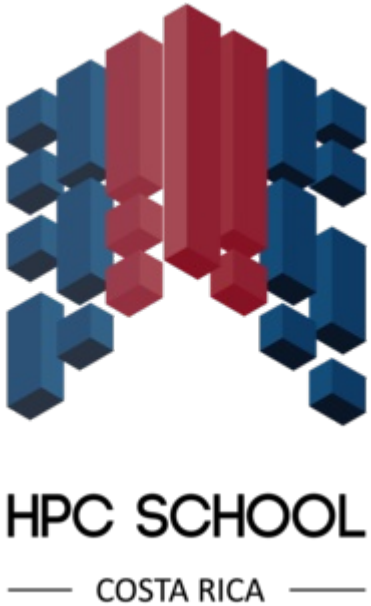
Costa Rica HPC School 2022

MPI Programming



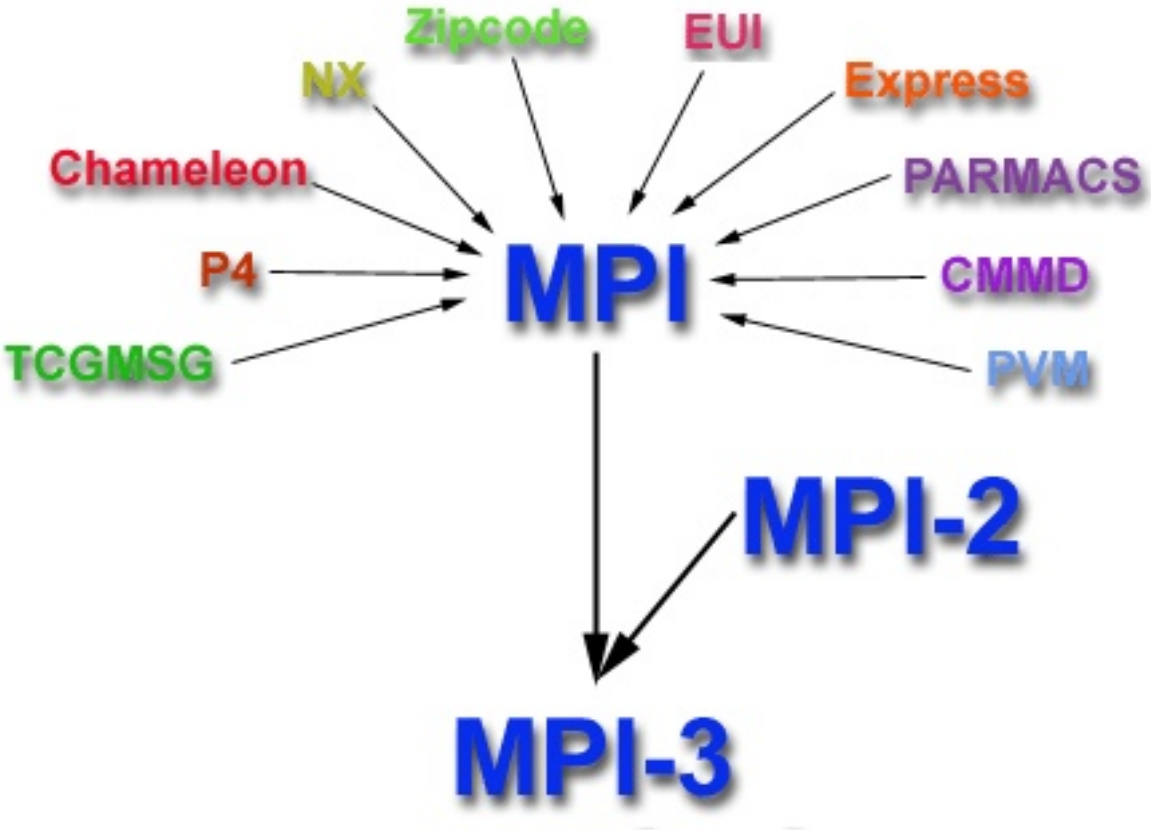
Gabriel P. Silva

MPI

What is MPI?

- It is a portable messaging standard that eases the development of parallel applications.
- It uses the message passing parallel programming paradigm and it can be used with clusters or networks of workstations.
- It is a library of functions to be used with programs written in C, C++ or Fortran.
- MPI was heavily influenced by work at the IBM T.J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, and PARMACS. Other important contributions also came from Zipcode, Chimp, PVM, Chameleon and PICL.

MPI History



https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

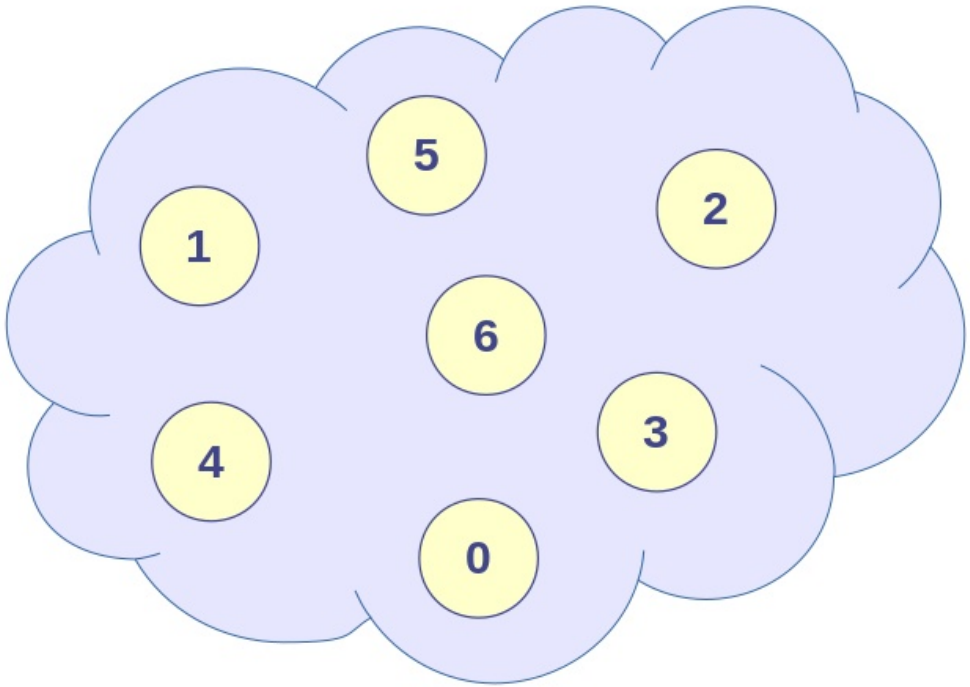
Which are MPI objectives?

- One of the goals of the MPI is to offer the possibility of an efficient implementation of communication:
  - Avoiding memory-to-memory copies;
  - Allowing superposition of communication and computation.
- Allow its use in heterogeneous environments.
- The communication interface is assumed to be reliable:
  - Communication failures must be handled by the platform's communication subsystem.

What is a communicator?

- The MPI library works with the concept of communicators to define the universe of processes involved in a communication operation, using group and context attributes:
  - Two processes belonging to the same group and using the same context can communicate directly.
  - The default communicator is named MPI_COMM_WORLD and contains all the processes that are started when the program is first executed.
  - Each process has a unique identifier called a rank, which ranges from 0 to P-1, where P is the number of processes in the communicator.

Communicator as a cloud



Initializing MPI

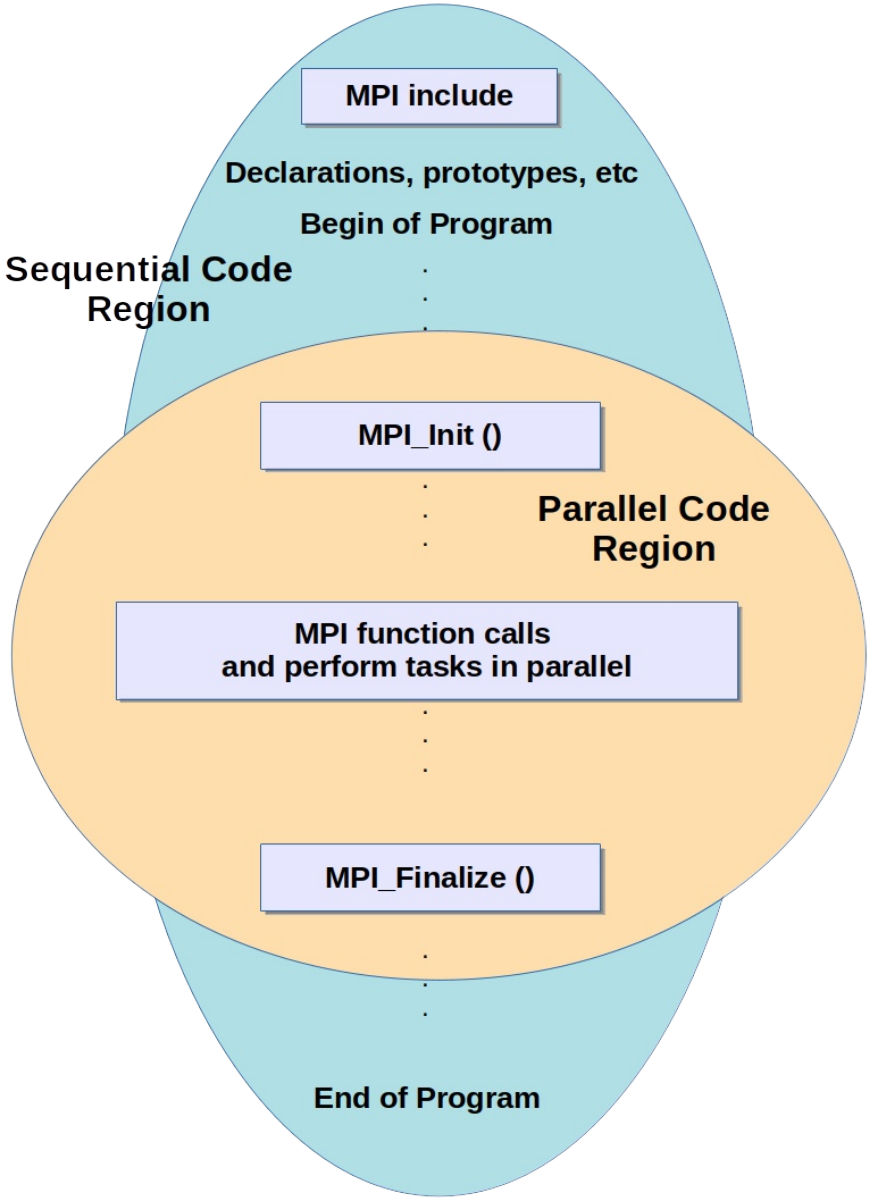
- Every MPI program must contain one of the following preprocessor directives:

  #include “mpi.h”

  #include <mpi.h>
- This file, mpi.h, contains the definitions, macros, and function prototypes needed to compile an MPI program.
- Before any other MPI function is called, the MPI_Init function must be called at least once.
- Its arguments are pointers to the main program parameters, argc and argv.
- This function allows the system to perform the necessary preparations for the MPI library to be used.

Finalizing MPI

- At the end of the program, the MPI_Finalize function must be called.
- This function clears any backlog left by MPI, eg., pending receptions that were never completed.
- Typically, an MPI program might have the following layout:


MPI include
Declarations, prototypes, etc
Begin of Program
Sequential Code Region
MPI_Init ()
Parallel Code Region
MPI function calls
and perform tasks in parallel
MPI_Finalize ()
End of Program

Initializing MPI

```
. . .
#include "mpi.h"
. . .
main(int argc, char** argv) {
. . .
/* No MPI functions can be called before this point  */
MPI_Init(&argc, &argv);
. . .
MPI_Finalize();
/* No MPI functions can be called after this point*/
. . .
/* main */
. . .
```

Checking if MPI has been initialized

- In some situations it may be necessary to check whether the MPI_Init and MPI_Finalize functions have already been called.
- The MPI_Initialized routine indicates whether the MPI_Init function was called, returning a logical value of true (1) or false (0).
- The MPI_Finalized routine indicates whether the MPI_Finalize function has been called.

```
int MPI_Initialized (int *flag)

int MPI_Finalized (int *flag)
```

MPI_Initialized

```
#include "mpi.h"
int main(int argc, char *argv[ ]) {
int initiaded, finalized;
...
    MPI_Initialized(&initiaded);
    if (!initiated)
        MPI_Init(&argc, &argv);
/* Do the work in parallel */
...
/* When the program is about to end */
    MPI_Finalized(&finalized);
    if (!finalized)
        MPI_Finalize();
    return(0); }
```

Who am I?

- The MPI_Comm_Rank function returns the rank of a process in its second argument.
- Its syntax is:

```
int MPI_Comm_Rank(MPI_Comm com, int *rank)
```

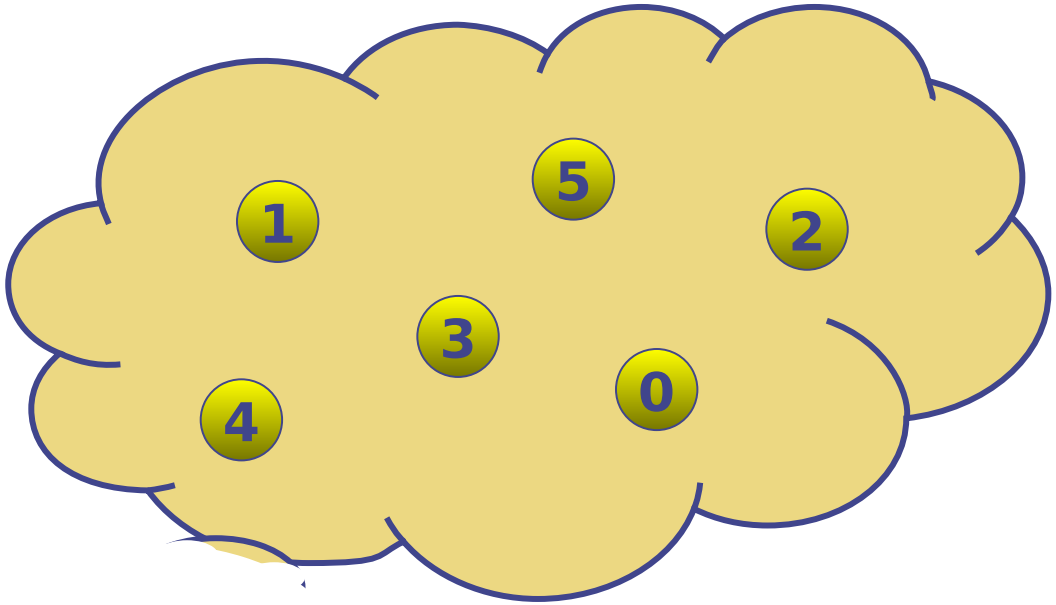
- The first argument is a communicator to which the process belongs.
- For basic programs, we will use the predefined communicator MPI_COMM_WORLD.

