24.11.2021

Parallel Computing with MPI

Luigi Santangelo
luigi.santangelo@unipv.it

University of Pavia

Roadmap

- Overview
  - What parallel computing is
  - Why it is so important
  - Parallel Computing Memory Architectures
  - Speed Up and Scalability
  - Functional and Domain Decomposition
- MPI: Point-to-Point functions
  - Communicators
  - Datatypes
  - Send and Receive functions
  - Synchronous, Blocking, Bufferend and Standard functions

Roadmap

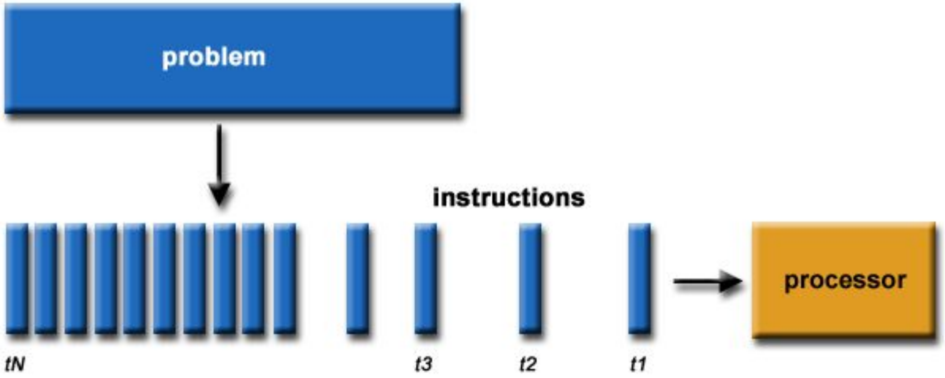
- MPI: Collective functions
    - Barrier, Broadcast
    - Scatter, Gather
    - AllGather, AllToAll
    - Reduce, AllReduce, ReduceScatter
    - Scan
- Building an MPI Cluster using Google Cloud

Parallel computing
an overview

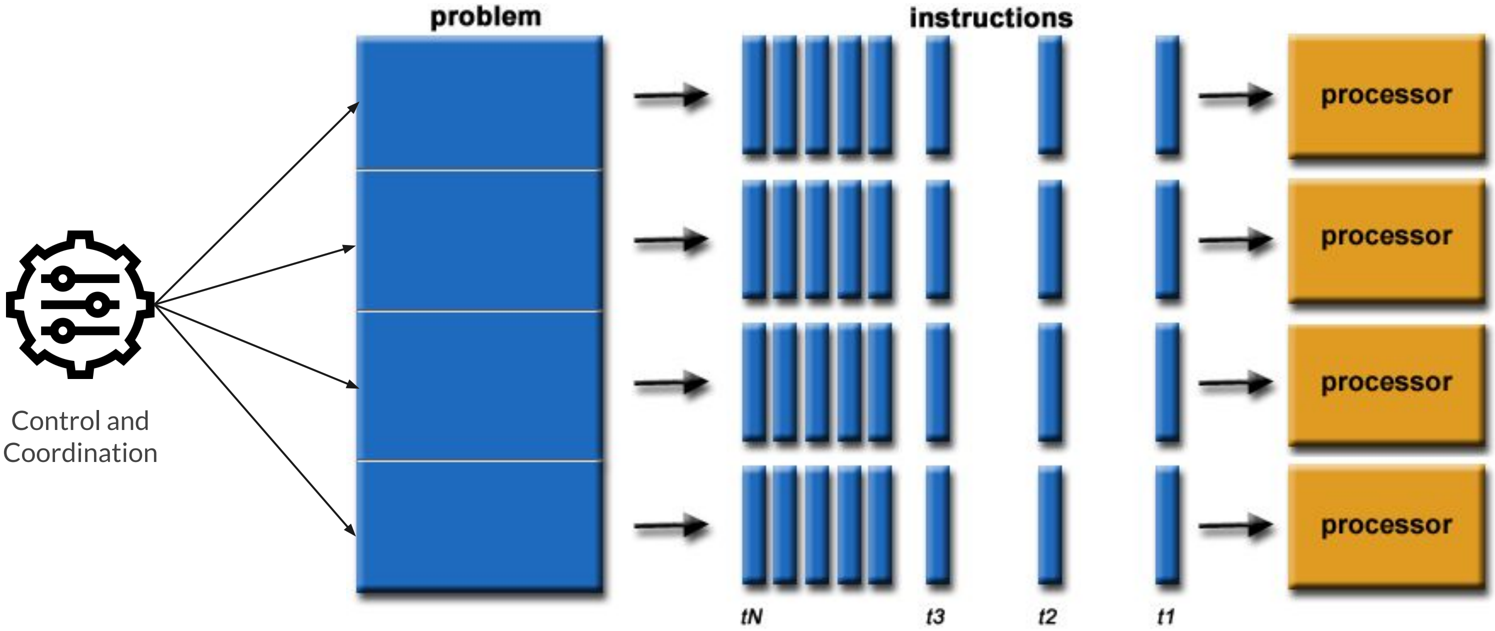
Serial Computing

Many techniques for speeding up the execution:

- pipelining
- loop unrolling
- branch prediction
- speculation
- register renaming
- dynamic scheduling
- out-of-order completion
- and so on



Parallel Computing



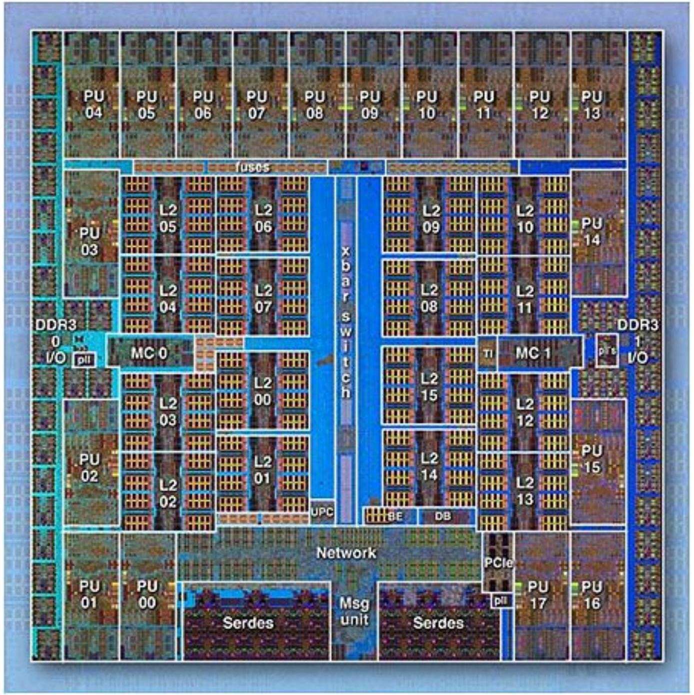
What do we need for parallel computing?

A single computer with multiple processors/cores

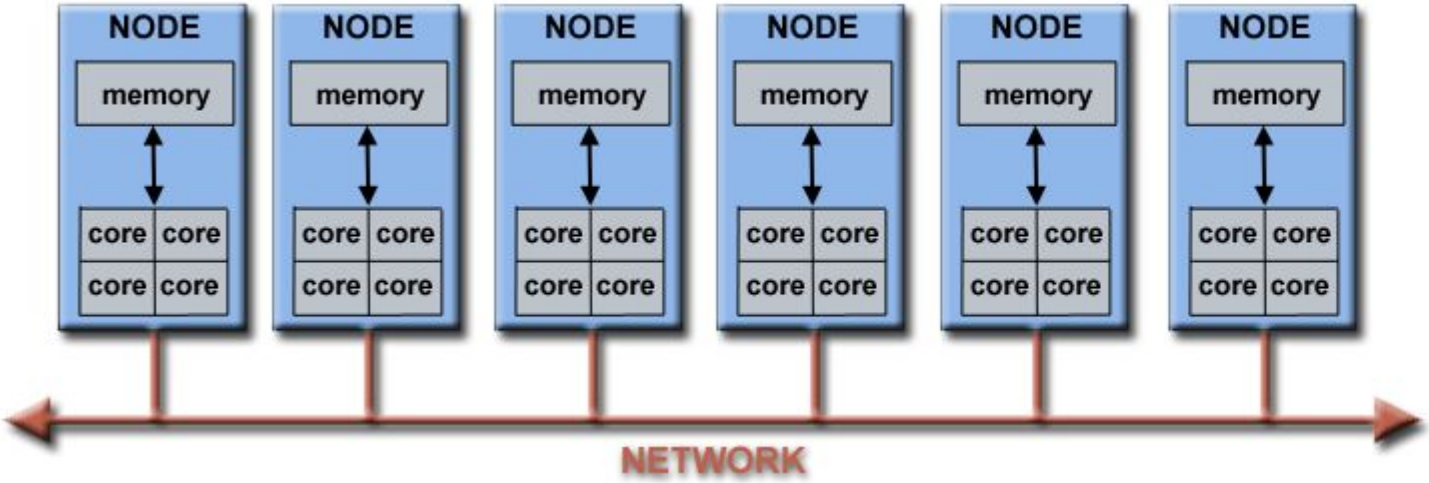
An arbitrary number of such computers connected by a network (**cluster**)

A single multicore system

- IBM BG/Q Compute Chip
- 18 cores
- 16 L2 cache units



A cluster of stand-alone computers



Parallel Computing in daily life

Galaxy Formation

Planetary Movments

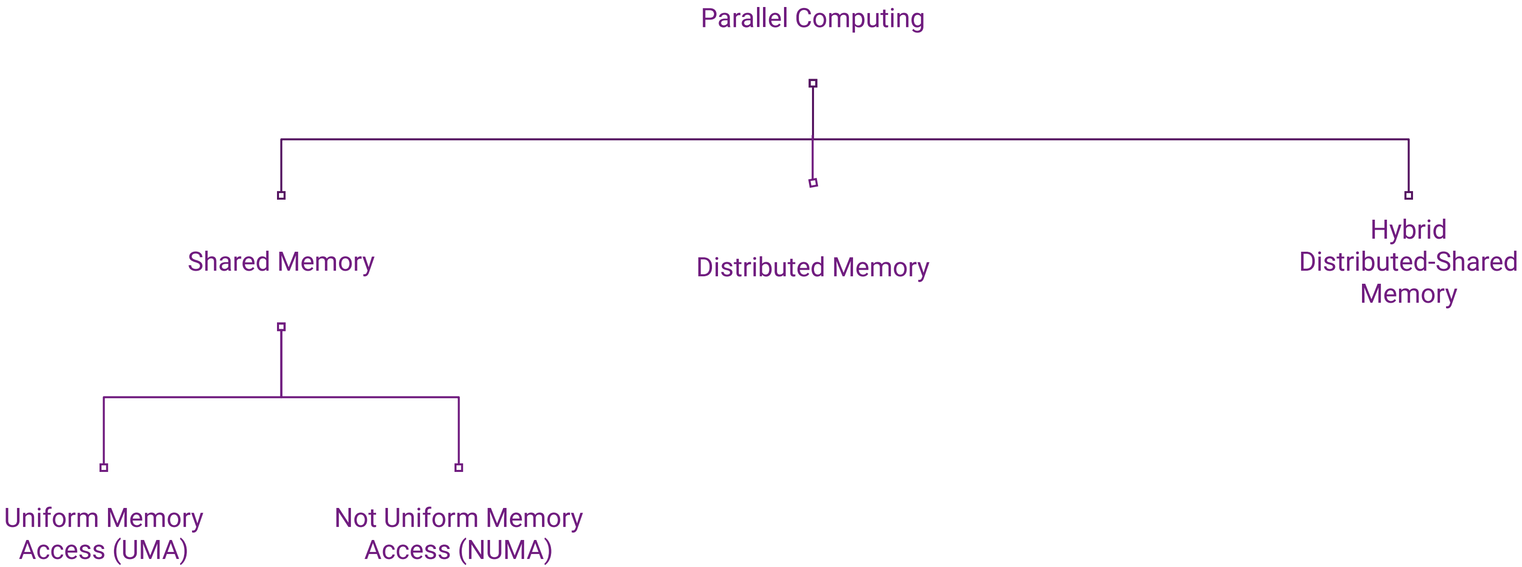
Climate Change

Rush Hour Traffic

Plate Tectonics

Weather

Parallel Computing Memory Architectures



Parallel Computing

Shared Memory

Distributed Memory

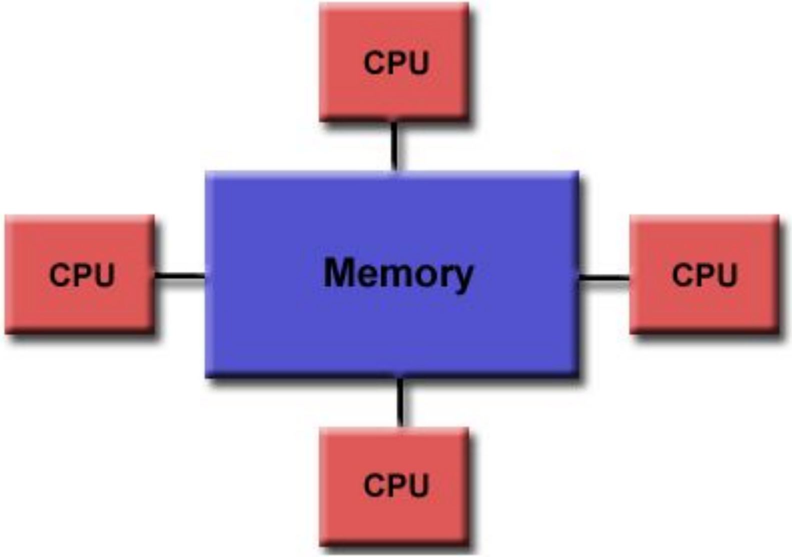
Hybrid Distributed-Shared Memory

Uniform Memory Access (UMA)

Not Uniform Memory Access (NUMA)

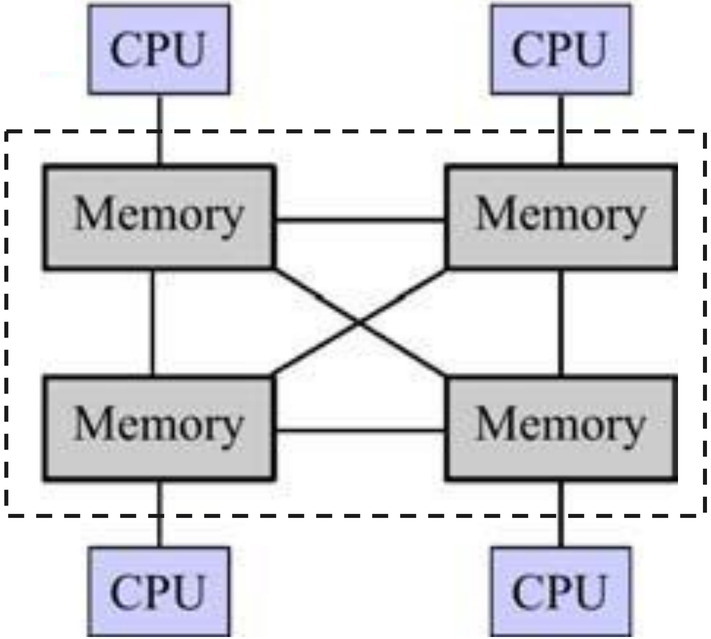
Shared Memory: Uniform Memory Access

- CPUs are identical to each others
- Memory is shared among all processes
- A single global space address
- Changes are seen by all processes
- Equal access times to memory
- Each CPU has got its own cache
  - Cache-coherency provided at hardware-level



Shared Memory: Not-Uniform Memory Access

- Memory is shared among CPUs
- A single global space address
- Memory access time is not uniform anymore
  - faster for accessing into local memory
  - slower for accessing into other cpu memories

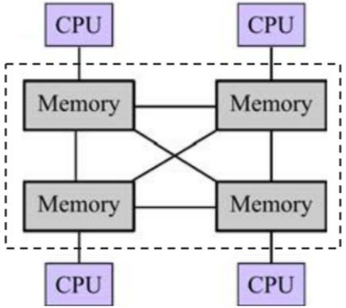


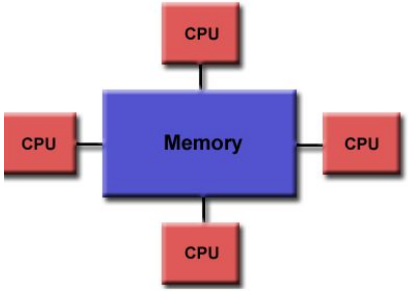
Shared Memory: trade-offs

👍

Global address space provides a user-friendly programming

Data sharing between tasks is both fast and uniform





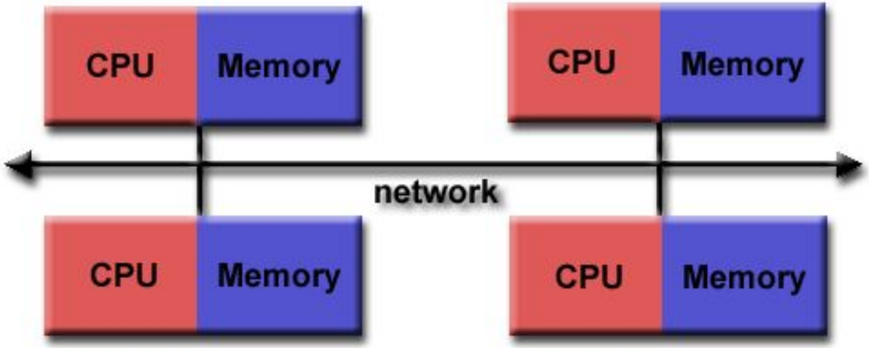
👎

Lack of scalability between memory and CPUs: adding more CPUs can geometrically **increases traffic** on the shared memory-CPU path

Programmer responsibility for **synchronization** constructs that ensure "correct" access of global memory

Distributed Memory

- Each CPU has its own local memory
- No Global Addressing
- Data changes don’t have any effect on the memory of other CPUs
- No Cache Coherence
- CPUs communicate through network



Distributed Memory: trade-offs

👍

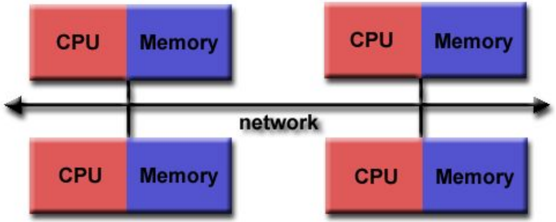
The system is **scalable** (increasing the number of CPUs, the size of memory increases too)

CPUs can fastly access to their own memory

👎

Programmer is responsible for data exchange

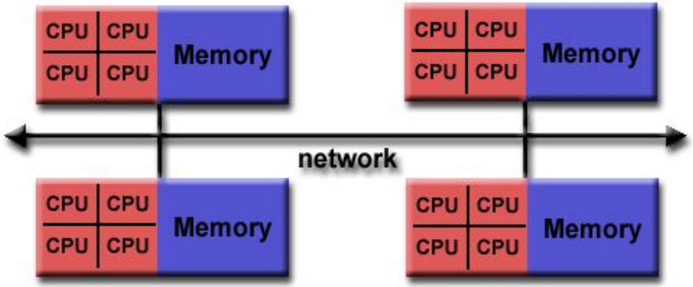
Getting data stored in remote node is slower than local data

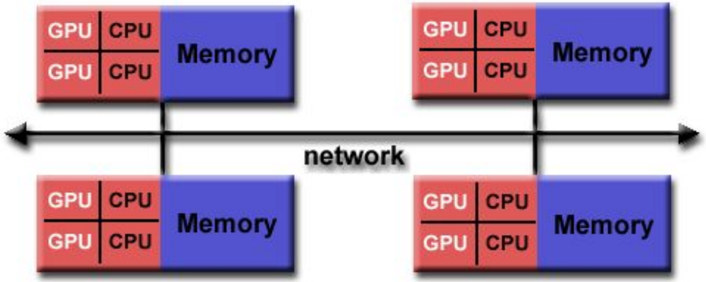


Hybrid Distributed-Shared Memory

The largest and fastest computers in the world today employ both shared and distributed memory architectures.

- CPUs
- GPUs
- Internode communication
- Intranode communication

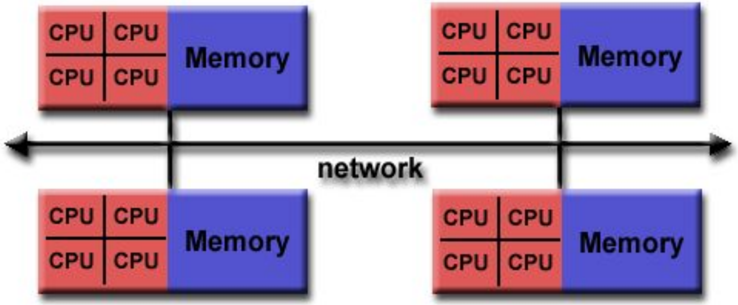




Hybrid Distributed-Shared Memory: trade-offs

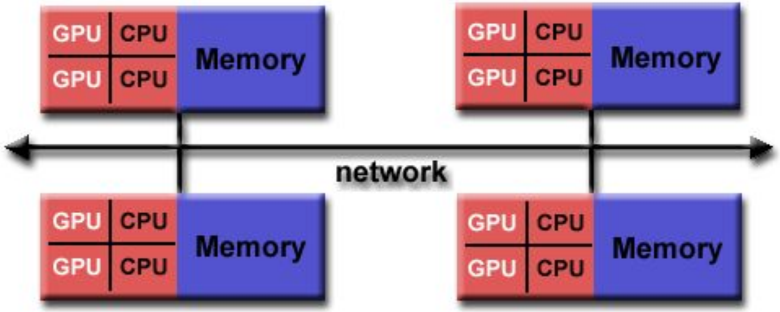
👍

All advantages of distributed and shared memory



👎

All disadvantages of distributed and shared memory



Speed Up

$$Speedup(N) = \frac{Time_{serial}}{Time_{parallel}(N)}$$

Linear SpeedUp: Ideally, doubling the number of CPUS the execution time is halved (Speedup = N)

In reality, this happens hardly ever, because some software cannot be completely parallelized

Amdalh's law

This law gives the **theoretical speedup** a parallel application can achieve

Be:

- S the fraction of the code that cannot be parallelized (serial execution)
- P the fraction of the code that can be parallelized
- S + P = 1

$$Speedup(N) = \frac{Time_{serial}}{Time_{parallel}(N)} = \frac{(S+P)T_{serial}}{S \cdot T_{serial} + \frac{P \cdot T_{serial}}{N}} = \frac{S+P}{S+\frac{P}{N}} = \frac{1}{S+\frac{P}{N}}$$

Amdalh’s law

$$\lim_{N \to \infty} \frac{1}{S + \frac{P}{N}} = \frac{1}{S}$$

$$\lim_{S \to 0} \frac{1}{S + \frac{P}{N}} = \frac{1}{S + \frac{1 - S}{N}} = N$$

Observations:

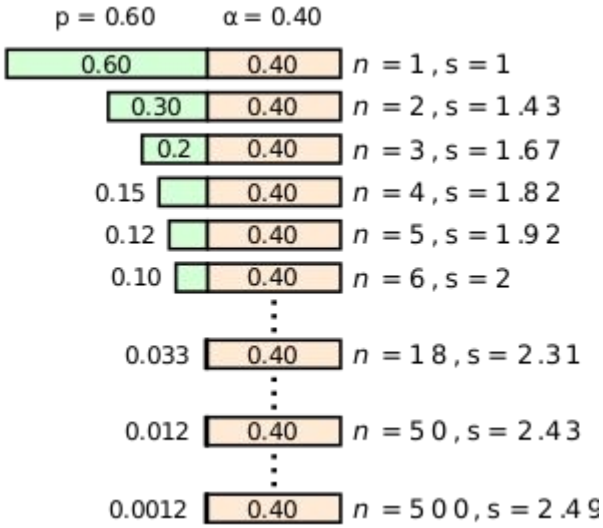
- From the first limit: the fraction of serial code **is a bound** of the scalability
- From the second limit: if there wasn’t any serial code, the speed up is equal to N (**linear speed**)

Amdalh's law: an example

If Serial Code is 10% (S = 0.10 and P = 0.90), the **highest speedup** we can get is 10, regardless the number of CPU used

$$\lim_{N \to \infty} \frac{1}{S + \frac{P}{N}} = \lim_{N \to \infty} \frac{1}{0.1 + \frac{0.9}{N}} = \frac{1}{0.1} = 10$$

Amdalh’s law: a graphical explanation



p = fraction of the code which can be executed in parallel mode

ɑ = fraction of the code which can be executed in serial mode

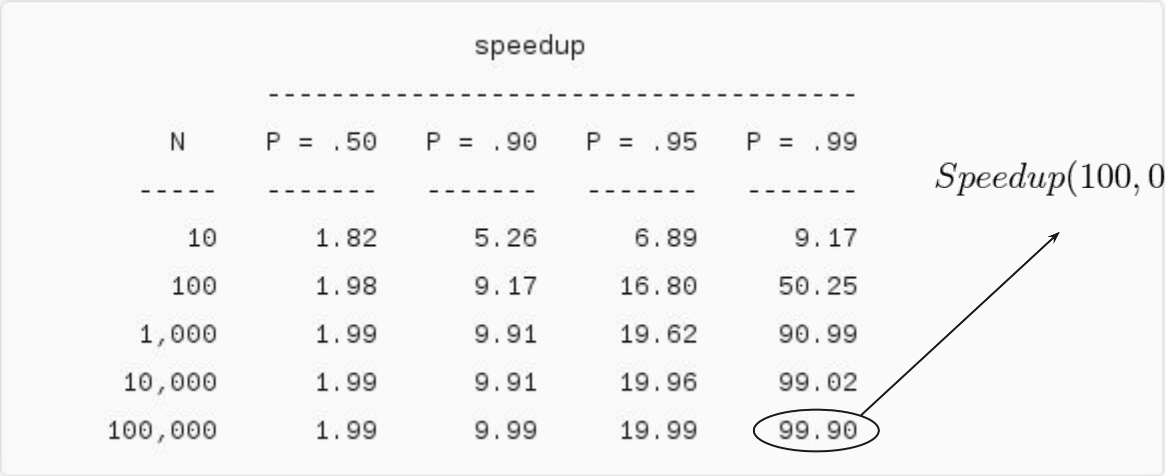
n = core number

s = SpeedUp

$$Speedup(2) = \frac{1}{S + \frac{P}{N}} = \frac{1}{0.40 + \frac{0.60}{2}} = 1.43$$

Limit of the Parallel Programming

Speedup is strongly affected by the fraction of serial code

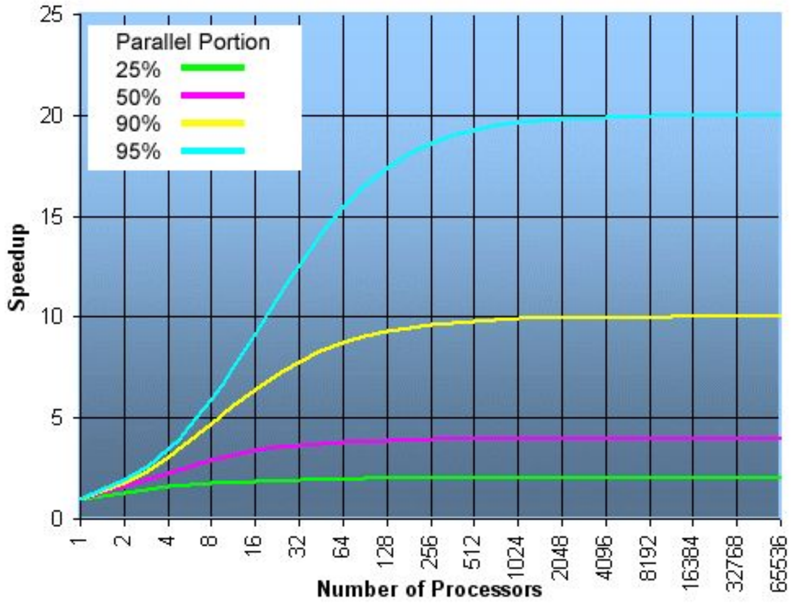


```
                         speedup
              ------------------------------------
      N       P = .50    P = .90    P = .95    P = .99
    -----     -------    -------    -------    -------
       10       1.82       5.26       6.89       9.17
      100       1.98       9.17      16.80      50.25
    1,000       1.99       9.91      19.62      90.99
   10,000       1.99       9.91      19.96      99.02
  100,000       1.99       9.99      19.99      99.90
```

$$Speedup(100,000) = \frac{1}{S + \frac{P}{N}} = \frac{1}{0.01 + \frac{0.99}{100{,}000}} = 99.90$$

Limit of the Parallel Programming

Speedup is strongly affected by the fraction of serial code



Superlinear speed

Some application might achieve performance even better than the linear speed, that is S > N

This might happen for several reasons, for example due to the CPU cache

Scalability

$$Scalability(N) = \frac{T_{parallel}(1)}{T_{parallel}(N)}$$

*Quite similar to the SpeedUp but instead of considering the execution time of the serial implementation, it takes the execution time of the parallel implementation with a **parallel degree equal to 1***

Scalability

- Strong Scaling (Amdahl)
    - The total problem size stays fixed as more processors are added.
    - Goal is to run the same problem size faster
    - Perfect scaling means problem is solved in 1/P time (compared to serial)
- Weak Scaling (Gustafson)
    - The problem size per processor stays fixed as more processors are added.
    - The total problem size is proportional to the number of processors used
    - Goal is to run larger problem in same amount of time
    - Perfect scaling means problem P runs in same time as single processor run

Limit of scalability

- Load balancing
- Synchronization
- Communication
- Overhead

Parallel Programming Models

There are several parallel programming models in common use

- Shared Memory (without threads, using semaphores or locks to prevent race conditions and deadlock)
- Threads (pThread, OpenMP)
- **Distributed Memory / Message Passing (MPI)**
- Data Parallel
- Hybrid (MPI + OpenMP/pThread)
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

Non-parallelizable applications

Not all applications can be parallelized.

