Programming Distributed Memory Machines with MPI

Ricardo Rocha and Fernando Silva

Computer Science Department
Faculty of Sciences
University of Porto

Parallel Computing 2017/2018

Programming Distributed Memory Machines

Applications are seen as a set of programs (or processes), each with its own memory, that **execute independently in different machines**. Cooperation between the programs is achieved by **exchanging messages**.

Programmers must control synchronization and operation details of the application but, do not want to waist too much (any?) time with the **low level communication details**.

Message passing libraries extend existing languages and provide programmers with an implementation abstraction for communication details without the programmer having to explicitly know how that is accomplished at the network level. They usually support/include:

- Remote execution of programs
- Send/receive message support between the programs
- Tools to monitor the execution state of the programs

Message Passing Libraries

Communication:

- No shared variables
- Pairwise or point-to-point functions to send and receive messages
- Collective functions to move data (broadcast, scatter, gather) and to resume data (reduce) from all/several programs

Synchronization:

- No locks (there are no shared variables to protect)
- Synchronous messages
- Barrier mechanisms

Inquiries:

- How many processes?
- Which one am I?
- Any messages waiting?

Message Passing Libraries

Many message passing libraries have been proposed:

- PVM, Parallel Virtual Machine (Oak Ridge National Laboratory, University of Tennessee)
- ACL Message Passing Library (Advanced Computing Lab, Los Alamos National Laboratory)
- CMMD (Thinking Machines Corporation)
- MPL, Message Passing Library (IBM SP2)
- NX Message Passing (Intel Paragon)
- ...

But, nowadays **MPI (Message Passing Interface)** is the industry standard.

Message Passing Interface (MPI)

Started in 1992 as a cooperation between universities and industries from Europe and the United States:

- First published in 1994 (**MPI-1**)
- Extensions have been proposed to handle dynamic execution and parallel IO (**MPI-2**) and non-blocking collectives (**MPI-3**)

MPI is just a specification (**MPI Forum** – `http://www.mpi-forum.org`)

- Not a programming language
- Not a implementation
- Initial libraries implemented only for the C/C++ and Fortran languages (now also for Perl, Python, Ruby, OCaml, Java, R, ...)

Major implementations:

- **MPICH** (`http://www.mcs.anl.gov/mpi/mpich`)
- **OpenMPI** (`hhtp://www.open-mpi.org`)

Goals and Novel Features of MPI

Main goals:

- Increase program's portability
- Increase and improve functionality
- Achieve efficient implementations on several different architectures
- Support heterogeneous environments

Novel Features:

- **Communicators** encapsulate communication spaces for library safety
- **Data types** reduce copying costs and permit heterogeneity
- Multiple **communication modes** allow precise buffer management
- Extensive **collective operations** for scalable global communication
- **Topologies** encapsulate different user views of process layout and permits efficient process placement
- **Profiling** interface encourages portable tools

Single Program Multiple Data (SPMD)

SPMD is a programming model in which all components that make the parallel application are included in **just one executable**. Each running process can then determine its own **rank** among all processes and thus separate its execution flow from the others when needed.

```
...
if (my_rank == 0) {
   // code for process 0
}
...
else if (my_rank == N) {
   // code for process N
}
...
```

MPI does not impose any constraint on the programming model and SPMD is just a possible option, but a more portable one.

MPI Execution Environment

A program initiates the MPI execution environment with a call to:

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

and terminates the MPI execution environment by calling:

```
MPI_Finalize(void)
```

All MPI functions return 0 if OK or a positive value in case of error.

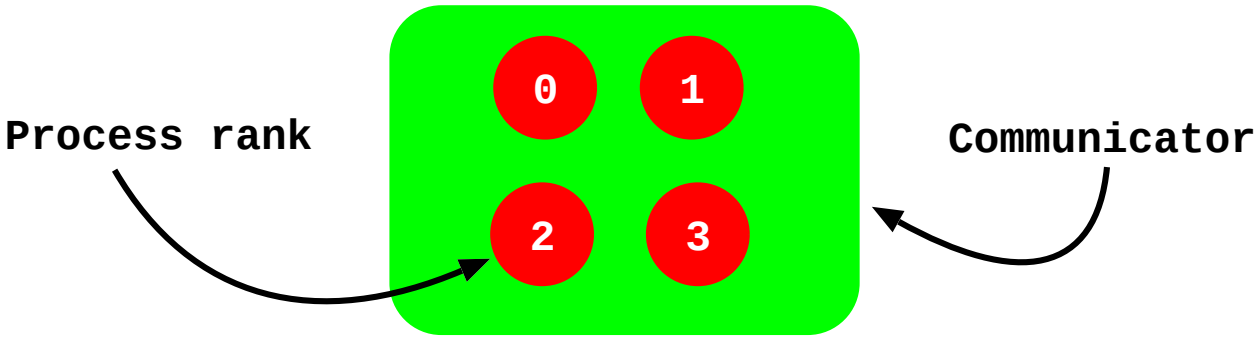
General Structure of a MPI Program

```
// include library of MPI function calls
#include <mpi.h>

main(int argc, char **argv) {
  ...
  // no MPI calls before this point
  MPI_Init(&argc, &argv);

  ...

  MPI_Finalize();
  // no MPI calls after this point
  ...
}
```