How many processes are out there?

- Many constructs in our programs also depend on the number of processes running the program.
- MPI provides the MPI_Comm_size function to determine this value.
- This function returns the number of processes in a communicator in its second argument.
- Its syntax is:

```
int MPI_Comm_size(MPI_Comm com, int *num_procs)
```

MPI_COMM_WORLD



Some useful functions

- Aborting a program:

  int MPI_Abort(MPI_Comm com, int error)

- Identifying the MPI version:

  int MPI_Get_version(int *version, int *subversion)

- Recovering computer's name:

  int MPI_Get_processor_name (char *name, int *lenght)

Measuring execution time

- The MPI_Wtime function returns the total elapsed clock time in seconds (double precision) since a given moment in the past, which depends on the implementation, but must always be the same for a given one.
- The MPI_Wtick function returns the resolution in seconds (in double precision) of the function MPI_Wtime.

Basic functions example

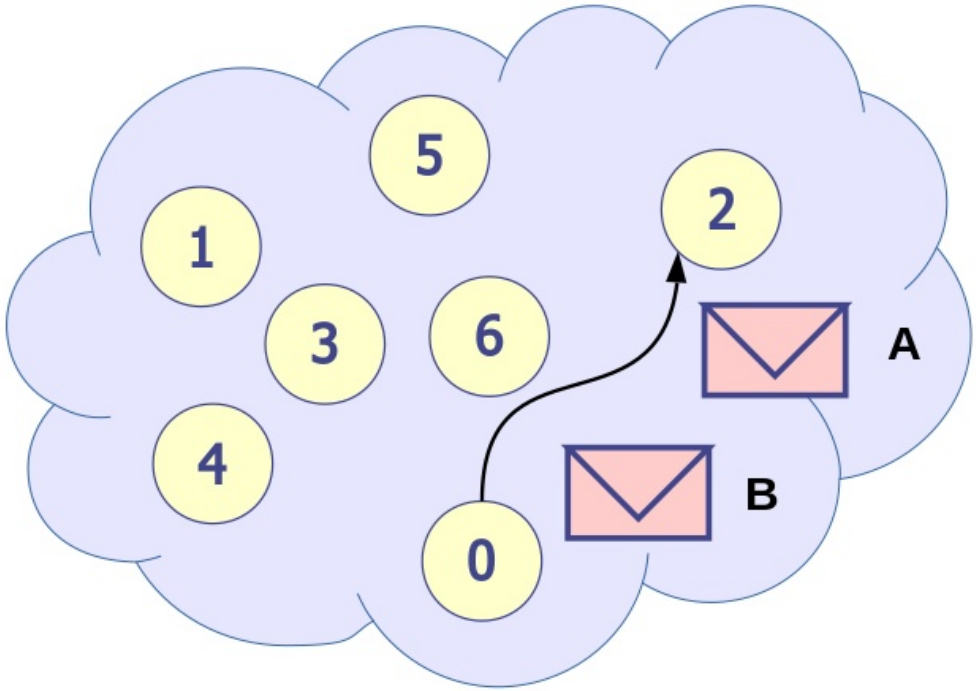
https://github.com/gpsilva2003/MPI/blob/main/src/mpi_funcoes.c

Point-to-Point Communication

What is a MPI message?

- Message = Payload + Envelope
- For the message to be communicated successfully, the system must attach some information to the payload.
- This additional information is the message envelope, which contains the following information:
  - The rank of the source process.
  - The rank of the destination process.
  - A label specifying the message type.
  - A communicator defining the communication domain.

How messages are ordered?



How messages are ordered?

- The messages do not overtake each other.
- For example, if the process with rank 0 sends two successive messages A and B and the rank 2 process calls two receive routines that match any of the messages, the order of messages is preserved, as A will always be received before B.
- If a process has a single thread of execution, then any two communications executed by this process are ordered.
- On the other hand, if the process is multi-threaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads.

How two processes can communicate to each other?

- At first, they must use the same communicator.
- There is a pre-defined communicator named MPI_COMM_WORLD, that includes all active processes since the program execution initiated.
- Second, they must exchange messages using MPI functions MPI_Send and MPI_Recv to send and receive messages, respectively.
- The first one sends the message to a given process and the second one receives the message from a process.
- Both are blocking, that means: blocking send waits until all data has been copied from the send buffers. Blocking reception waits until the receive buffer contains the entire message.

Which types of data are allowed?

- Correspondence between MPI and C types:

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED CHAR	unsigned char
MPI_UNSIGNED SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI_BYTE and MPI_PACKED

- The last two types, MPI_BYTE and MPI_PACKED do not correspond to any standard types in C.
- The MPI_BYTE type can be used if you do not want to perform any conversions between different data types.
- The MPI_PACKED type will be discussed later (?).

MPI_Send

int MPI_Send(void* message, int count, MPI_Datatype mpi_type, int dest, int tag, MPI_Comm com)

- message: initial address of the data to be sent.
- count: number of data.
- mpi_type: MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_BYTE, MPI_LONG, MPI_UNSIGNED_CHAR, etc.
- dest: rank of the destination process.
- tag: message tag.
- com: communicator that specifies the context of the communication and the processes participating in the group. The default communicator is MPI_COMM_WORLD.

MPI_Recv

int MPI_Recv(void* message, int count, MPI_Datatype mpi_type, int source, int tag, MPI_Comm com, MPI_Status* status)

- message: Receive buffer start address
- count: Maximum number of data to be received
- mpi_type: MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_BYTE, MPI_LONG, MPI_UNSIGNED_CHAR, etc.
- source: rank of source process (* = MPI_ANY_SOURCE)
- tag: message tag (* = MPI_ANY_TAG)
- com: communicator
- status: Structure with three fields: MPI_SOURCE, MPI_TAG, MPI_ERROR.

Point-to-point communication

- The dest and source arguments are, respectively, the rank of the receiving and sending processes.
- MPI allows source to be a wildcard (*), in this case we use MPI_ANY_SOURCE as parameter.
- There is no wildcard for destination.
- The tag is an integer, and for now, MPI_COMM_WORLD is our only communicator.
- There is a wildcard, MPI_ANY_TAG, which MPI_Recv can use as tag.
- There is no wildcard for the communicator.

Point-to-point communication

- Those items uniquely identify each incoming message.
- The source argument identifies the messages received from different processes.
- The tag, specified by the user, distinguishes among the messages sent by the same process.
- MPI guarantees that tags are integers at least between 0 and 32767, but many implementations allow greater values.
- A process A can send a message to process B if the arguments that A uses in MPI_Send are identical to the ones that B uses in MPI_Recv.

Point-to-point communication

- The last argument of MPI_Recv, status, returns information about the data received.
- This argument references a record with two fields: one for the source and the other one for the tag.
- So, for example, if the source of the receive function was MPI_ANY_SOURCE, then the status will contain the rank of the process that sent the message.
- Note that the amount of space allocated by the receive buffer does not have to be the same amount of space in the received message.
- MPI allows a message to be received as long as there is enough space to store it.

A simple example

https://github.com/gpsilva2003/MPI/blob/main/src/mpi_simples.c

Using handle status

- Reception information using wildcard is returned by the MPI_Recv function in the “handle status”.

Informação	C
remetente	status.MPI_SOURCE
etiqueta	status.MPI_TAG
erro	status.MPI_ERROR

- MPI_Get_count can be used to know the total number of elements received:

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

Status example

https://github.com/gpsilva2003/MPI/mpi_status.c

Checking for received messages

- Now that we've seen how the MPI_Status object works, we can use it together with the MPI_Probe routine to determine the size of a message before actually receiving it.
- It allows us to size the receive buffer appropriately rather than reserving a lot of space for all possible message sizes.
- The MPI_Probe function is very useful in master/worker applications that have an intensive exchange of variable length messages.

MPI_Probe

int MPI_Probe(int source, int tag, MPI_Comm com, MPI_Status* status)

- The MPI_Probe function is very similar to the MPI_Recv function.
- In fact, the first one performs the same functions as the second, except for receiving the message.
- The MPI_Probe function will block waiting for a message with the corresponding source and tag. When the message is available, it will fill status handle with the appropriate information.
- The user can then use MPI_Recv function to receive the actual message.

MPI_Probe example

https://github.com/gpsilva2003/MPI/mpi_probe.c

Case study – trapezoidal method

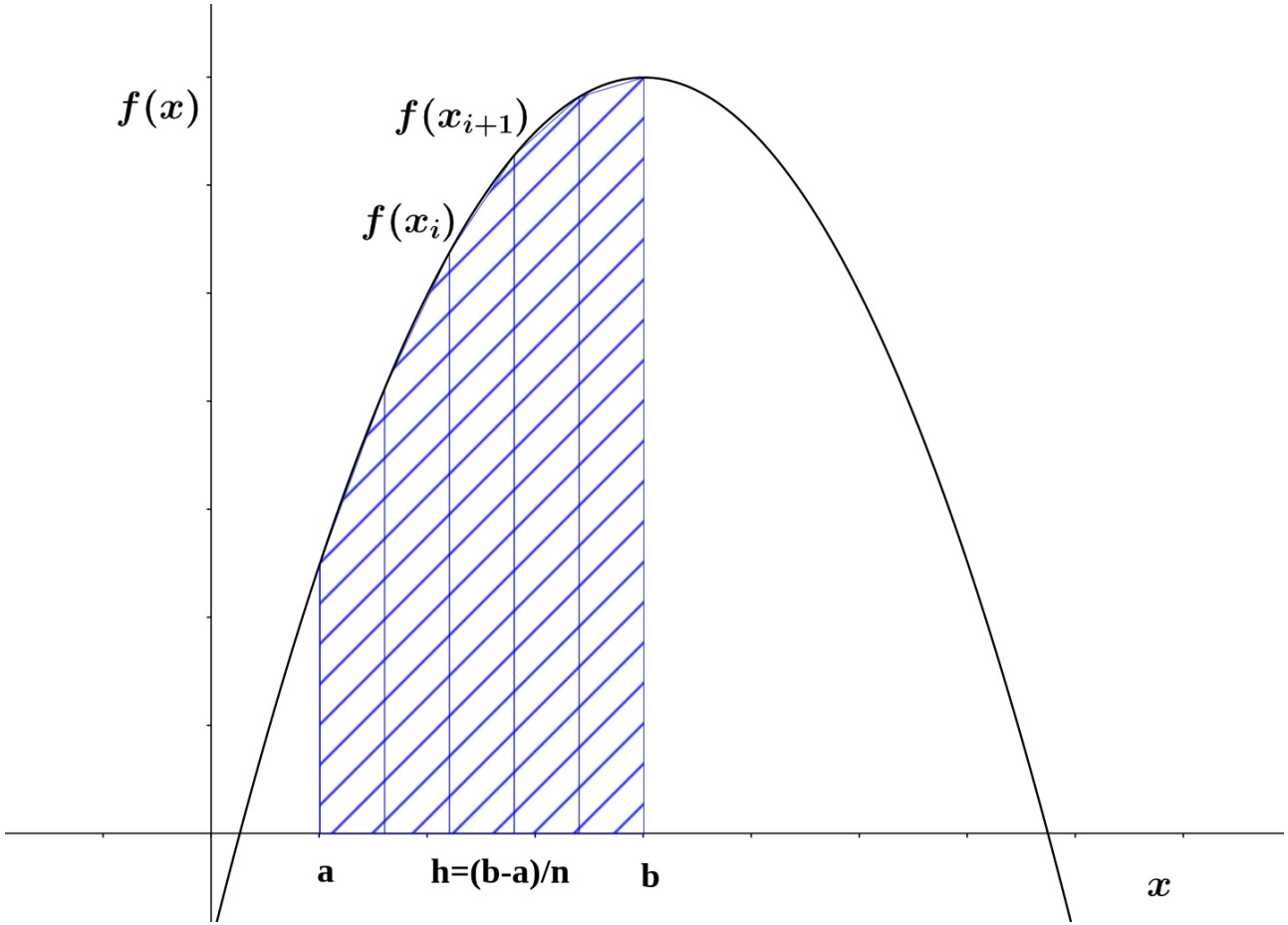
- Recall that the trapezoid method estimates the value of f(x) by dividing the interval [a; b] into "n" equal segments and calculating the following sum:

$$h * [\frac{f(x_o)}{2} + \frac{f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i)] \qquad (3.1)$$

$$h = \frac{(b-a)}{n} \text{ e } x_i = a + i * h, i = 1, ..., (n-1)$$

- By placing f(x) in a routine, we can write a program to calculate an integral using the trapezoidal method.

Case study – trapezoidal method



Case study – trapezoidal method

https://github.com/gpsilva2003/SERIAL/blob/main/src/trapezio_seq.c

Case study – trapezoidal method

```
/* The function f(x) is predefined.
* Input: a, b, n.
* Output: estimation of the integral from a to b of f(x).
*/
#include <stdio.h>
float f(float x) {
float return_val;
/* Calculate f(x). Store result in return_val. */
. . .
return return_val;
} /* f */
main() {
float integral;      /* Store result in integral */
float a, b;          /* Left and right limits */
int n;               /* Number of trapezoidals */
float h;             /* Trapezoidal base width */
```

Case study – trapezoidal method

```
float x;
int i;
    printf("Enter a, b, and n \n");
    scanf("%f %f %d", &a, &b, &n);
    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
    x = a;
    for (i = 1; i != n-1; i++) {
      x += h;
      integral += f(x);
    }
    integral *= h;
printf("With n = %d trapezoidals, the estimate \n",  n);
printf("from the integral from %f to %f = %f \n",  a, b, integral);
} /* main */
```

Case study – trapezoidal method

- The simplest way to parallelize this program is to divide the interval [a;b] evenly between the processes and each process estimating the value of the integral of f(x) in each subinterval.
- Suppose there are “p” processes and “n” trapezoids, and to simplify the discussion, we also assume that “n” is divisible by “p”.
- So it is natural that the first process calculates the area of the first “n/p” trapezoids, the second process calculates the next “n/p” and so on.
- The total value of the integral is determined by summing the local values calculated in each process.

Case study – trapezoidal method

- The process ***q*** will estimate the integral over the interval:

$$[a + q\frac{nh}{p}, a + (q+1)\frac{nh}{p}]$$

- Therefore, each process needs the following information:
  - The number of available processes, p.
  - Its rank, q.
  - The interval of integration, [a; b].
  - The number of subintervals, n.

