Message Passing Interface
Distributed-Memory Parallel Programming

Orian Louant

orian.louant@uliege.be

October 20, 2020

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

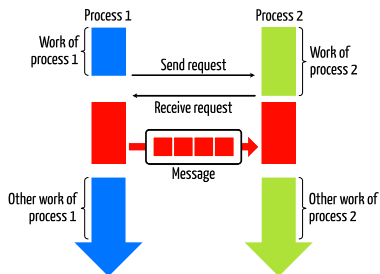
It is not a particular implementation.

- MPI standard defines a library by specifying the names and results of functions
- An MPI program should be able to run on all MPI implementations
- Vendors provide an MPI implementation for their machines and there is free, open source, implementations available as well

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

Initializing and Finalizing the MPI Environment

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

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

```
MPI_Finalize()
```

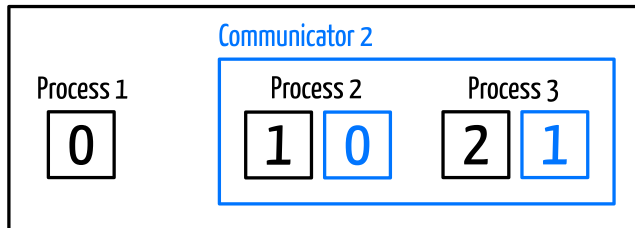
Terminates MPI execution environment. Once this function has been called, no MPI function may be called afterward.

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 rank inside each of them.

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 and Rank

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)
```

While the rank of a process in a communicator is obtained using the **MPI_Comm_rank** function.

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

I Love 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;
}
```

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` compiler wrapper

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

`mpicc` 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`.

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

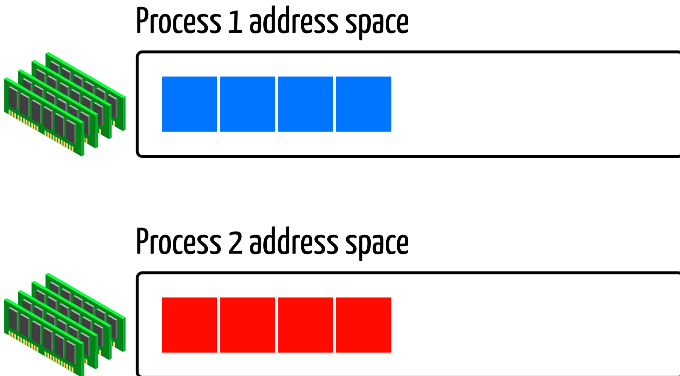
Save your job submission script and run it with `sbatch`

```
 $ 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.
```

Point to Point Communication

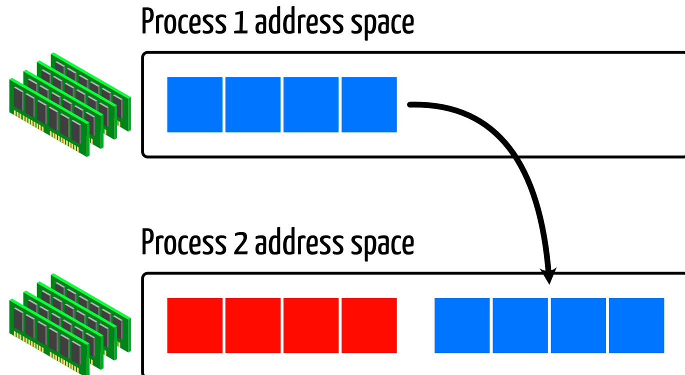
Point to Point Communication

Multiple processes are spawn to run in parallel and in the message-passing model of parallel computation, 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 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
         int tag,                   = identify the type of the message
         MPI_Comm communicator)     = the communicator used for this message
```

Receiving a Message

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

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

Receiving a Message

- For the communication to succeed, the receive buffer of which we pass the memory address as argument of the `MPI_Recv` function must be large enough to hold the message. If it is not, behaviour is undefined. The buffer may, however, be longer than the data received.
- The `count` argument is the 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.

MPI Data types

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.

The MPI Status Structure

When using the **MPI_Recv**, informations about the message can be obtained through the MPI_Status structure.

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

In addition you can use this structure and the **MPI_Get_count** function to get the actual size of the message.

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

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;
}
```

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.

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.

