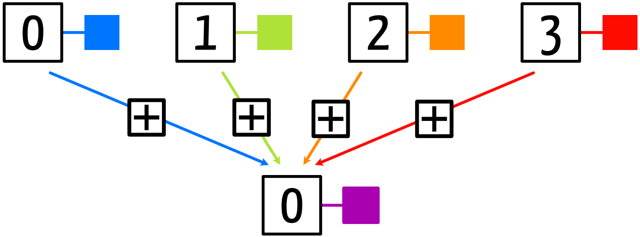
```fortran
      integer, dimension(:), allocatable :: sendbuf, recvbuf
      integer, dimension(:), allocatable :: displs, nelems

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      if(rank .eq. 0) then
        n = size*(size+1)/2
        allocate(sendbuf(n))
        allocate(displs(size), nelems(size))

        do i = 1,n
          sendbuf(i) = 100*i
        end do

        do i = 1,size
          displs(i) = i*(i-1)/2
          nelems(i) = i
        end do
      end if

      allocate(recvbuf(rank+1))
      call MPI_Scatterv(sendbuf, nelems, displs, MPI_INTEGER,
           recvbuf, rank+1, MPI_INTEGER, 0, MPI_COMM_WORLD,
           ierror)

      print 100, rank, recvbuf
100   format('Process ', i0, ' received values:', *(1x,i0))

      call MPI_Finalize(ierror)
```

Scatterv Example

```
 $ mpicc -o scatterv scatterv.c
 $ mpirun -np 4 ./scatterv
Process 0 received values: 100
Process 1 received values: 200 300
Process 3 received values: 700 800 900 1000
Process 2 received values: 400 500 600
```

MPI Reduce

Data reduction is reducing a set of numbers into a smaller set of numbers. For example, summing elements of an array or find the min/max value in an array.



Here we take one element from each process and sum them to the process of rank 0.

MPI Reduce

Reduction with MPI is done using the **MPI_Reduce** function.

```
MPI_Reduce(void* saddress,
           void* raddress,
           int count,
           MPI_Datatype datatype,
           MPI_Op operation,
           int root,
           MPI_Comm communicator)
```

= address of the data you want to reduce
= address of where to store the result
= the number of data elements
= the type of data we want to reduce
= the type operation to perform
= rank of the reduction root
= the communicator used for this reduction

```
MPI_Reduce(saddress, raddress, count, datatype, operation, root, &
&          communicator, ierror)
             type(*), dimension(..), intent(in) :: saddress
             type(*), dimension(..)             :: raddress
             integer, intent(in)                :: count, datatype, operation
             integer, intent(in)                :: root, communicator
             integer, intent(out)               :: ierror
```

MPI Reduction Operators

MPI has a number of elementary reduction operators, corresponding to the operators of the C programming language.

MPI Op	Operation	MPI Op	Operation
`MPI_MIN`	`min`	`MPI_LAND`	`&&`
`MPI_MAX`	`max`	`MPI_LOR`	`||`
`MPI_SUM`	`+`	`MPI_BAND`	`&`
`MPI_PROD`	`*`	`MPI_BOR`	`|`

In addition you can create your own custom operator type.

Back to the Sum of Integers

If we go back to the communication part of the sum of integer.

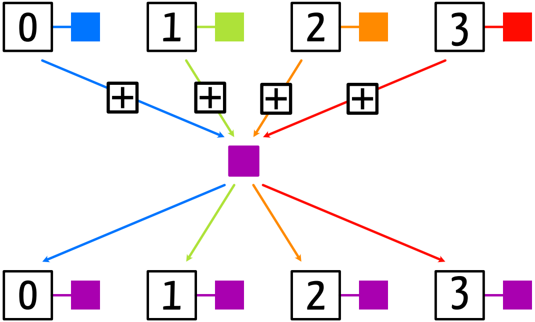
```
if (rank > 0) {
  MPI_Send(&proc_sum, 1, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD);
} else {
  unsigned int remote_sum;
  for(int src = 1; src < world_size; ++src) {
    MPI_Recv(&remote_sum, 1, MPI_UNSIGNED, src, 1, MPI_COMM_WORLD, &status);
    proc_sum += remote_sum;
  }
}
```

We can rewrite this part of the code with a reduction

```
unsigned int final_sum;
MPI_Reduce(&proc_sum, &final_sum, 1, MPI_UNSIGNED, MPI_SUM, 0, MPI_COMM_WORLD);
```

MPI All Reduce

You can also use an all reduce operation so that the result is available to all the processes in the communicator.