Case study – trapezoidal method

- Remember that the first two items can be found using the MPI functions:

  MPI_Comm_size

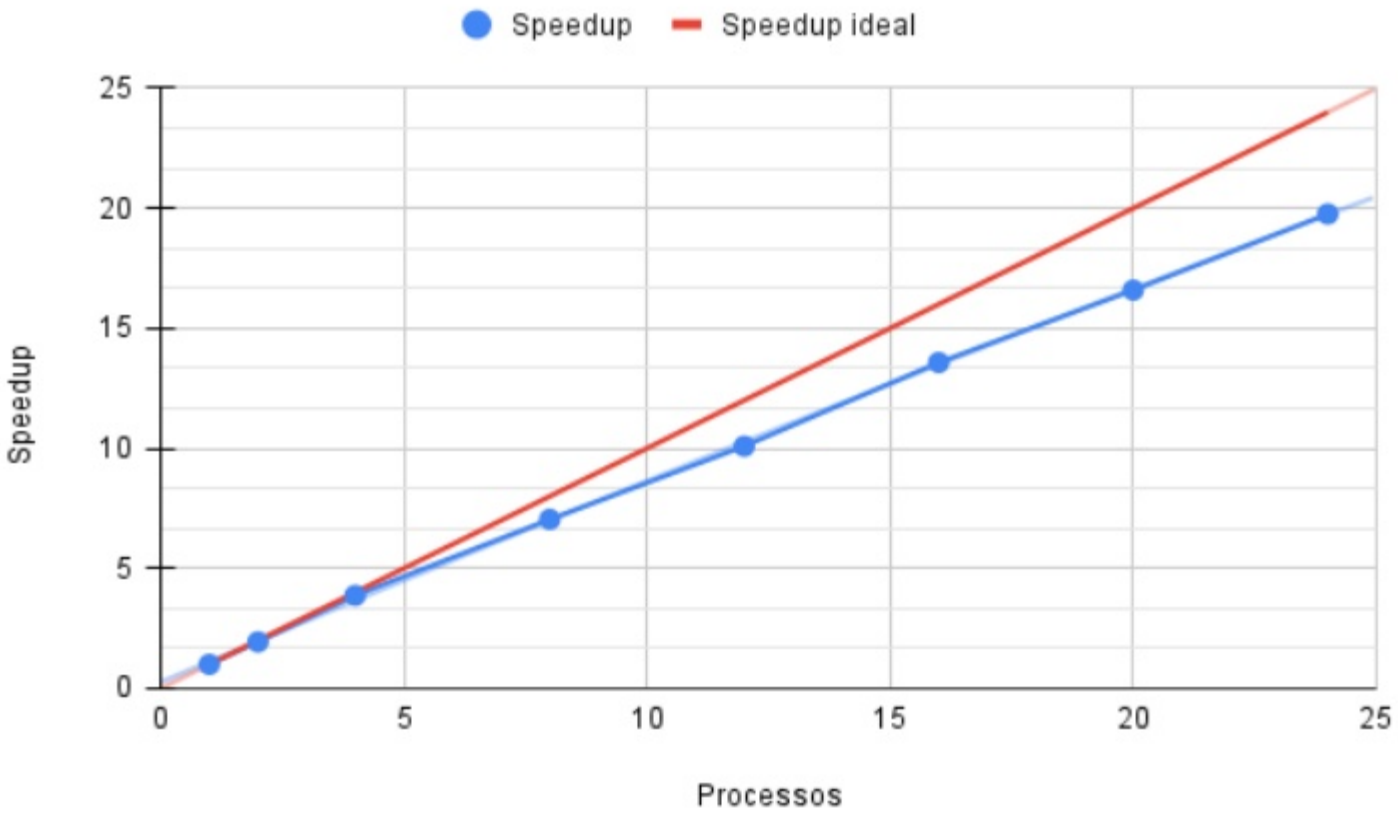
  MPI_Comm_rank

- The user could inform the last two items, but for our first attempt at parallelization, we are assigning fixed values to them.
- A simple way to calculate the sum of all local values is each process sending its partial result to process 0 and let that process do the final sum.

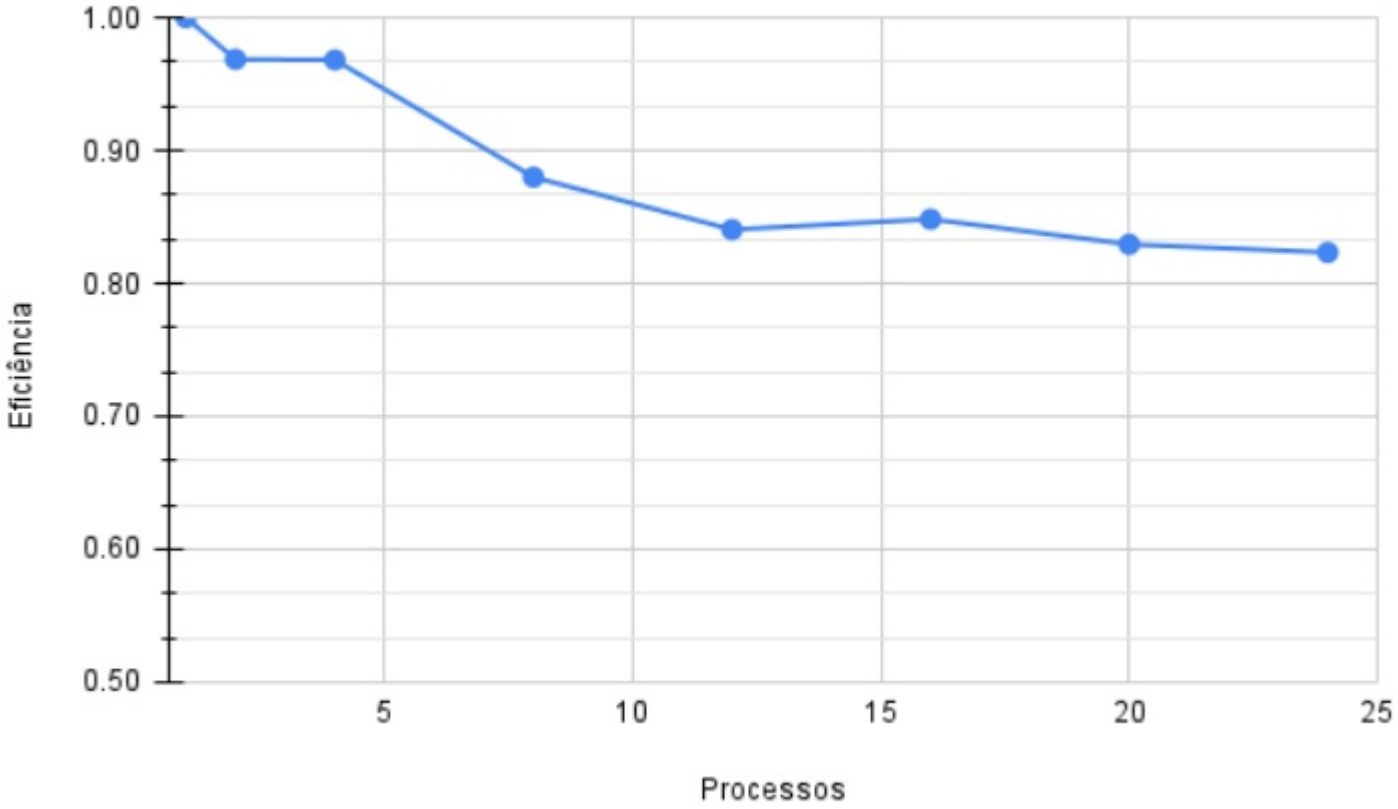
Trapezoidal method example

https://github.com/gpsilva2003/MPI/blob/main/src/mpi_trapezio.c

Trapezoidal method – Speedup



Trapezoidal method – Efficiency



Collective Communication

Collective communication

- Collective communications operations are more restrictive than peer-to-peer communications:
  - The amount of data sent must exactly match the amount of data specified by the receiver.
  - The tag argument does not exist.
  - Only the blocking version of the functions is available.*.
- All processes participating in a collective communication call the same function with compatible arguments.

* In the latest versions of the MPI standard there are already non-blocking versions of collective communication routines.

Collective communication

- When a collective operation has a single source process or a single target process, this process is called a root.
- *Barrier*

  Blocks all processes until all processes in the group call the function.
- *Broadcast*

  Sends the same message to all processes.
- *Gather*

  Data is collected from all processes in a single process.
- *Scatter*

  Data is distributed from one process to the others.

Collective communication

- *Allgather*

  A gather followed by a broadcast.

- *Reduce*

  It performs the collective operations of sum, maximum, minimum, etc.

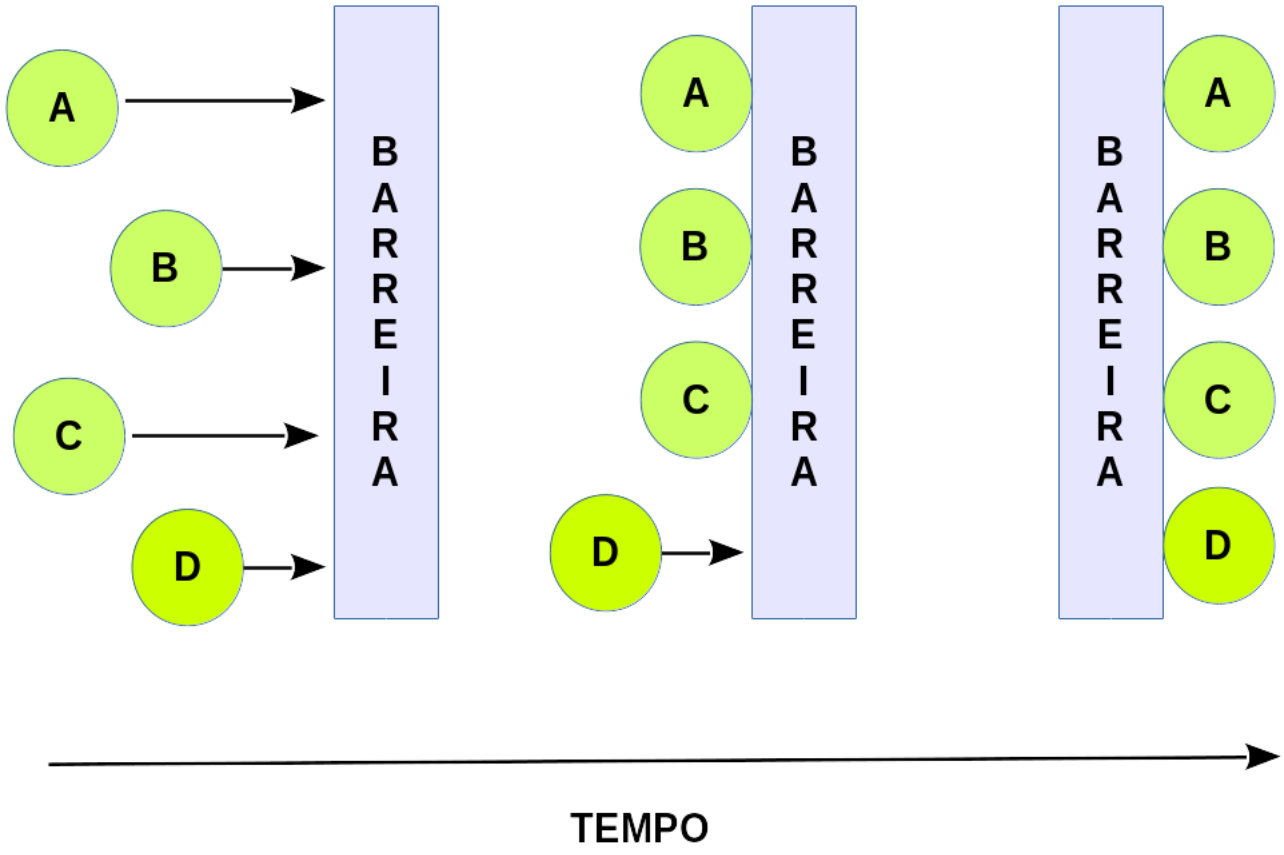
- *Allreduce*

  A reduction followed by a diffusion.

- *Alltoall*

  A set of gather operations where each process receives different data.

Barrier



MPI_Barrier

int MPI_Barrier(MPI_Comm com)

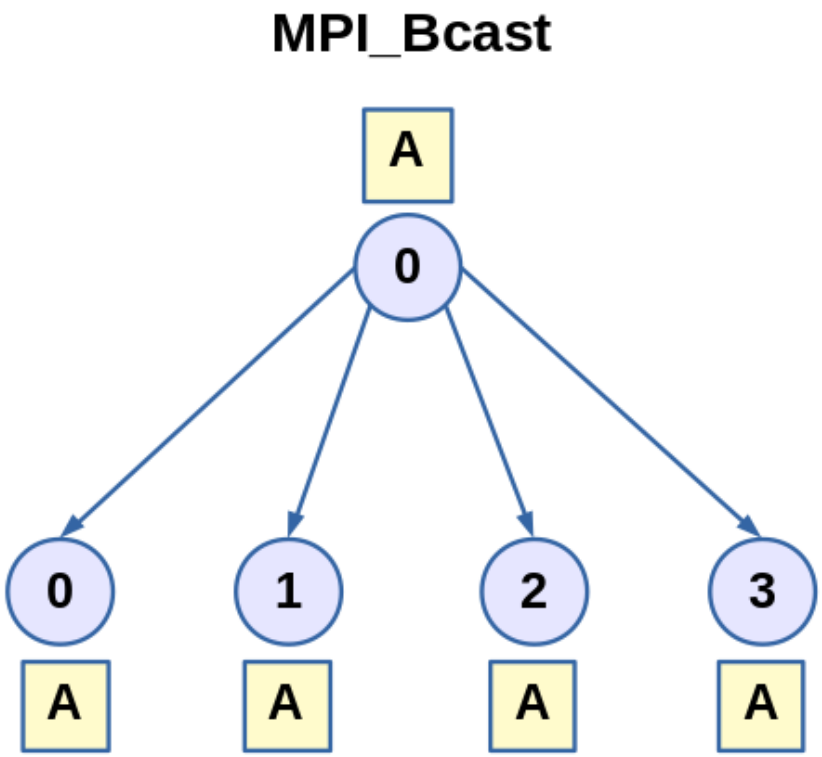
- The MPI_Barrier function provides a mechanism to synchronize all processes on the communicator com.
- Each process blocks (i.e., stops) until all processes in com have called MPI_Barrier.