Communicators

A MPI program sees its execution environment as **groups of processes**:

- The **communicator** data structure encapsulates the concept of group of processes and defines a **communication space for the set of processes in a group**
- All processes have a unique identifier, named **rank**, that determines their position (from 0 to N-1) within the communicator



Communicators

All communication functions take place within the context of a communicator:

- By default, the MPI execution environment sets a universal communicator `MPI_COMM_WORLD` including all processes in execution
- A process can be part of more than one communicator and assume different rankings in each of them

Getting Information About Communicators

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

`MPI_Comm_rank()` returns in `rank` the position of the current process in the communicator `comm`.

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

`MPI_Comm_size()` returns in `size` the number of processes participating in the communicator `comm`.

MPI Messages

In its essence, messages are just data packets being exchanged among processes. For a message to be exchanged, the MPI execution environment needs to know at least the following data:

- Sender process
- Receiver process
- Location of data at origin
- Location of data at destination
- Size of data
- Type of data

As we will see, a very relevant information in MPI messages is the **type of data**.

Basic Data Types

MPI	C
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_PACKED	

Sending Messages

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

`MPI_Send()` is the basic function for sending messages:

- `buf` is the starting address of the data to be sent
- `count` is the number of elements of type `datatype` to be sent
- `datatype` is the type of data to be sent
- `dest` is the rank of the receiver process within communicator `comm`
- `tag` is an identification tag for the message being sent, which allows to group/distinguish the messages being exchanged
- `comm` is the communicator for the processes involved in the communication

Receiving Messages

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

`MPI_Recv()` is the basic function for receiving messages:

- `buf` is the starting address where received data must be placed
- `count` is the maximum number of elements of type `datatype` to be received (must be $\geq$ to the number of elements being sent)
- `datatype` is the type of data to be received

Receiving Messages

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

- `source` is the rank of the sending process within communicator `comm` (`MPI_ANY_SOURCE` allows to receive from any process)
- `tag` is the identification tag for the message being received (`MPI_ANY_TAG` allows to receive any message)
- `comm` is the communicator for the processes involved
- `status` returns information about the sending process and message tag (`status.MPI_SOURCE` and `status.MPI_TAG`) (if not important, can be ignored using `MPI_STATUS_IGNORE`)

Getting Information About Received Messages

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

`MPI_Get_count()` returns in `count` the number of elements of type `datatype` received in the message associated with `status`.

```
MPI_Probe(int source, int tag, MPI_Comm comm,
  MPI_Status *status)
```

`MPI_Probe()` synchronizes the reception of the next message, by returning in `status` information about the message, but without receiving it:

- To effectively receive the message, a call to `MPI_Recv()` is required
- Useful when we do not know beforehand the size of the message, thus allowing to avoid overflowing the receiving buffer

I'm Alive! (mpi-alive.c)

```
#define MY_TAG 0

main(int argc, char **argv) {
  int i, my_rank, n_procs; char msg[100]; MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
  if (my_rank != 0) {
    sprintf(msg, "I'm Alive!");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 0, MY_TAG, MPI_COMM_WORLD);
  } else {
    for (i= 1; i < n_procs; i++) {
      MPI_Recv(msg, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
               MPI_COMM_WORLD, &status);
      printf("%d: %s\n", status.MPI_SOURCE, msg);
    }
  }
  MPI_Finalize();
}
```

Communication Modes

MPI allows different communication modes for sending messages:

- Standard send: `MPI_Send()`
- Synchronous send: `MPI_Ssend()`
- Buffered send: `MPI_Bsend()`