Here we take one element from each process and sum them. The result of the reduction is then broadcasted to all the processes.

MPI All Reduce

Reduction with MPI is done using the **`MPI_Allreduce`** function.

```
MPI_Allreduce(void* saddress,            = address of the data you want to reduce
              void* raddress,            = address of where to store the result
              int count,                 = the number of data elements
              MPI_Datatype datatype,     = the type of data we want to reduce
              MPI_Op operation,          = the type operation to perform
              MPI_Comm communicator)     = the communicator used for this reduction
```

```
MPI_Allreduce(saddress, raddress, count, datatype, operation, root, &
&             communicator, ierror)
              type(*), dimension(..), intent(in) :: saddress
              type(*), dimension(..)             :: raddress
              integer, intent(in)                :: count, datatype, operation
              integer, intent(in)                :: communicator
              integer, intent(out)               :: ierror
```

Reduce and Allreduce Example

As an example of the `MPI_Allreduce` and `MPI_Reduce` functions, we will consider the computation of standard deviation

$$\sigma = \sqrt{\frac{\sum_i (x_i - \mu)^2}{N}}$$

- x_i: value of the population
- μ: mean of the population
- N: size of the population

Reduce and Allreduce Example

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

srand(time(NULL) * rank);

for (int i = 0; i < nelems_per_rank; ++i)
  values[i] = (rand() / (double)RAND_MAX);

for (int i = 0; i < nelems_per_rank; ++i)
  local_sum += values[i];

MPI_Allreduce(&local_sum, &global_sum, 1, MPI_DOUBLE,
              MPI_SUM, MPI_COMM_WORLD);
mean = global_sum / (nelems_per_rank * size);

for (int i = 0; i < nelems_per_rank; ++i)
  local_sq_diff += (values[i] - mean) * (values[i] - mean);

MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
  double stddev = sqrt(global_sq_diff / (nelems_per_rank * size));
  printf("Mean = %lf, Standard deviation = %lf\n", mean, stddev);
}
```

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

      call random_seed()
      call random_number(values)

      loc_sum = 0.0
      do i = 1, nelems_per_rank
        loc_sum = loc_sum + values(i)
      enddo

      call MPI_Allreduce(loc_sum, global_sum, 1, &
&           MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD, ierror)

      mean = global_sum / (nelems_per_rank * size)
      loc_sq_diff = 0.0
      do i = 1, nelems_per_rank
        loc_sq_diff = loc_sq_diff &
&                    + (values(i) - mean) * (values(i) - mean)
      end do

      call MPI_Reduce(loc_sq_diff, global_sq_diff, 1, &
&           MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD,
       ierror)
      if (rank .eq. 0) then
        stddev = dsqrt(global_sq_diff / dble(nelems_per_rank *
               size))
        print 100, mean, stddev
100     format('Mean = ', f12.6, ', Standard deviation = ', f12.6,
        f12.6)
      end if
```

MPI Barrier

A barrier can be used to synchronize all processes in a communicator. Each process wait until all processes reach this point before proceeding further.

```
MPI_Barrier(MPI_Comm communicator)
```

For example:

```c
MPI_Init(&argc, &argv);

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

printf("Process %d: I start waiting at the barrier.\n",
       rank);

MPI_Barrier(MPI_COMM_WORLD);

printf("Process %d: I'm on the other side of the barrier.\n",
       rank);

MPI_Finalize();
```

```fortran
    integer :: rank, ierror

    call MPI_Init(ierror)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

    print 100, rank
100 format('Process ', i0, ': I start waiting at the barrier.')

    call MPI_Barrier(MPI_COMM_WORLD, ierror);

    print 200, rank
200 format('Process ', i0, &
&             ': I am on the other side of the barrier.')

    call MPI_Finalize(ierror)
```

Conclusion

Summary

Today, we covered the following topics:

- Point to point communication
- Non-blocking communication
- Collective communication

But we only scratch the surface: the possibilities offered by MPI are much broader than what we have discussed.

Going further

The possibilities offered by MPI are much broader than what we have discussed.

- User-defined datatype
- Persistent communication
- One-sided communication
- File I/O
- Topologies