MPI_Bcast

- A communication pattern that involves all processes on a communicator is called collective communication.
- A broadcast is a collective communication in which a single process sends the same data to each process.
- The MPI function for broadcast is:

```
int MPI_Bcast (void* message, int count,
MPI_Datatype mpi_type, int root, MPI_Comm com)
```

Broadcast



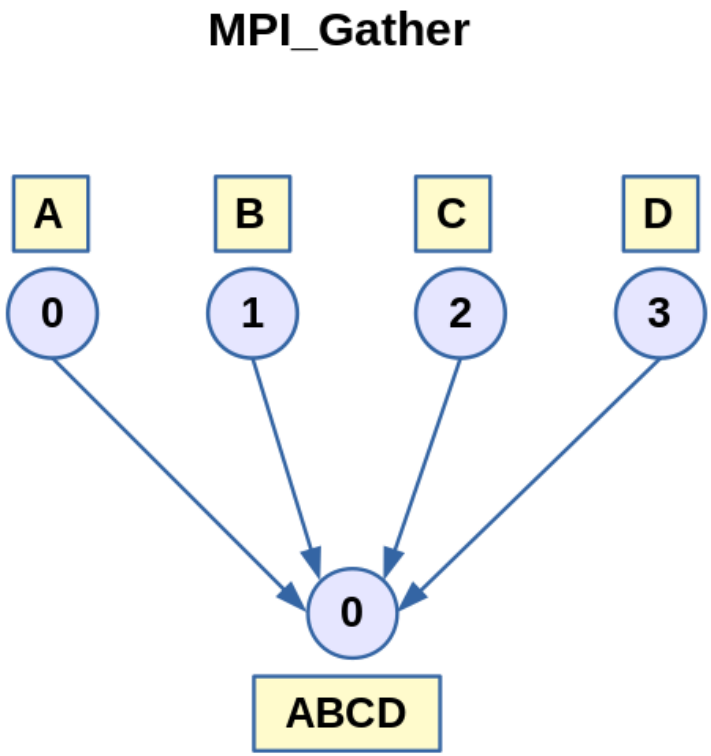
MPI_Bcast

- It simply sends a copy of the message on the root process to each process on com communicator.
- It must be called by all processes on the communicator using the same arguments for root and com.
- MPI_Recv cannot be used to receive a broadcast message.
- The count and mpi_type parameters have the same function as the MPI_Send and MPI_Recv functions: they specify the size of the message.

MPI_Bcast

- However, unlike peer-to-peer functions, the MPI standard requires that count and mpi_type parameters are the same for all processes on the same communicator for collective communication.
- That is because a process can receive data from more than one process, and to determine the total amount of data received, an integer vector of return status would be required.

Gather

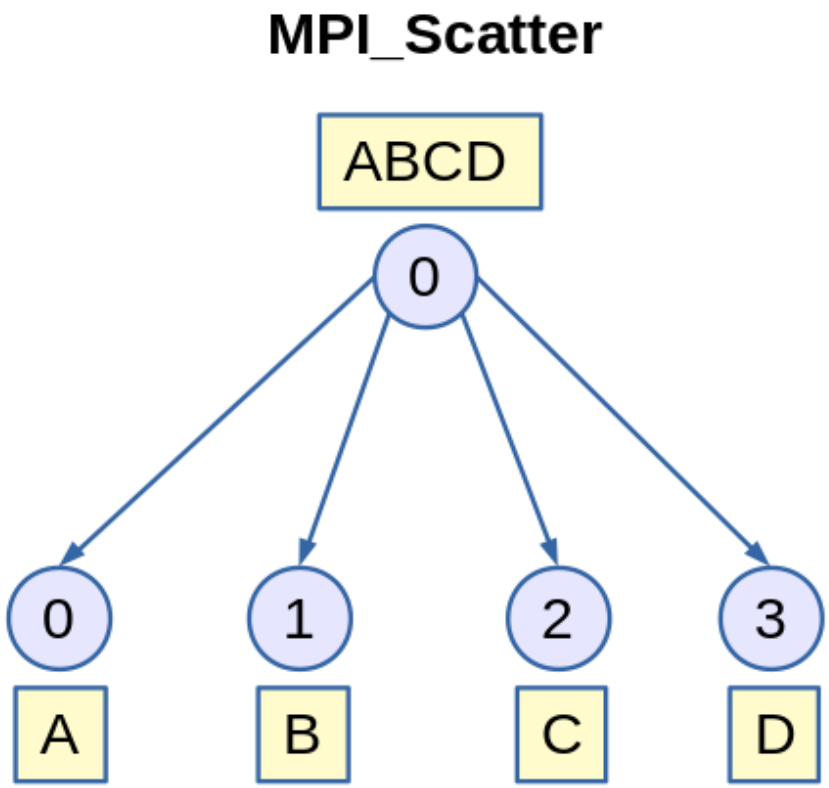


MPI_Gather

int MPI_Gather(void *send_buf, int send_count, MPI_Datatype send_type, void *recv_buf, int recv_count, MPI_Datatype recv_type, int root, MPI_Comm com)

- Each process in com sends the contents of buf_send to the process with rank equal to root.
- The root process concatenates the received data in recv_buf in an order that is defined by the rank of each process.
- The receiving arguments are only meaningful in the process with rank equal to root.
- The recv_count argument indicates the number of items sent by each process, not the total number of items received by the root process, and normally is equal to send_count.

Scatter



MPI_Scatter

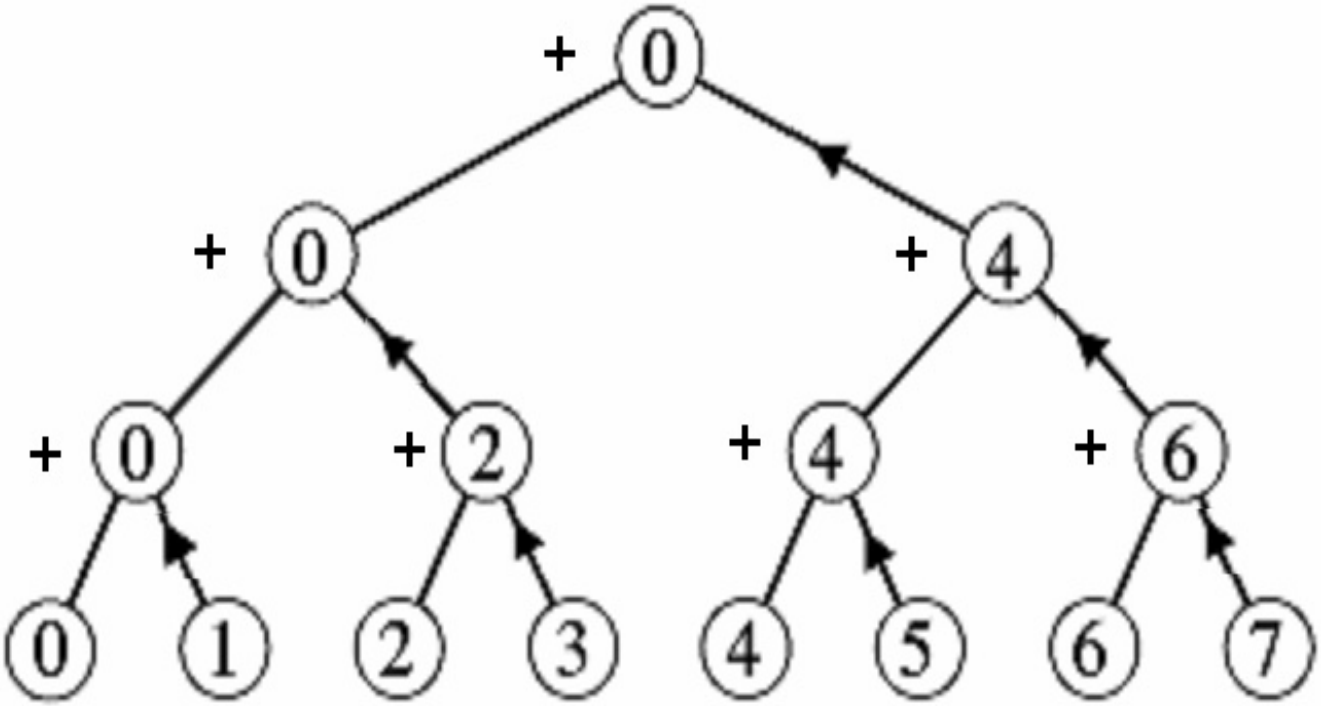
```
int MPI_Scatter(const void *send_buf, int
send_count, MPI_Datatype send_type, void
*recv_buf, int recvcount, MPI_Datatype recv_type, int
root, MPI_Comm com)
```

- The process with a rank equal to root distributes the contents of send_buf among all processes.
- The contents of send_buf are divided into p segments, each one with send_count items each.
- The first segment goes to the process with rank 0, the second to the process with rank 1, and so on.
- The send_buf argument is only meaningful in the root process.

Reduction

- You might recall that in the trapezoidal method program, each processor executes the same amount of work until the end of the local summing phase.
- However, after that, processor 0 receives the partial sums of each processor sequentially, causing a workload imbalance.
- But, we can use the following procedure instead:
  a) 1 sends its result to 0, 3 to 2, 5 to 4, 7 to 6.
  b) 0 adds its integral to that of 1, 2 adds to that of 3, etc.
  c) 2 sends to 0, 6 sends to 4.
  d) 0 sum, 4 sum.
  e) 4 sends to 0.
  f) 0 sum.

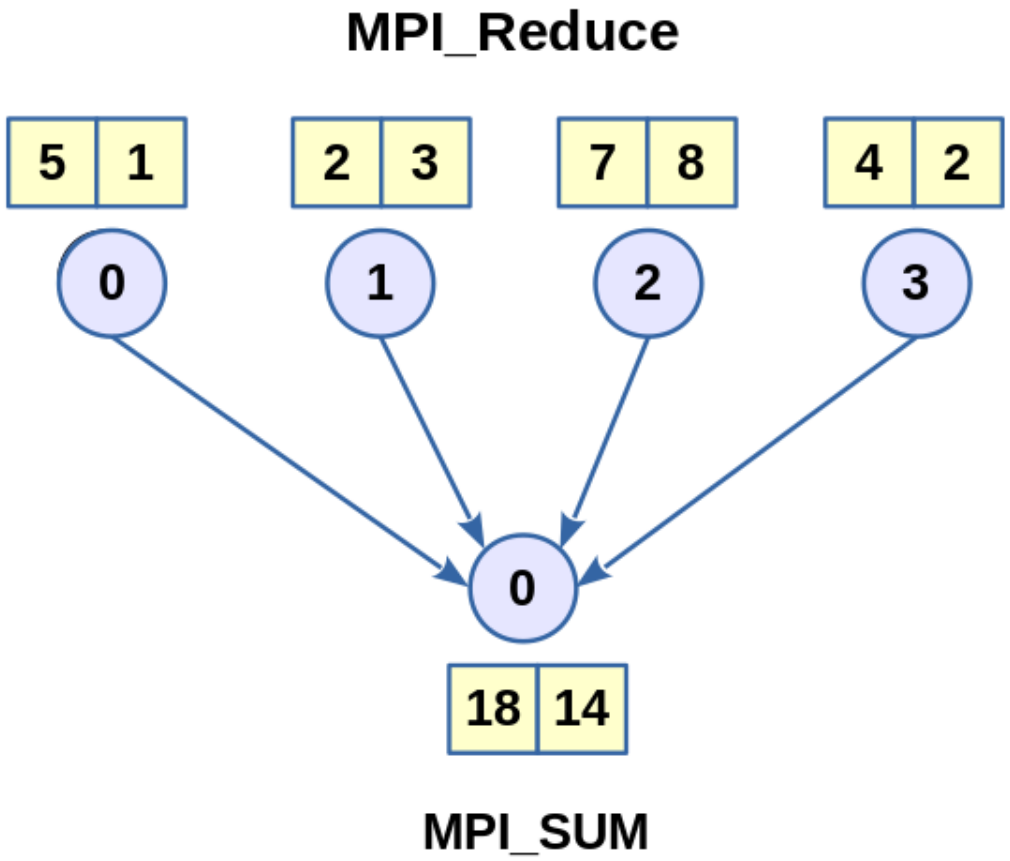
Reduction



Reduction

- The global sum we are trying to calculate is an example of a general class of collective communication operations called reduction operations.
- In a global reduction operation, data sent by all processes in a communicator is combined with binary operations.
- Typical binary operations are addition, maximum, minimum, binary and logical operations, etc.
- You can also define other operations beyond those pre-defined for the MPI_Reduce function.

Reduction



MPI_Reduce

int MPI_Reduce(void* operand, void* result, int count, MPI_Datatype mpi_type, MPI_Op oper, int root, MPI_Comm com)

- The MPI_Reduce operation combines the operands stored in *operand using the oper operation and stores the result in *result only in the root process.
- Both operand and result refer to count memory locations with type mpi_type.
- All processes on the communicator must call MPI_Reduce with an identical count, mpi_type, and oper values.