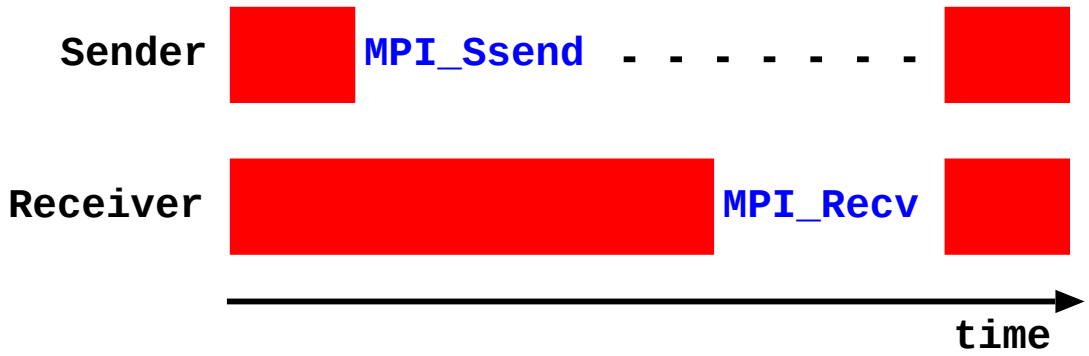
In any case, message ordering is always preserved and reception should be done using `MPI_Recv()`. If a process A sends N messages to a process B by making N calls to `MPI_Send()`/`MPI_Ssend()`/`MPI_Bsend()` and process B makes N calls to `MPI_Recv()` to receive the N messages, the MPI execution environment ensures that the 1^{st} send call is matched with the 1^{st} receive call, the 2^{nd} send call is matched with the 2^{nd} receive call, and so on.

Synchronous Send

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

Sender waits to transmit until the receiver confirms it is ready to receive:

- Although this communication mode may be useful in certain cases, its use **delays the sender until the receiver is ready**
- It should be used only when the sender needs to ensure that the message has been received before proceeding with execution

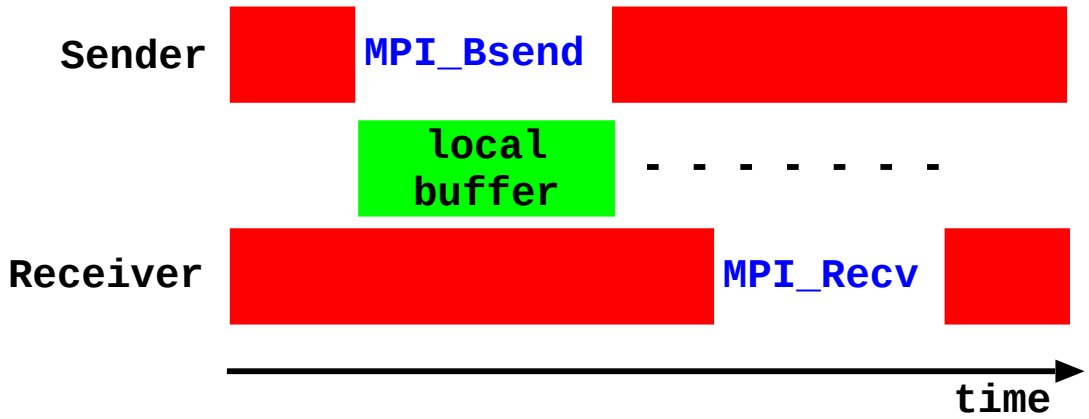


Buffered Send

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

The message is first copied into a local buffer and only then sent to the receiver:

- The sender does not depends on synchronizing with the receiver and can continue its execution without any delay
- Requires the association of an **explicit local buffer** to the sender



Attaching and Detaching a Local Buffer

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

`MPI_Buffer_attach()` tells the MPI execution environment that the memory space starting at `buf` and with size `size` can be used for local buffering of messages. At any instant, only one local buffer can be attached to a process.

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

`MPI_Buffer_detach()` tells the MPI execution environment to stop using the buffer pointed by `buf` for local buffering of messages. If there are pending messages in the buffer, it **returns only when all messages have been delivered**. `MPI_Buffer_detach()` does not free the buffer's memory, and for that one must call the `free()` system call.