Let’s consider the Fibonacci sequence: {1, 1, 2, 3, 5, 8, 13, 21, ...}

$$f(n) = \begin{cases} 1 & if \;\; n = 0 \;\; or \; n = 1 \\ f(n-1) + f(n-2) & if \;\; n \geq 2 \end{cases}$$

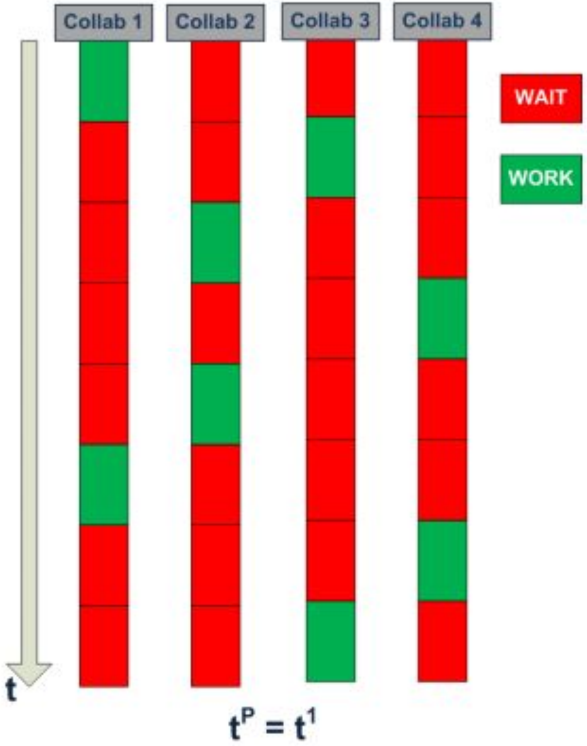
This problem is not parallelizable because each f(n) can be computed only when f(n-1) and f(n-2) have been computed

Non-parallelizable applications

Using 4 different CPUs is useless because each CPU needs to wait until the required results are available.

So, even using N cores, the time to complete the parallel execution is the same as the serial execution

Data dependency problem

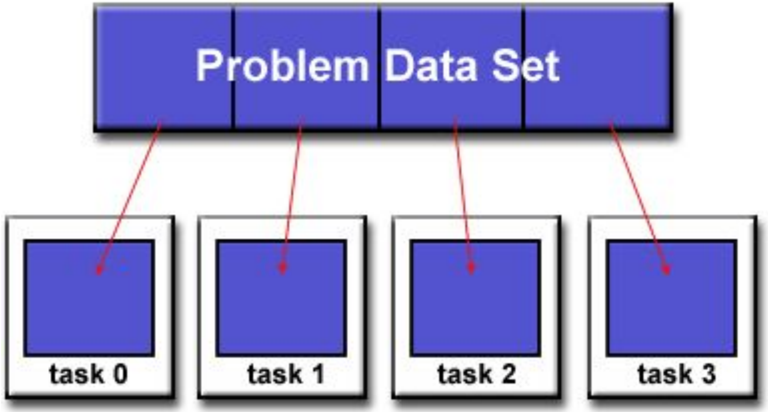


Parallelizable applications: decomposition

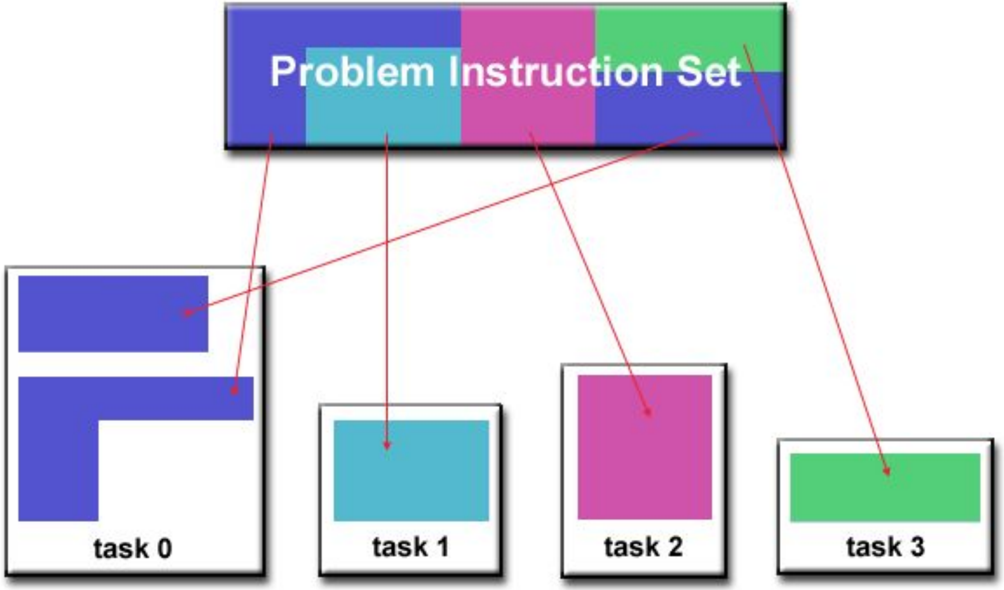
One of the first steps in designing a parallel program is to break the problem into discrete **chunks** of work that can be distributed to multiple tasks:

- Domain Decomposition
- Functional Decomposition

Domain Decomposition



Functional Decomposition



A example of Domain Decomposition

Suppose we want to compute the **Geometric Series**

$$G_N = \sum_{i=1}^{N} x^i$$

Suppose that we want to parallelize this computation using P = 4 different cores and N is a multiple of P. Then, the original formula can be rewritten into

$$G_N = \sum_{i=1}^{N} x^i = \sum_{i=1}^{P} \left(\sum_{j=1}^{\frac{N}{P}} x^{\frac{N}{P}(i-1)+j} \right) = \sum_{i=1}^{P} S_i$$

$$S_i = \sum_{j=1}^{\frac{N}{P}} x^{\frac{N}{P}(i-1)+j}$$

A example of Domain Decomposition

$$G_{16} = 2^1+2^2+2^3+2^4+2^5+2^6+2^7+2^8+2^9+2^{10}+2^{11}+2^{12}+2^{13}+2^{14}+2^{15}+2^{16}$$

Processor 1	$2^1+2^2+2^3+2^4$
Processor 2	$2^5+2^6+2^7+2^8$
Processor 3	$2^9+2^{10}+2^{11}+2^{12}$
Processor 4	$2^{13}+2^{14}+2^{15}+2^{16}$

A example of Domain Decomposition

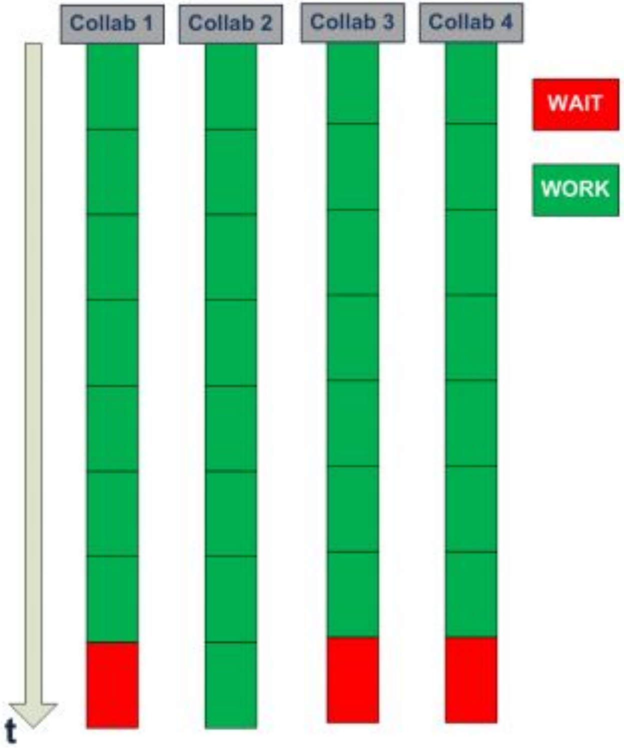
$$G_N = \sum_{i=1}^{N} x^i = \sum_{i=1}^{P} \left(\sum_{j=1}^{\frac{N}{P}} x^{\frac{N}{P}(i-1)+j} \right) = \sum_{i=1}^{P} S_i$$

Each S_i can be assigned to each core.

In the last step, the outermost sum is computed.

Serial computation time: T

Parallel computation time: T/P



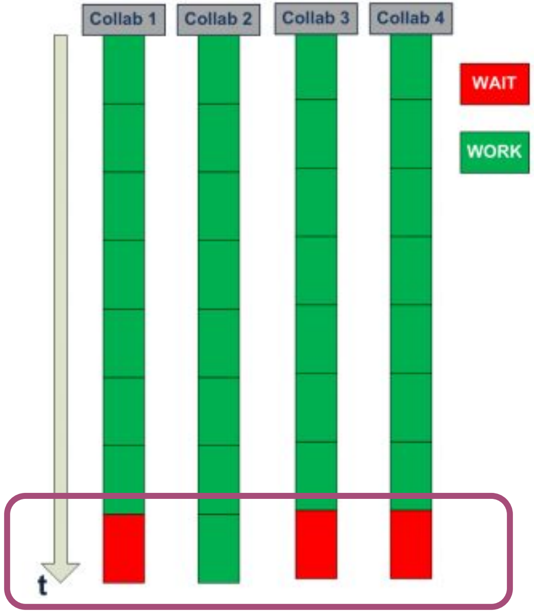
Parallel Computing Tradeoffs

If domain or problem can be decomposed, using P concurrent processes we can:

1. Reduce the **execution time**
2. Reduce the **amount of memory** required by each process

On the opposite

1. **Communication** between processes is needed
   a. In the Geometric Series, each process communicates the result to process #2
2. **Synchronization** between processes is required
   a. process A cannot send a message to process B if B is not waiting for the message sent by A



Synchronization and Communication

MPI: an introduction

What is MPI: Message Passing Interface

- It is **not:**
  - a compiler
  - a library
  - a framework
  - a programming language
- It's a specification, a standard, an interface, **not an implementation**
  - **MPI Forum:** https://www.mpi-forum.org/
  - Version 4.0 released on June, the 9th 2021
  - The complete specification can be found at:
    - https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
    - 1139 pages

What is MPI

- There are many implementation
  - **OpenMPI (https://www.open-mpi.org/) - Open Source**
  - IntelMPI (https://www.intel.com/)
  - MVAPICH (http://mvapich.cse.ohio-state.edu/)
  - Rookie (https://www.rookiehpc.com/)
  - ...
- Each implementation has its own version
  - OpenMPI 4.1.1 implements the specification 4.0
- There are many implementation for different programming language
  - **C/C++**
  - Fortran
  - Java

How to Install OpenMPI

on Centos8

- yum install gcc
- yum install gcc-c++
- yum install make
- cd ~/Download
- wget https://download.open-mpi.org/release/open-mpi/v4.1/openmpi-4.1.1.tar.gz
- tar -xzvf openmpi-4.1.1.tar.gz
- cd openmpi-4.1.1/
- ./configure --prefix=/usr/local/openmpi-4.1.1
- make -j 4 all
- make install
- vi ~/.bashrc
    - MPI_HOME=/usr/local/openmpi-4.1.1
    - PATH=$MPI_HOME/bin:$HOME

MPI call format in C/C++

All functions in MPI have a similar format:

```
err = MPI_Xxxxxx(parameters, …)
```

- prefix MPI_
- only first letter uppercase
- all functions return an integer error code
- parameters can be either “in” or “out”

Where to get help

- man pages
  - i.e. `man MPI_Init`
- https://www.open-mpi.org/doc/current/
- https://www.rookiehpc.com/mpi/docs/

Communication Environment

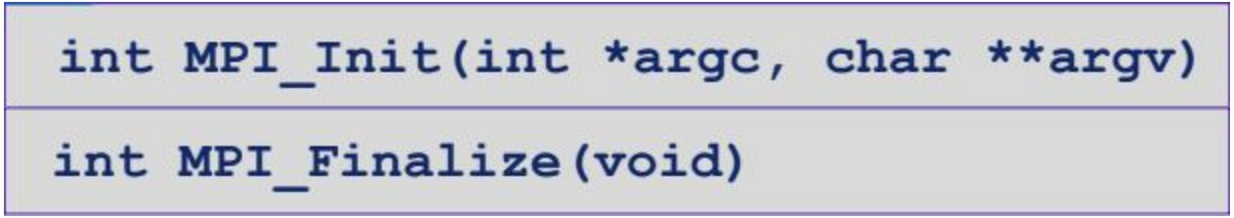
`MPI_Init`

- it initializes the communication environment
- all processes need to use it before any other
- it is called only one time per process

`MPI_Finalize`

- it finalizes the communication environment
- it is not possible to use any other MPI function after that

MPI_Init and MPI_Finalizes



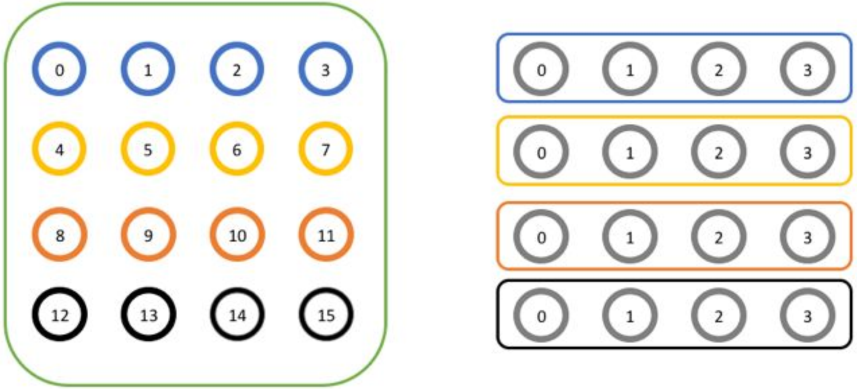
```
int MPI_Init(int *argc, char **argv)
int MPI_Finalize(void)
```

parameters `argc` and `argv` are those that are taken by main method

```
void main (int argc, char *argv[]) {
…
}
```

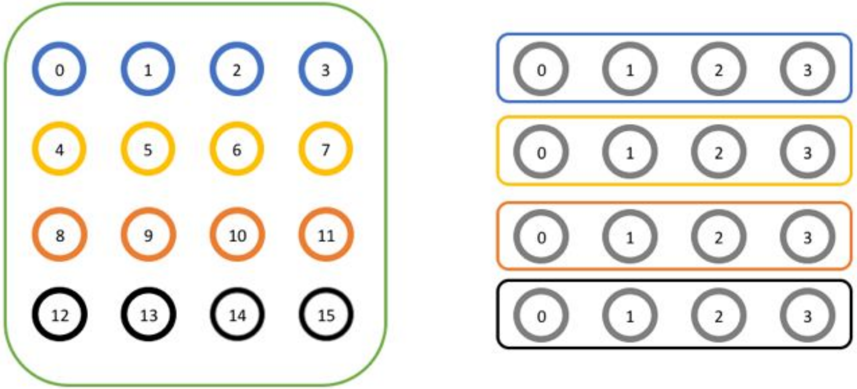
Communicator

- A communicator is a **collection** of processes sharing attributes
- Each process is **identified** inside its communicator (**rank**)
- Two processes can **communicate** only if they belong to the same communicator
- **MPI_COMM_WORLD** is the default communicator

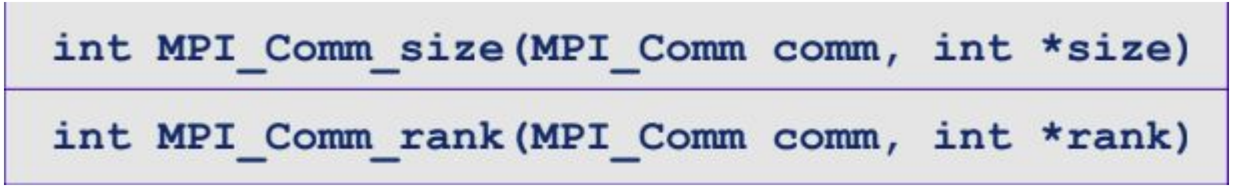


Communicator size

- It is the number of the processes belonging to the same communicator
- **`MPI_Comm_size`**: function for getting the size of a communicator
- **`MPI_Comm_rank`**: function for getting the ID of a process
- convention: process 0 = master



MPI_Comm_size and MPI_Comm_rank



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

- **comm**: the communication we want to know
- **size**: pointer to an integer variable (it will contain the size of the Comm)
- **rank**: pointer to an integer variable (it will contain the rank of the process)

The first program: example01.c

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {
    int myrank, size;
    /* 1. Initialize MPI */
    MPI_Init(&argc, &argv);
    /* 2. Get my rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    /* 3. Get the total number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* 4. Print myrank and size */
    printf("Process %d of %d \n", myrank, size);
    /* 5. Terminate MPI */
    MPI_Finalize();
}
```

There is no reference to the number of processes to be executed

Compiling and running the program

```
> mpicc example01.c -o example01

> mpirun -n 4 example01
```

all 4 processes are executed on localhost

```
Process 0 of 4
Process 2 of 4
Process 1 of 4
Process 3 of 4
```

Your turn

- run the program using a different number of processes
- why indexes in the output do appear out of order?
- different program executions will give the output in the same order?

Keep in mind

The correct execution of the program must **not depend on the process number**

This means that your code has to work properly **regardless the number of processes** used

The message structure

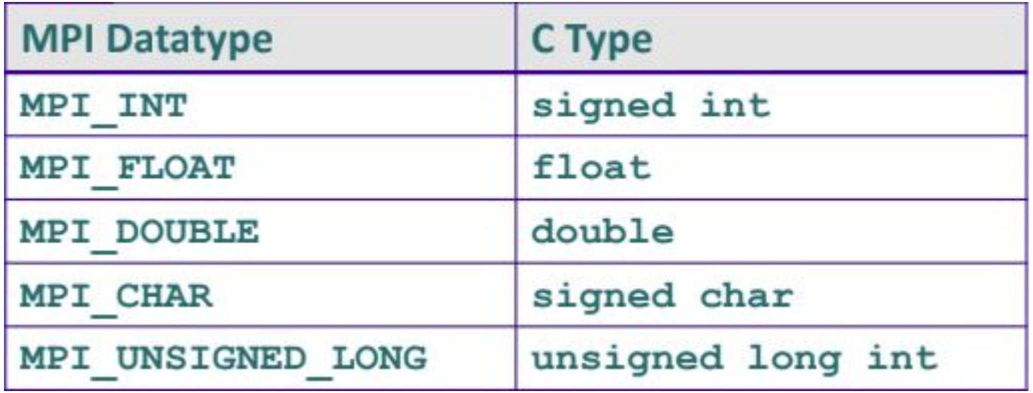
Processes communicate using **messages**

A message is made by:

- Envelope
  - **sender/receiver**: the ID of the sender/receiver process
  - **communicator**: the ID of the group where processes belong to
  - **tag**: the ID of the message
- Body
  - **buffer**: the message to send / receive
  - **datatype**: the type of the message *(see next)*
  - **count**: the number of occurrences of the datatype

All sender and receiving functions manage all these parameters

MPI_Datatype



MPI Datatype	C Type
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	signed char
MPI_UNSIGNED_LONG	unsigned long int

Complex datatype (for example structured data) can be defined as well.

Different types of communications

Processes can communicate using:

1. Point to Point functions

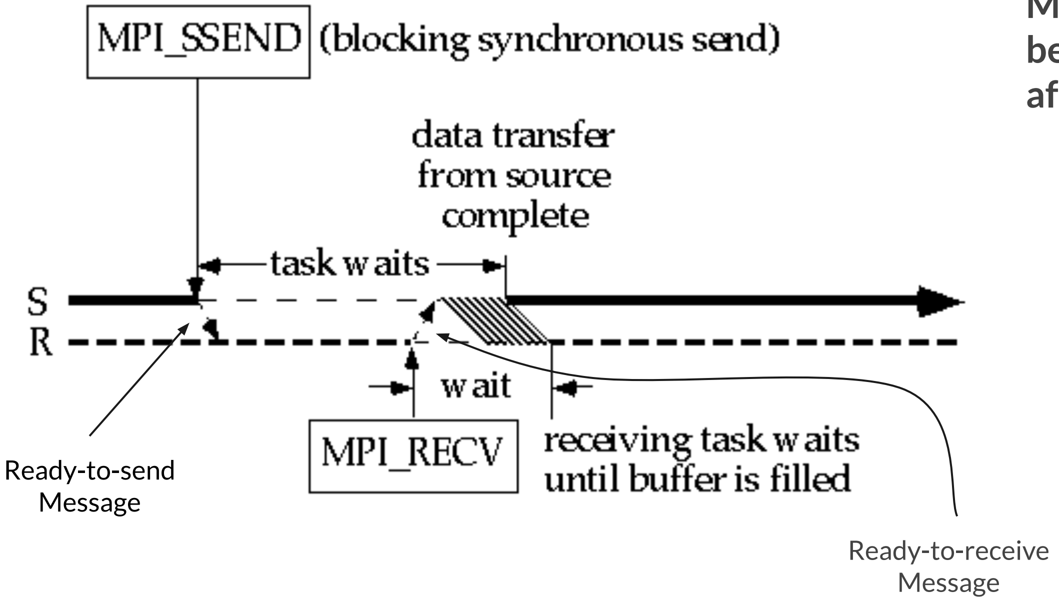
2. Collective functions

MPI: point-to-point functions

Point-to-point functions

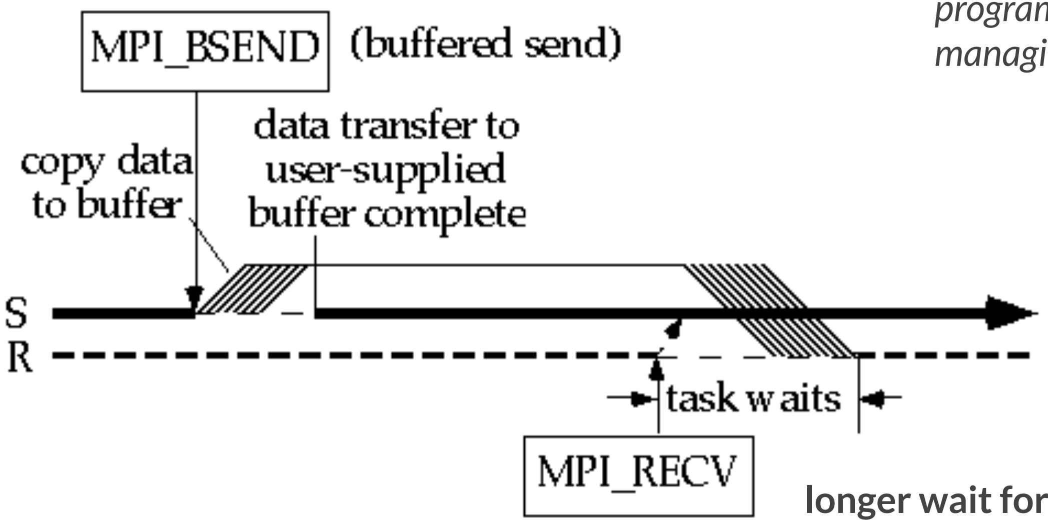
Communication mode	Blocking Routines	Non-Blocking Routines
Synchronous	MPI_Ssend / MPI_Recv	MPI_Issend / MPI_Recv
Buffered	MPI_Bsend / MPI_Recv	MPI_Ibsend / MPI_Recv
Ready	MPI_Rsend / MPI_Recv	MPI_Irsend / MPI_Recv
Standard	MPI_Send / MPI_Recv	MPI_Isend / MPI_Irecv

Blocking Synchronous Send (MPI_Ssend)


MPI_SSEND
(blocking synchronous send)
data transfer from source complete
task waits
S
R
wait
MPI_RECV
receiving task waits until buffer is filled
Ready-to-send Message
Ready-to-receive Message

MPI_Recv can either be invoked before or after MPI_Ssend

Blocking Buffered Send (MPI_Bsend)



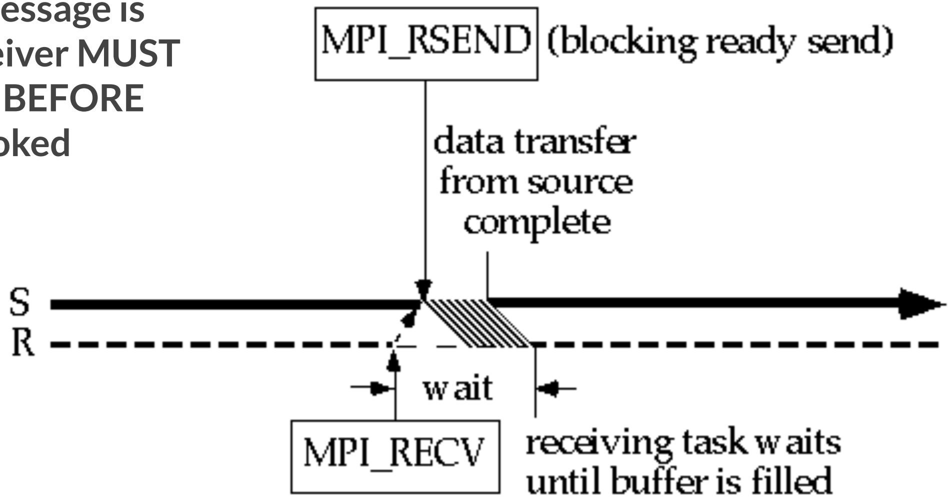
BE CAREFUL:
programmer is responsible for managing user-supplied buffer

MPI_Recv can either be invoked before or after MPI_Ssend

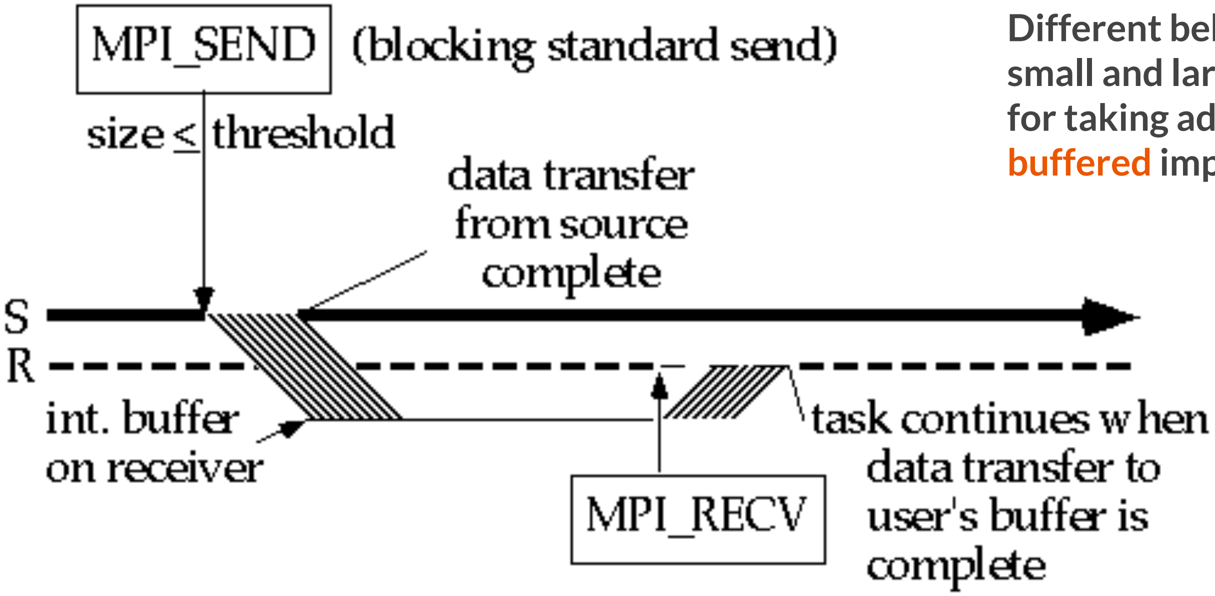
longer wait for the receiver because of the system overhead (due to the copy of the message from the buffer)

Blocking Ready Send (MPI_Rsend)

Ready-To-Send message is dropped, but receiver MUST invoke MPI_Recv BEFORE MPI_Rsend is invoked

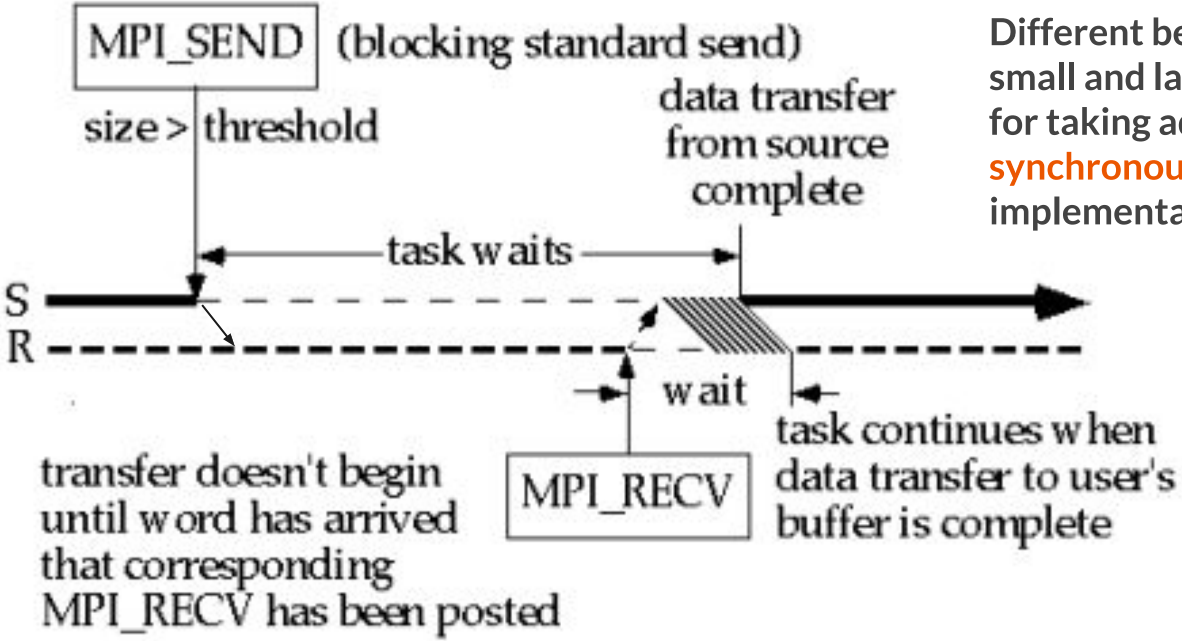

MPI_RSEND
(blocking ready send)
data transfer from source complete
S
R
wait
MPI_RECV
receiving task waits until buffer is filled

Standard Send (MPI_Send) [eager protocol]


MPI_SEND
(blocking standard send)
size ≤ threshold
data transfer from source complete
S
R
int. buffer on receiver
MPI_RECV
task continues w hen data transfer to user's buffer is complete

Different behaviour for small and large messages for taking advantages of buffered implementation

Standard Send (MPI_Send) [rendezvous protocol]


MPI_SEND
(blocking standard send)
size > threshold
data transfer from source complete
task waits
S
R
wait
transfer doesn't begin until word has arrived that corresponding MPI_RECV has been posted
MPI_RECV
task continues when data transfer to user's buffer is complete

Different behaviour for small and large messages for taking advantages of synchronous implementation

How to know the threshold?