MPI_Reduce

- The oper argument can have one of the following predefined values:

Operation name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	“AND” logic
MPI_BAND	“AND” bitwise
MPI_LOR	“OR” logic
MPI_BOR	“OR” bitwise
MPI_LXOR	“OR EXCLUSIVE” logic
MPI_BXOR	“OR EXCLUSIVE” bitwise
MPI_MAXLOC	Maximum and position
MPI_MINLOC	Minimum and position

MPI_Reduce

- As an example, let's rewrite the last lines of the trapezoidal method program:

```
. . .
/* Adds the integrals calculated by each process */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
MPI_SUM, 0, MPI_COMM_WORLD);
/* Print the result */
. . .
```

Reduction

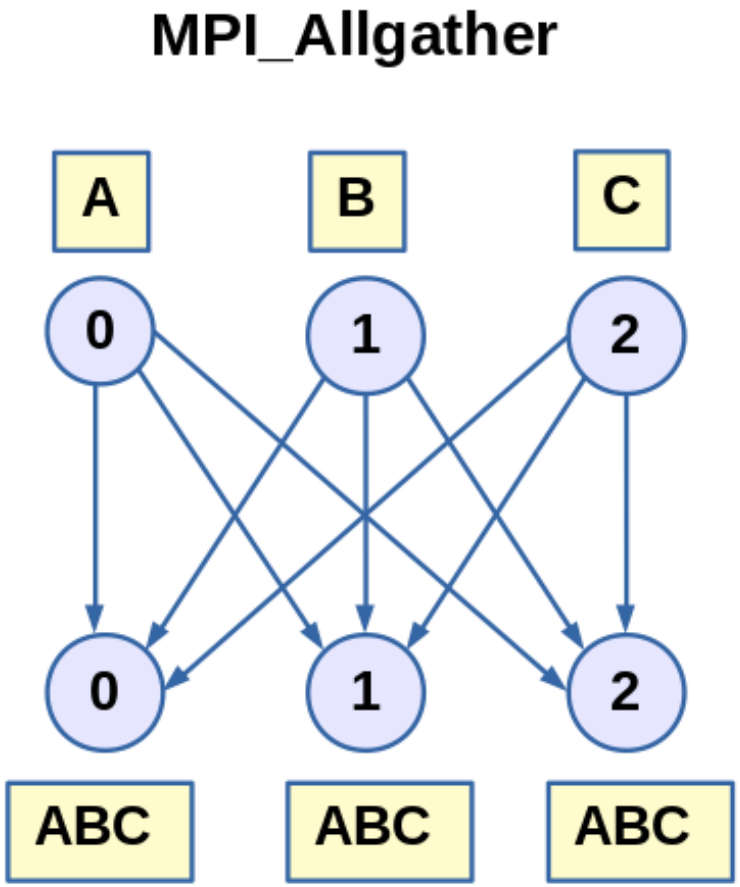
https://github.com/gpsilva2003/MPI/src/mpi_reduce.c

Allgather

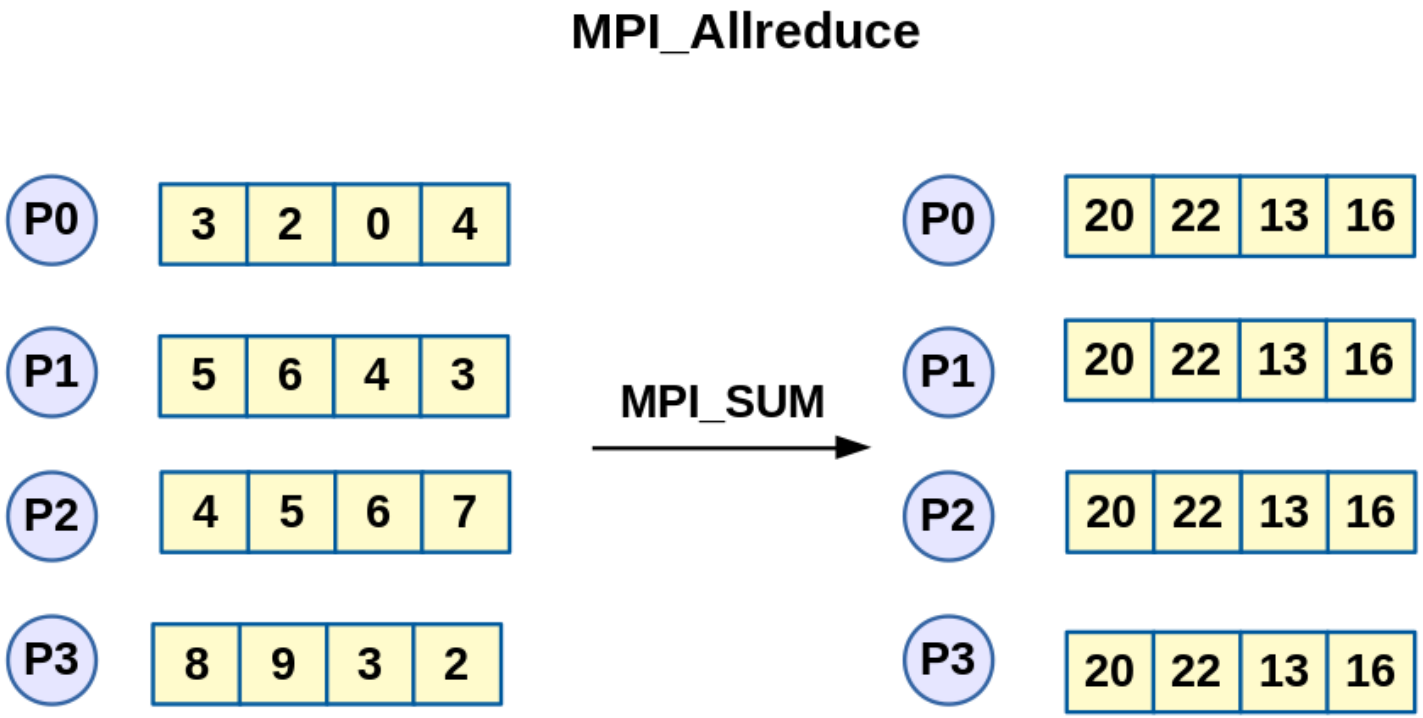
```
int MPI_Allgather(const void *send_buf, int
send_count, MPI_Datatype send_type, void
*recv_buf, int recv_count, MPI_Datatype recv_type,
MPI_Comm com)
```

- MPI_Allgather gathers the contents of send_buf in each process.
- Its effect is equivalent to a sequence of **p** calls to MPI_Gather, each with a different process acting as the root process.

Allgather



Reduction with Broadcast



MPI_Allreduce

```
int MPI_Allreduce(void *send_buf, void *recv_buf, int
    count, MPI_Datatype mpi_type, MPI_Op oper,
    MPI_Comm com)
```

- MPI_Allreduce stores the result of the oper reduce operation in the recv_buf buffer of each process.

Matrix-Vector Multiplication Case Study

$$A_{m,n} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ a_{2,0} & a_{2,1} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \quad b_n = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

$$c_m = \mathrm{Ab} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{m-1} \end{bmatrix}$$

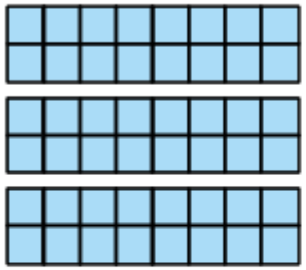
$$c_i = a_{i,0}.b_0 + a_{i,1}.b_1 + a_{i,2}.b_2 + \cdots + a_{i,n-1}.b_{n-1}$$

Matrix-Vector Multiplication Case Study

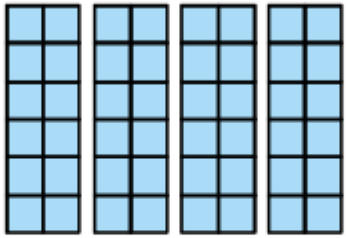
$$A \times b = c$$

$$\begin{bmatrix} 2 & 1 & 3 & 4 & 0 \\ 5 & -1 & 2 & -2 & 4 \\ 0 & 3 & 4 & 1 & 2 \\ 2 & 3 & 1 & -3 & 0 \end{bmatrix} \times \begin{bmatrix} 3 \\ 1 \\ 4 \\ 0 \\ 3 \end{bmatrix} = \begin{bmatrix} 19 \\ 34 \\ 25 \\ 13 \end{bmatrix}$$

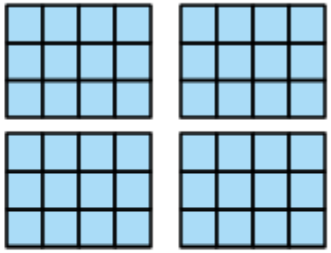
Matrix-Vector Multiplication Case Study



Decomposição em blocos no sentido das linhas



Decomposição em blocos no sentido das colunas



Decomposição em blocos j x k

Matrix-Vector Multiplication Case Study

- Each of these forms of decomposition has its advantages and disadvantages, as well as distinct complexities.
- For simplicity, we will assume that we will use the first alternative of data distribution, with each process having a block of rows of matrix A and vectors b and c replicated in each process.
- A simple complexity analysis, assuming m = n, indicates a sequential computational complexity of $O(n^2)$. When p processes are used, the computational complexity per process, without communication costs, is equal to $O(n^2/p)$.

Matrix-Vector Multiplication Case Study

- For the communication costs, not including the cost of sending line i and vector b, and just collecting and spreading the result vector c for all processes, we have the following situation.
- An efficient algorithm for the MPI_Allgather function requires each process to send $[\log_2 p]$ messages, with the total number of elements sent per process equal to $n(p - 1)/p$.
- So the communication complexity is equal to $O(n + \log_2 p)$, and the total complexity of this algorithm is equal to $O(n^2 /p + n + \log2 p)$ (J et al., 2007).

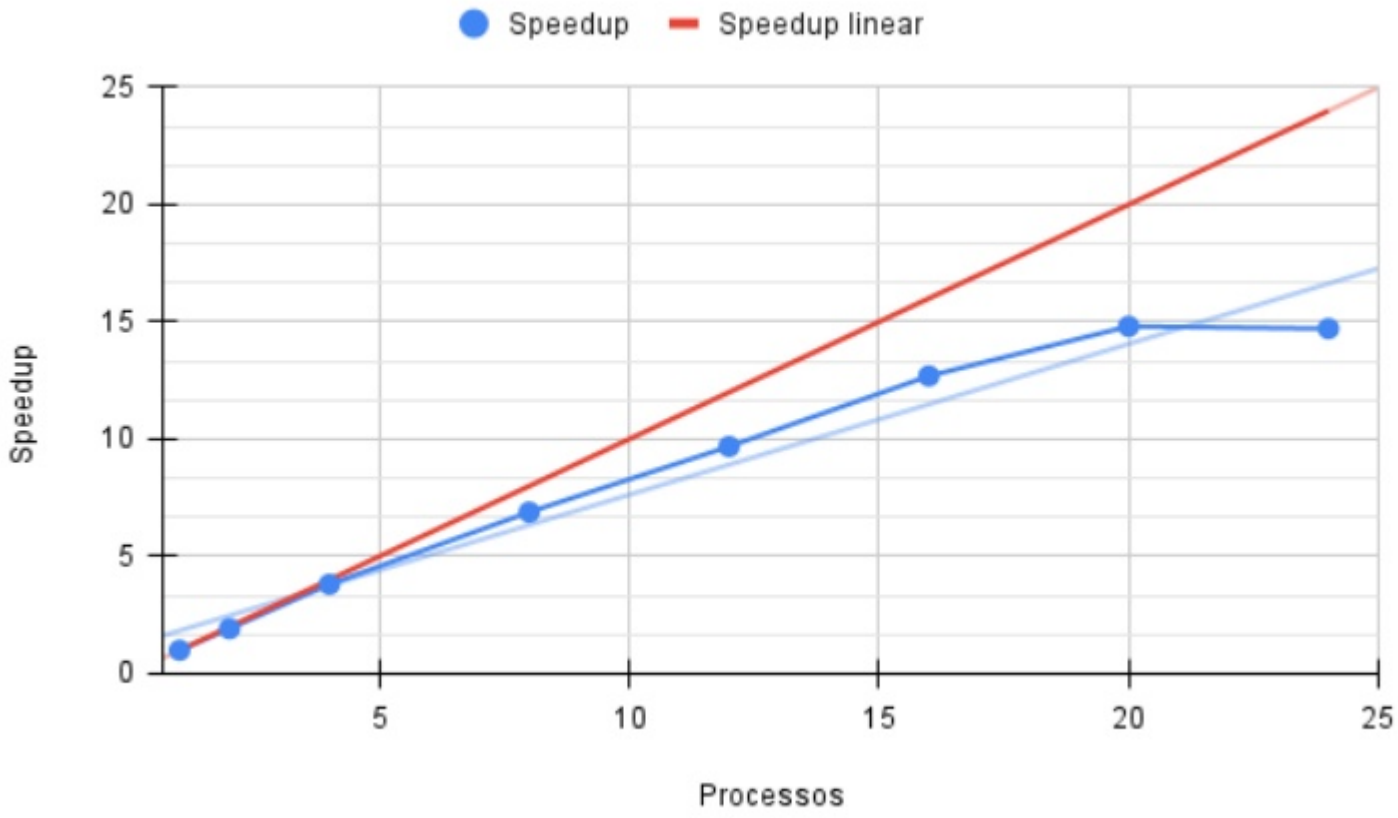
Matrix-Vector Multiplication Sequential Solution

https://github.com/gpsilva2003/SERIAL/blob/main/src/mvx_seq.c

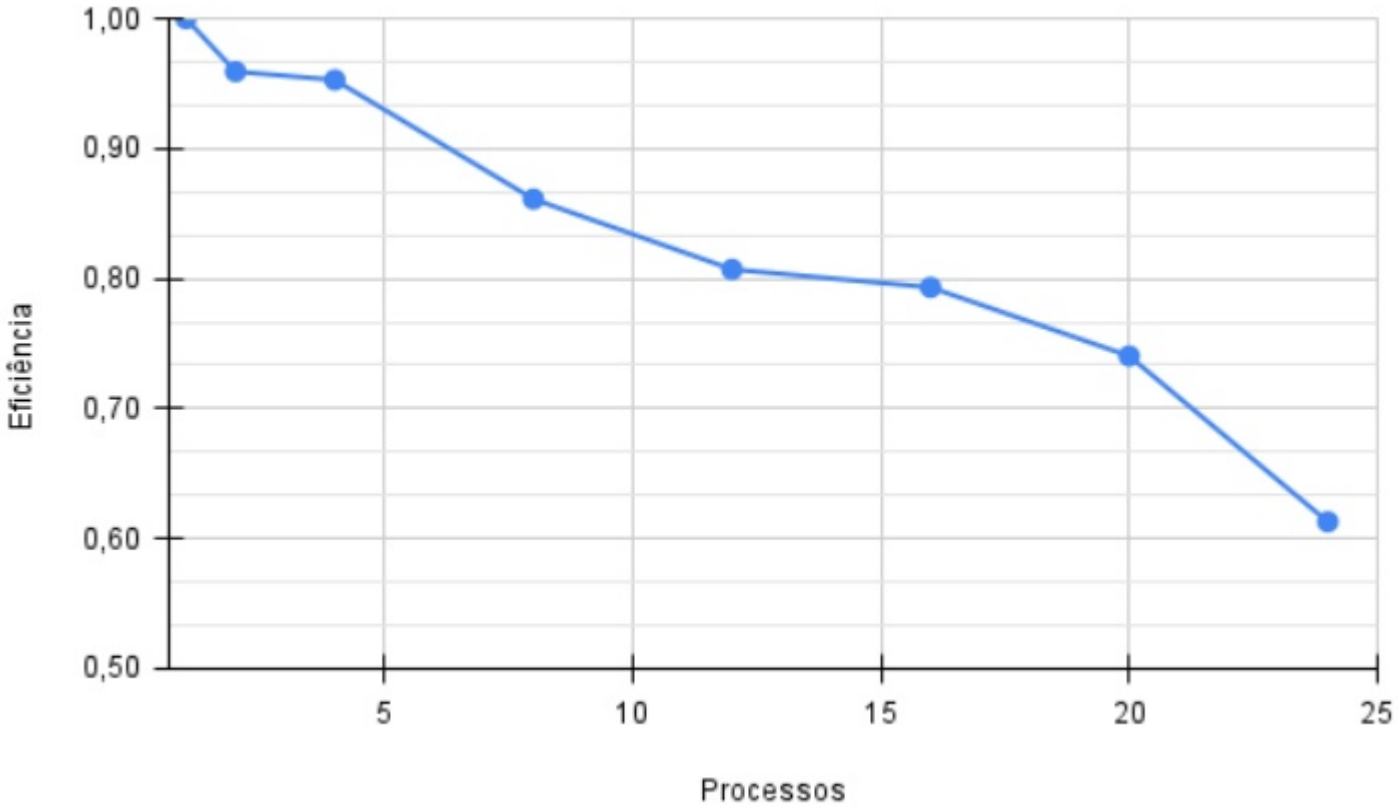
Matrix-Vector Multiplication Parallel Solution