Welcome! (mpi-welcome.c)

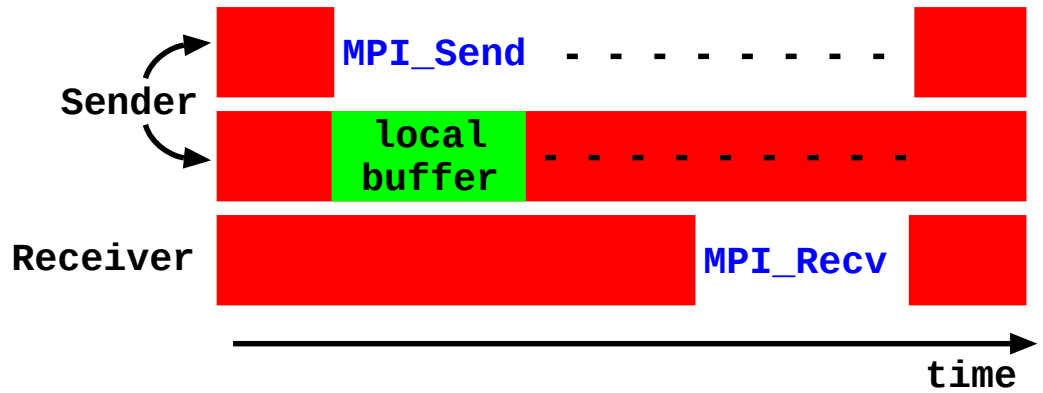
```
main(int argc, char **argv) {
  int buf_size; char *local_buf;
  ...
  buf_size = BSIZE; local_buf = (char *) malloc(buf_size);
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
  MPI_Buffer_attach(local_buf, buf_size);
  sprintf(msg, "Welcome!");
  for (i = 0; i < n_procs; i++) if (i != my_rank)
    MPI_Bsend(msg, strlen(msg)+1, MPI_CHAR, i, MY_TAG, MPI_COMM_WORLD);
  for (i = 0; i < n_procs; i++) if (i != my_rank) {
    sprintf(msg, "Argh!");
    MPI_Recv(msg, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
    printf("%d->%d: %s\n", status.MPI_SOURCE, my_rank, msg);
  }
  MPI_Buffer_detach(&local_buf, &buf_size);
  free(local_buf);
  MPI_Finalize();
}
```

Standard Send

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

Terminates when the message is sent but this does not imply that it has been delivered to the receiver. The MPI execution environment may keep the message on hold for a while in a local buffer:

- Typically, small messages are buffered while larger messages are synchronized (**MPI implementation dependent**)
- For portability, programmers should assume synchronization



Simultaneous Send and Receive

```
MPI_Sendrecv(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,
  int recvcount, MPI_Datatype recvtype, int source,
  int recvtag, MPI_Comm comm, MPI_Status *status)
```

`MPI_Sendrecv()` allows for the simultaneous sending and receiving of messages. Useful when one wants circular communications on a set of processes, thus avoiding mismatches and potential deadlocks:

- `sendbuf` is the starting address of the data to be sent
- `sendcount` is the number of elements of type `sendtype` to be sent
- `sendtype` is the type of data to be sent
- `dest` is the rank of the receiver process within communicator `comm`
- `sendtag` is the identification tag for the message being sent

Simultaneous Send and Receive

```
MPI_Sendrecv(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,
  int recvcount, MPI_Datatype recvtype, int source,
  int recvtag, MPI_Comm comm, MPI_Status *status)
```

- `recvbuf` is the starting address where received data must be placed
- `recvcount` is the maximum number of elements of type `recvtype` to be received
- `recvtype` is the type of data to be received
- `source` is the rank of the sending process within communicator `comm`
- `recvtag` is the identification tag for the message being received
- `comm` is the communicator for the processes involved
- `status` returns information about the sending process

Simultaneous Send and Receive

```
MPI_Sendrecv(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf,
  int recvcount, MPI_Datatype recvtype, int source,
  int recvtag, MPI_Comm comm, MPI_Status *status)
```

Important aspects of the `MPI_Sendrecv()` communication mode:

- The buffers `sendbuf` and `recvbuf` must be different
- The tags `sendtag` and `recvtag`, the sizes `sendcount` and `recvcount`, and the types of data `sendtype` and `recvtype`, can be different
- A message that is sent using `MPI_Sendrecv()` can be received by any other receiving method
- A message that is received with `MPI_Sendrecv()` may have been sent by any other sending method

Simultaneous Send and Receive

```
MPI_Sendrecv_replace(void *buf, int count,
  MPI_Datatype datatype, int dest, int sendtag, int source,
  int recvtag, MPI_Comm comm, MPI_Status *status)
```

`MPI_Sendrecv_replace()` allows for the simultaneous sending and receiving of messages **using the same buffer to send and to receive**:

- At the end of the communication, the message being sent is replaced by the one being received
- The buffer `buf`, the size `count` and the type of data `datatype` are used to define both the messages being sent and being received
- A message that is sent using `MPI_Sendrecv_replace()` can be received by any other receiving method
- A message that is received with `MPI_Sendrecv_replace()` may have been sent by any other sending method

Non-Blocking Communications

A communication is said to be **blocking** if it suspends the execution until the communication succeeds. A blocking communication succeeds when the message buffer associated with the communication can be reused.