```
ompi_info --all | grep btl_tcp_eager_limit

ompi_info --all | grep btl_sm_eager_limit
```

tcp: communications happen through network (internode communications)

sm: communications happen through shared memory (internode communications)

Choosing the right communication mode

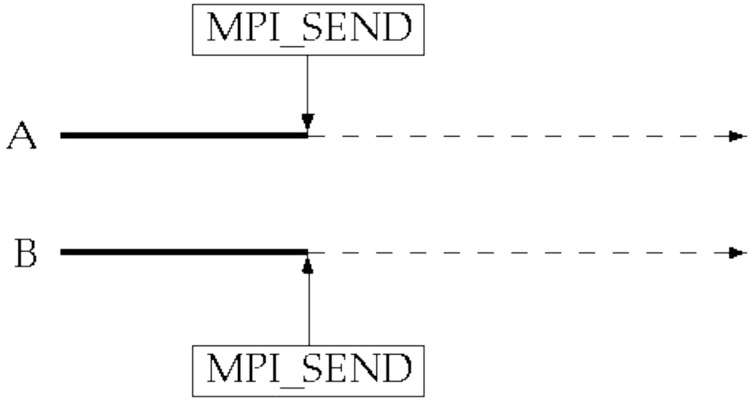
	Advantages	**Disadvantages**
Synchronous	Send/Recv order is not critical; No extra buffer space;	Can incur synchronous overhead; Handshake required (ready-messages);
Ready	Lowest total overhead No extra buffer space; No handshake	Recv must precede Send;
Buffered	Decouples Send from Recv *(message is buffered on sender side)* No sync overhead on Sender; Programmer **must** control buffer; Send/Recv order is not critical;	System overhead due to the copy of the buffer
Standard	Good for many cases	Protocol is determined by MPI implementation

Deadlock

Using **blocking functions** might incur in deadlock

This means that both functions stop program execution until the message is not received/sent by the counterpart.

For example: if P_1 invokes MPI_Send for sending a message to P_2, the execution of P_1 is blocked until P_2 does not invoke the corresponding MPI_Recv



Non-Blocking functions

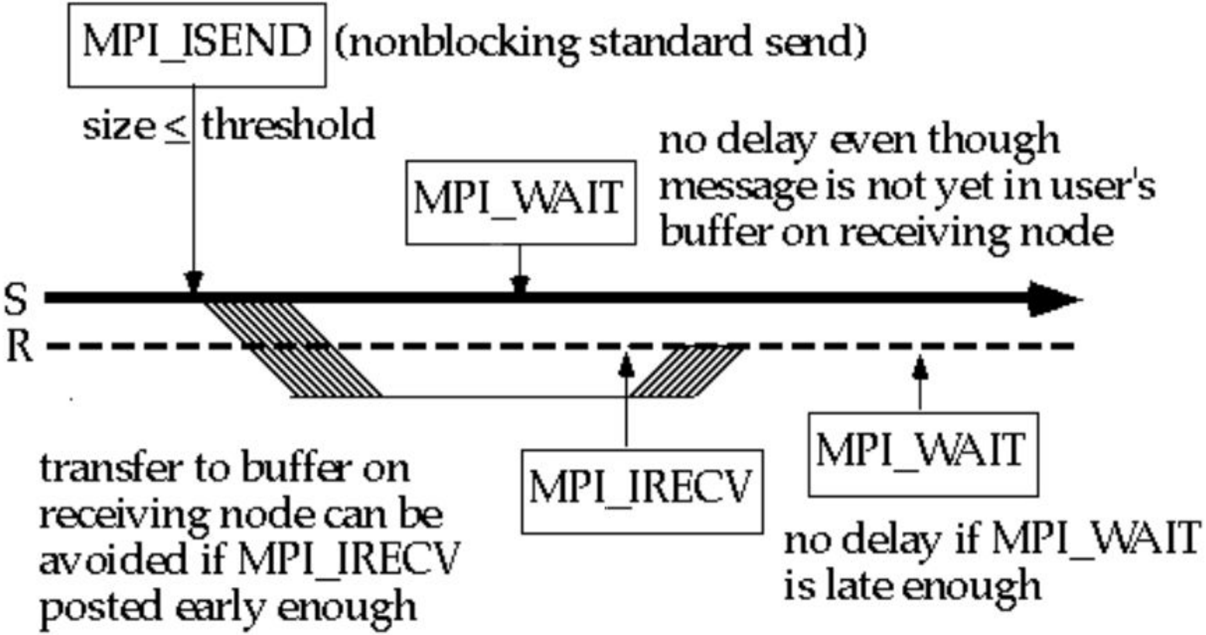
For each blocking function (synchronized, buffered, ready and standard) there is a corresponding non-blocking function with a similar behaviour.

The only difference is that sending a receiving functions never block the task execution.

But we need to use **MPI_Wait** function.

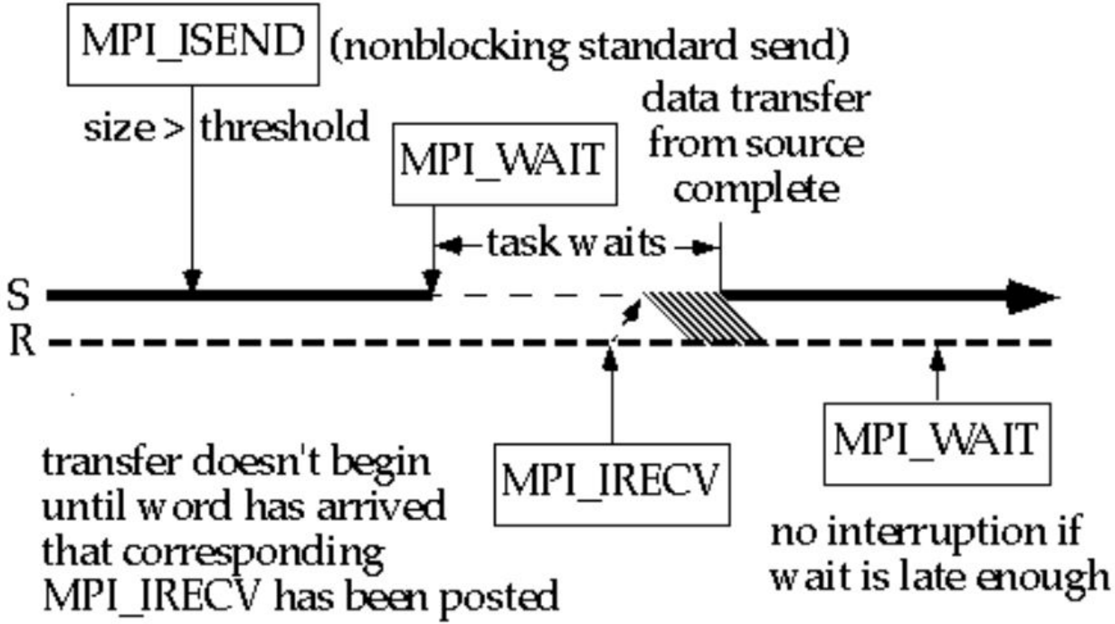
Let’s describe only the MPI_Isend and MPI_Irecv functions (non-blocking standard functions)

Non-Blocking Standard Sender (MPI_Isend)



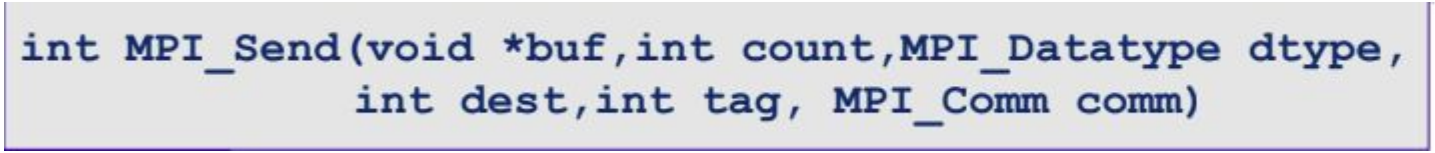
MPI_ISEND (nonblocking standard send)
size ≤ threshold
MPI_WAIT
no delay even though message is not yet in user's buffer on receiving node
S
R
transfer to buffer on receiving node can be avoided if MPI_IRECV posted early enough
MPI_IRECV
MPI_WAIT
no delay if MPI_WAIT is late enough

Non-Blocking Standard Sender (MPI_Isend)


MPI_ISEND
(nonblocking standard send)
size > threshold
MPI_WAIT
data transfer from source complete
task waits
S
R
transfer doesn't begin until word has arrived that corresponding MPI_IRECV has been posted
MPI_IRECV
MPI_WAIT
no interruption if wait is late enough

MPI: point-to-point functions Examples

MPI_Send



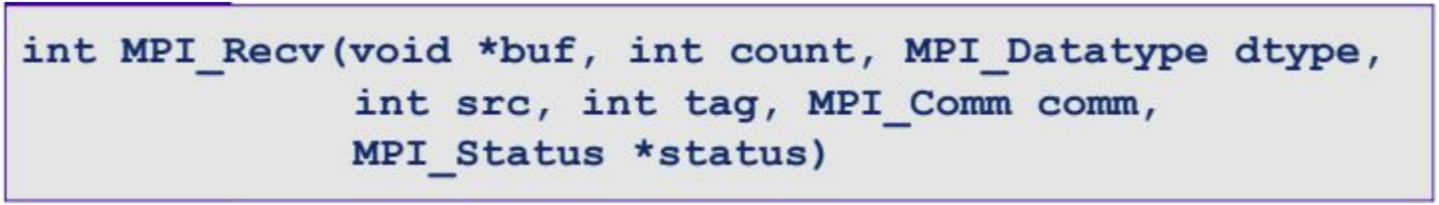
```
int MPI_Send(void *buf,int count,MPI_Datatype dtype,
             int dest,int tag, MPI_Comm comm)
```

- **buf**: Initial address of send buffer
- **count**: Number of elements send
- **dtype**: Datatype of each send buffer element
- **dest**: Rank of destination
- **tag**: Message tag
- **comm**: Communicator

message

envelope

MPI_Recv



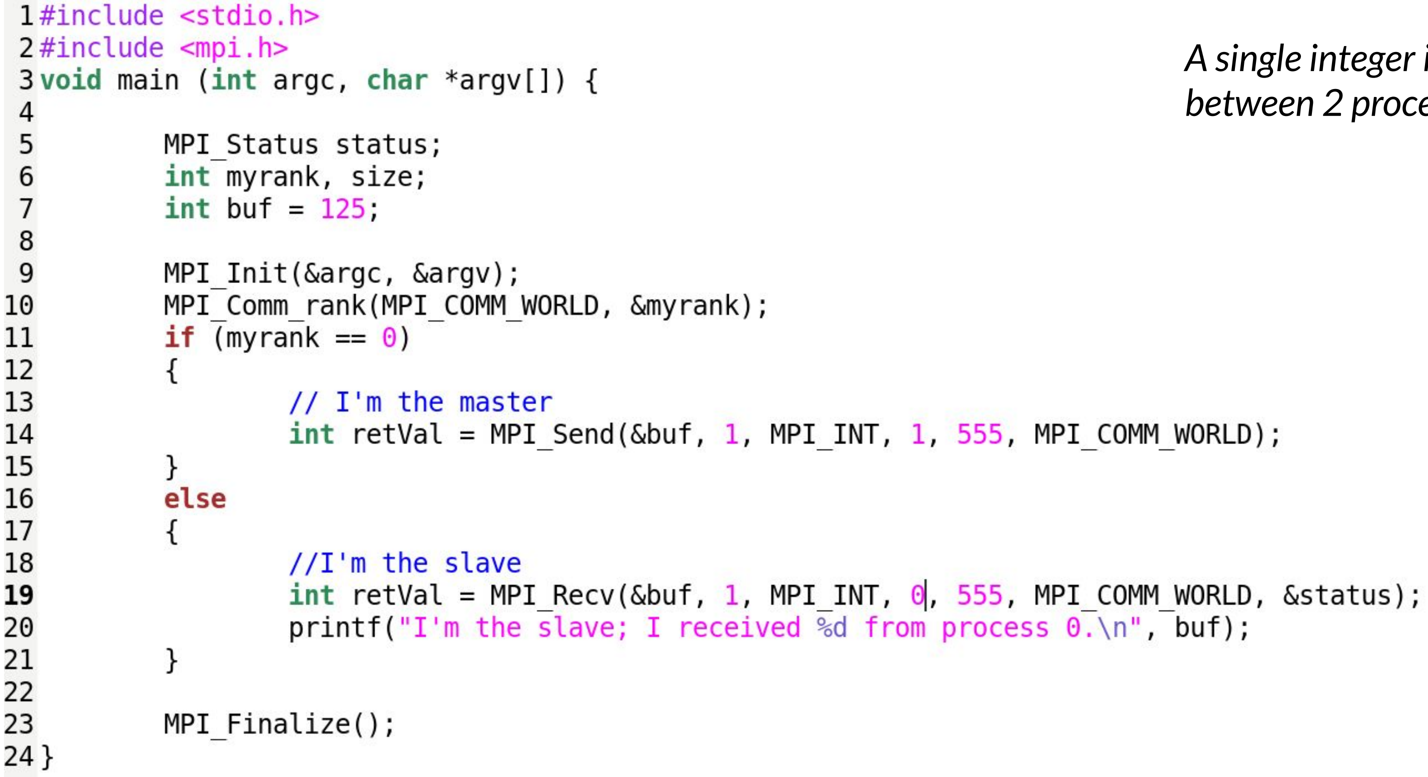
```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
                 int src, int tag, MPI_Comm comm,
                 MPI_Status *status)
```

- **buf**: Initial address of receive buffer
- **count**: Maximum number of elements to receive
- **dtype**: Datatype of each receive buffer entry
- **src**: Rank of source
- **tag**: Message tag
- **comm**: communicator
- **status**: Status object (information about the received message)

message

envelope

Sending and receiving data: example02.c



```c
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {

    MPI_Status status;
    int myrank, size;
    int buf = 125;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0)
    {
        // I'm the master
        int retVal = MPI_Send(&buf, 1, MPI_INT, 1, 555, MPI_COMM_WORLD);
    }
    else
    {
        //I'm the slave
        int retVal = MPI_Recv(&buf, 1, MPI_INT, 0, 555, MPI_COMM_WORLD, &status);
        printf("I'm the slave; I received %d from process 0.\n", buf);
    }

    MPI_Finalize();
}
```

A single integer is sent between 2 processes

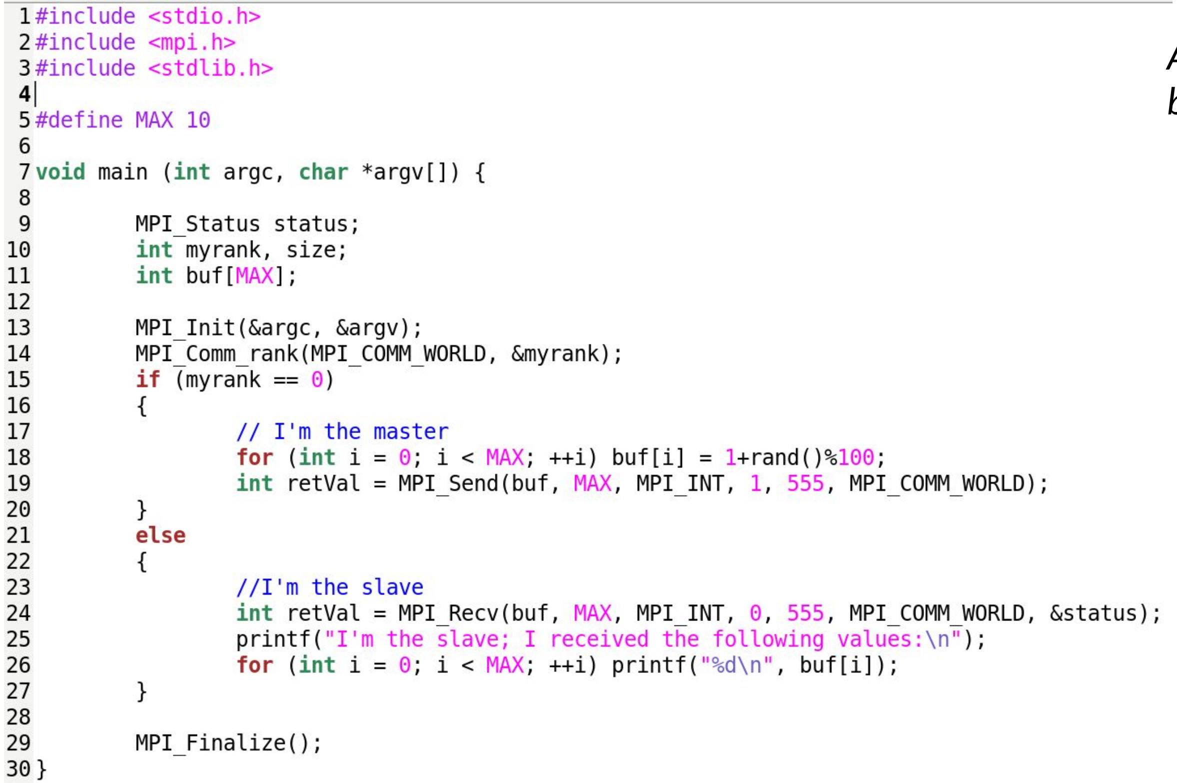
Some considerations

- The receiver can get the message if the **envelope** specified by the receiver is exactly the same as the envelope specified by sender
    - REMIND: envelop = source /destination + communicator + tag
- MPI_Send and MPI_Recv are **blocking**
    - Sender waits until receiver gets the message
    - Receiver waits until sender sends the message

Your turn

a. Can the application work even using a different process number?
b. What happens if I run the code with `-n 4`? Why?
c. Can we avoid blocking application execution?
    a. solution: `else if (myrank == 1)`
d. What happens if I run the code replacing the tag on the receiver side to 554? Why?
e. Try to send a char instead of a integer
f. Try to send an array of 10 integers (see example03)

Sending and receiving data: example03.c



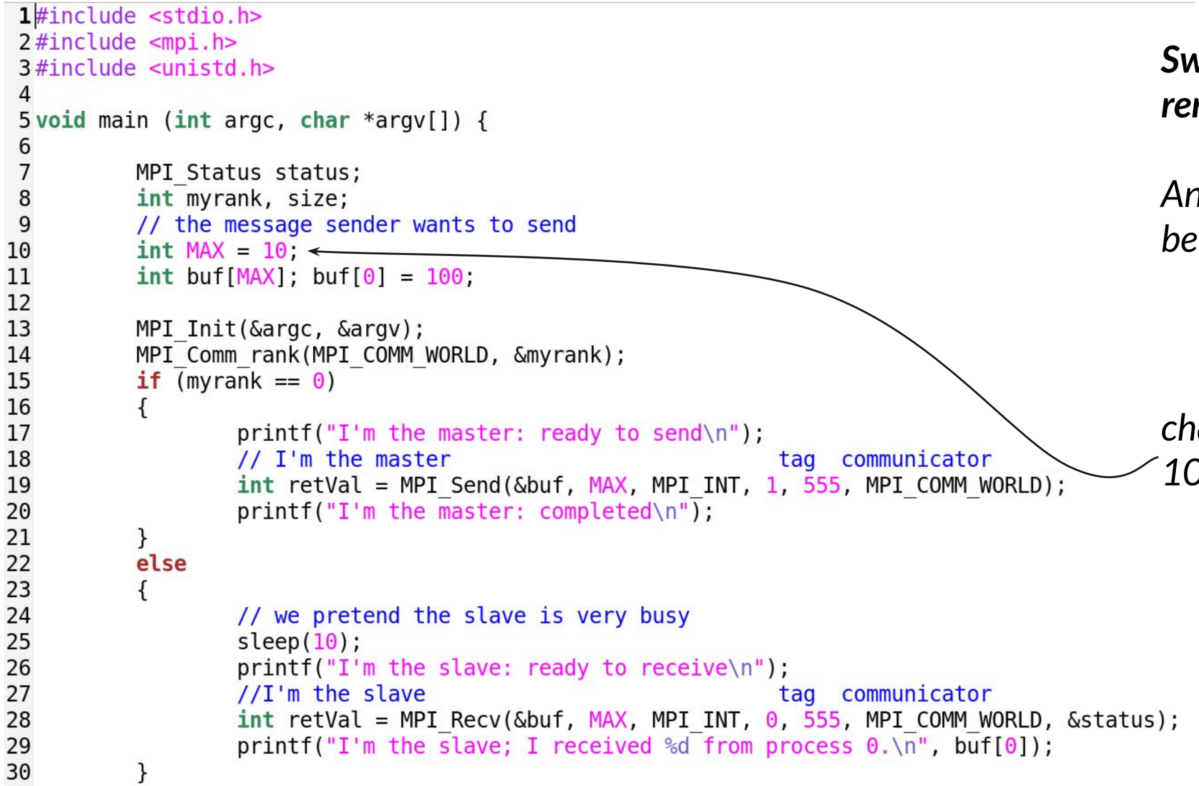
```c
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>

#define MAX 10

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

        MPI_Status status;
        int myrank, size;
        int buf[MAX];

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        if (myrank == 0)
        {
                // I'm the master
                for (int i = 0; i < MAX; ++i) buf[i] = 1+rand()%100;
                int retVal = MPI_Send(buf, MAX, MPI_INT, 1, 555, MPI_COMM_WORLD);
        }
        else
        {
                //I'm the slave
                int retVal = MPI_Recv(buf, MAX, MPI_INT, 0, 555, MPI_COMM_WORLD, &status);
                printf("I'm the slave; I received the following values:\n");
                for (int i = 0; i < MAX; ++i) printf("%d\n", buf[i]);
        }

        MPI_Finalize();
}
```

A vector of integers is sent between 2 processes

Your turn

a. Instead of sending a vector of 10 integers in one shot, let’s send the vector in ten steps (one integer per send). Here again, only two processes involved in the communication

Switching protocols: example03.1.c



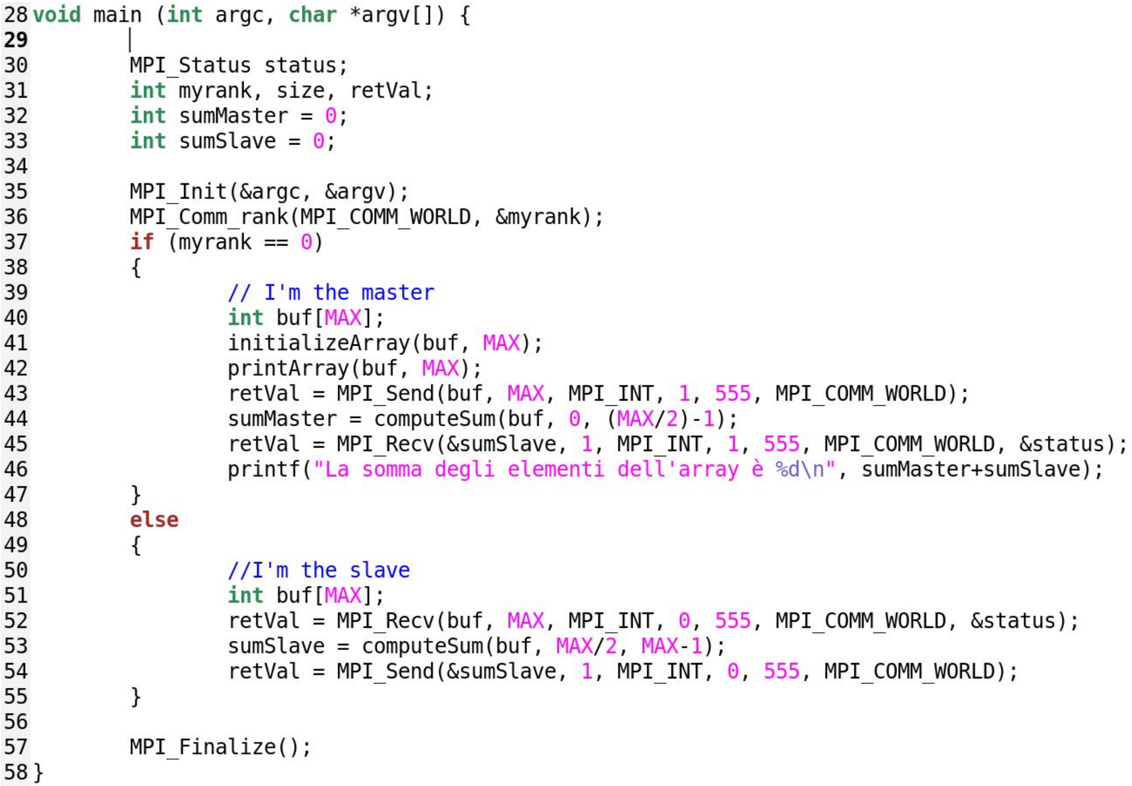
```c
#include <stdio.h>
#include <mpi.h>
#include <unistd.h>

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

        MPI_Status status;
        int myrank, size;
        // the message sender wants to send
        int MAX = 10;
        int buf[MAX]; buf[0] = 100;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        if (myrank == 0)
        {
                printf("I'm the master: ready to send\n");
                // I'm the master                          tag  communicator
                int retVal = MPI_Send(&buf, MAX, MPI_INT, 1, 555, MPI_COMM_WORLD);
                printf("I'm the master: completed\n");
        }
        else
        {
                // we pretend the slave is very busy
                sleep(10);
                printf("I'm the slave: ready to receive\n");
                //I'm the slave                            tag  communicator
                int retVal = MPI_Recv(&buf, MAX, MPI_INT, 0, 555, MPI_COMM_WORLD, &status);
                printf("I'm the slave; I received %d from process 0.\n", buf[0]);
        }
```

Switch from eager to rendezvous protocol

An array of integers is sent between 2 processes

change this variable from 10 to 1024

Summing up integers: example04.c



```c
void main (int argc, char *argv[]) {

        MPI_Status status;
        int myrank, size, retVal;
        int sumMaster = 0;
        int sumSlave = 0;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        if (myrank == 0)
        {
                // I'm the master
                int buf[MAX];
                initializeArray(buf, MAX);
                printArray(buf, MAX);
                retVal = MPI_Send(buf, MAX, MPI_INT, 1, 555, MPI_COMM_WORLD);
                sumMaster = computeSum(buf, 0, (MAX/2)-1);
                retVal = MPI_Recv(&sumSlave, 1, MPI_INT, 1, 555, MPI_COMM_WORLD, &status);
                printf("La somma degli elementi dell'array è %d\n", sumMaster+sumSlave);
        }
        else
        {
                //I'm the slave
                int buf[MAX];
                retVal = MPI_Recv(buf, MAX, MPI_INT, 0, 555, MPI_COMM_WORLD, &status);
                sumSlave = computeSum(buf, MAX/2, MAX-1);
                retVal = MPI_Send(&sumSlave, 1, MPI_INT, 0, 555, MPI_COMM_WORLD);
        }