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);
}
```

Example: Sum of Integer

Now, consider a program that sums the integer between 1 and N. The serial code will be

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

int main(int argc, char* argv[]) {
  if(argc != 2) {
    printf("usage: isum <size>\n");
    exit(1);
  }

  int N = atoi(argv[1]);

  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;
}
```

Example: Sum of Integer

The easy way to parallelize this code is using OpenMP. On line of magic and we are done.

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

int main(int argc, char* argv[]) {
  if(argc != 2) {
    printf("usage: isum <size>\n");
    exit(1);
  }

  int N = atoi(argv[1]);

  unsigned int sum = 0;
  #pragma omp parallel for reduction(+:sum)
  for (int i = 1; i <= N; ++i) {
      sum += i;
  }

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

  return 0;
}
```

Example: Sum of Integer

To parallelize our program with MPI, we first start by initializing MPI and by getting information about our environment and divide the work among the processes.

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

if (rank == 0)
  printf("Running %d process with %d threads each.\n", world_size, omp_get_max_threads());

int start = (N * rank / world_size) + 1;
int   end = N * (rank+1) / world_size;
```

Example: Sum of Integer

In the next step, each process carries out the sum for its range of integers.

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

if (rank == 0)
  printf("Running %d process with %d threads each.\n", world_size, omp_get_max_threads());

int start = (N * rank / world_size) + 1;
int   end = N * (rank+1) / world_size;

unsigned int proc_sum = 0;
#pragma omp parallel for reduction(+:proc_sum)
for (int i = start; i <= end; ++i)
    proc_sum += i;
```

Example: Sum of Integer

Finally, each process send the result of its computation to the process with rank 0. The process with rank 0 receive the value and compute the final sum.

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

if (rank == 0)
  printf("Running %d processes with %d threads each.\n", world_size, omp_get_max_threads());

int start = (N * rank / world_size) + 1;
int   end = N * (rank+1) / world_size;

unsigned int proc_sum = 0;
#pragma omp parallel for reduction(+:proc_sum)
for (int i = start; i <= end; ++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;
  }
}
```

Example: Sum of Integer (final, part. 1)

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <omp.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("Running %d process with %d threads each.\n", world_size, omp_get_max_threads());
```

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

```
  int start = (N * rank / world_size) + 1;
  int   end = N * (rank+1) / world_size;

  unsigned int proc_sum = 0;
  #pragma omp parallel for reduction(+:proc_sum)
  for (int i = start; i <= end; ++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;
}
```

Compile an OpenMP+MPI Program

As `mpicc` is wrapper to a host compiler, we can use all the usual flags available with the host compiler. Compiling an OpenMP + MPI program is thus not different that without MPI.

For our sum of integer example:

```
$ mpicc -fopenmp -O3 -o isum integer_sum.c
```

Running an MPI+OpenMP Program on a CÉCI Cluster

We use `ntasks` to define the number of processes while the number of threads for each of these processes is defined by `cpus-per-task`. For example to run 4 processes with 4 threads each:

```
#!/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=4
#SBATCH --mem-per-cpu=1024 # megabytes
#SBATCH --partition=batch
#SBATCH --output=mpi_openmp.out

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

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

mpirun -np $SLURM_NTASKS ./isum 10000000
```

Running an MPI+OpenMP Program on a CÉCI Cluster

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

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

[...]

 $ cat mpi_openmp.out
Running 4 processes with 4 threads each.
Process 2 has local sum 4205194448
Process 0 has local sum 2560025808
Process 3 has local sum 732811472
Process 1 has local sum 3382610128
The sum of 1 to 10000000 is 2290707264.
```

Collective Communication

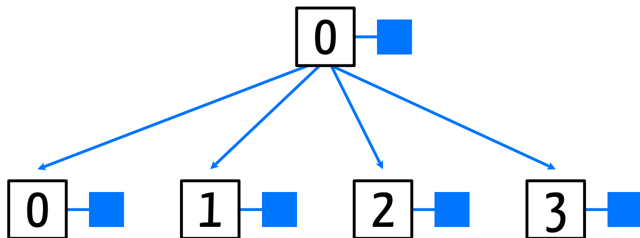
Collective Communication

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 betwen 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
```

Broadcast Example

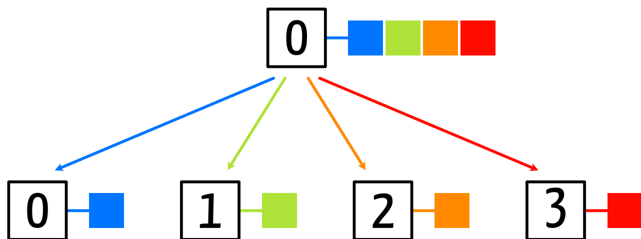
```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