A communication is said to be **non-blocking** if the continuation of the execution does not depend on the success of the communication. Nevertheless, the message buffer associated with the communication should not be reused until the communication succeeds.

- The advantage of non-blocking communication is to start sending the messages as early as possible and only later verify their success
- The call to a non-blocking function returns immediately since it only announces to the MPI execution environment the existence of a message to be sent or received
- The communication completes when, in a later moment, the process gets to know the success of the communication

Non-Blocking Send and Receive

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

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

Both functions return in **req** the identifier that permits later verification on the success of the communication.

Non-Blocking Probing of Messages

```
MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
  MPI_Status *status)
```

`MPI_Iprobe()` checks (without blocking) for the arrival of a message associated with `source`, `tag` and `comm` without receiving it:

- Returns in `flag` the logical value that indicates the arrival of some message, and `status` provides information about it
- To receive the message, one has to use a receiving function

Success of Non-Blocking Communications

```
MPI_Wait(MPI_Request *req, MPI_Status *status)
```

`MPI_Wait()` blocks the calling process until the communication identified by `req` succeeds. Returns in `status` information about the message.

```
MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)
```

`MPI_Test()` tests whether the communication identified by `req` has succeeded. Returns in `flag` the logical value that indicates the success of the communication and, in case of success, returns in `status` information about the message.

Hello! (mpi-hello.c)

```
main(int argc, char **argv) {
  char recv_msg[100]; MPI_Request req[100];
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
  for (i = 0; i < n_procs; i++) if (my_rank != i) {
    sprintf(msg, "Hello proc %d!", i);
    MPI_Irecv(recv_msg, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
              MPI_COMM_WORLD, &(req[i]));
    MPI_Isend(msg, strlen(msg)+1, MPI_CHAR, i, MY_TAG, MPI_COMM_WORLD,
              &(req[i + n_procs]));
  }
  for (i = 0; i < n_procs; i++) if (my_rank != i) {
    sprintf(recv_msg, "Argh!");
    MPI_Wait(&(req[i]), &status);
    printf("%d->%d: %s\n", status.MPI_SOURCE, my_rank, recv_msg);
    MPI_Wait(&(req[i + n_procs]), &status);
  }
  MPI_Finalize();
}
```

Which Communications Should I Use?

Most users use standard communications to send and receive messages, specially when the MPI implementation is efficient. However, their use do not guarantee the functionality and portability of the application.

Alternatively, the use of **synchronous and non-blocking standard communications** is sufficient to build robust applications.

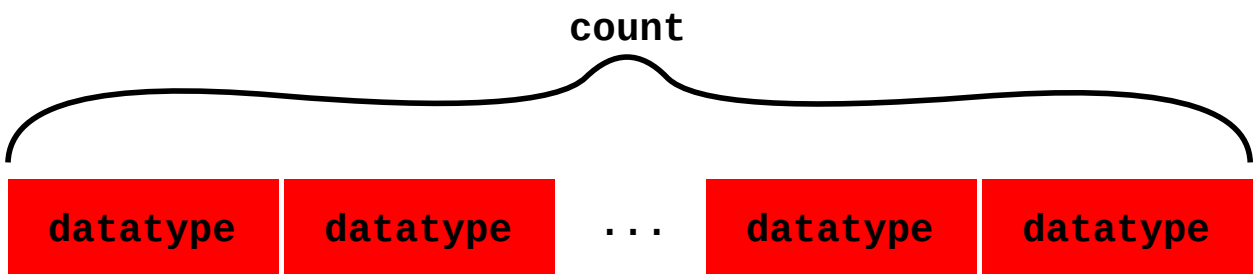
Non-blocking communications do not always lead to the best results. Their use should only be considered when there is a clear computation overlap.

Very commonly, sending is done with non-blocking functions and receiving with blocking functions.