        MPI_Finalize();
}
```

Summing integer elements between 2 processes

mpirun -n 2 example04.o

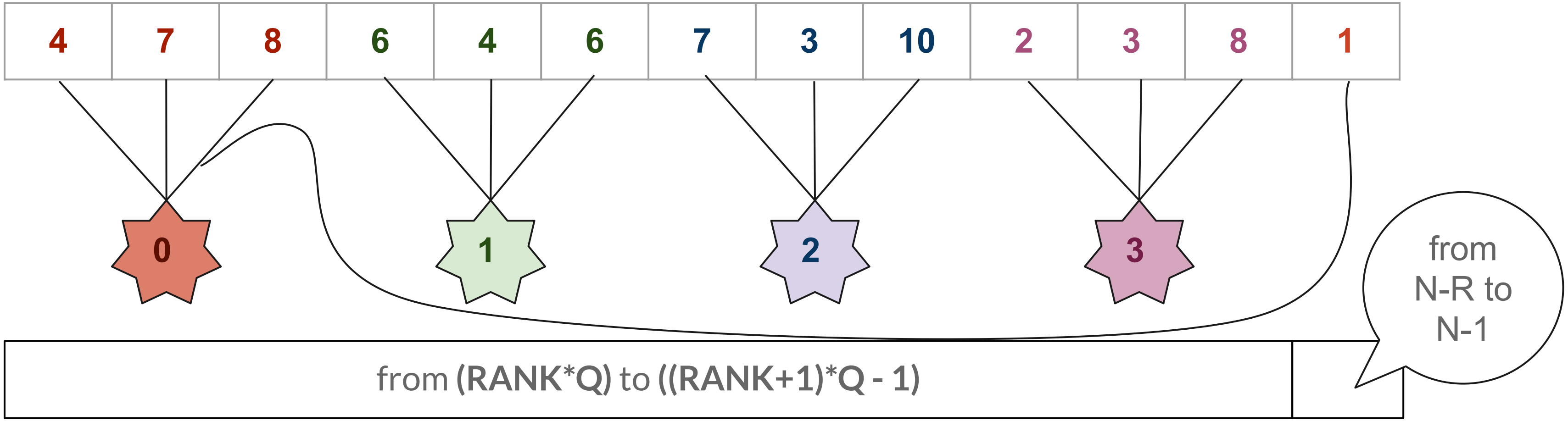
Good to know

- All the former examples have been working only using two processes
- For all of them, using more processes never work (the completion time is always the same)
- The following examples will use several processes
    - No assumption will be done on the number of processes (the parallel applications will work regardless the process number)

Summing up integers: example05.c

Suppose we have the following vector of 13 integers and we want to sum up all the elements using 4 different processes

N = 13
P = 4
Q = N/P = 3
R = N%P = 1

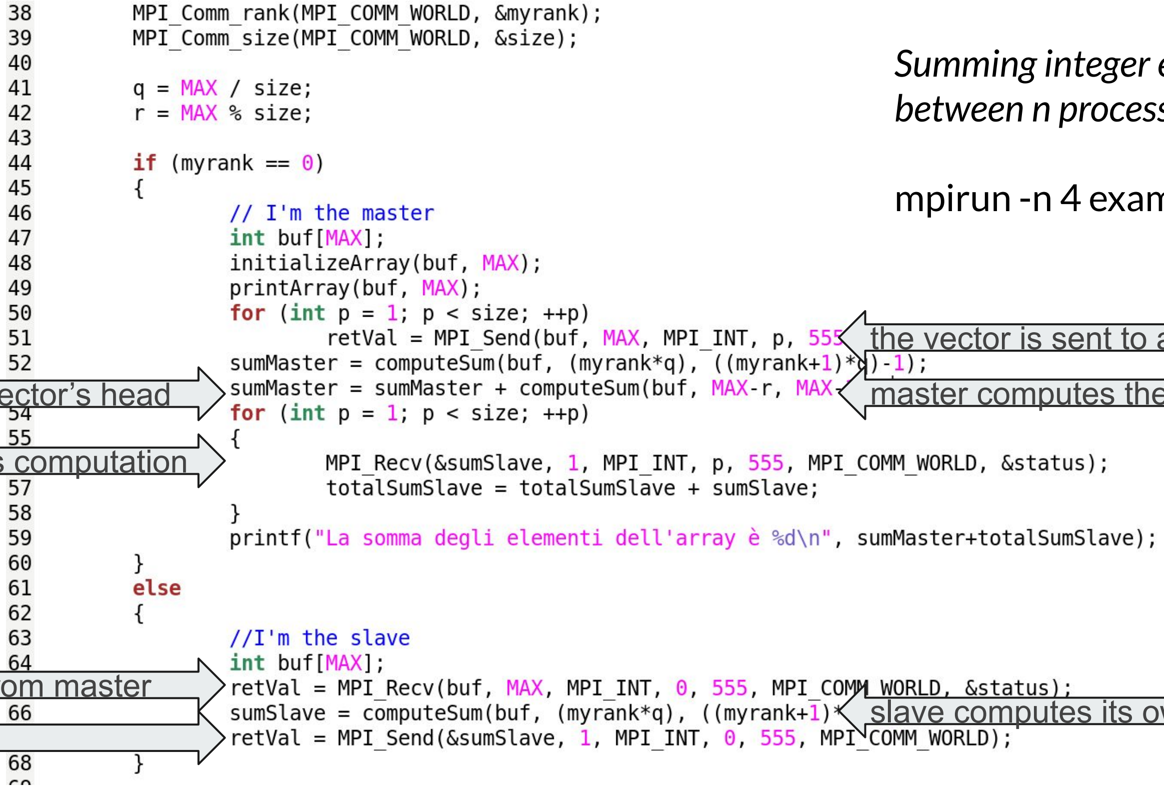


4	7	8	6	4	6	7	3	10	2	3	8	1

from **(RANK*Q)** to **((RANK+1)*Q - 1)**

from N-R to N-1

Summing up integers: example05.c



```
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        q = MAX / size;
        r = MAX % size;

        if (myrank == 0)
        {
                // I'm the master
                int buf[MAX];
                initializeArray(buf, MAX);
                printArray(buf, MAX);
                for (int p = 1; p < size; ++p)
                        retVal = MPI_Send(buf, MAX, MPI_INT, p, 555
                sumMaster = computeSum(buf, (myrank*q), ((myrank+1)*q)-1);
                sumMaster = sumMaster + computeSum(buf, MAX-r, MAX
                for (int p = 1; p < size; ++p)
                {
                        MPI_Recv(&sumSlave, 1, MPI_INT, p, 555, MPI_COMM_WORLD, &status);
                        totalSumSlave = totalSumSlave + sumSlave;
                }
                printf("La somma degli elementi dell'array è %d\n", sumMaster+totalSumSlave);
        }
        else
        {

                //I'm the slave
                int buf[MAX];
                retVal = MPI_Recv(buf, MAX, MPI_INT, 0, 555, MPI_COMM_WORLD, &status);
                sumSlave = computeSum(buf, (myrank*q), ((myrank+1)*
                retVal = MPI_Send(&sumSlave, 1, MPI_INT, 0, 555, MPI_COMM_WORLD);
        }
```

the vector is sent to all slaves

master computes the vector's head

master computes the vector's tail

master receives slave's computation

slave receives vector from master

slave computes its own data

sum is sent to master

Summing integer elements between n processes

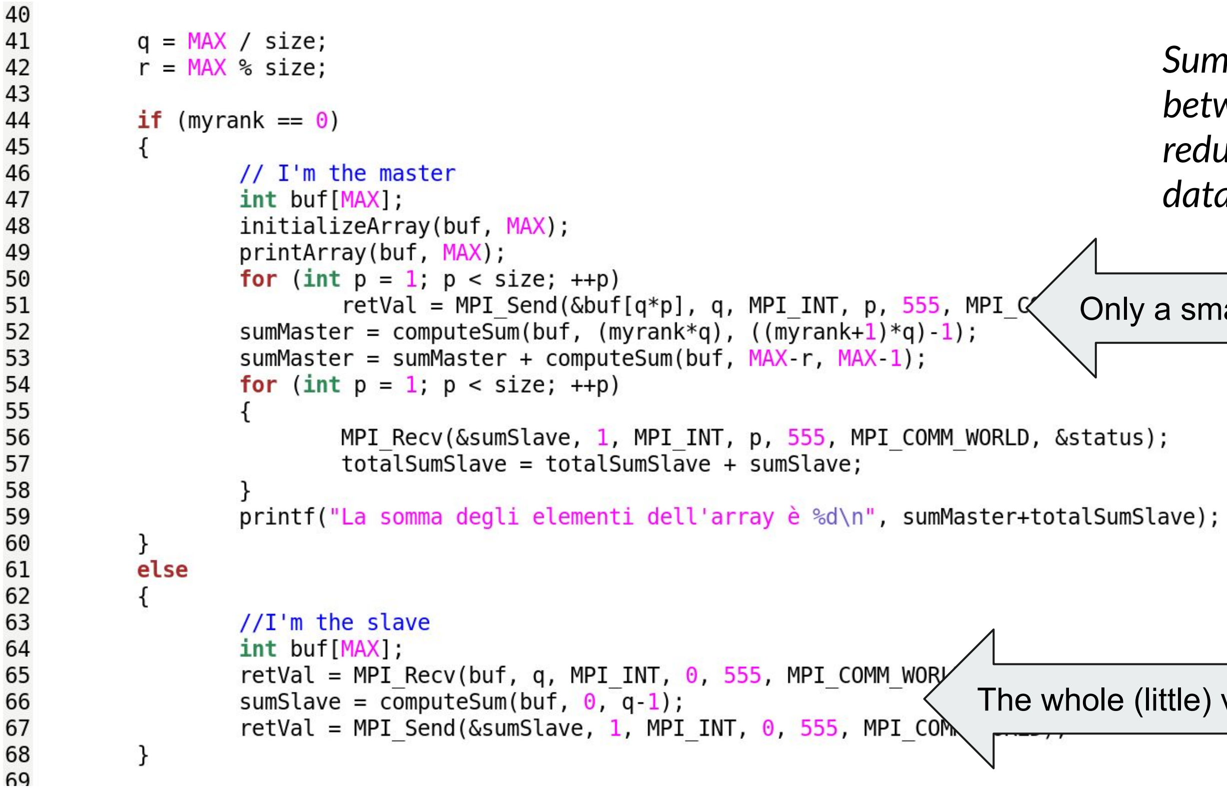
mpirun -n 4 example04.o

Observation

- Parallel execution among 4 processes
- Slave processes compute sum operation using only 3 integers
- Master instead makes more work (because of the vector’s tail)
- The execution ends when all processes complete their own execution
  - the slower processes slows down the application execution
  - balancing problem
- The whole vector is sent to all slave processes (not just the data each slave should work on)
  - communication time problem

How can we send to the slave processes only the data they really need?

Summing up integers: example06.c



```
        q = MAX / size;
        r = MAX % size;

        if (myrank == 0)
        {
                // I'm the master
                int buf[MAX];
                initializeArray(buf, MAX);
                printArray(buf, MAX);
                for (int p = 1; p < size; ++p)
                        retVal = MPI_Send(&buf[q*p], q, MPI_INT, p, 555, MPI_C
                sumMaster = computeSum(buf, (myrank*q), ((myrank+1)*q)-1);
                sumMaster = sumMaster + computeSum(buf, MAX-r, MAX-1);
                for (int p = 1; p < size; ++p)
                {
                        MPI_Recv(&sumSlave, 1, MPI_INT, p, 555, MPI_COMM_WORLD, &status);
                        totalSumSlave = totalSumSlave + sumSlave;
                }
                printf("La somma degli elementi dell'array è %d\n", sumMaster+totalSumSlave);
        }
        else
        {

                //I'm the slave
                int buf[MAX];
                retVal = MPI_Recv(buf, q, MPI_INT, 0, 555, MPI_COMM_WORL
                sumSlave = computeSum(buf, 0, q-1);
                retVal = MPI_Send(&sumSlave, 1, MPI_INT, 0, 555, MPI_COM
        }
```

Summing integer elements between N processes but reducing the total amount of data sent

Only a small section of the vector is sent

The whole (little) vector is computed

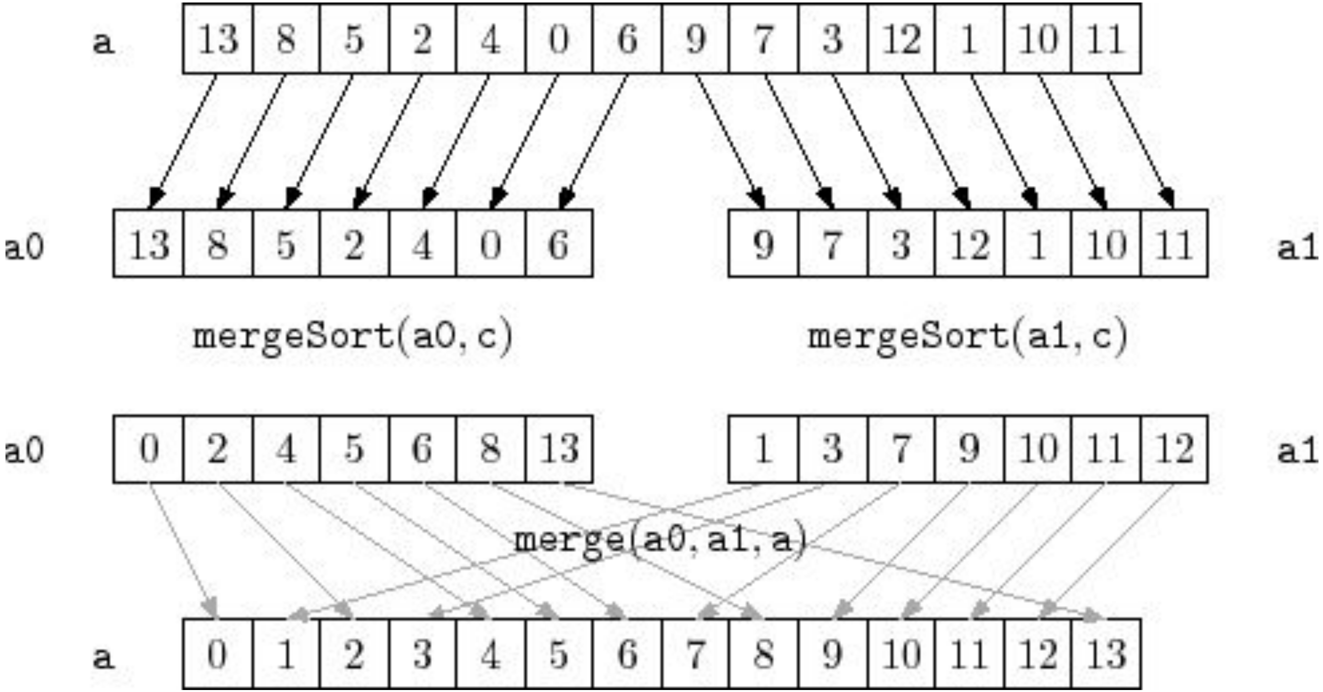
Your turn

- Can the code on example05 and example06 be executed with 3 parallel process without any changes? And with 100?

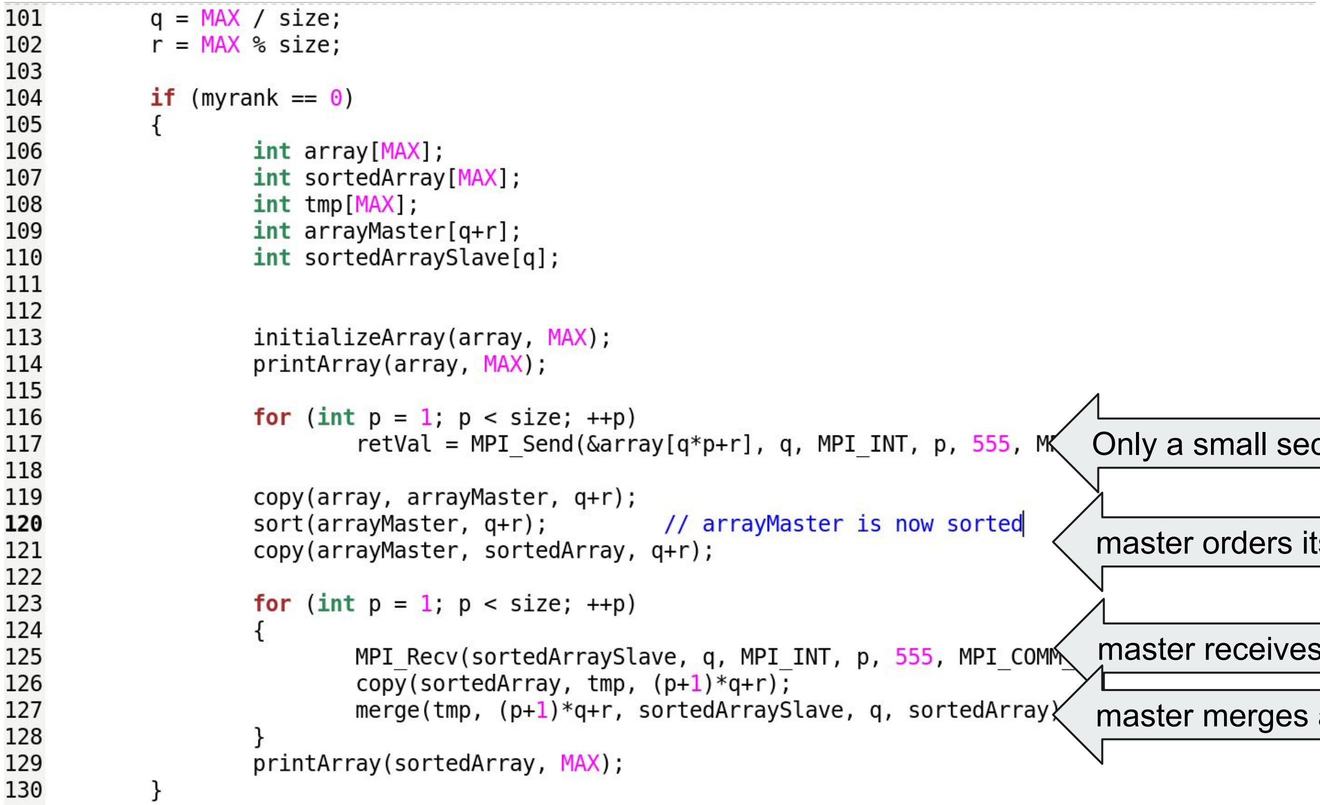
- In example05 and example06, initialization is only made by master (in serial way). Is there a way to parallelize initialization process?

- Think about a method to reduce the long-tail effect on the master

Parallel Sort: overview



Parallel Sort (with merging): example07.c



```
    q = MAX / size;
    r = MAX % size;

    if (myrank == 0)
    {
        int array[MAX];
        int sortedArray[MAX];
        int tmp[MAX];
        int arrayMaster[q+r];
        int sortedArraySlave[q];


        initializeArray(array, MAX);
        printArray(array, MAX);

        for (int p = 1; p < size; ++p)
            retVal = MPI_Send(&array[q*p+r], q, MPI_INT, p, 555, M

        copy(array, arrayMaster, q+r);
        sort(arrayMaster, q+r);         // arrayMaster is now sorted
        copy(arrayMaster, sortedArray, q+r);

        for (int p = 1; p < size; ++p)
        {
            MPI_Recv(sortedArraySlave, q, MPI_INT, p, 555, MPI_COMM_
            copy(sortedArray, tmp, (p+1)*q+r);
            merge(tmp, (p+1)*q+r, sortedArraySlave, q, sortedArray)
        }
        printArray(sortedArray, MAX);
    }
```

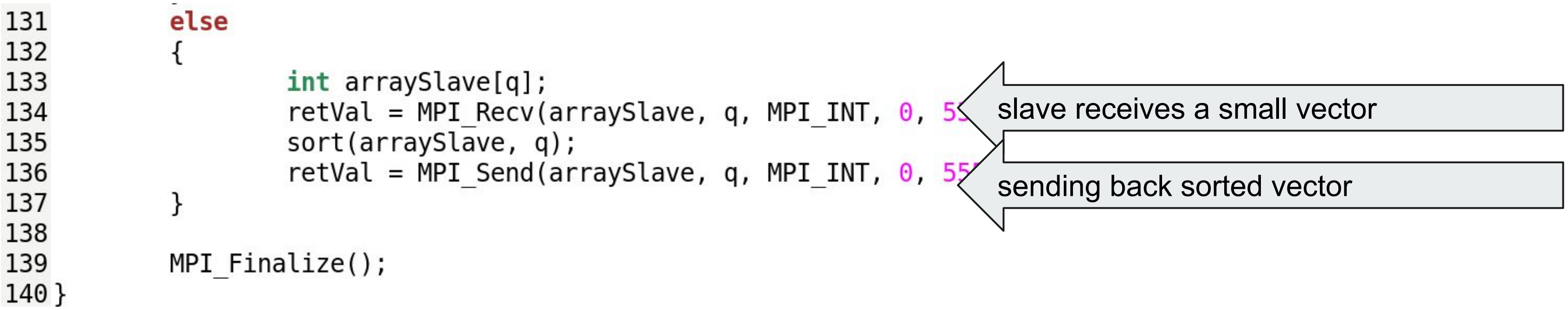
Only a small section of the vector is sent

master orders its own subvector

master receives ordered vectors from slaves

master merges all subvectors

Parallel Sort (with merging): example07.c

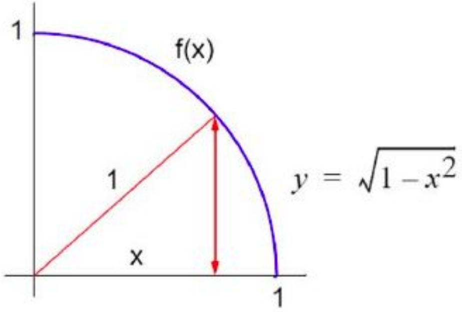


```
        else
        {
                int arraySlave[q];
                retVal = MPI_Recv(arraySlave, q, MPI_INT, 0, 5
                sort(arraySlave, q);
                retVal = MPI_Send(arraySlave, q, MPI_INT, 0, 55
        }

        MPI_Finalize();
}
```

slave receives a small vector

sending back sorted vector

$$\int_0^1 \sqrt{1-x^2}\,dx = \frac{\pi}{4}$$



Computing π in parallel: overview

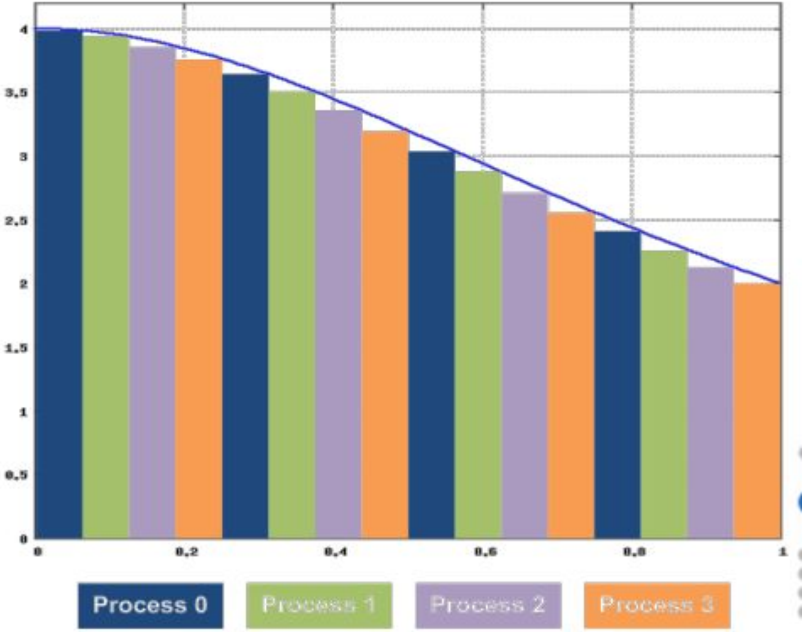
We know that in general if *f(x)* is a integrable function:

$$\int_a^b f(x)\,dx = \lim_{N\to\infty} \sum_{i=1}^{N} f_i h \quad \textit{with} \quad f_i = f(a+ih) \quad \textit{and} \quad h = \frac{b-a}{N}$$

To compute π $$\pi = \int_0^1 \frac{4}{1+x^2}\,dx = \lim_{N\to\infty} \sum_{i=1}^{N} \frac{4h}{1+(ih)^2} \quad \textit{with } h = \frac{1}{N}$$

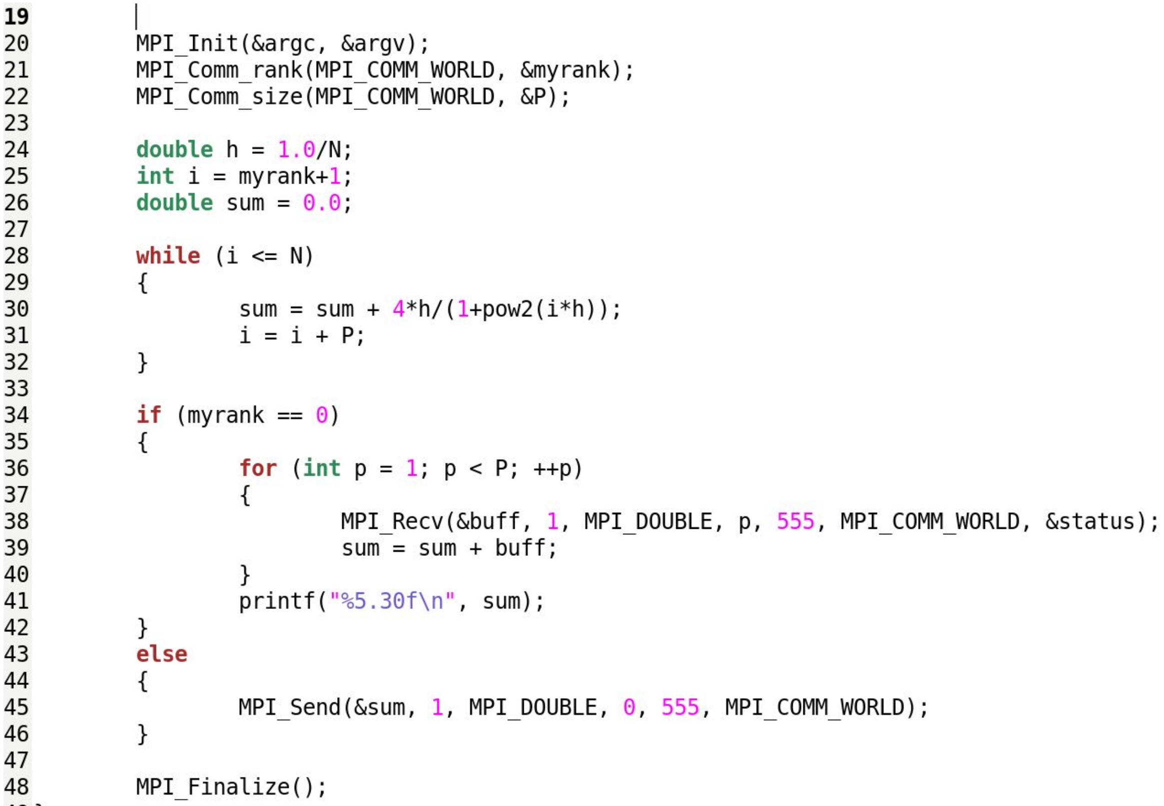
Using a very enough large N $$\pi \cong \sum_{i=1}^{N} \frac{4h}{1+(ih)^2} \quad \text{with} \quad h = \frac{1}{N}$$

Computing π in parallel: overview



- The interval [0, 1] is split into ***N*** parts
- Each part is assigned to a process $\boldsymbol{p_i}$
- each $\boldsymbol{p_i}$ process works on its own sub-interval
- process $\boldsymbol{p_0}$ gathers all results and sum them up all together

Computing π in parallel: example08.c



```

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &P);

        double h = 1.0/N;
        int i = myrank+1;
        double sum = 0.0;

        while (i <= N)
        {
                sum = sum + 4*h/(1+pow2(i*h));
                i = i + P;
        }

        if (myrank == 0)
        {
                for (int p = 1; p < P; ++p)
                {
                        MPI_Recv(&buff, 1, MPI_DOUBLE, p, 555, MPI_COMM_WORLD, &status);
                        sum = sum + buff;
                }
                printf("%5.30f\n", sum);
        }
        else
        {
                MPI_Send(&sum, 1, MPI_DOUBLE, 0, 555, MPI_COMM_WORLD);
        }

        MPI_Finalize();
```

Embarrassingly parallelism

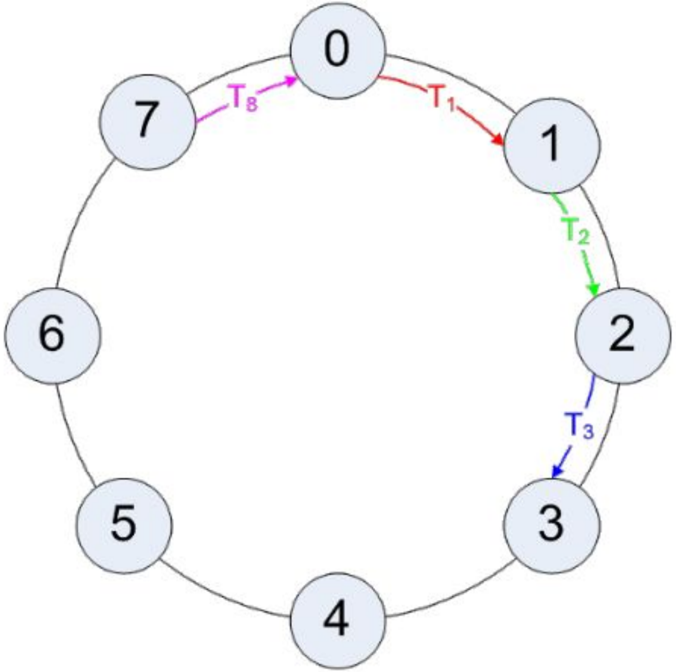
An embarrassingly parallel program is one where little or no effort is needed to separate the problem into a number of parallel tasks.

This is often the case where there is little or no dependency or need for communication between those parallel tasks, or for results between them

Computing the PI is an **embarrassingly parallel problem**

Your turn: communication ring

- process 0 reads an integer from standard standard input
- process 0 sends the integer to process 1
- process 1 receives the integer, decrease it and sends forward to process 2
- the cycle goes on until the last process gets the integer.
- The last process sends back the integer to 0, that displays the number

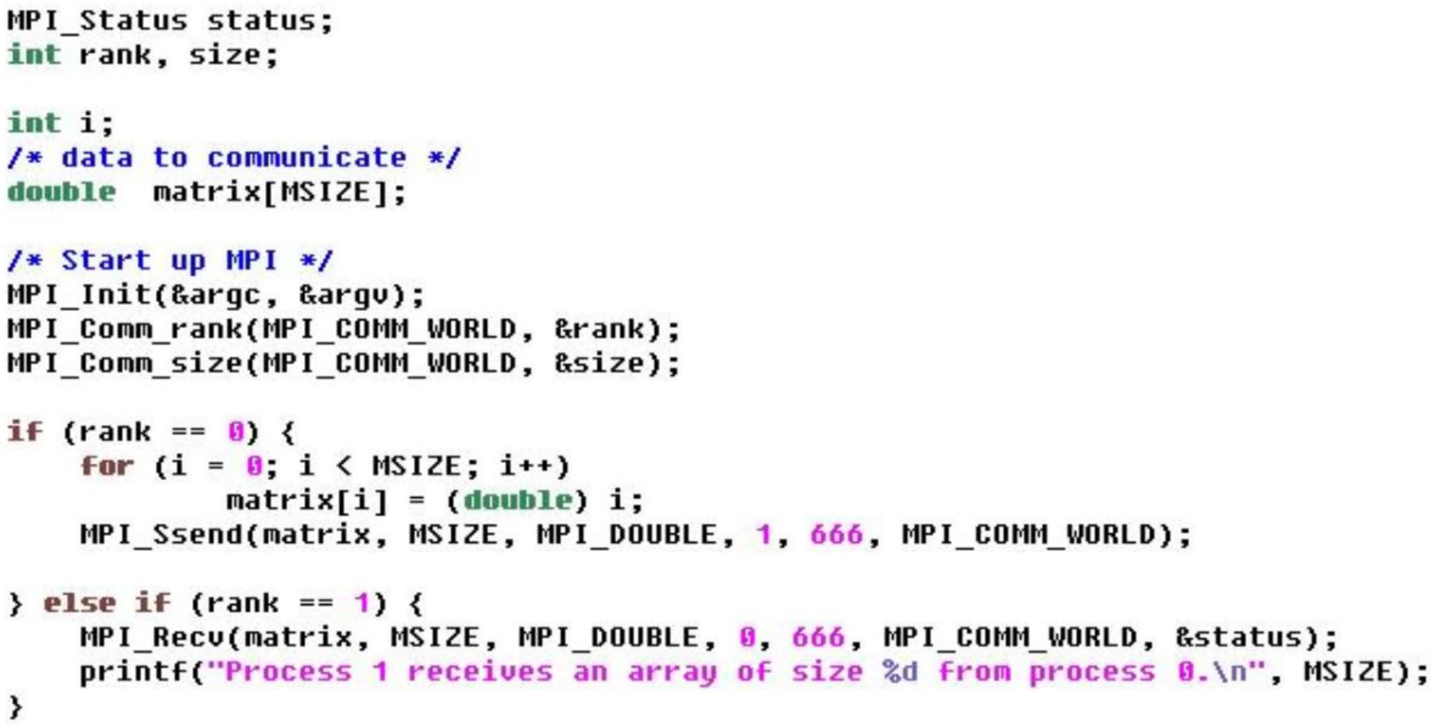


MPI_Ssend

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

- buf: initial buffer
- `count`: number of elements in send buffer
- datatype: datatype of each send buffer element
- dest: rank of destination
- tag: message tag
- comm: communicator

MPI_Ssend: example13.c