https://github.com/gpsilva2003/MPI/mpi_mxv.c

Matrix-Vector Multiplication Speedup



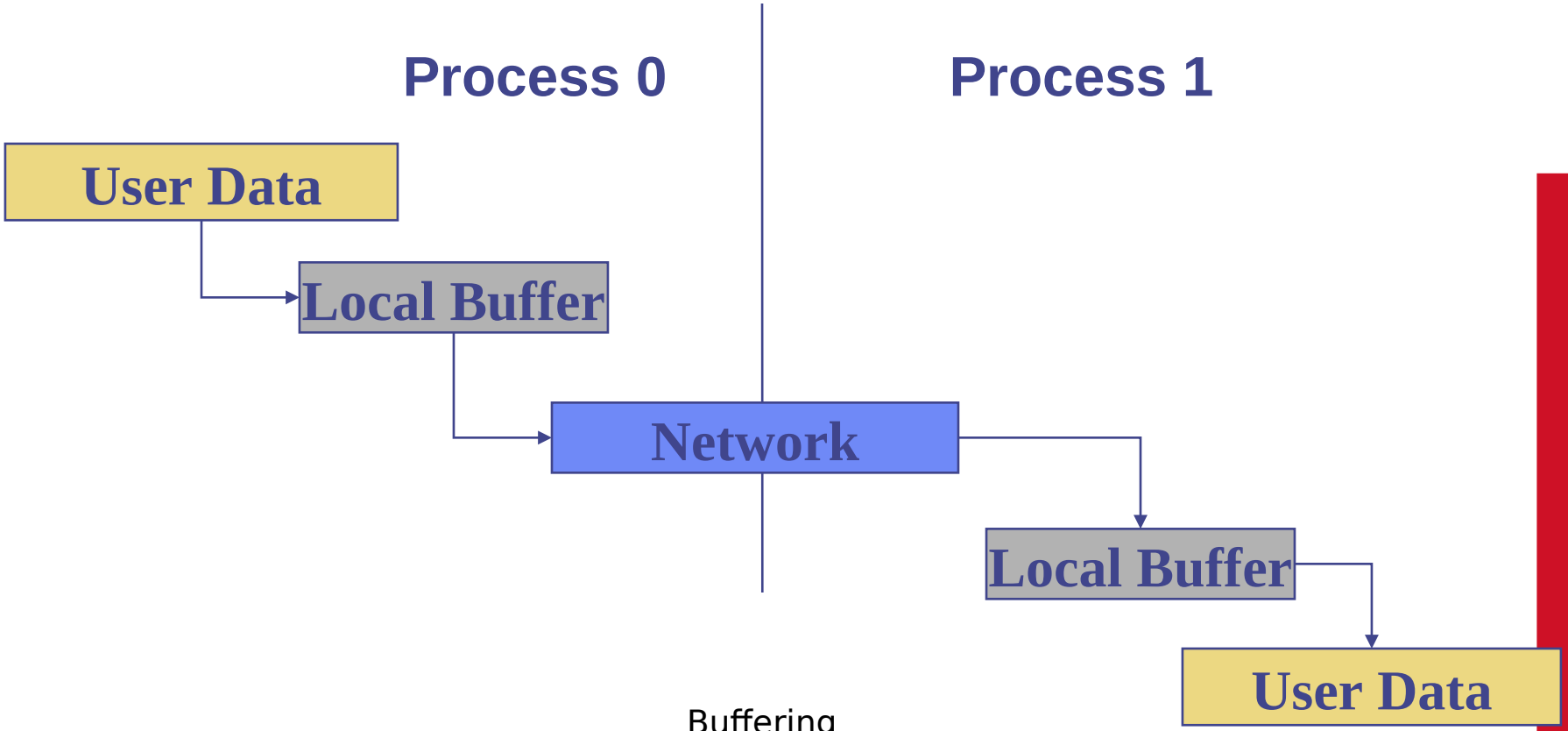
Matrix-Vector Multiplication Efficiency



Communication Modes

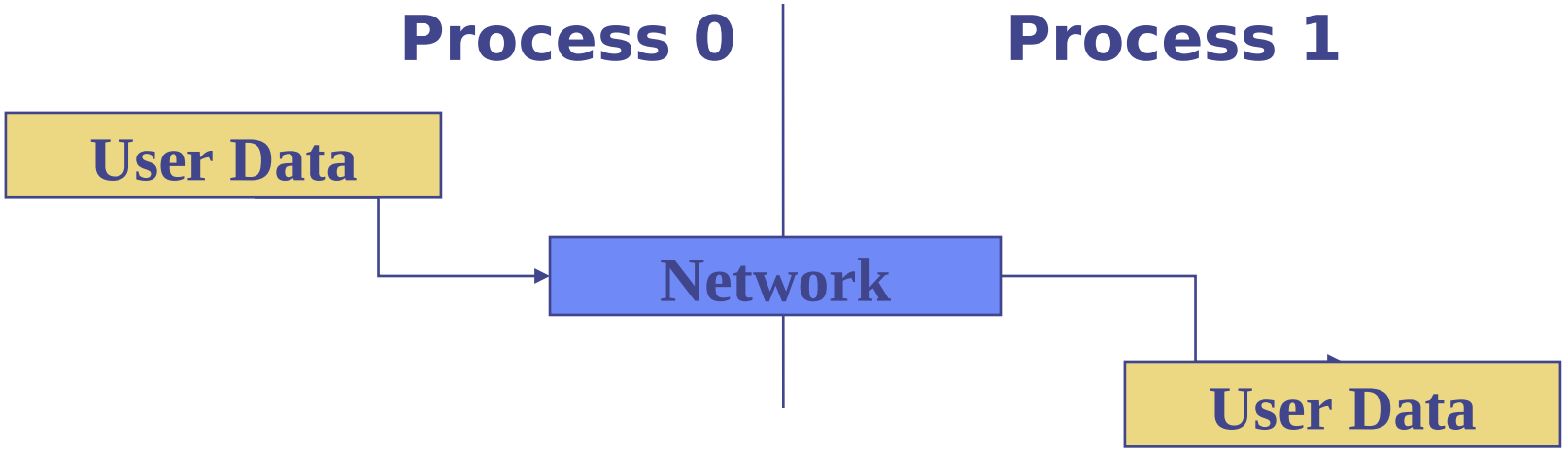
Sending a Message

- When you send a message, where does it go? One possibility is:



Improving Performance

- It is better to avoid copies:



- This requires changes in the way messages are sent and received so that the operation can complete successfully.

Blocking Communication

- In blocking communication:
  - MPI_Recv does not complete until the userspace receive buffer is full (message available for use).
  - MPI_Send does not complete until the send buffer in userspace is empty (buffer available for reuse).
- The success of the communication operation depends on the size of the message and the size of the system buffer, and messages larger than the available space in the system buffer will be sent in synchronous mode.
- A correct MPI program cannot rely on using a system buffer. Such programs are called unsafe, although they may run and produce correct results most of the time.

Blocking Communication

- A blocking send routine will only return after there is a guarantee that the application data (your send data) can be reused.
- As a guarantee, it is understood that subsequent modifications will not affect the data being sent. This guarantee does not imply that the data was actually received by another process - it could very well be situated in a system buffer.
- A blocking transmission can be synchronous, forcing a confirmation protocol with the reception routine to ensure the complete sending of the message.
- A blocking send can also be asynchronous, as long as a system buffer is used to store the data before it is distributed to the receiving routine.

Non-Blocking Communication

- If we are not using buffers, we must use non-blocking send/receive routines, to guarantee that there will be no “deadlock”.
- In blocking sending, the program stops until the userspace message buffer can be used safely.
- In non-blocking sending, the computation continues and, when necessary, we check whether the operation has already completed or not.

Non-Blocking Communication

- The only difference in non-blocking operations is that they (immediately) return a “handle request”.
- This handle can be tested only once or used to “loop” waiting for the message to arrive.
- Therefore, if the programming is done properly, we can perform computation and communication in parallel, improving the final performance of the program.

Non-Blocking Communication

int MPI_Isend(void* message, int count, MPI_Datatype mpi_type, int dest, int tag, MPI_Comm com, MPI_Request *request)

int MPI_Irecv(void* message, int count, MPI_Datatype mpi_type, int source, int tag, MPI_Comm com, MPI_Request *request)

Waiting for a Message

- Waiting for the message:

  int MPI_Wait(MPI_Request *request, MPI_Status *status)

- You can also test withou waiting:

  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

Multiple Wait

- Sometimes it is desirable to wait for multiple “requests”:

  int MPI_Waitall(int cont, MPI_Request vetor_de_pedidos[ ], MPI_Status vetor_de_estados[ ])

  int MPI_Waitany(int cont, MPI_Request vetor_de_pedidos[ ], int *indice, MPI_Status *estado)

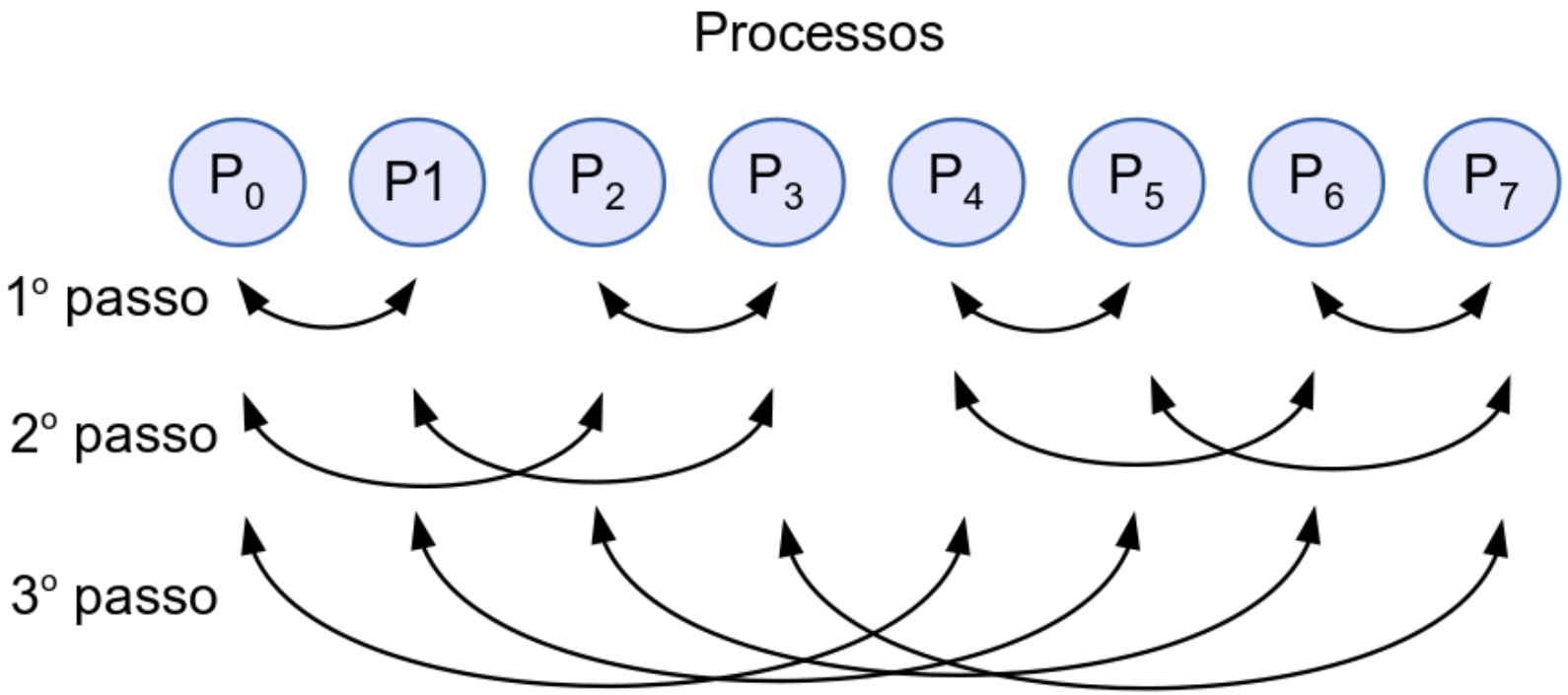
  int MPI_Waitsome(int cont_entra, MPI_Request vetor_de_pedidos[ ], int *cont_saida, int vetor_de_indices[ ], MPI_Status vetor_de_estados[ ])

- There are corresponding test versions for each of the above functions.

Reduction with Broadcast

- The following examples present an implementation of the allreduce operation using several MPI communication modes and non-blocking send and receive routines.
- The algorithm of the MPICH implementation of MPI, which uses a recursive doubling technique, was used as a basis for these examples.
- For simplicity, let's assume that P, the number of processes, is a power of 2 and implement only the MPI_MAX reduction operation (get the maximum of all values).
- The algorithm only requires log2 P steps to perform the reduction algorithm and broadcast the result to all nodes.

Reduction with Broadcast



Non-Blocking Communication Example

https://github.com/gpsilva2003/MPI/mpi_isend.c

Communication Modes

- MPI defines four modes of communication:
  - Synchronous mode (the most secure)
  - Ready mode (less overhead for the system)
  - “Buffered” mode (decouples the sender from the receiver)
  - Default mode (compromise solution, but no one really knows what happens)
- The communication mode is selected according to the sending routine used.

Communication Modes

- Standard
  - The system decides whether the message will be “buffered”.
- Buffered
  - The application must explicitly provide a “buffer” for the sent message.
- Synchronous
  - The send operation does not complete until the receive operation has started.
- Ready
  - The send operation can only be started after the receive operation has already started.