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

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int bcast_root = 0;

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

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

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

  MPI_Finalize();

  return 0;
}
```

Broadcast Example

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

MPI Scatter

During a scatter, the elements of an array are ditributed 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
```

Scatter Example

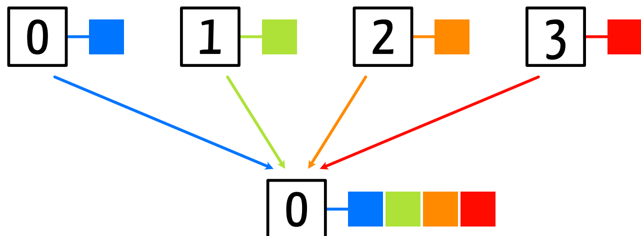
```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

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

    return EXIT_SUCCESS;
}
```

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
```

Gather Example (part one)

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

double *create_rand_nums(int num_elements) {
  double *rand_nums = (double *)malloc(sizeof(double) * num_elements);

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

  return rand_nums;
}

double compute_avg(double *array, int num_elements) {
  double sum = 0.f;

  for (int i = 0; i < num_elements; ++i) {
    sum += array[i];
  }

  return sum / num_elements;
}
```

Gather Example (continued)

```
int main(int argc, char** argv) {
  int nelems_pp = atoi(argv[1]);

  MPI_Init(&argc, &argv);

  int rank, world_size, gath_root = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  srand(time(NULL) + rank);

  double* rand_nums = create_rand_nums(nelems_pp);
  double       pavg = compute_avg(rand_nums, nelems_pp);

  double *sub_avgs = NULL;
  if (rank == gath_root) {
    sub_avgs = (double *)malloc(sizeof(double) * world_size);
  }

  MPI_Gather(&pavg, 1, MPI_DOUBLE, sub_avgs, 1, MPI_DOUBLE, gath_root, MPI_COMM_WORLD);

  if (rank == gath_root) {
    double avg = compute_avg(sub_avgs, world_size); free(sub_avgs);
    printf("Average of all elements is %lf\n", avg);
  }

  free(rand_nums);
  MPI_Finalize();

  return 0;
}
```

Back to the Sum of Integer

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

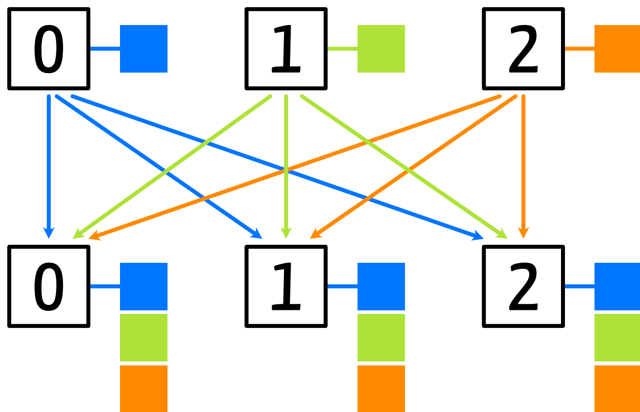
```
if (rank > 0) {
  MPI_Send(&proc_sum, 1, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD);
} else {
  unsigned int remote_sum;
  for(int src = 1; src < world_size; ++src) {
    MPI_Recv(&remote_sum, 1, MPI_UNSIGNED, src, 1, MPI_COMM_WORLD, &status);
    proc_sum += remote_sum;
  }
}
```

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.

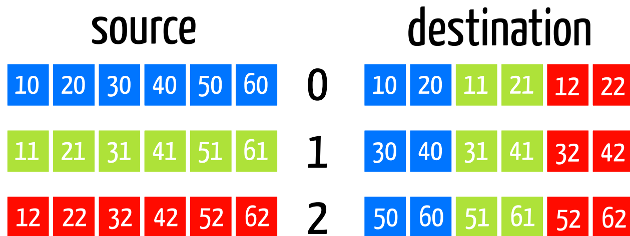
```
MPI_Allgather(void* saddress,                 = address of the data you want to gather
              int scount,                     = number of elements to gather
              MPI_Datatype sdatatype,         = the type of data we want to gather
              void* raddress,                 = where 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
```