```
MPI_Status status;
int rank, size;

int i;
/* data to communicate */
double  matrix[MSIZE];

/* Start up MPI */
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (rank == 0) {
    for (i = 0; i < MSIZE; i++)
            matrix[i] = (double) i;
    MPI_Ssend(matrix, MSIZE, MPI_DOUBLE, 1, 666, MPI_COMM_WORLD);

} else if (rank == 1) {
    MPI_Recv(matrix, MSIZE, MPI_DOUBLE, 0, 666, MPI_COMM_WORLD, &status);
    printf("Process 1 receives an array of size %d from process 0.\n", MSIZE);
}
```

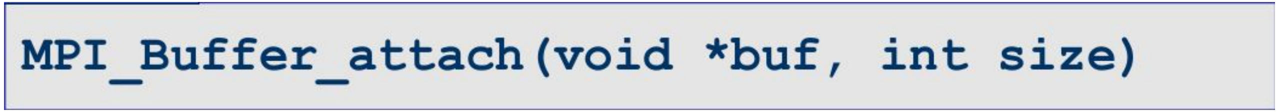
MPI_Bsend

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

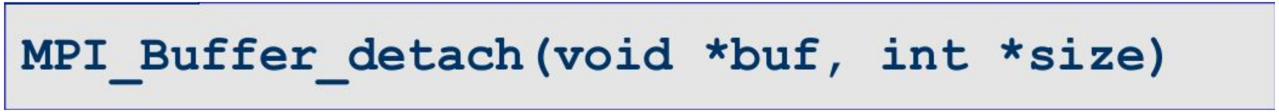
- buf: initial buffer
- `count`: number of elements in send buffer
- datatype: datatype of each send buffer element
- dest: rank of destination
- tag: message tag
- comm: communicator

Before using BSend, the buffer needs to be attached

Attaching and detaching



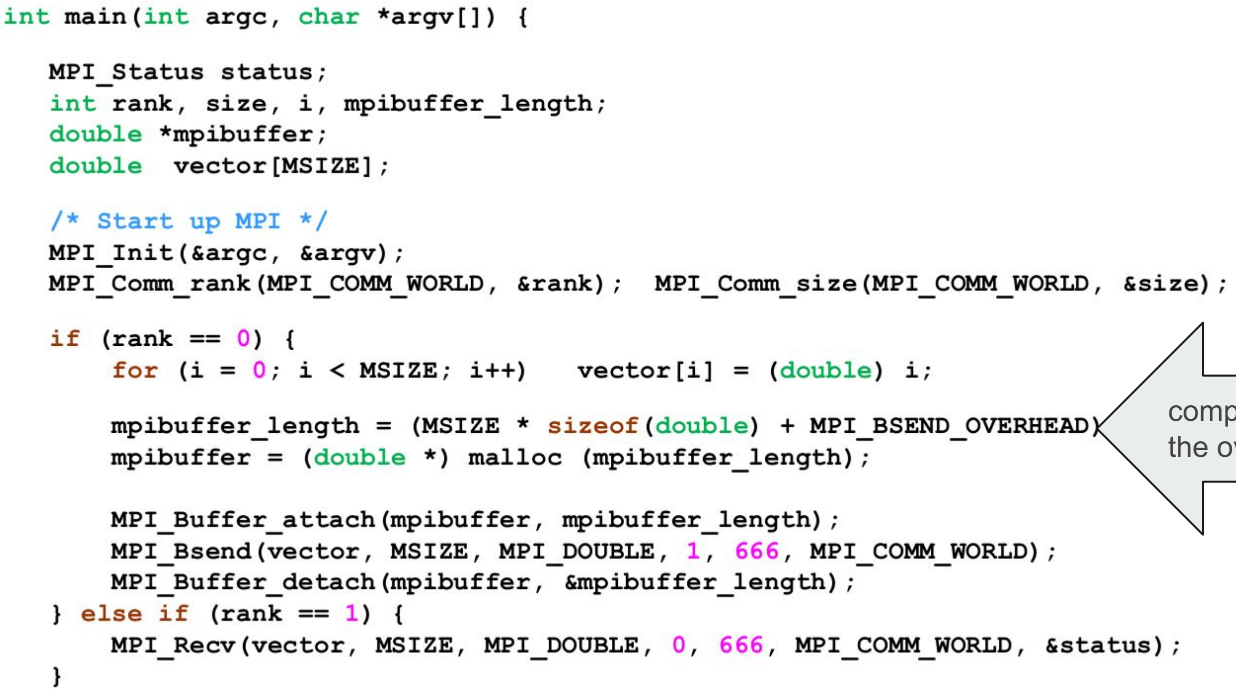
```
MPI_Buffer_attach(void *buf, int size)
```



```
MPI_Buffer_detach(void *buf, int *size)
```

MPI_BSEND_OVERHEAD
represents the size, in bytes, of the memory overhead generated everytime an MPI_Bsend or MPI_Ibsend is issued.

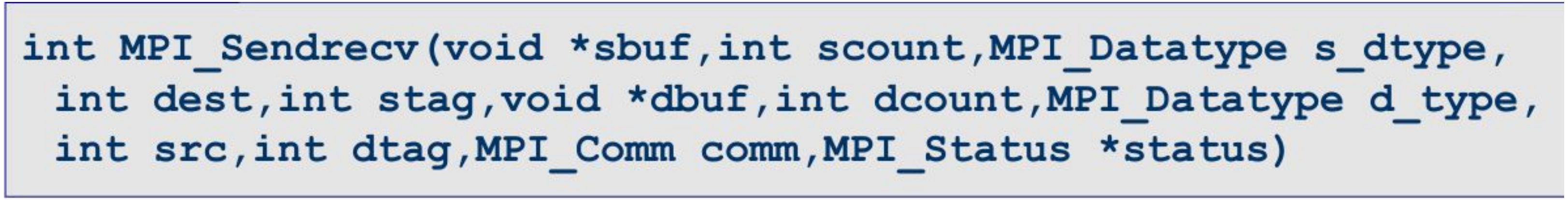
MPI_Bsend: example14.c



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

   MPI_Status status;
   int rank, size, i, mpibuffer_length;
   double *mpibuffer;
   double  vector[MSIZE];

   /* Start up MPI */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);  MPI_Comm_size(MPI_COMM_WORLD, &size);

   if (rank == 0) {
       for (i = 0; i < MSIZE; i++)   vector[i] = (double) i;

       mpibuffer_length = (MSIZE * sizeof(double) + MPI_BSEND_OVERHEAD);
       mpibuffer = (double *) malloc (mpibuffer_length);

       MPI_Buffer_attach(mpibuffer, mpibuffer_length);
       MPI_Bsend(vector, MSIZE, MPI_DOUBLE, 1, 666, MPI_COMM_WORLD);
       MPI_Buffer_detach(mpibuffer, &mpibuffer_length);
   } else if (rank == 1) {
       MPI_Recv(vector, MSIZE, MPI_DOUBLE, 0, 666, MPI_COMM_WORLD, &status);
   }
```

computes the size of the buffer considering the overhead introduced by BSEND

MPI_Sendrecv



```
int MPI_Sendrecv(void *sbuf,int scount,MPI_Datatype s_dtype,
  int dest,int stag,void *dbuf,int dcount,MPI_Datatype d_type,
  int src,int dtag,MPI_Comm comm,MPI_Status *status)
```

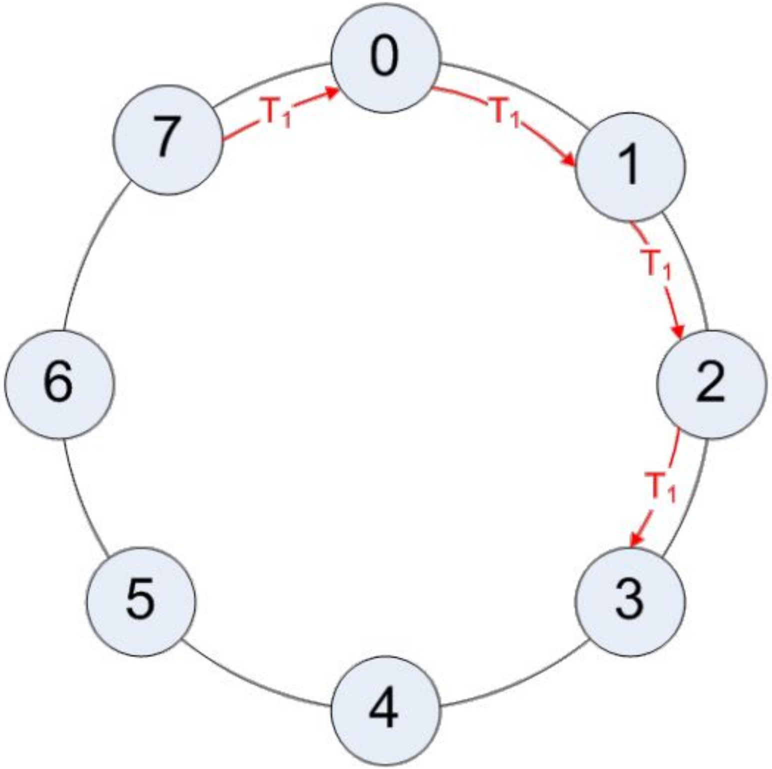
- sbuf: initial buffer for sender
- scount: number of elements in send buffer
- s_dtype: datatype of each buffer element sent
- dest: rank of destination
- stag: message tag for sending
- comm: communicator

- dbuf : initial buffer for receiver
- dcount: number of elements in receiver buffer
- d_type: datatype of each buffer element received
- src: the sender’s rank
- dtag: receive tag
- status

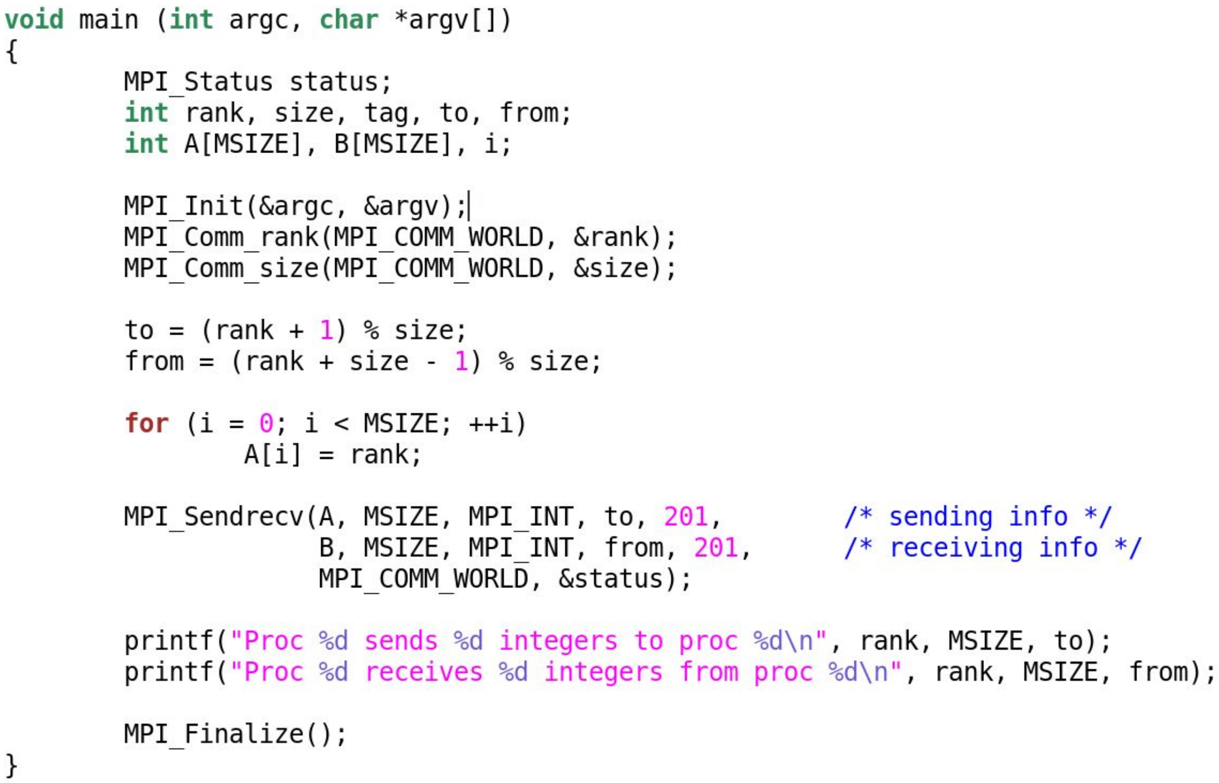
Circular Shift

Let’s suppose now that, differently from the previous one, we want that all processes send a message to the neighbor at the same time (all of the in T_1)

Using SEND and RECEIVE functions arise a **Deadlock** (because there is correspondence between sending and receiving, but using MPI_Sendrecv we drop the problem



Circular Shift: example09.c



```c
void main (int argc, char *argv[])
{
        MPI_Status status;
        int rank, size, tag, to, from;
        int A[MSIZE], B[MSIZE], i;

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

        to = (rank + 1) % size;
        from = (rank + size - 1) % size;

        for (i = 0; i < MSIZE; ++i)
                A[i] = rank;

        MPI_Sendrecv(A, MSIZE, MPI_INT, to, 201,        /* sending info */
                     B, MSIZE, MPI_INT, from, 201,      /* receiving info */
                     MPI_COMM_WORLD, &status);

        printf("Proc %d sends %d integers to proc %d\n", rank, MSIZE, to);
        printf("Proc %d receives %d integers from proc %d\n", rank, MSIZE, from);

        MPI_Finalize();
}
```

MPI_Isend

```
int MPI_Isend(void *buf, int count,
   MPI_Datatype datatype, int dest, int tag,
   MPI_Comm comm, MPI_Request *request)
```

- similar to MPI_Send
- `request`: pointer to be used in MPI_Wait

MPI_Issend

```
int MPI_Issend(void *buf, int count,
   MPI_Datatype datatype, int dest, int tag,
   MPI_Comm comm, MPI_Request *request)
```

- similar to MPI_Ssend
- `request`: pointer to be used in MPI_Wait

MPI_Ibsend

```
int MPI_Ibsend(void *buf, int count,
   MPI_Datatype datatype, int dest, int tag,
   MPI_Comm comm, MPI_Request *request)
```

- similar to MPI_Bsend
- `request`: pointer to be used in MPI_Wait

MPI_Irsend

```
int MPI_Irsend(void *buf, int count,
   MPI_Datatype datatype, int dest, int tag,
   MPI_Comm comm, MPI_Request *request)
```

- similar to MPI_Rsend
- `request`: pointer to be used in MPI_Wait

MPI_Irecv

```
int MPI_Irecv(void *buf, int count,
   MPI_Datatype datatype, int source, int tag,
   MPI_Comm comm, MPI_Request *request)
```

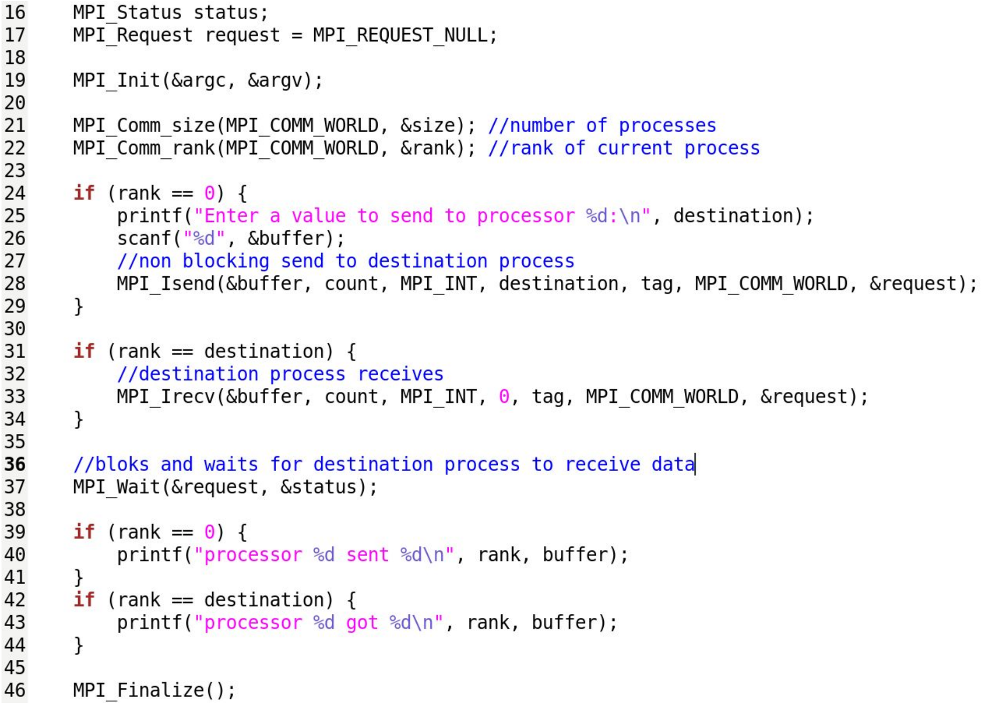
- similar to MPI_Recv
- `request`: pointer to be used in MPI_Wait

MPI_Wait

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- `request`: pointer used in MPI_I*send

Using non-blocking functions: example10.c

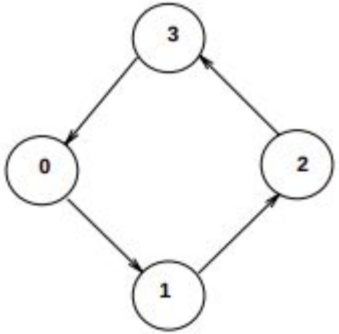


```c
    MPI_Status status;
    MPI_Request request = MPI_REQUEST_NULL;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size); //number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); //rank of current process

    if (rank == 0) {
        printf("Enter a value to send to processor %d:\n", destination);
        scanf("%d", &buffer);
        //non blocking send to destination process
        MPI_Isend(&buffer, count, MPI_INT, destination, tag, MPI_COMM_WORLD, &request);
    }

    if (rank == destination) {
        //destination process receives
        MPI_Irecv(&buffer, count, MPI_INT, 0, tag, MPI_COMM_WORLD, &request);
    }

    //bloks and waits for destination process to receive data
    MPI_Wait(&request, &status);

    if (rank == 0) {
        printf("processor %d sent %d\n", rank, buffer);
    }
    if (rank == destination) {
        printf("processor %d got %d\n", rank, buffer);
    }

    MPI_Finalize();
```

Exercises

Exercises

- Ping Pong: write a program in which two processes repeatedly pass a message back and forth
- Rotating: each process stores it own rank, then sends this value to the process on its right. The process continues passing on the values they receive until they get their own rank back. Each process should finish by printing out the sum of the values.



Exercises

- Ordering: consider a 2-dimensional matrix. Each row is ordered

4	0	3
5	2	7
2	3	1

0	3	4
2	5	7
1	2	3

Exercises

- Simple Array Assignment: The master task initiates numtasks-1 number of worker tasks and then distributes an equal portion of the array to each worker. Each worker receives its portion of the array and performs a simple value assignment to each of its elements. The value assigned to each element is simply that element's index in the array plus 1. Each worker then sends its portion of the array back to the master. As the master receives a portion of the array from each worker, selected elements are displayed.

Exercises

- Matrix Multiplication: This example is a simple matrix multiplication program, i.e. AxB=C. Matrix A is copied to every processor. Matrix B is divided into blocks and distributed among processors. The data is distributed among the workers who perform the actual multiplication in smaller blocks and send back their results to the master.

MPI: collective functions

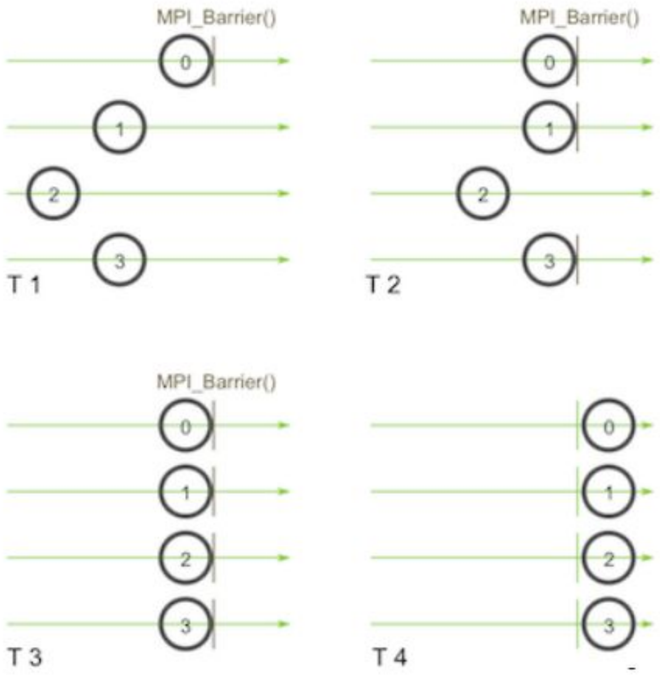
Collective functions

When communication involves all processes, instead of using point-to-point functions. Three classes:

- Synchronization
  - MPI_Barrier
- Global Communication (data movement)
  - MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Allgather, MPI_Alltoall
- Global Reduction (collective computation)
  - MPI_Reduce, MPI_Allreduce, MPI_Reduce_scatter, MPI_Scan

Synchronization: MPI_Barrier

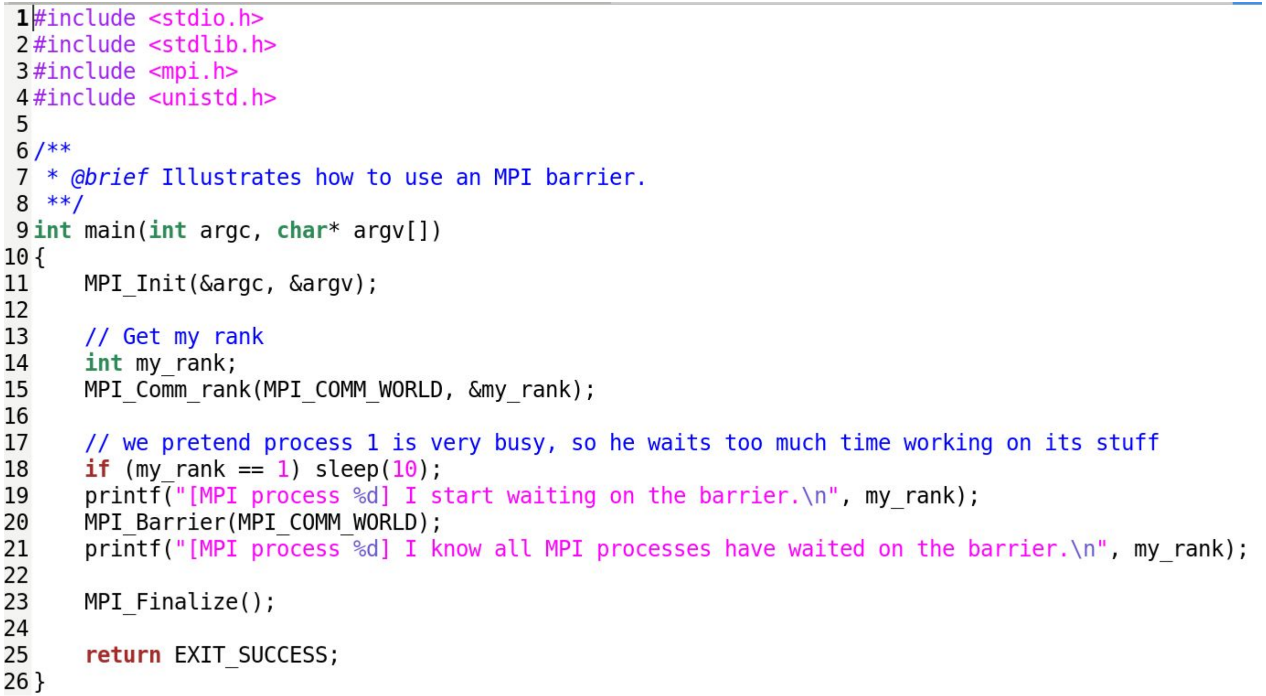
- Blocks until all processes in the group of the same communicator
- Used for synchronization



MPI_Barrier

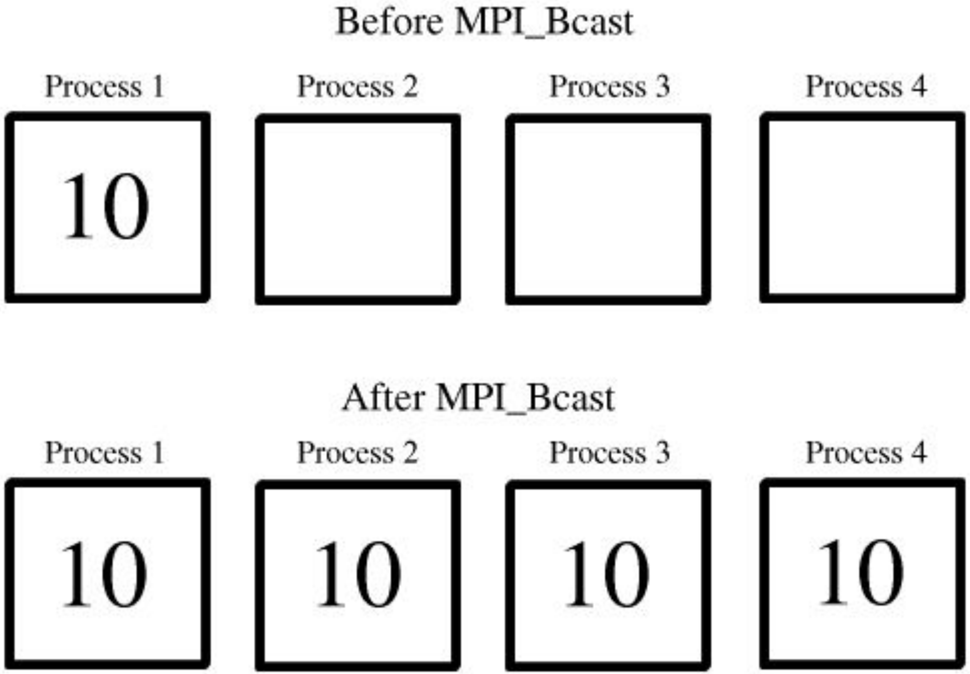
```
int MPI_Barrier(MPI_Comm comm)
```

MPI_Barrier: example15.c



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

/**
 * @brief Illustrates how to use an MPI barrier.
 **/
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // we pretend process 1 is very busy, so he waits too much time working on its stuff
    if (my_rank == 1) sleep(10);
    printf("[MPI process %d] I start waiting on the barrier.\n", my_rank);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("[MPI process %d] I know all MPI processes have waited on the barrier.\n", my_rank);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Global communication: MPI_Bcast



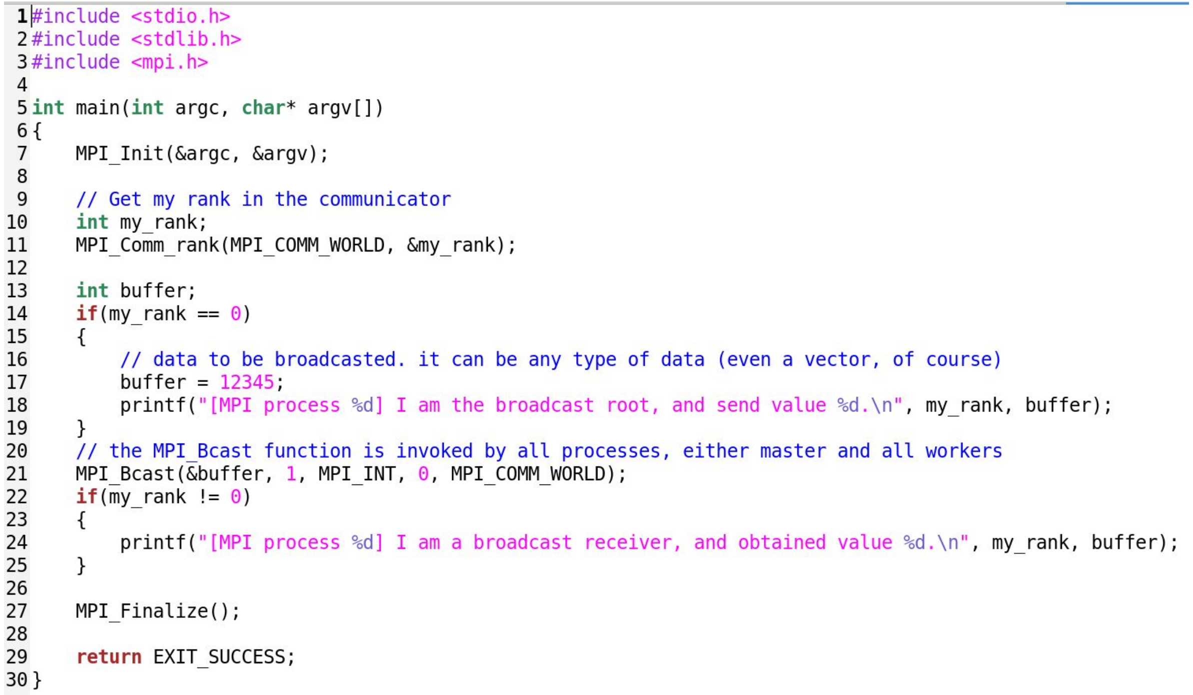
The same data is sent from the master process to the other processes