Communication Modes

- Each mode has a corresponding non-blocking version

Communication Mode	Blocking Routines	Non-blocking Routines
Synchronous	MPI_Ssend	MPI_ISsend
Ready	MPI_Rsend	MPI_IRsend
Buffered	MPI_Bsend	MPI_IBsend
Standard	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv

Communication Modes

int MPI_Bsend(void* message, int count, MPI_Datatype mpi_type, int dest, int tag, MPI_Comm com)

int MPI_Ssend(void* message, int count, MPI_Datatype mpi_type, int dest, int tag, MPI_Comm com)

int MPI_Rsend(void* message, int count, MPI_Datatype mpi_type, int dest, int tag, MPI_Comm com)

Communication Modes

- For a given message sending routine, one says that a corresponding reception operation has been posted, when any process starts a reception routine with a communicator and tag that satisfies the communicator and tag used by the sending routine.
- Note that wildcards are allowed in reception operations for both the sender (MPI_ANY_SOURCE) and the tag (MPI_ANY_TAG), which must be considered in this case.
- MPI_Recv receives messages sent in any mode.

Buffered Mode

- The send operation can be started whether or not a corresponding receive operation is started. The send operation may complete before a corresponding receipt has been posted.
- There is a need to use additional functions for allocating and releasing space for storing messages.
- It is up to the user, not the system, to manage the allocation of buffers.
- It is guaranteed that send and receive operations are not synchronized.

Buffered Mode

```
int MPI_Buffer_attach (void *buffer, int size);
```

- There can only be one active buffer at a time.
- The amount of space allocated must be sufficient to guarantee the correct functioning of the program.

```
int MPI_Buffer_detach(void *buffer_addr,
int *size);
```

- This routine returns a pointer to the buffer being deactivated and a pointer to its size.
- This is done to allow a library to overwrite and restore the buffer.
- The allocated space is not automatically freed.

MPI_Pack_Size

```
int MPI_Pack_size(int count, MPI_Datatype mpi_type,
    MPI_Comm com, int *size)
```

```
MPI_Pack_size(20, MPI_INT, com, &tam1);
MPI_Pack_size(40, MPI_FLOAT, com, &tam2);
tam_buffer = tam1 + tam2 +2*MPI_BSEND_OVERHEAD;
```

- In order to calculate the space required for each message, the MPI_Pack_size routine must be used.
- The MPI_BSEND_OVERHEAD constant specifies an additional space required by the system to send each message (such as envelope information).
- This constant has a specific value for each MPI implementation.

Buffered Mode Example

https://github.com/gpsilva2003/MPI/mpi_bsend.c

Synchronous Mode

- The send routine can be started whether or not there is a corresponding receive routine posted.
- However, the sending will complete successfully only when a corresponding reception has been posted and reception of the message sent by the synchronous send routine is initiated by the receive operation.
- This mode does not require the use of system buffering.
- You can assure that a program is safe if it runs correctly using only send routines in synchronous mode.

Synchronous Mode Example

https://github.com/gpsilva2003/MPI/mpi_ssend.c

Ready Mode

- Sending can be started only if there is a corresponding receive routine already started. If not, the program will terminate with an error.
- Although expected, the implementation of routines in ready mode is not guaranteed to be more efficient than in standard mode.
- It is necessary to use synchronization functions (eg barriers) to ensure that reception in one process is posted before sending by the other.
- This is the most difficult mode to program and should only be used when performance is important.

Ready Mode Example

https://github.com/gpsilva2003/MPI/mpi_rsend.c

Standard Mode

- In this mode, MPI decides whether the messages sent will be buffered or sent synchronously.
- A standard mode send routine can be started whether or not there is a corresponding receive routine posted.
- It cannot be assumed that the send operation will end before or after the corresponding reception is started.
- As a result, programs may work well on one system and not on another if the program is programmed in an unsafe way.

Standard Mode Example

https://github.com/gpsilva2003/MPI/mpi_padrao.c

Deadlock

- Sending a large message from process 0 to process 1

  If the storage space on the destination is insufficient, the sending routine must wait until the user provides enough memory space (by calling a receive routine).

- What happens to this code?

Processo 0	Processo 1
MPI_Send(1)	**MPI_Send(0)**
MPI_Recv(1)	**MPI_Recv(0)**

- This is called “unsafe” because it depends on the availability of system buffers.

Some Solutions

- Order the sending and receiving operations properly:

Process 0	Process 1
MPI_Send(1)	MPI_Recv(0)
MPI_Recv(1)	MPI_Send(0)

- Fornecer um buffer de recepção ao mesmo tempo que envia a mensagem:

More Solutions

- The user explicitly provides a buffer to send:

Process 0	Process 1
MPI_BSend(1)	MPI_BSend(0)
MPI_Recv(1)	MPI_Recv(0)

- Use of non-blocking operations:

Process 0	Process 1
MPI_Isend(1)	MPI_ISend(0)
MPI_Irecv(1)	MPI_IRecv(0)
MPI_Waitall	MPI_Waitall

MPI_Sendrecv

- Allows simultaneous sending and receiving.
- Provides a receive buffer while sending the message.
- Send and receive data types may be different.
- You can use MPI_Sendrecv with common MPI_Recv or MPI_Send (or MPI_Irecv, MPI_Ssend, etc.)

Process 0	Process 1
MPI_Sendrecv(1)	MPI_Sendrecv(0)

MPI_Sendrecv

int MPI_Sendrecv(const void *send_buf, int send_count, MPI_Datatype send_type, int dest, int send_tag, void *recv_buf, int recv_count, MPI_Datatype recv_type, int source, int recv_tag, MPI_Comm com, MPI_Status * status)

MPI_Sendrecv Example

https://github.com/gpsilva2003/MPI/mpi_sendrecv.c

Prime Numbers Case Study

- In this section, our case study will be a program to calculate the number of prime numbers between 0 and a given integer value N.
- It basically checks if N is divisible by some odd number between 0 and the square root of N, with the even numbers being discarded right away.

```
$ mpicc -O3 -o mpi_primos mpi_primos.c -lm
$ mpirun -np 4 ./mpi_primos 100000000
```