All Gather Example

```c
int main(int argc, char** argv) {
  int nelems_pp = atoi(argv[1]);

  MPI_Init(&argc, &argv);

  int rank, world_size, gath_root = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  srand(time(NULL) + rank);

  double* rand_nums = create_rand_nums(nelems_pp);
  double       pavg = compute_avg(rand_nums, nelems_pp);

  double* sub_avgs = (double *)malloc(sizeof(double) * world_size);

  MPI_Allgather(&pavg, 1, MPI_DOUBLE, sub_avgs, 1, MPI_DOUBLE, MPI_COMM_WORLD);

  double avg = compute_avg(sub_avgs, world_size);
  printf("Average of all elements from process %d is %lf\n", rank, avg);

  free(sub_avgs);
  free(rand_nums);
  MPI_Finalize();

  return 0;
}
```

MPI All to All

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



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
```

All to All Example

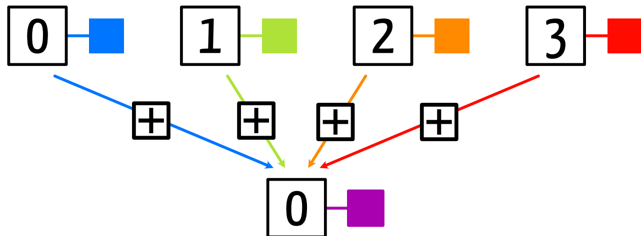
```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

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

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

    int* values = (int*)malloc(sizeof(int)*size);
    int*   recv = (int*)malloc(sizeof(int)*size);

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

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

    MPI_Alltoall(values, 1, MPI_INT, recv, 1, MPI_INT, MPI_COMM_WORLD);

    printf("Value collected on process %d: values = %d", rank, recv[0]);
    for (int i = 1; i < size; ++i) printf(", %d", recv[i]);
    printf("\n");

    free(values); free(recv);
    MPI_Finalize();

    return 0;
}
```

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
```

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,            = 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
           int root,                  = rank of the reduction root
           MPI_Comm communicator)     = the communicator used for this reduction
```

MPI Reduction Operators

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

<table>
<tr><th>MPI Op</th><th>Operation</th><th>MPI Op</th><th>Operation</th></tr>
<tr><td>MPI_MIN</td><td>min</td><td>MPI_LAND</td><td>&&</td></tr>
<tr><td>MPI_MAX</td><td>max</td><td>MPI_LOR</td><td>||</td></tr>
<tr><td>MPI_SUM</td><td>+</td><td>MPI_BAND</td><td>&</td></tr>
<tr><td>MPI_PROD</td><td>*</td><td>MPI_BOR</td><td>|</td></tr>
</table>

In addition you can create your own custom operator type.

Back to the Sum of Integer (again)

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 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:

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

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

    return 0;
}
```

Performance

Performance

- In order to extract the best performance from MPI, you need to hide the communication. This means that your code should spend most of its time doing the actual computation rather than communication.
- The same is true with the communication. Small messages are dominated by the latency. Try to have a message large enough to hide it.

Performance: Case Study

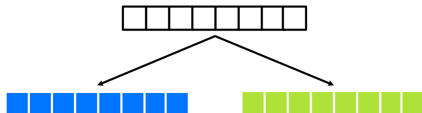
We will consider an hypothethical 1D stencil calculation where each cells are updated this way.

$$u_i^{new} = u_{i+1}^{old} - u_{i-1}^{old}$$

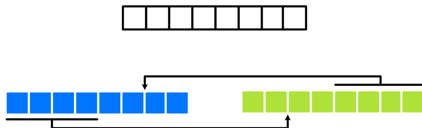
In order to update a cell, we require the value of the left and right cells.

Performance: Case Study

One way to do it, is to replicate the entire array to each of the processes.

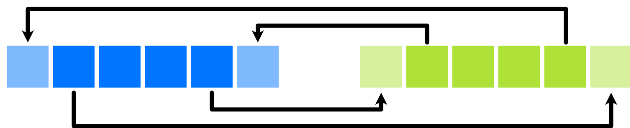


But then, at each step of the calculation, we will copy a large chunk of the data. Another problem is that the communication size increase with the size of the problem.



Performance: Case Study

The best way to solve this is to use a halo cells which contain the neighbour's value required for the calculation. Only these cells need to be updated through communication at each step.



This way, the size of the message will be the same whatever the problem size. Hopefully, the time spend in the calculation part, will hide the latency cost of the small message size.