MPI_Bcast

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

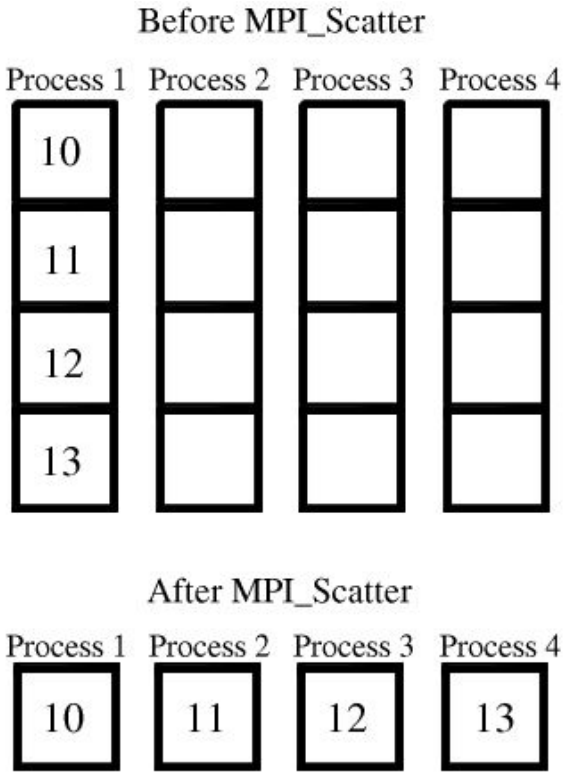
- buffer: point to the buffer
- count: number of entries in the buffer
- root: rank of process master (who sends data to each others)

MPI_Bcast: example16.c



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

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

    // Get my rank in the communicator
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    int buffer;
    if(my_rank == 0)
    {
        // data to be broadcasted. it can be any type of data (even a vector, of course)
        buffer = 12345;
        printf("[MPI process %d] I am the broadcast root, and send value %d.\n", my_rank, buffer);
    }
    // the MPI_Bcast function is invoked by all processes, either master and all workers
    MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if(my_rank != 0)
    {
        printf("[MPI process %d] I am a broadcast receiver, and obtained value %d.\n", my_rank, buffer);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Global communication: MPI_Scatter



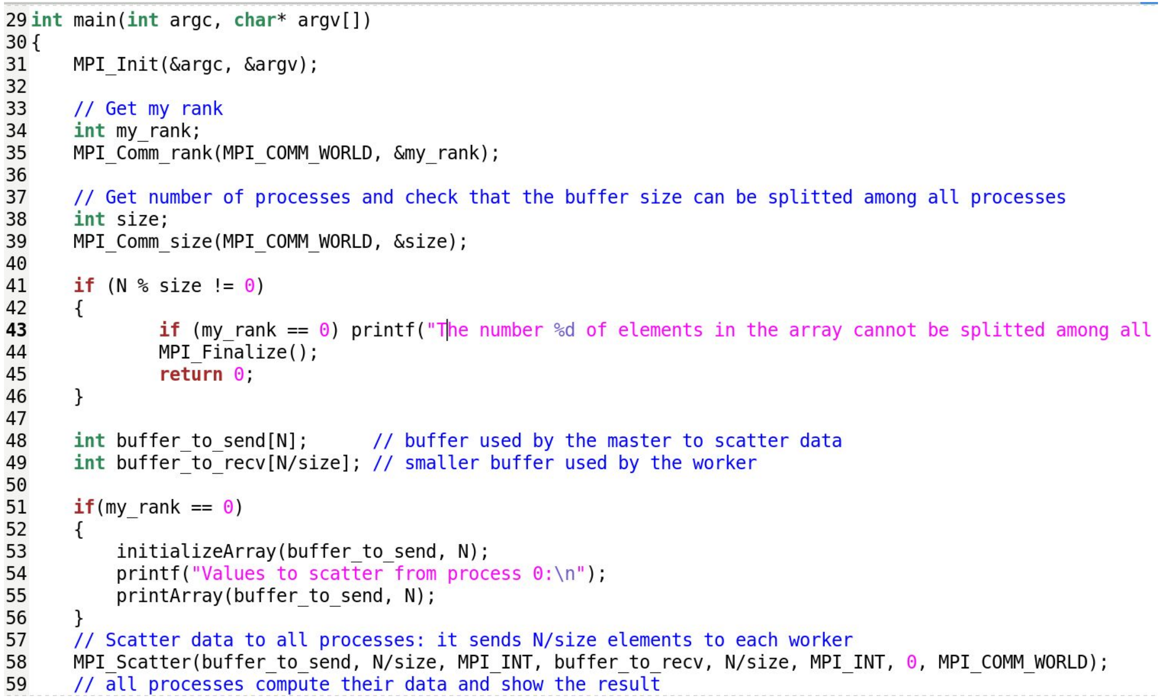
The vector of data is split in N parts (where N is the number of processes). Each part is sent to each process

MPI_Scatter

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

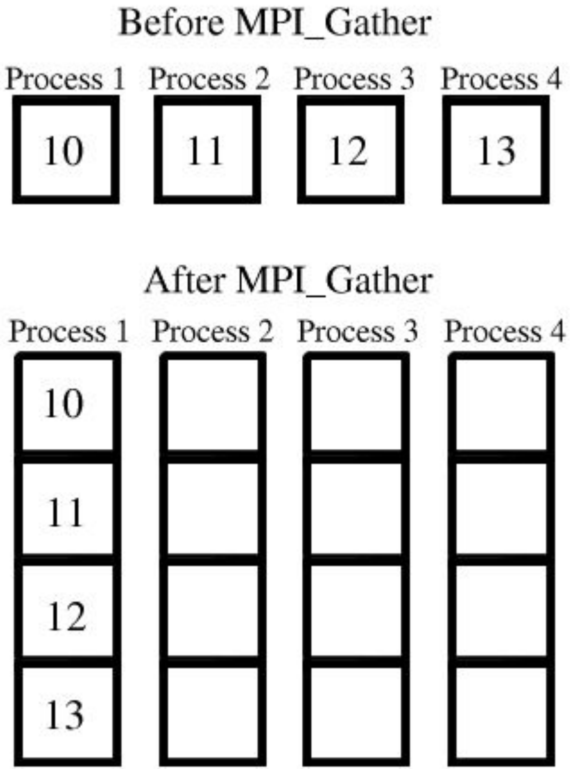
- sendbuf: address of send buffer (significant only for root)
- sendcount: number of elements sent to each process
- recvbuf: address of receive buffer
- recvcount: number of elements received
- root: rank of process master (who sends data to each others)

MPI_Scatter: example17.c



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

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get number of processes and check that the buffer size can be splitted among all processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (N % size != 0)
    {
            if (my_rank == 0) printf("The number %d of elements in the array cannot be splitted among all
            MPI_Finalize();
            return 0;
    }

    int buffer_to_send[N];      // buffer used by the master to scatter data
    int buffer_to_recv[N/size]; // smaller buffer used by the worker

    if(my_rank == 0)
    {
        initializeArray(buffer_to_send, N);
        printf("Values to scatter from process 0:\n");
        printArray(buffer_to_send, N);
    }
    // Scatter data to all processes: it sends N/size elements to each worker
    MPI_Scatter(buffer_to_send, N/size, MPI_INT, buffer_to_recv, N/size, MPI_INT, 0, MPI_COMM_WORLD);
    // all processes compute their data and show the result
```

Global communication: MPI_Gather

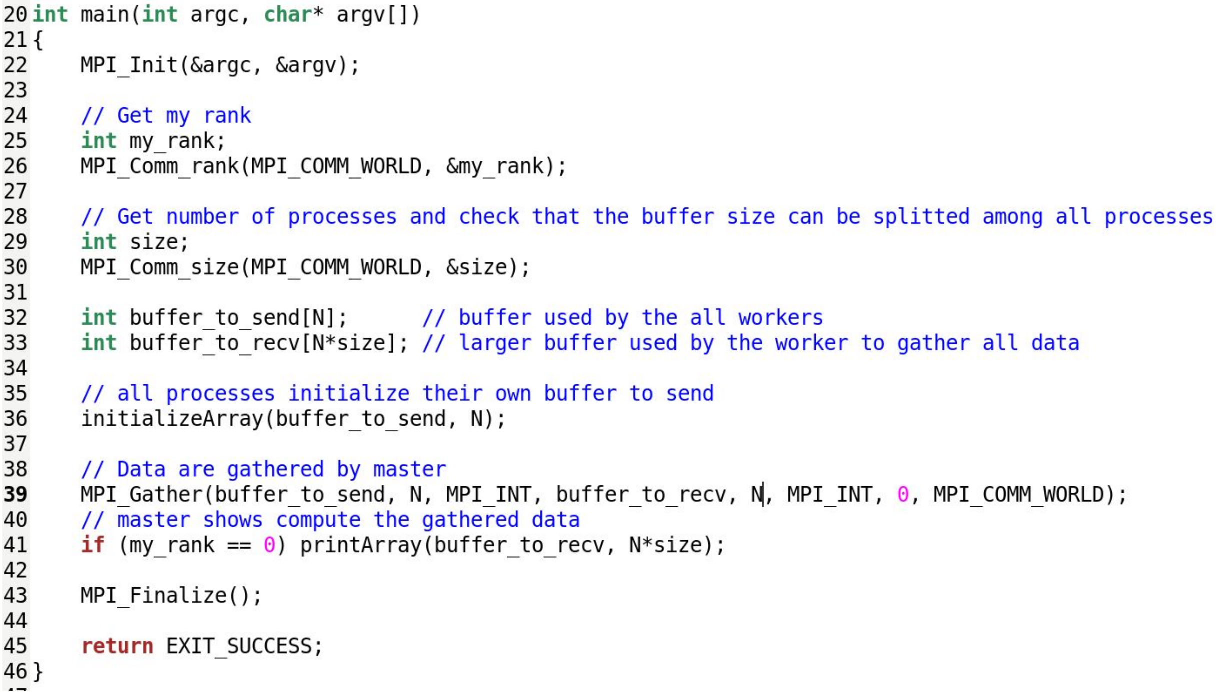


MPI_Gather

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

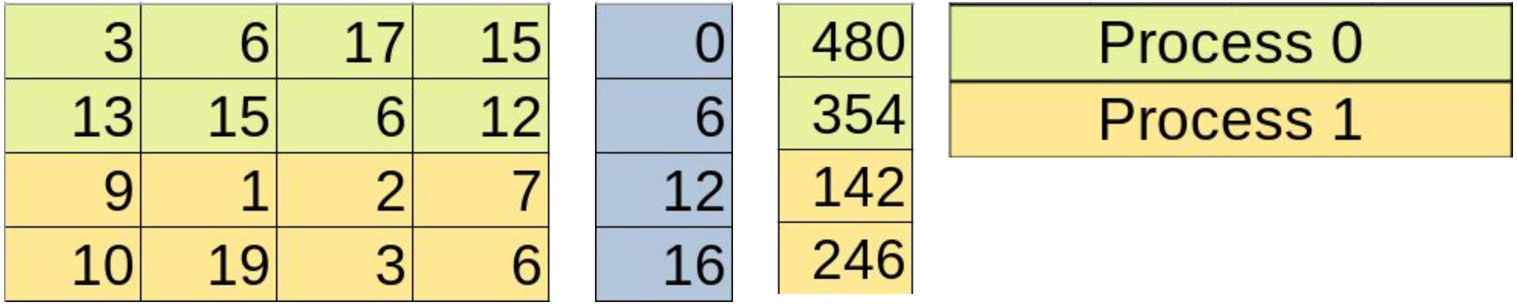
- sendbuf: address of send buffer
- sendcount: number of elements sent from each process
- recvbuf: address of receive buffer
- recvcount: number of elements received
- root: rank of process master (who receives data to each others)

MPI_Gather: example18.c



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

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get number of processes and check that the buffer size can be splitted among all processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int buffer_to_send[N];      // buffer used by the all workers
    int buffer_to_recv[N*size]; // larger buffer used by the worker to gather all data

    // all processes initialize their own buffer to send
    initializeArray(buffer_to_send, N);

    // Data are gathered by master
    MPI_Gather(buffer_to_send, N, MPI_INT, buffer_to_recv, N, MPI_INT, 0, MPI_COMM_WORLD);
    // master shows compute the gathered data
    if (my_rank == 0) printArray(buffer_to_recv, N*size);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Putting gather, scatter and broadcast together



3	6	17	15
13	15	6	12
9	1	2	7
10	19	3	6

Matrix

0
6
12
16

Vector

480
354
142
246

Result

Process 0
Process 1

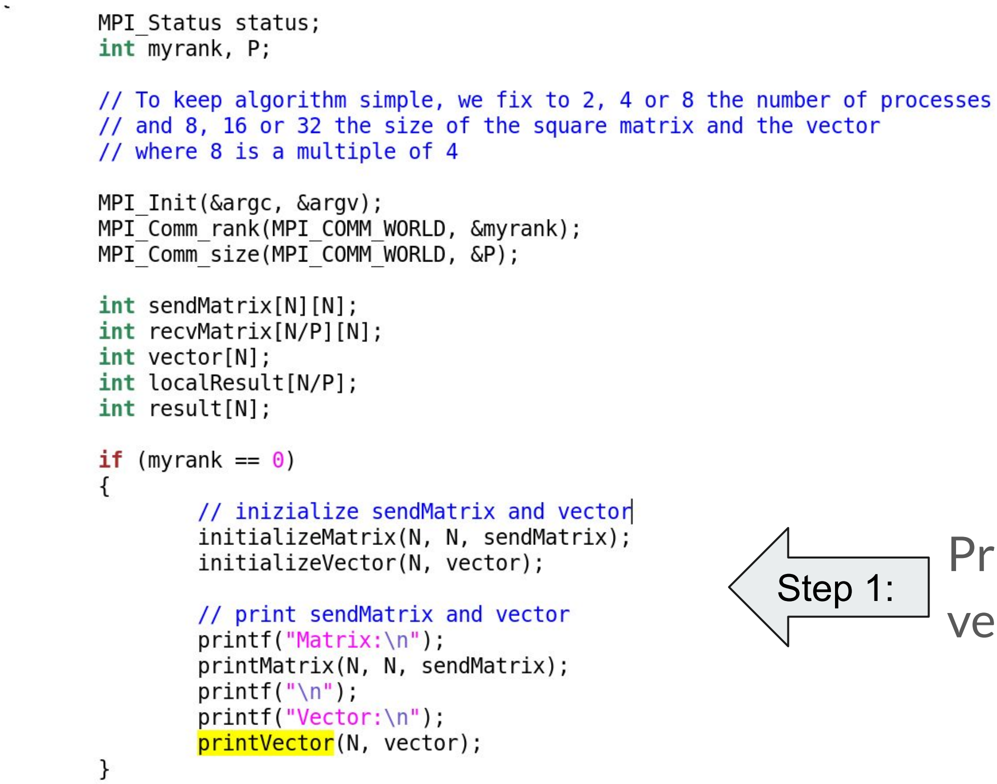
The original matrix is split in P parts, where P is the number of processes. Each process computes multiplication of the submatrix and the vector. Then the result is stores in a subvector. All subvectors are concat together

Matrix multiplication: example12.c

For simplicity, we suppose that **N is a multiple of P**

1. Process 0 initializes matrix and vector, then print both
2. Process 0 scatters matrix to all processes
3. Process 0 broadcasts vectors to all processes
4. Each process computes matrix multiplication, the stores the results in a local vector
5. Process 0 gathers all local vectors, getting the final result
6. Process 0 visualizes the final result

Matrix multiplication: example12.c

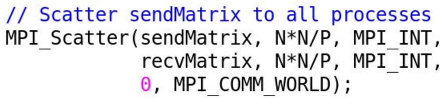


```
`
        MPI_Status status;
        int myrank, P;

        // To keep algorithm simple, we fix to 2, 4 or 8 the number of processes
        // and 8, 16 or 32 the size of the square matrix and the vector
        // where 8 is a multiple of 4

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &P);

        int sendMatrix[N][N];
        int recvMatrix[N/P][N];
        int vector[N];
        int localResult[N/P];
        int result[N];

        if (myrank == 0)
        {
                // inizialize sendMatrix and vector
                initializeMatrix(N, N, sendMatrix);
                initializeVector(N, vector);

                // print sendMatrix and vector
                printf("Matrix:\n");
                printMatrix(N, N, sendMatrix);
                printf("\n");
                printf("Vector:\n");
                printVector(N, vector);
        }
```

Step 1:

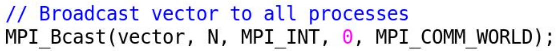
Process 0 initializes matrix and vector, then prints both

Matrix multiplication: example12.c



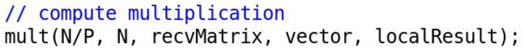
```
// Scatter sendMatrix to all processes
MPI_Scatter(sendMatrix, N*N/P, MPI_INT,
            recvMatrix, N*N/P, MPI_INT,
            0, MPI_COMM_WORLD);
```

Step 2: Process 0 scatters matrix to all processes



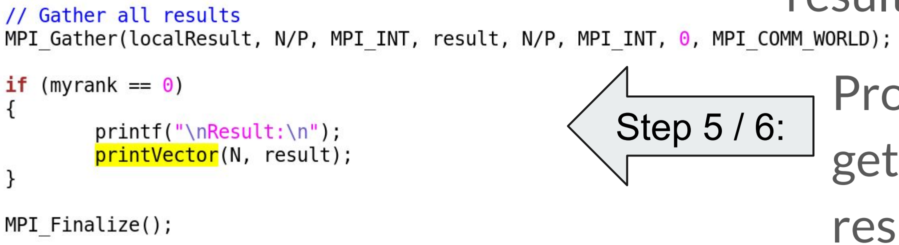
```
// Broadcast vector to all processes
MPI_Bcast(vector, N, MPI_INT, 0, MPI_COMM_WORLD);
```

Step 3: Process 0 broadcasts vector to all processes



```
// compute multiplication
mult(N/P, N, recvMatrix, vector, localResult);
```

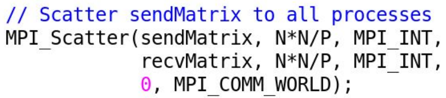
Step 4: Each process computes matrix multiplication, the stores the results in a local vector



```
// Gather all results
MPI_Gather(localResult, N/P, MPI_INT, result, N/P, MPI_INT, 0, MPI_COMM_WORLD);

if (myrank == 0)
{
        printf("\nResult:\n");
        printVector(N, result);
}

MPI_Finalize();
```

Step 5 / 6: Process 0 gathers all local vectors, getting the final result + print result

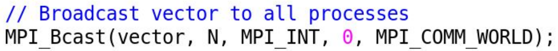
Matrix multiplication: example12.c



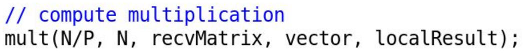
```
// Scatter sendMatrix to all processes
MPI_Scatter(sendMatrix, N*N/P, MPI_INT,
            recvMatrix, N*N/P, MPI_INT,
            0, MPI_COMM_WORLD);
```

Summing integer elements between 2 processes

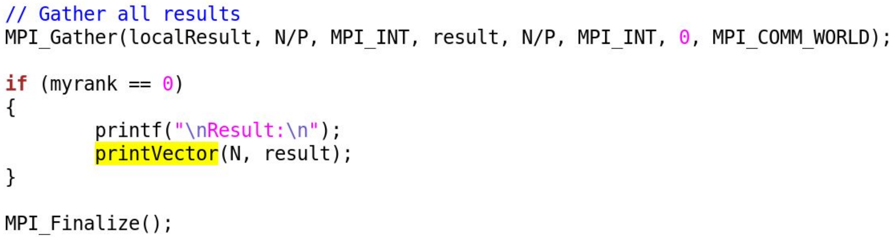
mpirun -n 4 example12.o



```
// Broadcast vector to all processes
MPI_Bcast(vector, N, MPI_INT, 0, MPI_COMM_WORLD);
```



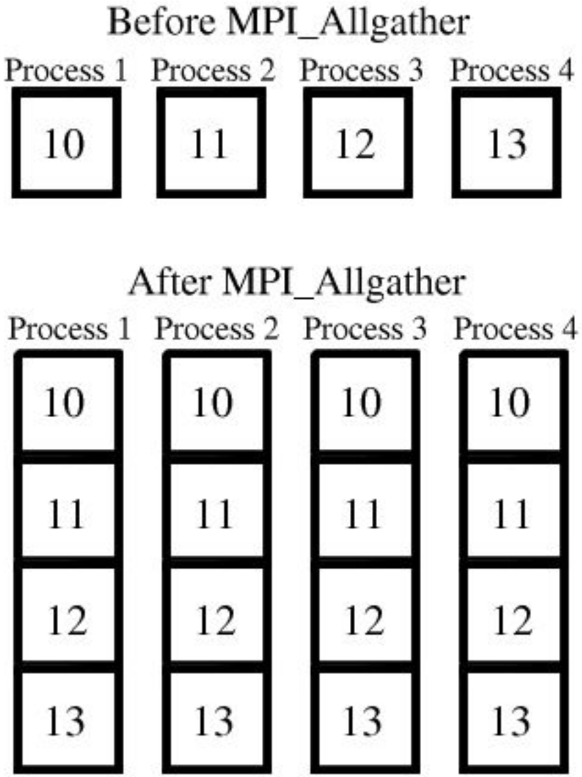
```
// compute multiplication
mult(N/P, N, recvMatrix, vector, localResult);
```



```
// Gather all results
MPI_Gather(localResult, N/P, MPI_INT, result, N/P, MPI_INT, 0, MPI_COMM_WORLD);

if (myrank == 0)
{
        printf("\nResult:\n");
        printVector(N, result);
}

MPI_Finalize();
```

Global communication: MPI_Allgather

This operation is equivalent to GATHER+BROADCAST but of course more efficient



Before MPI_Allgather

Process 1	Process 2	Process 3	Process 4
10	11	12	13

After MPI_Allgather

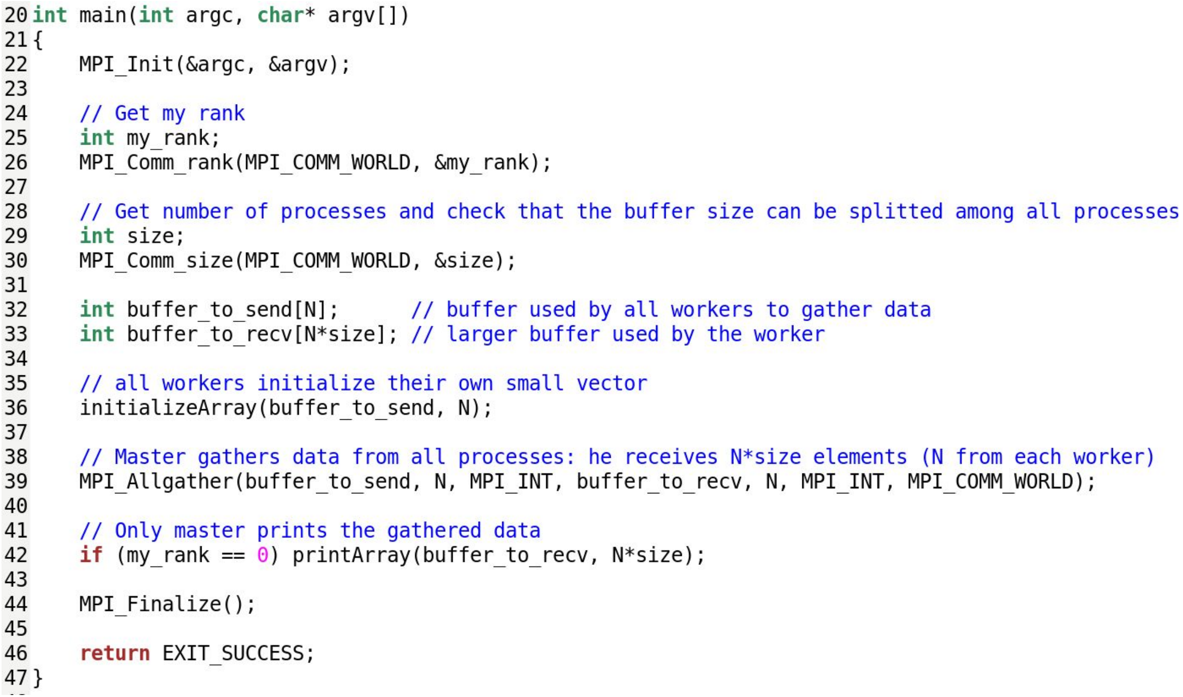
Process 1	Process 2	Process 3	Process 4
10	10	10	10
11	11	11	11
12	12	12	12
13	13	13	13

MPI_Allgather

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

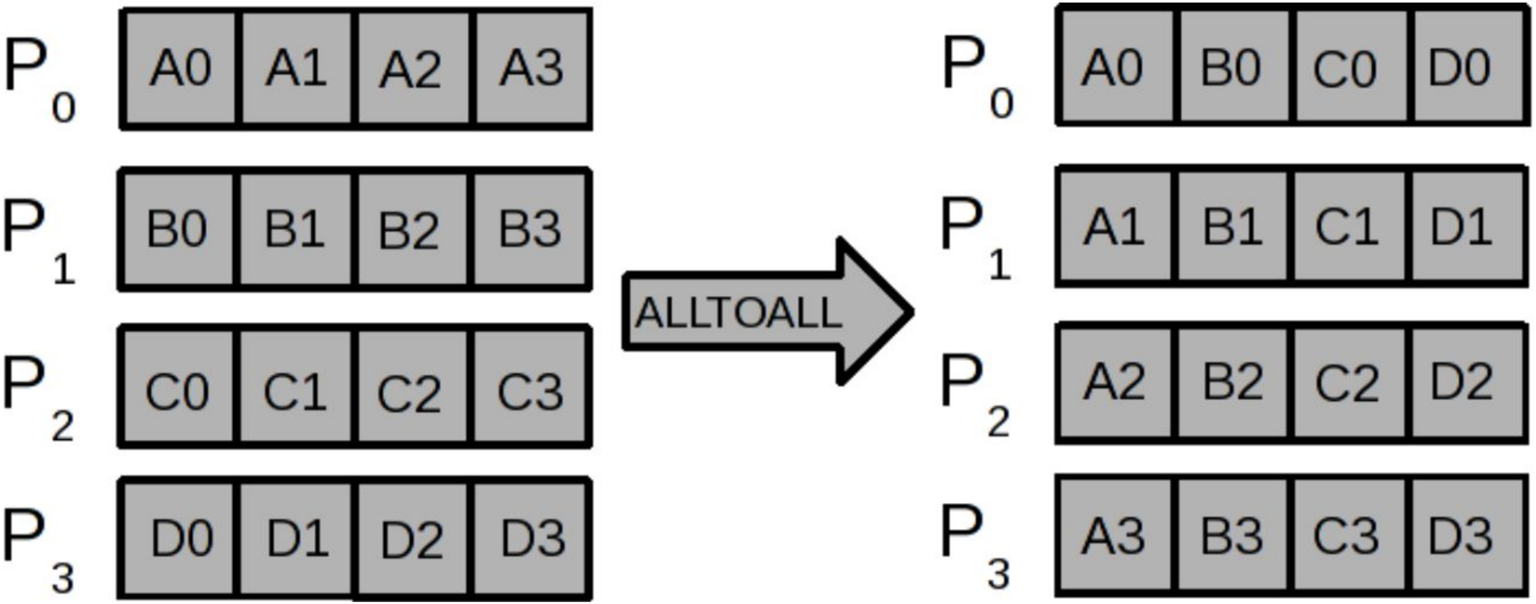
- sendbuf: address of send buffer
- sendcount: number of elements sent from each worker
- recvbuf: address of receive buffer
- recvcount: number of elements received from each worker

MPI_Allgather: example19.c



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

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get number of processes and check that the buffer size can be splitted among all processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int buffer_to_send[N];      // buffer used by all workers to gather data
    int buffer_to_recv[N*size]; // larger buffer used by the worker

    // all workers initialize their own small vector
    initializeArray(buffer_to_send, N);

    // Master gathers data from all processes: he receives N*size elements (N from each worker)
    MPI_Allgather(buffer_to_send, N, MPI_INT, buffer_to_recv, N, MPI_INT, MPI_COMM_WORLD);

    // Only master prints the gathered data
    if (my_rank == 0) printArray(buffer_to_recv, N*size);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Global communication: MPI_Alltoall



P_0	A0	A1	A2	A3
P_1	B0	B1	B2	B3
P_2	C0	C1	C2	C3
P_3	D0	D1	D2	D3

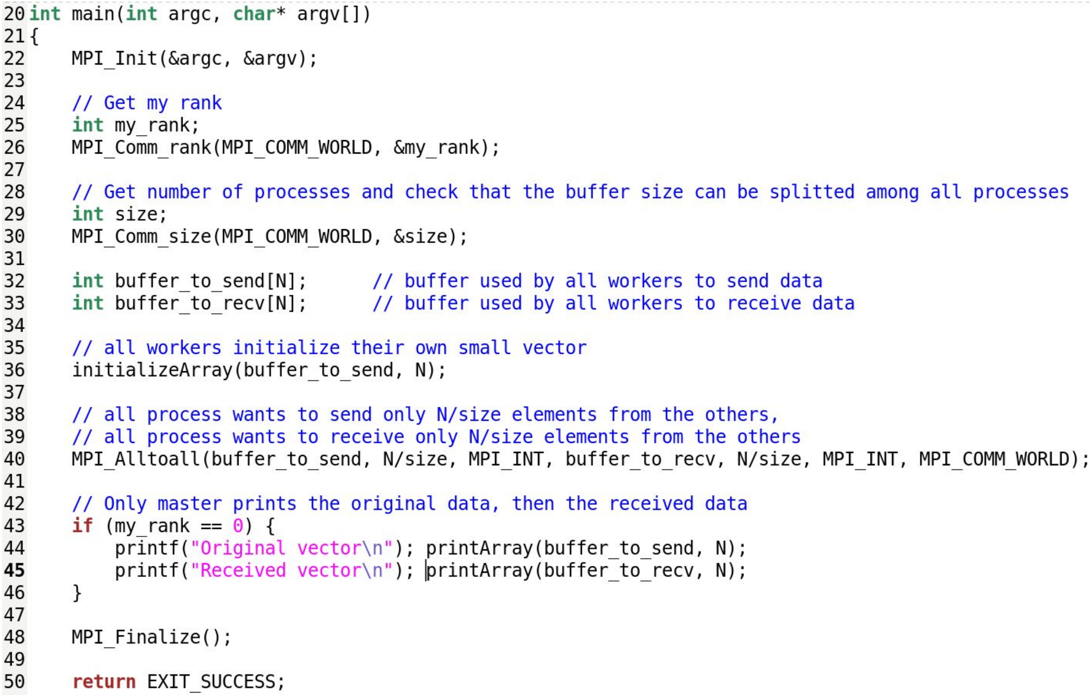
P_0	A0	B0	C0	D0
P_1	A1	B1	C1	D1
P_2	A2	B2	C2	D2
P_3	A3	B3	C3	D3

MPI_Alltoall

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

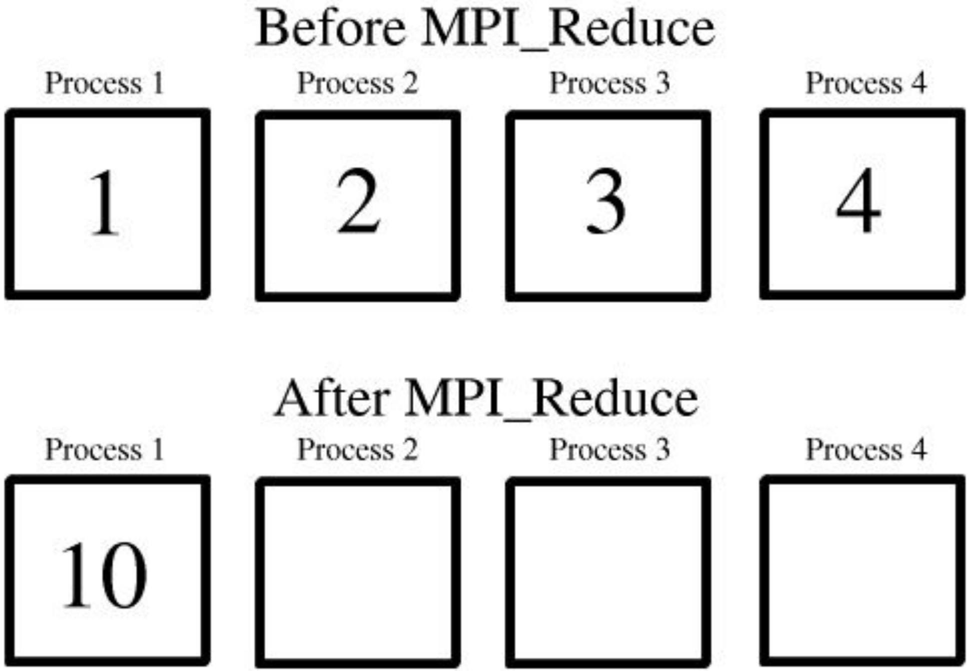
- sendbuf: address of send buffer (significant only for root)
- sendcount: number of elements to send to each process
- recvbuf: address of receive buffer
- recvcount: number of elements to receive from each process
- root: rank of process master (who sends data to each others)

MPI_Alltoall: example20.c



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

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get number of processes and check that the buffer size can be splitted among all processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int buffer_to_send[N];      // buffer used by all workers to send data
    int buffer_to_recv[N];      // buffer used by all workers to receive data

    // all workers initialize their own small vector
    initializeArray(buffer_to_send, N);

    // all process wants to send only N/size elements from the others,
    // all process wants to receive only N/size elements from the others
    MPI_Alltoall(buffer_to_send, N/size, MPI_INT, buffer_to_recv, N/size, MPI_INT, MPI_COMM_WORLD);

    // Only master prints the original data, then the received data
    if (my_rank == 0) {
        printf("Original vector\n"); printArray(buffer_to_send, N);
        printf("Received vector\n"); printArray(buffer_to_recv, N);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
```