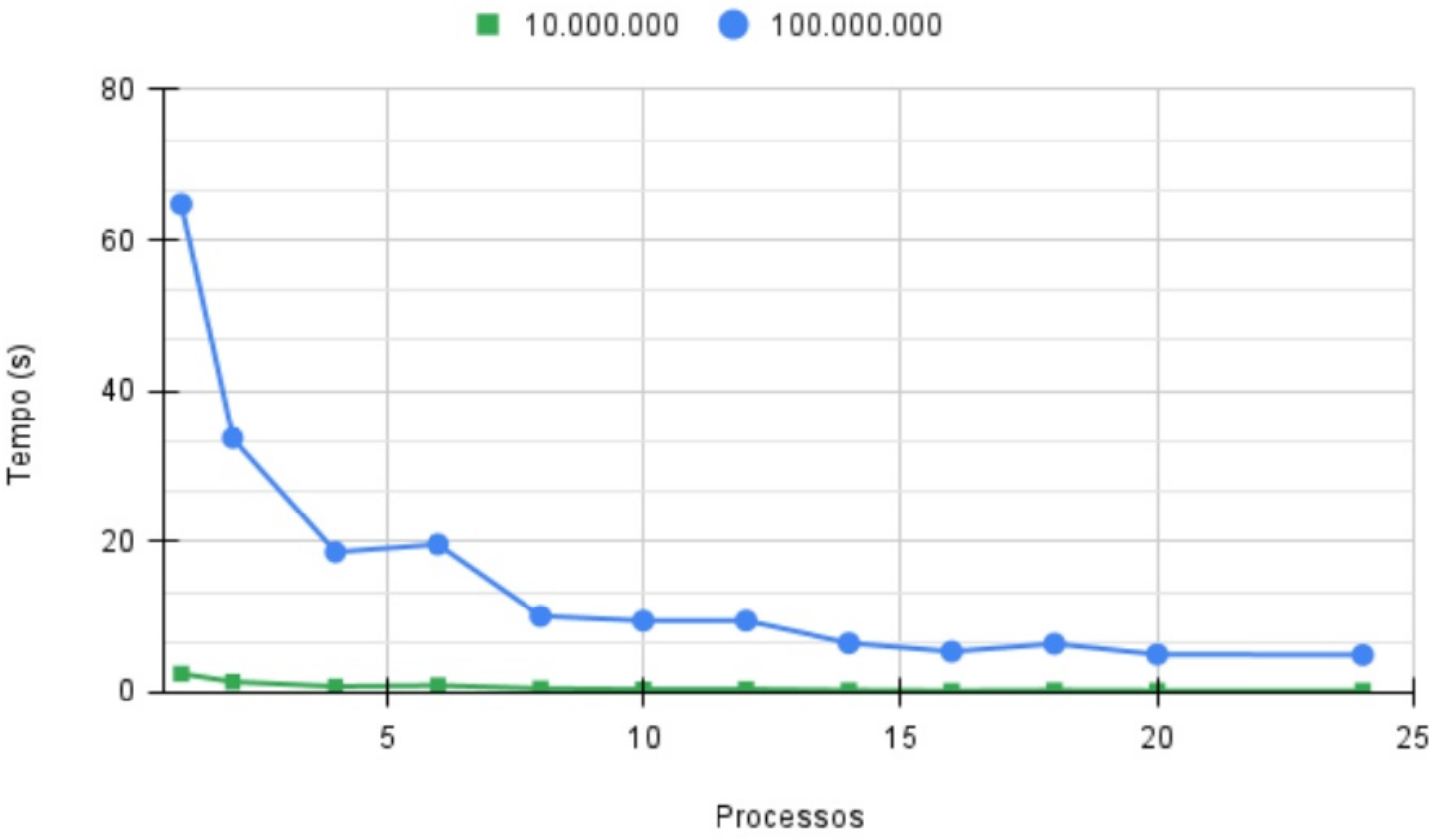
Prime Numbers
Naive Solution

https://github.com/gpsilva2003/MPI/mpi_primos.c

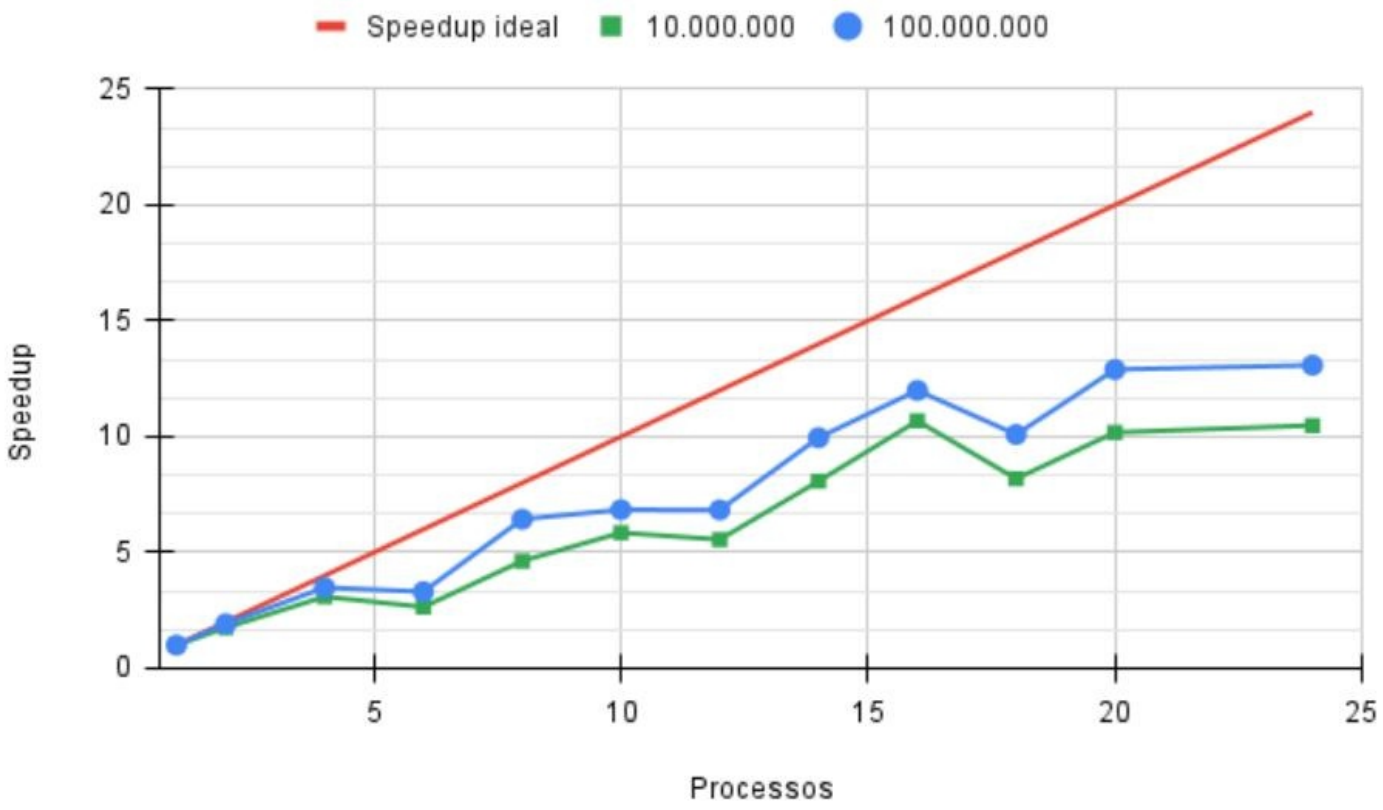
Prime Numbers
Sequential Solution

https://github.com/gpsilva2003/SERIAL/blob/main/src/primos_seq.c

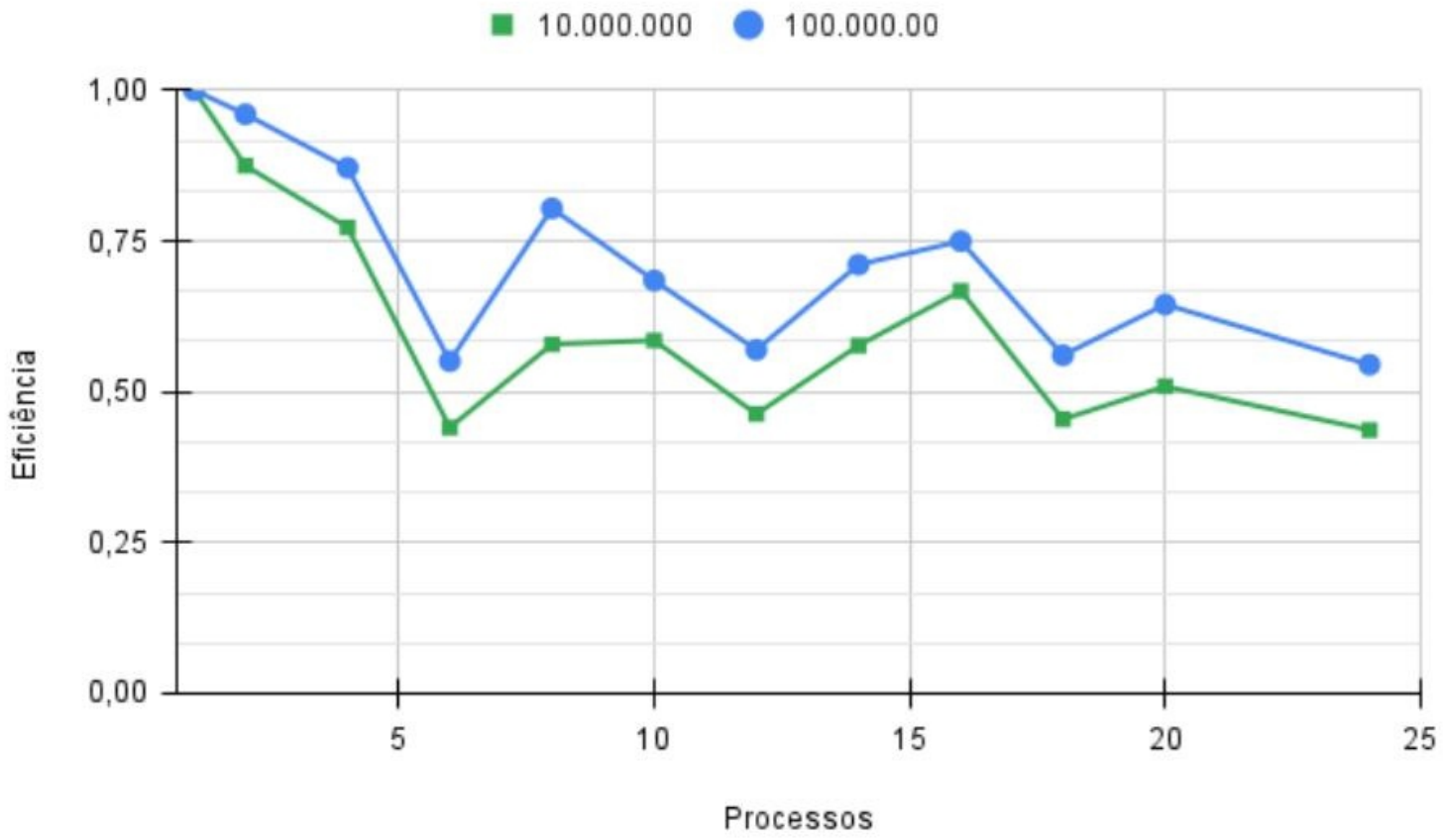
Primes – Naive – Execution Time



Primos – Naive – Speedup



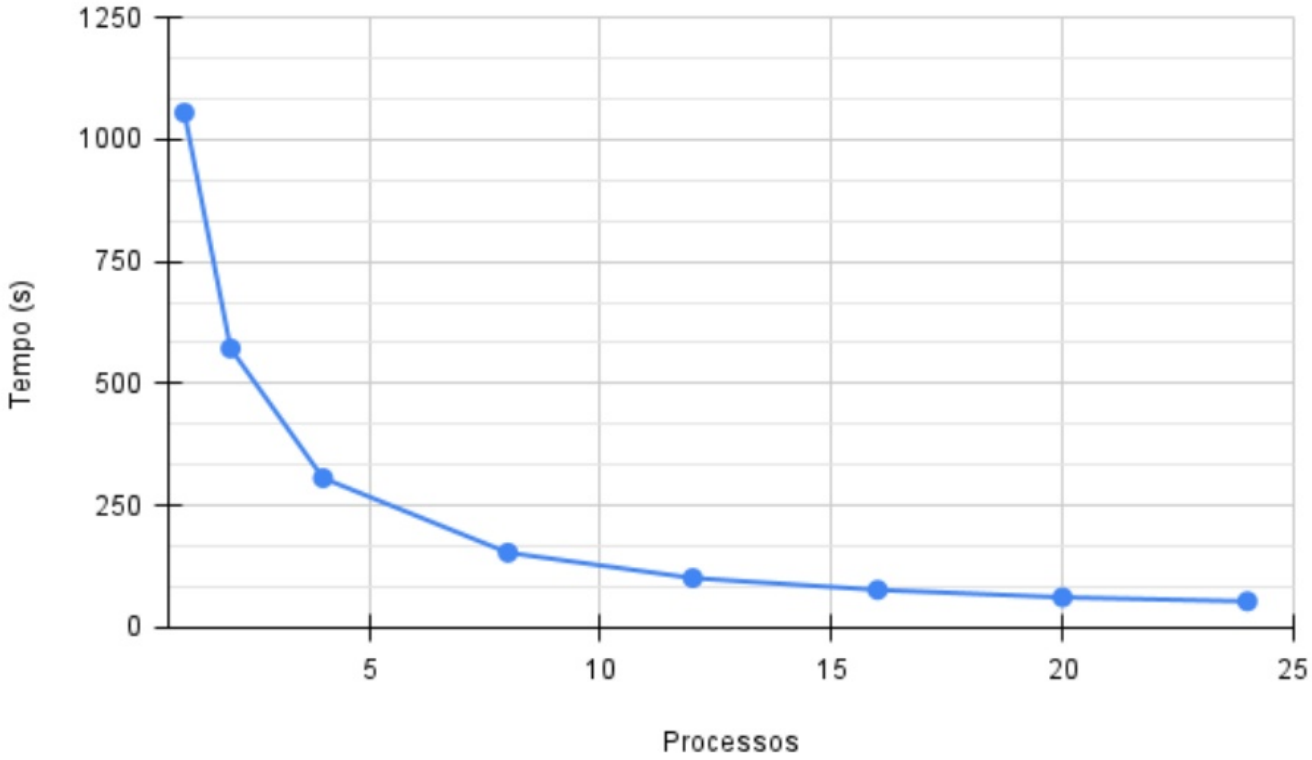
Primos – Naive – Efficiency



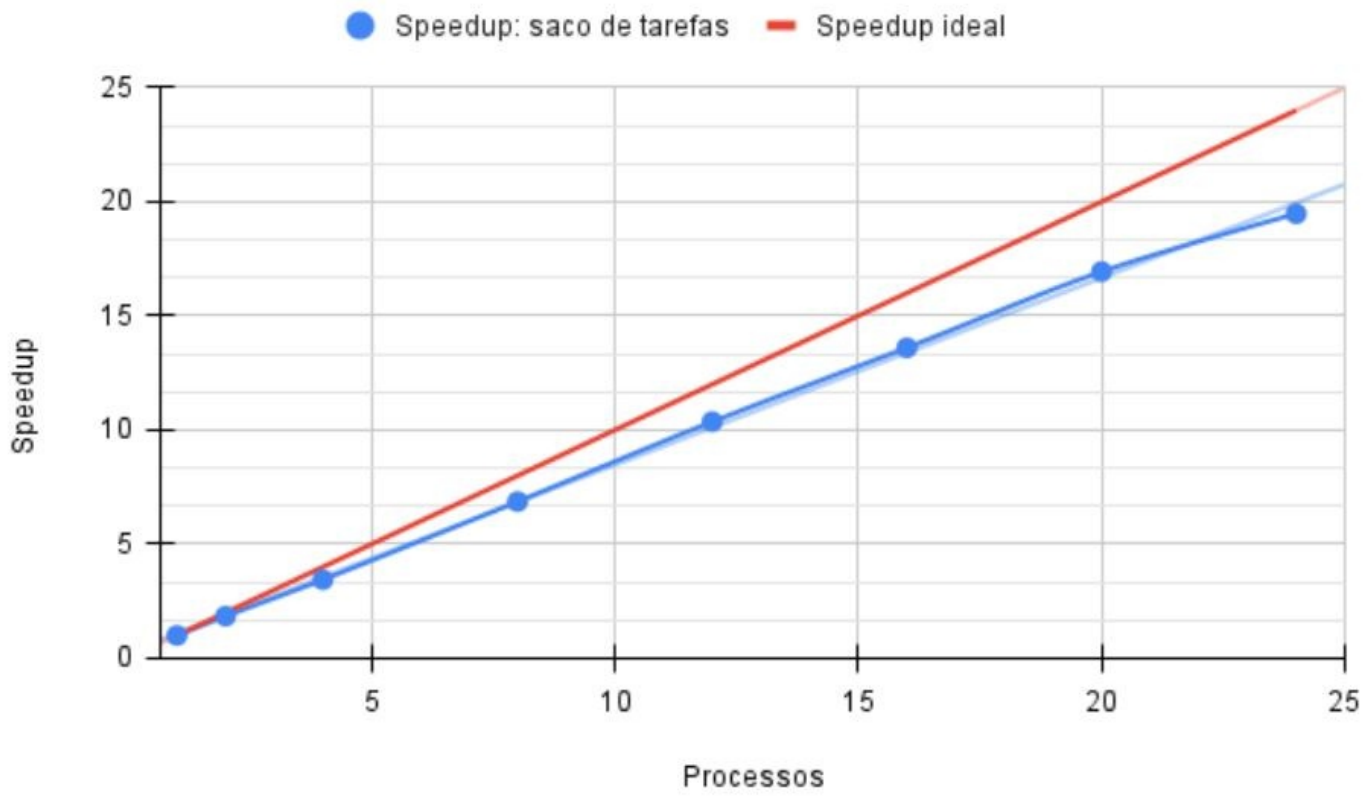
Prime Numbers
Bag of tasks

https://github.com/gpsilva2003/MPI/mpi_primosbag.c

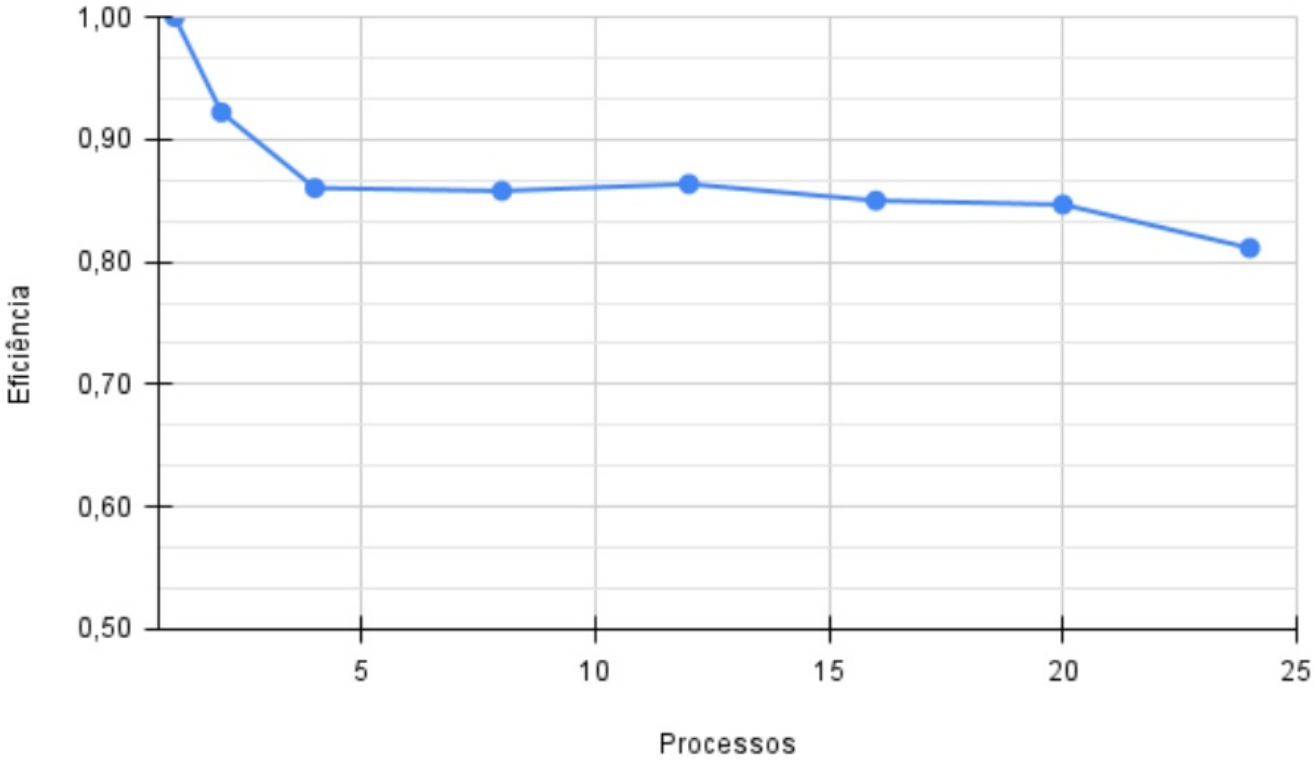
Primes – Bag of Tasks
Execution Time



Primes – Bag of Tasks
Speedup



Primes – Bag of Tasks
Efficiency



OpenMPI

OpenMPI

- You can install openmpi directly from OS repositories
- Fedora:

  $ sudo dnf install openmpi

- Ubuntu:

  $ sudo apt-get update

  $ sudo apt-get install openmpi-bin openmpi-doc

- WSL:

  $ sudo apt-get update

  $ sudo apt install openmpi-bin libopenmpi-dev

https://docs.microsoft.com/en-us/windows/wsl/install

OpenMPI

- The following enviroment variables must be configured:

  PATH: /usr/lib64/openmpi/bin
  LD_LIBRARY_PATH: /usr/lib64/openmpi

- To: verify all the features available

  $ ompi_info

OpenMPI

- You need type the following command to compile a source file named prog.c:

  $ mpicc -o prog prog.c

- To execute this program, let's say, with 4 process, you need type either:

  $ mpirun -n 4 prog

- Or:

  $ mpiexec -n 4 prog

OpenMPI

- The mpiexec/mpirun commands allow more elaborated options:

```
$ mpiexec -n 1 --host paraty: -n 19 slave
```

- Dispatch the process with rank equal 0 in the machine with name paraty and other 19 processes divided between the remaing machines.

OpenMPI

- Running OpenMPI with multiple machines:

```
$ mpirun --hostfile my_hostfile -np 8 parallel_app
```

- Where my_hostfile is a file containing the name or IP from the machines where you want to run the program

- To know more please visit:

https://www.open-mpi.org/faq/

References

1) Gabriel P. Silva, Calebe P. Bianchini e Evaldo B. Costa “Programação Paralela e Distribuída com MPI, OpenMP e OpenACC para computação de alto desempenho ” Editora Casa do Código, 2022
2) Neil MacDonald et alii, Writing Message Passing Programs with MPI, Edinburgh Parallel Computer Centre
3) Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, 2nd ed., Englewood Cliffs, NJ, Prentice--Hall, 1988.
4) Peter S. Pacheco, Parallel Programming with MPI, Morgan Kaufman Pub, 1997.
5) Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 3.1, 2015 Acesso em https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

Thanks!

Gabriel P. Silva

gabriel@ic.ufrj.br

http://github.com/gpsilva2003