Global reduction: MPI_Reduce



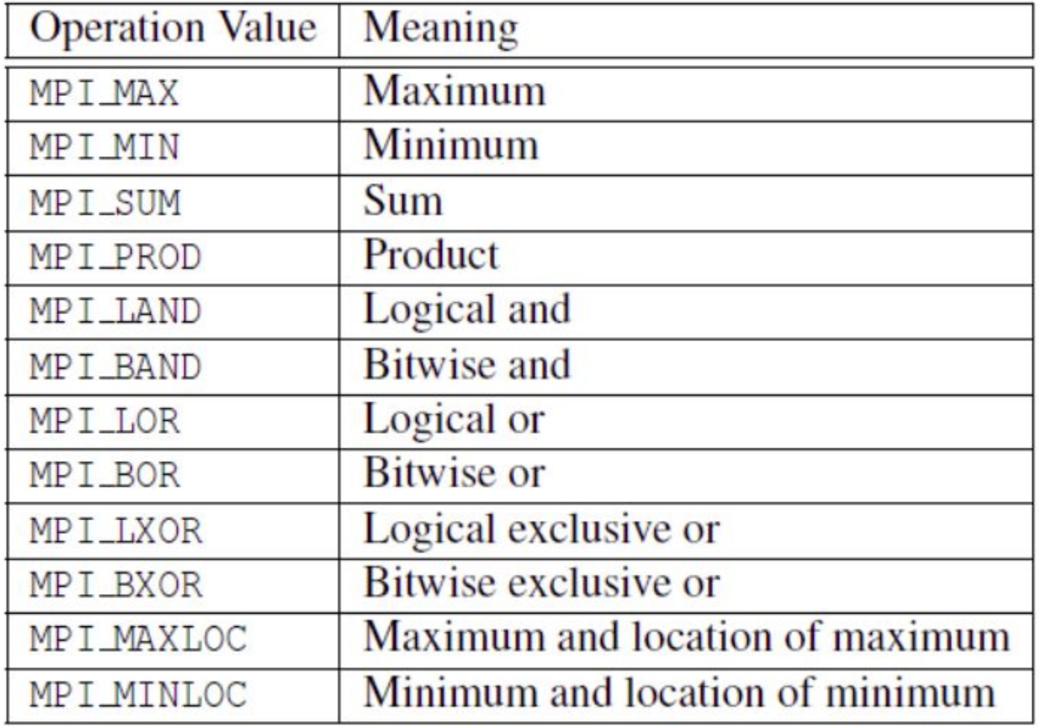
MPI_Reduce

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

- sendbuf: address of send buffer (significant only for root)
- recvbuf: address of receive buffer
- count: number of elements sent to each process
- op: reduction operation (*see later*)
- root: rank of process master (who sends data to each others)

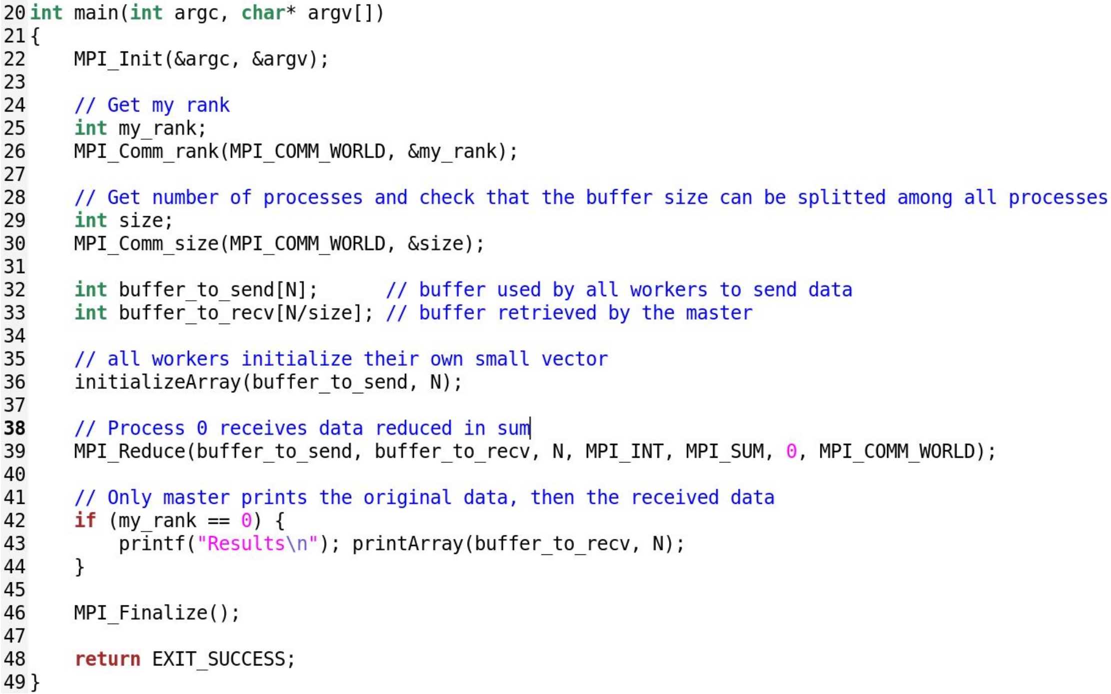
MPI_Reduce: predefined operations

User-defined operation can also be defined



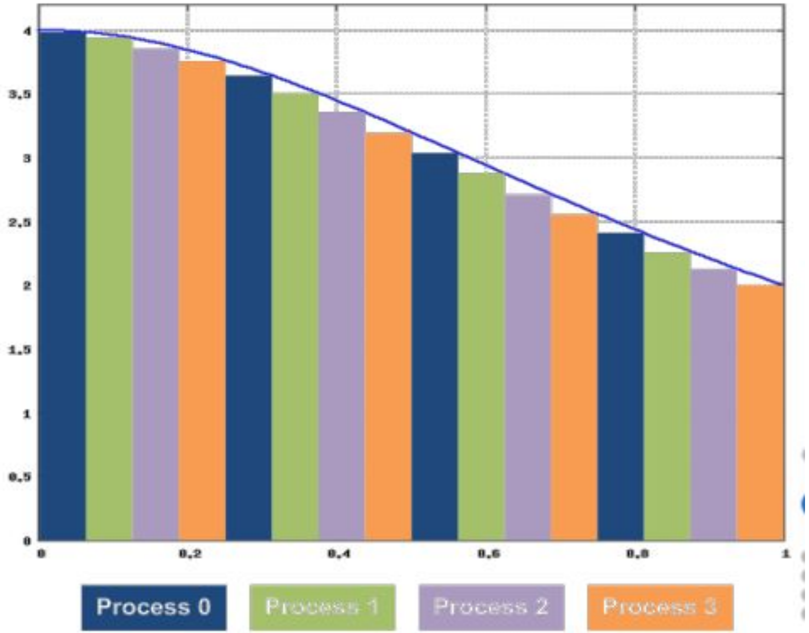
Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

MPI_Reduce: example21.c



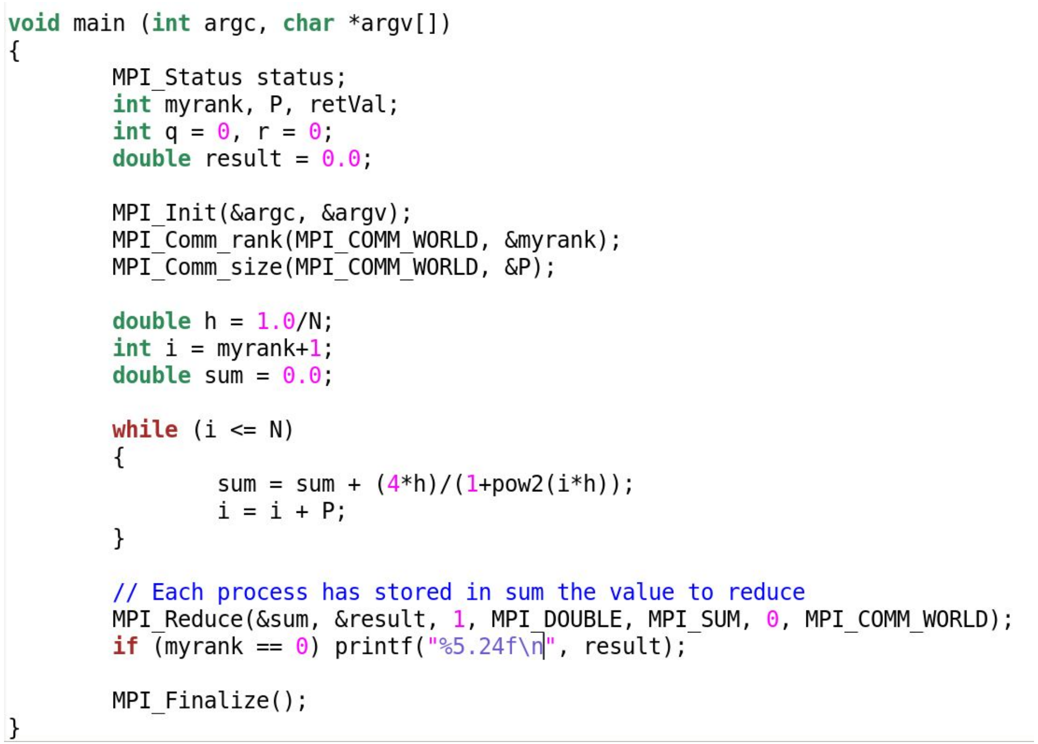
```c
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get number of processes and check that the buffer size can be splitted among all processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int buffer_to_send[N];      // buffer used by all workers to send data
    int buffer_to_recv[N/size]; // buffer retrieved by the master

    // all workers initialize their own small vector
    initializeArray(buffer_to_send, N);

    // Process 0 receives data reduced in sum
    MPI_Reduce(buffer_to_send, buffer_to_recv, N, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Only master prints the original data, then the received data
    if (my_rank == 0) {
        printf("Results\n"); printArray(buffer_to_recv, N);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Computing π in parallel (using MPI_Reduce): example11.c



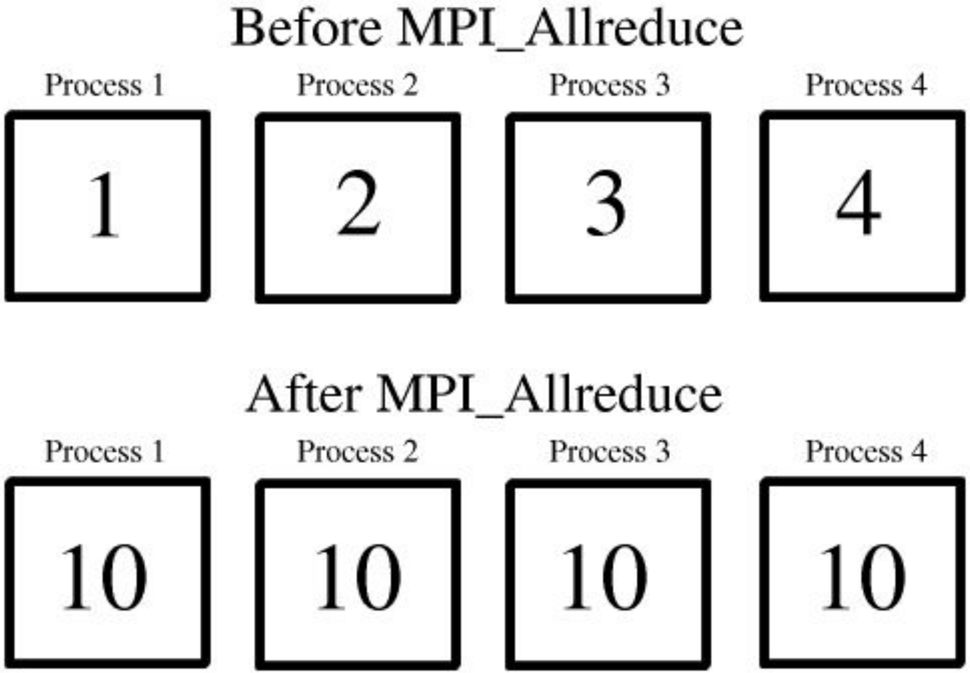
- The interval [0, 1] is split into ***N*** parts
- Each part is assigned to a process $\boldsymbol{p}_i$
- each $\boldsymbol{p}_i$ process works on its own sub-interval
- process $\boldsymbol{p}_0$ gathers all results and sum them up all together

Computing π in parallel (using MPI_Reduce): example11.c



```c
void main (int argc, char *argv[])
{
        MPI_Status status;
        int myrank, P, retVal;
        int q = 0, r = 0;
        double result = 0.0;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        MPI_Comm_size(MPI_COMM_WORLD, &P);

        double h = 1.0/N;
        int i = myrank+1;
        double sum = 0.0;

        while (i <= N)
        {
                sum = sum + (4*h)/(1+pow2(i*h));
                i = i + P;
        }

        // Each process has stored in sum the value to reduce
        MPI_Reduce(&sum, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if (myrank == 0) printf("%5.24f\n", result);

        MPI_Finalize();
}
```

Global reduction: MPI_Allreduce



Before MPI_Allreduce

Process 1	Process 2	Process 3	Process 4
1	2	3	4

After MPI_Allreduce

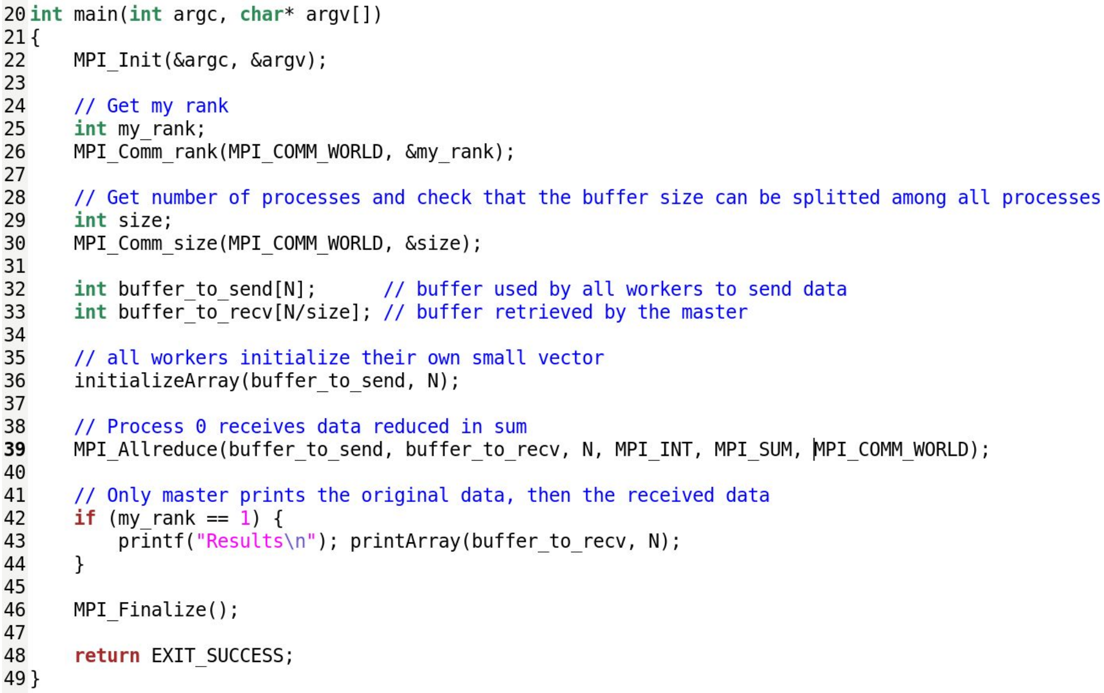
Process 1	Process 2	Process 3	Process 4
10	10	10	10

MPI_Allreduce

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

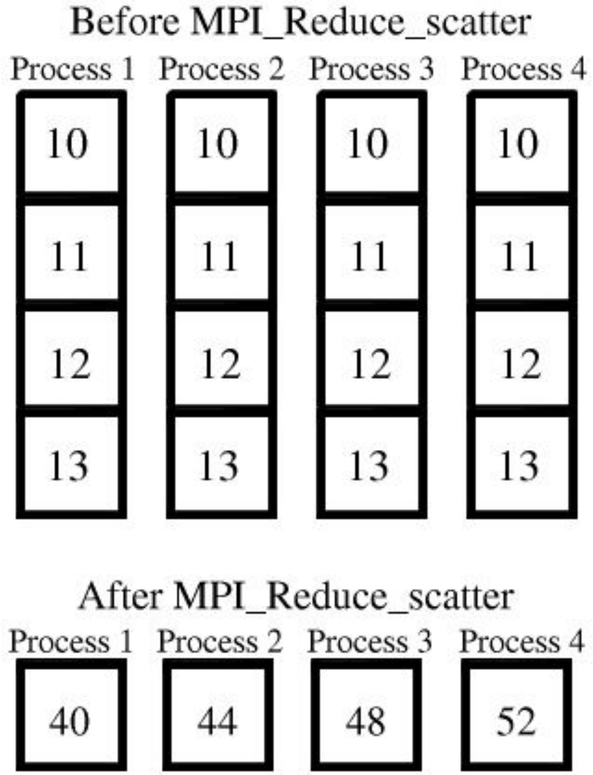
- sendbuf: address of send buffer (significant only for root)
- recvbuf: address of receive buffer
- count: number of elements sent to each process
- op: reduction operation

MPI_Allreduce: example22.c



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

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get number of processes and check that the buffer size can be splitted among all processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int buffer_to_send[N];      // buffer used by all workers to send data
    int buffer_to_recv[N/size]; // buffer retrieved by the master

    // all workers initialize their own small vector
    initializeArray(buffer_to_send, N);

    // Process 0 receives data reduced in sum
    MPI_Allreduce(buffer_to_send, buffer_to_recv, N, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    // Only master prints the original data, then the received data
    if (my_rank == 1) {
        printf("Results\n"); printArray(buffer_to_recv, N);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Global reduction: MPI_Reduce_scatter



Before MPI_Reduce_scatter

Process 1	Process 2	Process 3	Process 4
10	10	10	10
11	11	11	11
12	12	12	12
13	13	13	13

After MPI_Reduce_scatter

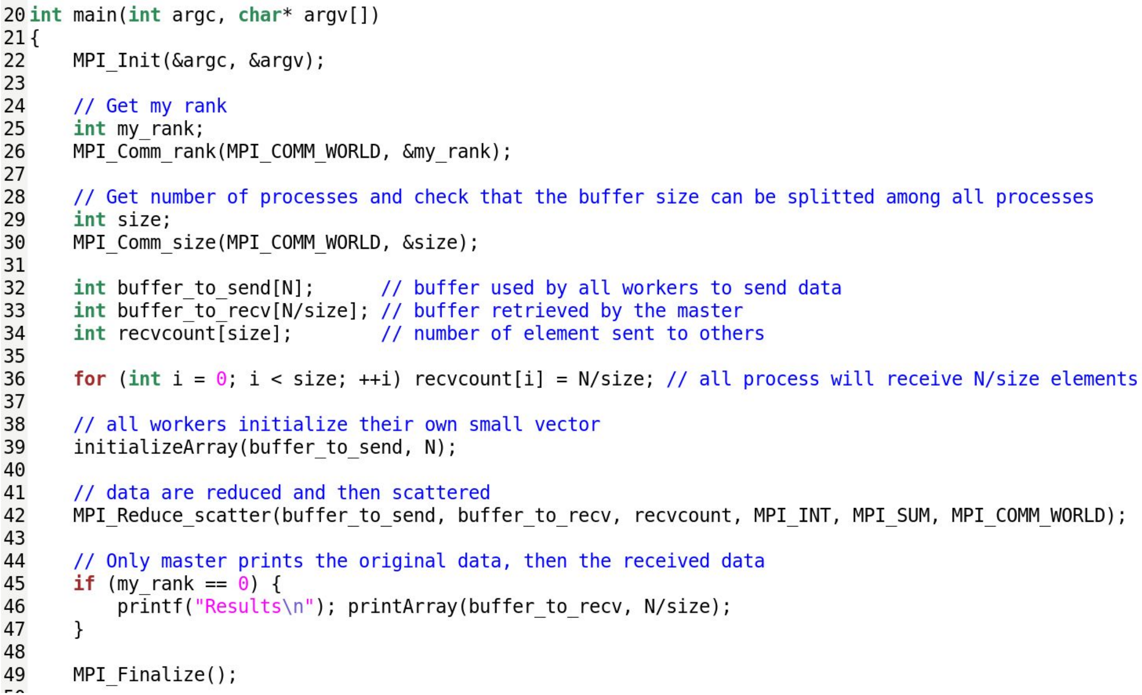
Process 1	Process 2	Process 3	Process 4
40	44	48	52

MPI_Reduce_scatter

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

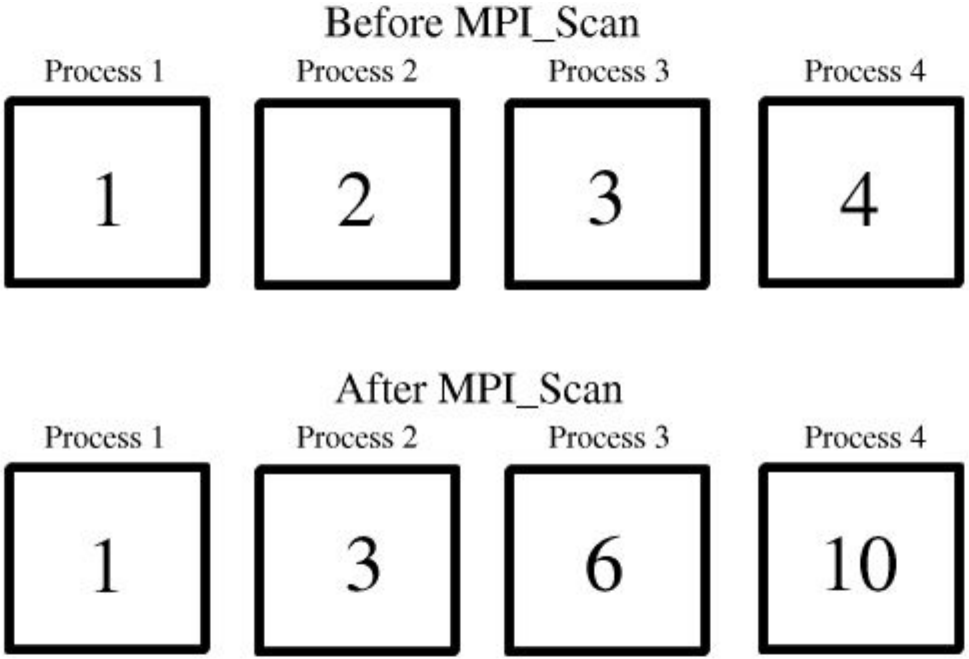
- sendbuf: address of send buffer (significant only for root)
- recvbuf: address of receive buffer
- count: integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
- op: reduction operation

MPI_Reduce_scatter: example23.c



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

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get number of processes and check that the buffer size can be splitted among all processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int buffer_to_send[N];      // buffer used by all workers to send data
    int buffer_to_recv[N/size]; // buffer retrieved by the master
    int recvcount[size];        // number of element sent to others

    for (int i = 0; i < size; ++i) recvcount[i] = N/size; // all process will receive N/size elements

    // all workers initialize their own small vector
    initializeArray(buffer_to_send, N);

    // data are reduced and then scattered
    MPI_Reduce_scatter(buffer_to_send, buffer_to_recv, recvcount, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    // Only master prints the original data, then the received data
    if (my_rank == 0) {
        printf("Results\n"); printArray(buffer_to_recv, N/size);
    }

    MPI_Finalize();
```

Global reduction: MPI_Scan



Before MPI_Scan

Process 1	Process 2	Process 3	Process 4
1	2	3	4

After MPI_Scan

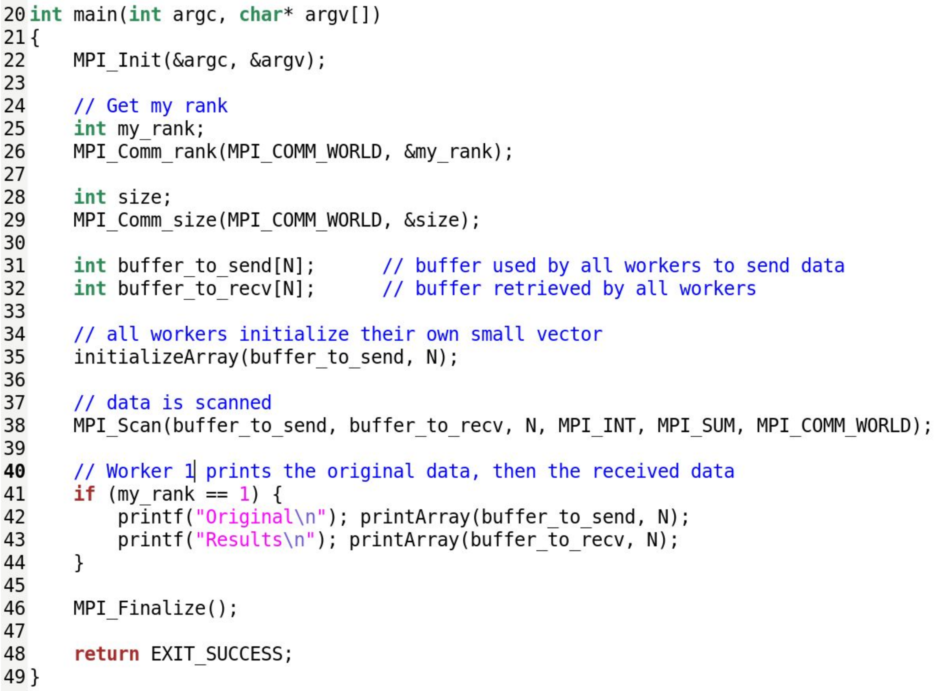
Process 1	Process 2	Process 3	Process 4
1	3	6	10

MPI_Scan

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

- sendbuf: address of send buffer (significant only for root)
- recvbuf: address of receive buffer
- count: number of elements in input buffer (integer).
- op: reduction operation
- comm: communicator

MPI_Scan: example24.c



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

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int buffer_to_send[N];      // buffer used by all workers to send data
    int buffer_to_recv[N];      // buffer retrieved by all workers

    // all workers initialize their own small vector
    initializeArray(buffer_to_send, N);

    // data is scanned
    MPI_Scan(buffer_to_send, buffer_to_recv, N, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    // Worker 1 prints the original data, then the received data
    if (my_rank == 1) {
        printf("Original\n"); printArray(buffer_to_send, N);
        printf("Results\n"); printArray(buffer_to_recv, N);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Immediate collective functions

- MPI_Ibcast
- MPI_Iscatter
- MPI_Igather
- MPI_Iallgather
- MPI_Ialltoall
- MPI_Ireduce
- MPI_Iallreduce
- MPI_Ireduce_scatter
- MPI_Iscan

Don’t forget to use MPI_Wait()

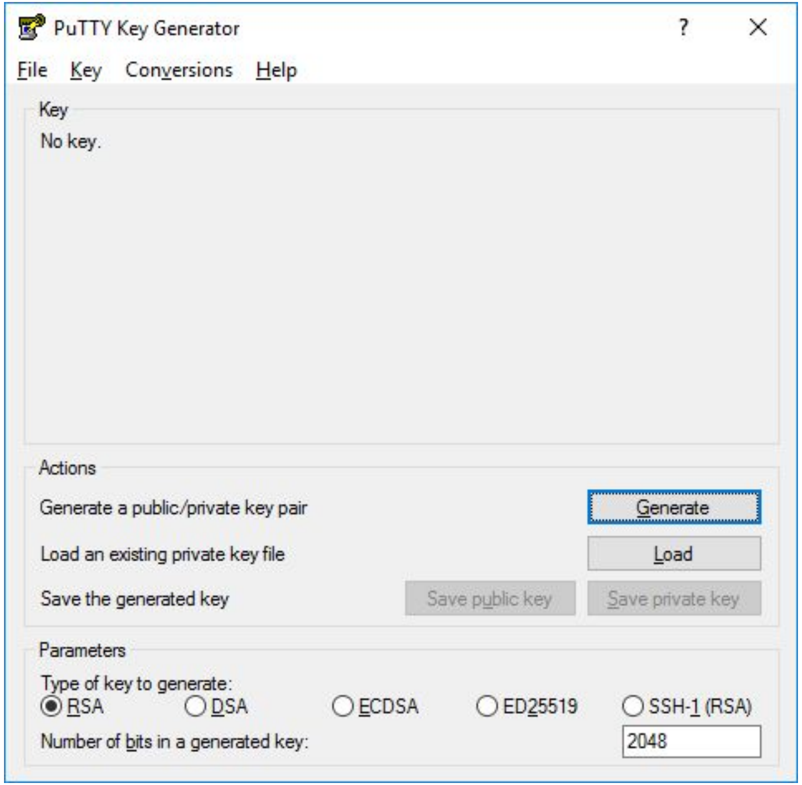
Building an MPI Cluster using Google Cloud Platform

Before starting: creating a SSH key (using Linux)

- mkdir myGoogleKey
- cd myGoogleKey
- ssh-keygen -t rsa -b 4096 -f ./id_rsa
  - The system will create the private key and ask for protecting it using a password. Leave empty for no password. If provided, don’t forget the password, it will be asked at login time
  - At the end, two files are created: id_rsa (the private key) and id_rsa.pub (the public key)
  - Keep safe both files as everybody could get access to your virtual instance

Before starting: creating a SSH key (using Win)

- download PuttyGen from https://www.puttygen.com/
- start the tool
- generate a RSA key
- save and keep safe public and private keys



Key

No key.

Actions

Generate a public/private key pair

Load an existing private key file

Save the generated key

Parameters

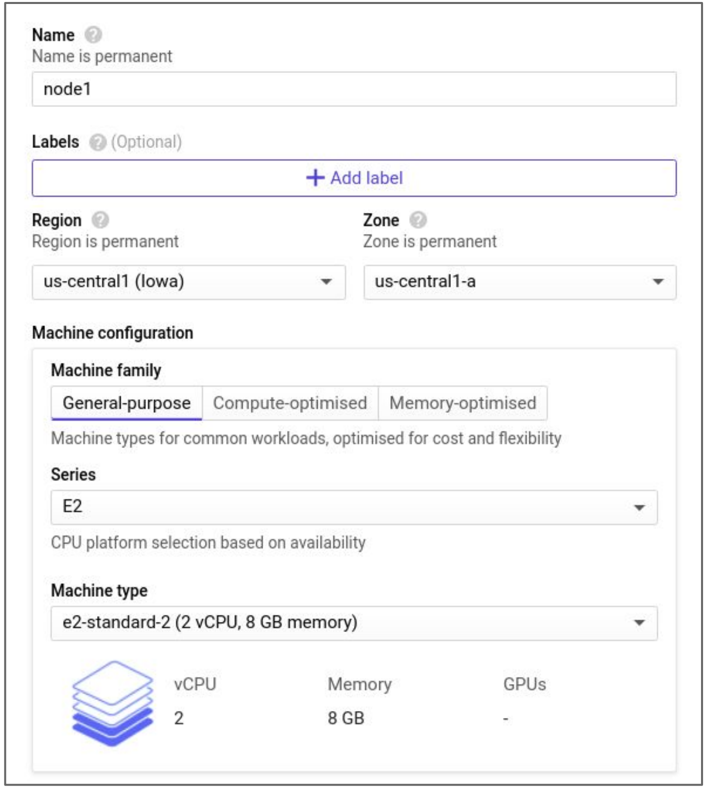
Type of key to generate:

● RSA ○ DSA ○ ECDSA ○ ED25519 ○ SSH-1 (RSA)

Number of bits in a generated key: 2048

Building a Virtual Instance (used as template)

- Log in https://console.cloud.google.com using your institutional email credentials
- Select Compute Engine > Virtual Instances
- Create a new instance having the following configuration:
  - name: node1
  - region: us-central1
  - cpu: 2
  - memory: 8GB



Name
Name is permanent
node1

Labels (Optional)
+ Add label

Region
Region is permanent
us-central1 (Iowa)

Zone
Zone is permanent
us-central1-a

Machine configuration

Machine family
General-purpose | Compute-optimised | Memory-optimised
Machine types for common workloads, optimised for cost and flexibility

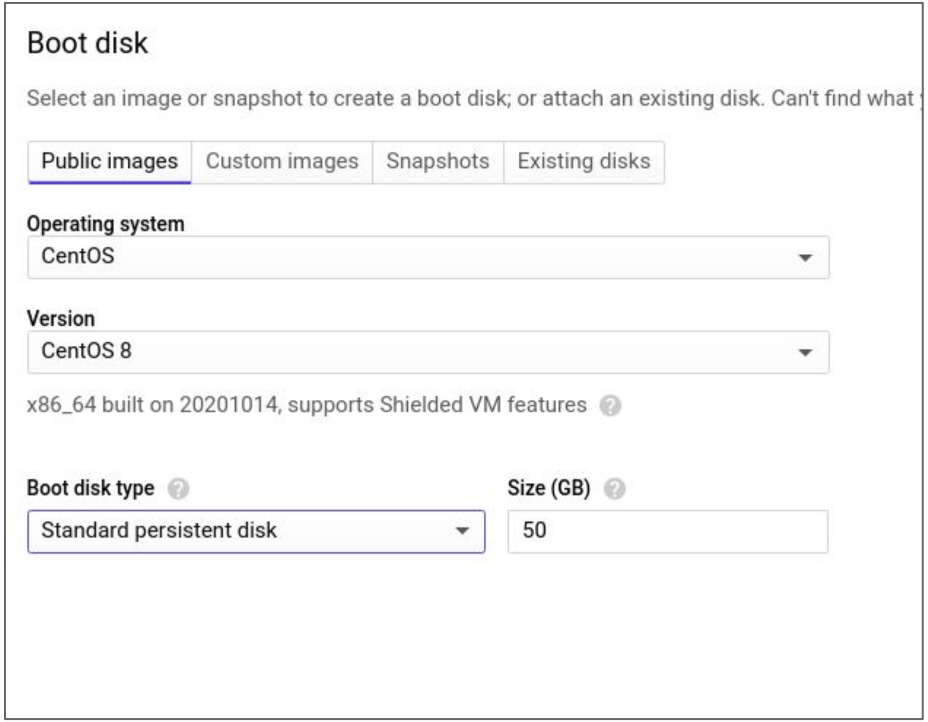
Series
E2
CPU platform selection based on availability

Machine type
e2-standard-2 (2 vCPU, 8 GB memory)

vCPU	Memory	GPUs
2	8 GB	-

Building a Virtual Instance (used as template)

- Create a new instance having the following configuration:
  - OS: centos
  - Version: 8
  - Boot Disk: Standard
  - Size: 50GB



Boot disk

Select an image or snapshot to create a boot disk; or attach an existing disk. Can't find what

Operating system: CentOS

Version: CentOS 8

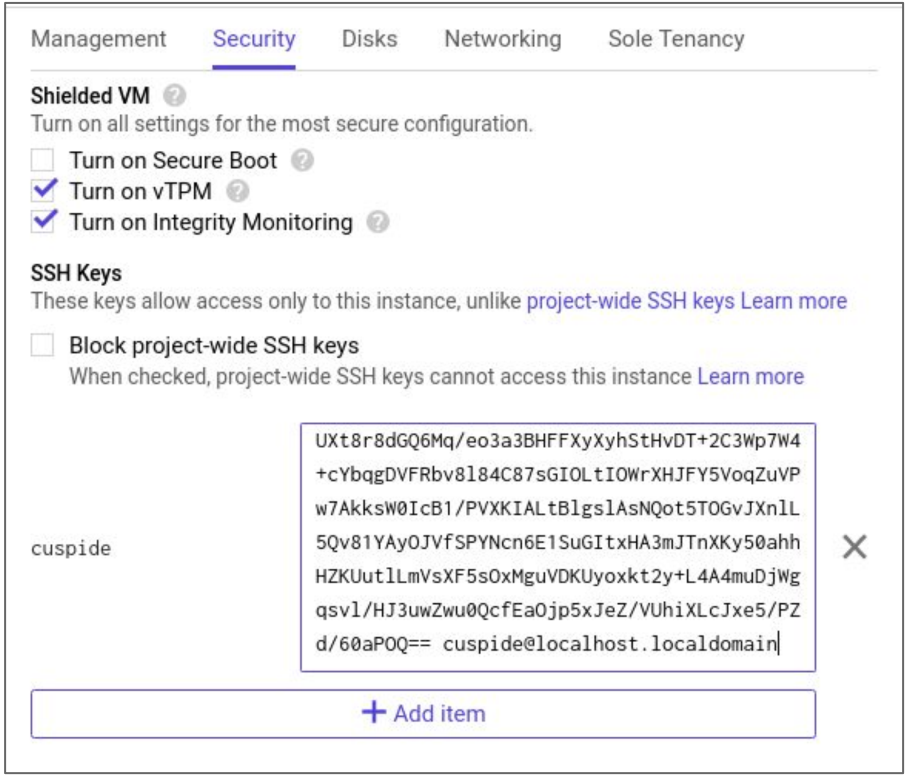
x86_64 built on 20201014, supports Shielded VM features

Boot disk type: Standard persistent disk

Size (GB): 50

Building a Virtual Instance (used as template)

- Using a text editor, open the public key created before (id_rsa.pub), copy the content and paste it into the right field (Security Tab)
- Take a look at the username assigned to the key (which is the same username who created the key)
- Let’s select the Create button to build the virtual instance.
- The VI is started up straightaway.



Shielded VM

Turn on all settings for the most secure configuration.

□ Turn on Secure Boot

☑ Turn on vTPM

☑ Turn on Integrity Monitoring

SSH Keys

These keys allow access only to this instance, unlike project-wide SSH keys Learn more

□ Block project-wide SSH keys

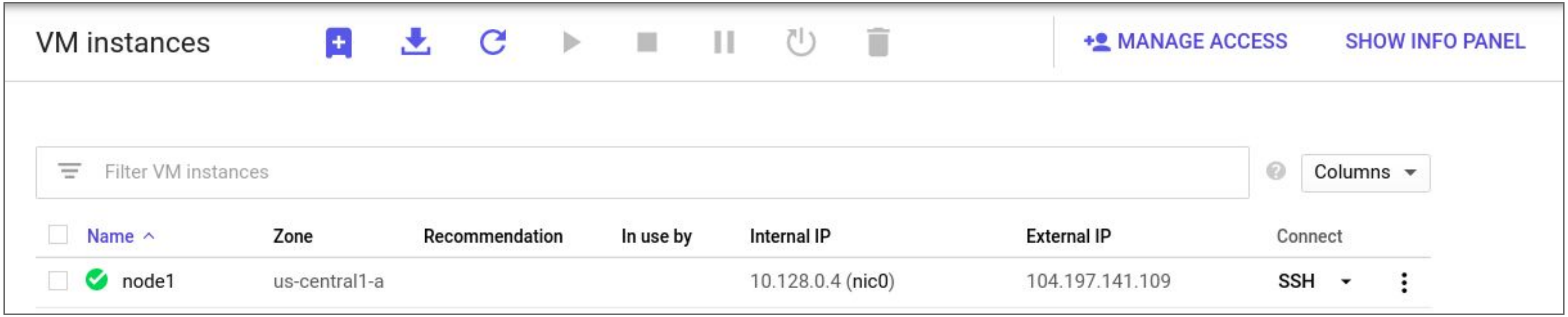
When checked, project-wide SSH keys cannot access this instance Learn more

cuspide

```
UXt8r8dGQ6Mq/eo3a3BHFFXyXyhStHvDT+2C3Wp7W4
+cYbqgDVFRbv8l84C87sGIOLtIOWrXHJFY5VoqZuVP
w7AkksW0IcB1/PVXKIALtBlgslAsNQot5TOGvJXnlL
5Qv81YAyOJVfSPYNcn6E1SuGItxHA3mJTnXKy50ahh
HZKUutlLmVsXF5sOxMguVDKUyoxkt2y+L4A4muDjWg
qsvl/HJ3uwZwu0QcfEaOjp5xJeZ/VUhiXLcJxe5/PZ
d/60aPOQ== cuspide@localhost.localdomain
```

Getting an access to the virtual instance

- Using the Dashboard, let’s take a look to the virtual instance. The green button means it is running
- The Virtual Instance is assigned to an external IP. Take note of that and keep in mind that it is going to stay the same as long as the virtual instance is left running. After that, the address might change



	Name	Zone	Recommendation	In use by	Internal IP	External IP	Connect
□	✓ node1	us-central1-a			10.128.0.4 (nic0)	104.197.141.109	SSH

Getting an access to the virtual instance

- Using your shell, run the following command:
  - ssh -l cuspide -i ./id_rsa 104.197.141.109
- Where:
  - cuspide: is the username showed in the security section
  - id_rsa: is the name of the private key created at the beginning
  - 104.197.141.109: the is virtual instance IP address showed by the dashbord
- If everything went well, you are inside your remote virtual instance. You can see that the prompt is different as it is something similar to cuspide@node1

Download and install OpenMPI

- sudo su
- yum install wget
- yum install perl
- yum install gcc
- yum install gcc-c++
- mkdir /usr/local/openMPI
- cd ~
- mkdir openMPI
- wget https://download.open-mpi.org/release/open-mpi/v4.1/openmpi-4.1.1.tar.gz
    - Please, verify before downloading if a new release is available
- tar -xvzf openmpi-4.1.1.tar.gz

Download and install OpenMPI

- cd openmpi-4.1.1
- mkdir build
- ../configure --prefix=/usr/local/openMPI
- make all install
- exit (getting back to the non-admin user)
- vi ~/.bashrc
  - export PATH=$PATH:/usr/local/openMPI/bin

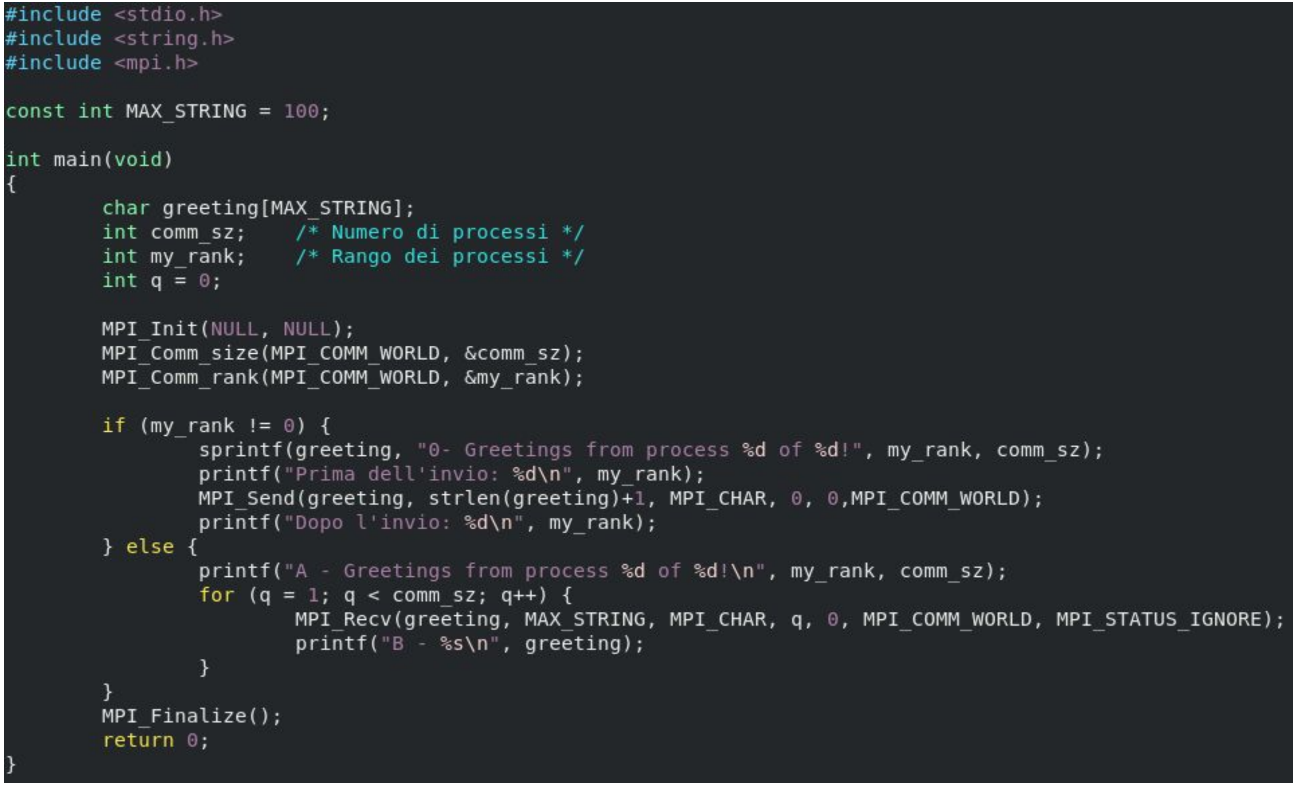
Copy the key pair

Copy on each virtual instance the key pair you created at the beginning:

```
scp -i id_rsa id_rsa* cuspide@104.197.141.109:/home/cuspide/.ssh
```

Be careful: the command should be run for each virtual instance changing properly username, IP address and home directory

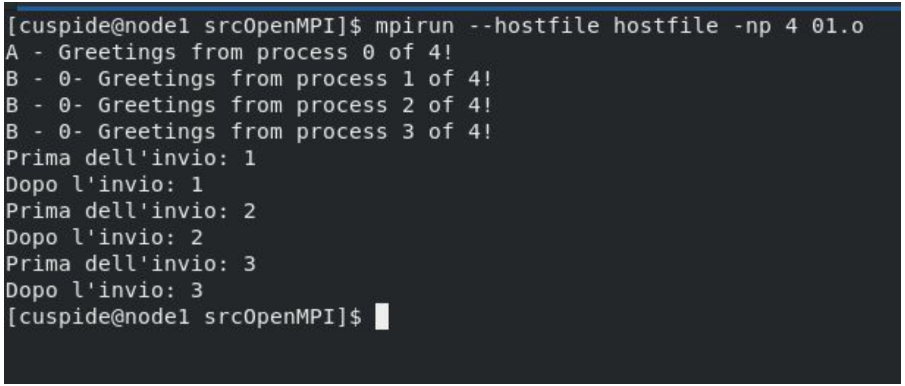
The first program



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

const int MAX_STRING = 100;

int main(void)
{
        char greeting[MAX_STRING];
        int comm_sz;    /* Numero di processi */
        int my_rank;    /* Rango dei processi */
        int q = 0;

        MPI_Init(NULL, NULL);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        if (my_rank != 0) {
                sprintf(greeting, "0- Greetings from process %d of %d!", my_rank, comm_sz);
                printf("Prima dell'invio: %d\n", my_rank);
                MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,MPI_COMM_WORLD);
                printf("Dopo l'invio: %d\n", my_rank);
        } else {
                printf("A - Greetings from process %d of %d!\n", my_rank, comm_sz);
                for (q = 1; q < comm_sz; q++) {
                        MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        printf("B - %s\n", greeting);
                }
        }
        MPI_Finalize();
        return 0;
}
```

Compiling and running the first application

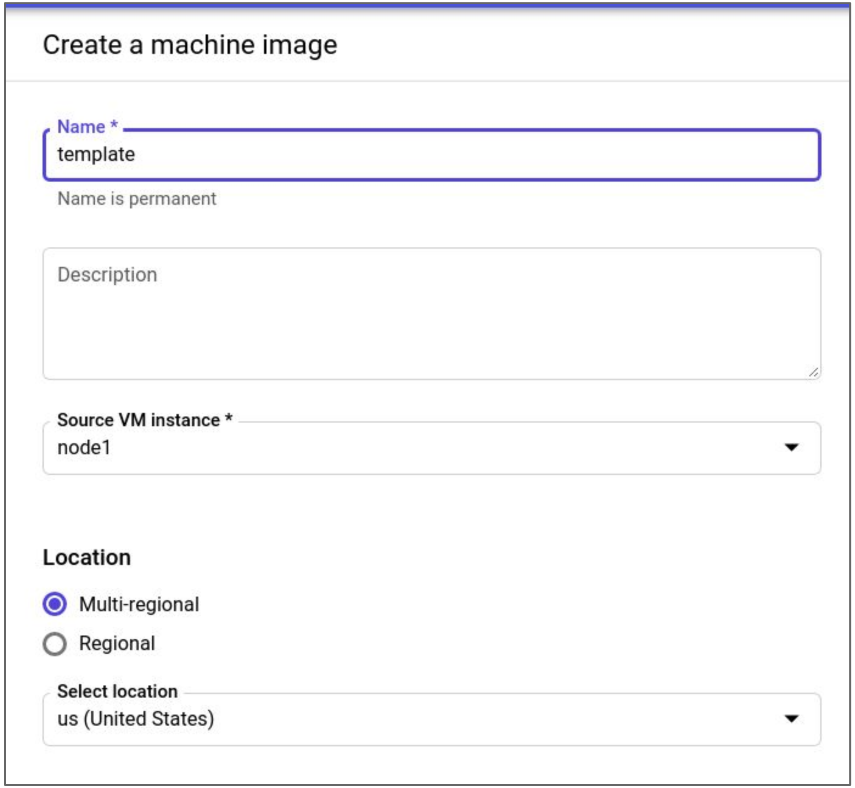
- vi hostfile
  - localhost slots=4
- mpicc 01.c -o 01.o
- mpirun --hostfile hostfile -np 4 01.o



```
[cuspide@node1 srcOpenMPI]$ mpirun --hostfile hostfile -np 4 01.o
A - Greetings from process 0 of 4!
B - 0- Greetings from process 1 of 4!
B - 0- Greetings from process 2 of 4!
B - 0- Greetings from process 3 of 4!
Prima dell'invio: 1
Dopo l'invio: 1
Prima dell'invio: 2
Dopo l'invio: 2
Prima dell'invio: 3
Dopo l'invio: 3
[cuspide@node1 srcOpenMPI]$ 
```

Create the cluster

- Stop the running virtual instance
- Select and Open the Virtual Instance
- Click on “Create Machine Image” button
- Set “template” as name
- Create the image



Create a machine image

Name *
template

Name is permanent

Description

Source VM instance *
node1

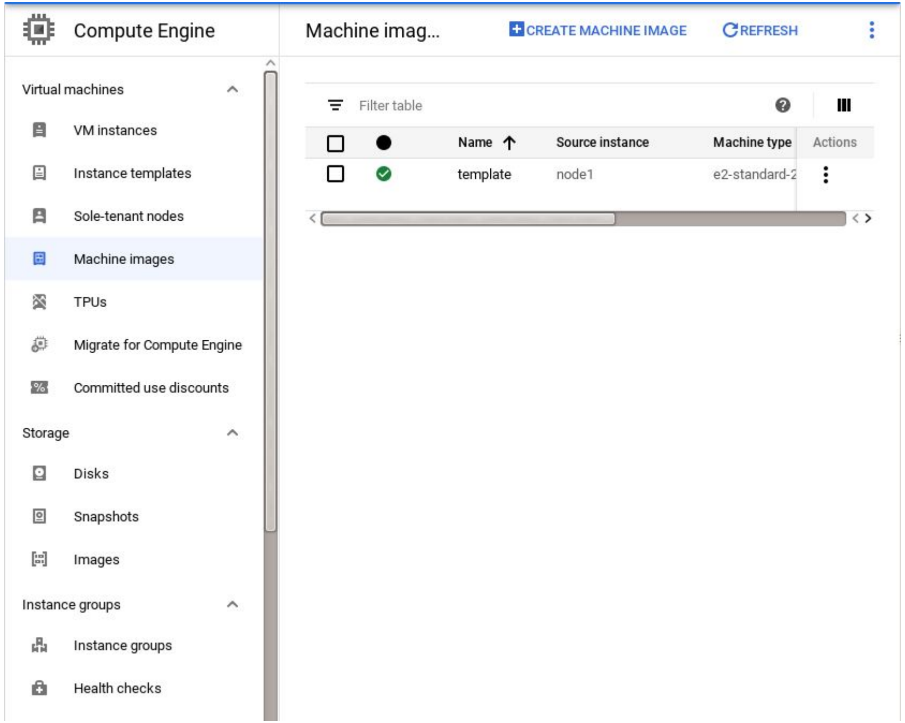
Location

● Multi-regional
○ Regional

Select location
us (United States)

Create the cluster

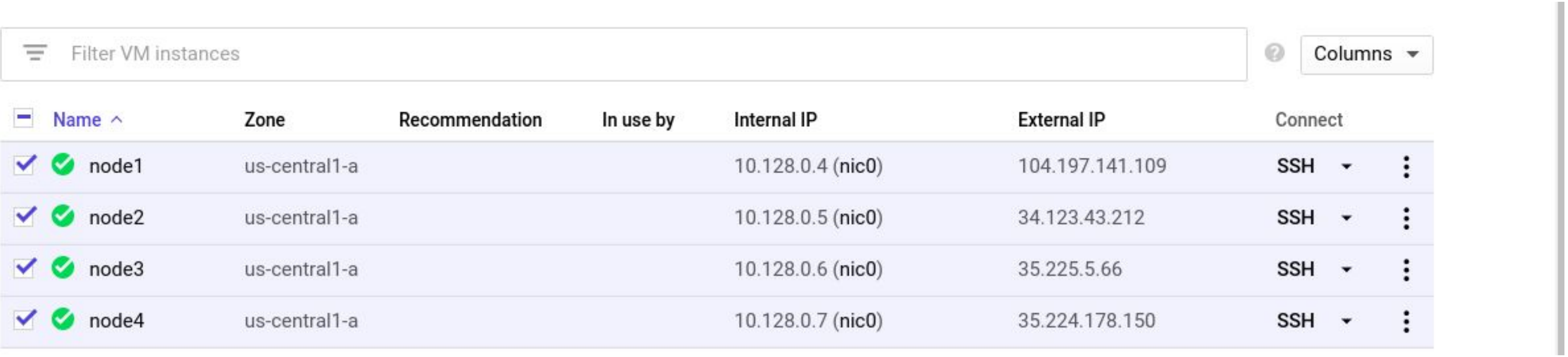
- From Compute Engine > Machine images, select the template called as “template” and select “Create instance”
- Set the new instance name as node2
- Do the same for node2, node3 and node4



□	●	Name ↑	Source instance	Machine type	Actions
□	✓	template	node1	e2-standard-2	⋮

Create the cluster

Start all nodes and note that each Virtual Instance has got its own external IP as well as the Internal IP. This last one will be used to connect the virtual instance to each others



	Name	Zone	Recommendation	In use by	Internal IP	External IP	Connect
☑	✓ node1	us-central1-a			10.128.0.4 (nic0)	104.197.141.109	SSH
☑	✓ node2	us-central1-a			10.128.0.5 (nic0)	34.123.43.212	SSH
☑	✓ node3	us-central1-a			10.128.0.6 (nic0)	35.225.5.66	SSH
☑	✓ node4	us-central1-a			10.128.0.7 (nic0)	35.224.178.150	SSH

Try the cluster interconnection

- Get an access to the first node (node1):
  - ssh -l cuspide -i ./id_rsa 104.197.141.109
- Try to connect using ssh to all other virtual instances using private network:
  - ssh 10.128.0.4
  - ssh 10.128.0.5
  - ssh 10.128.0.6
  - ssh 10.128.0.7
- Modify the hostfile
  - 10.128.0.4 slots=2
  - 10.128.0.5 slots=2
  - 10.128.0.6 slots=2
  - 10.128.0.7 slots=2
- Run the application again
  - mpirun --hostfile hostfile -np 8 01.o