Grouping Data for Communication

With message passing, a natural heuristic to maximize performance is to **minimize the number of messages being exchanged**:

- By default, all sending and receiving functions allow grouping in a single message, data of the same type that is contiguously stored in memory



On top of this basic functionality, MPI allows one to:

- Define new data types that group data of various types
- Pack and unpack data into/from a buffer

Derived Data Types

MPI allows the dynamic definition (during execution time) of new data types **built from the existing basic data types**:

- Initially, all processes must build the derived data types
- Then, they must make the derived data type known to the MPI execution environment
- When the derived data type is no longer needed, each process frees it from the MPI execution environment

Derived data types are **used in communication messages similarly to the basic data types**. Thus, sender and receiver must know about them. This is normally accomplished in the common part of the code.

There is a cost in building derived data types, hence it should only be used when one expects a significant number of messages to be exchanged.

Derived Data Types for Regular Data Intervals

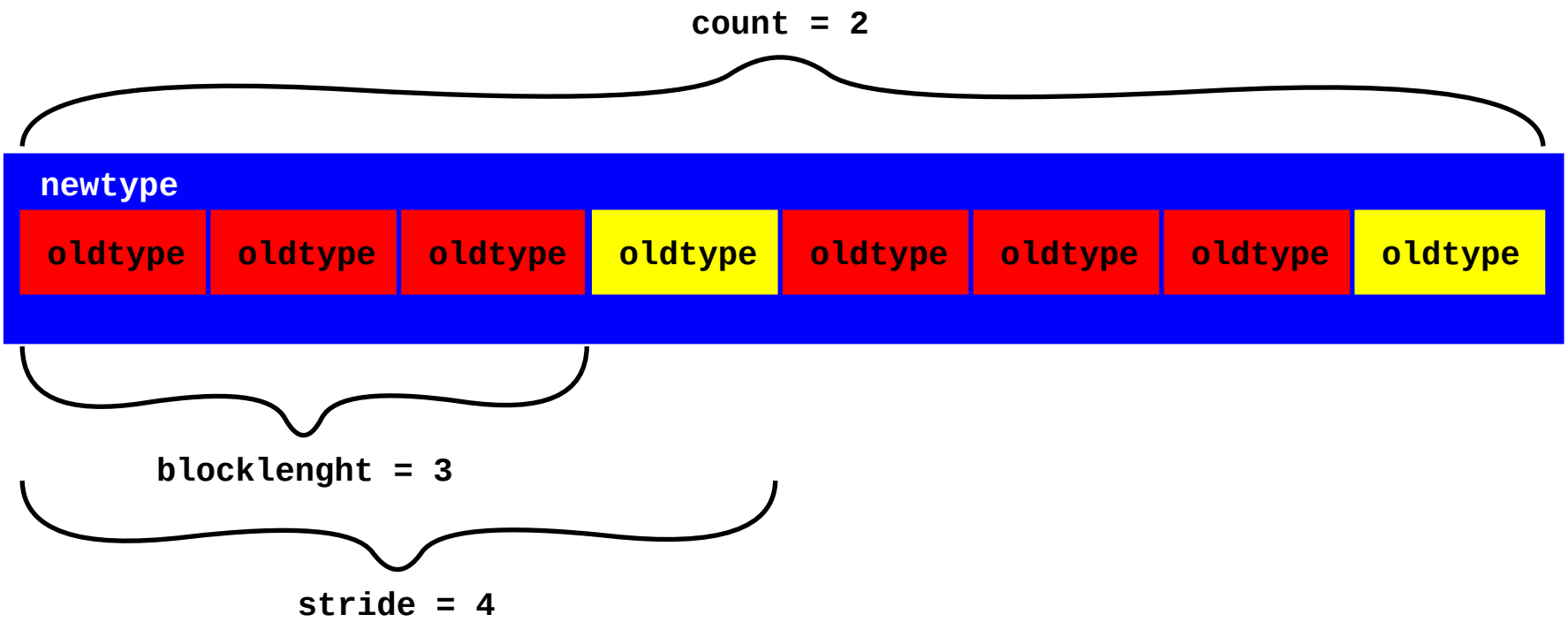
```
MPI_Type_vector(int count, int blocklength, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

`MPI_Type_vector()` builds a new data type from an existing vector of data:

- `count` is the number of data blocks in the new derived data type
- `blocklength` is the number of contiguous elements in each block
- `stride` is the number of contiguous elements that separate the start of each block (i.e. the displacement)
- `oldtype` is the data type of the elements in the existing vector
- `newtype` is the identifier of the new derived data type

Derived Data Types for Regular Data Intervals

```
MPI_Type_vector(int count, int blocklength, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```



Derived Data Types for Heterogeneous Data

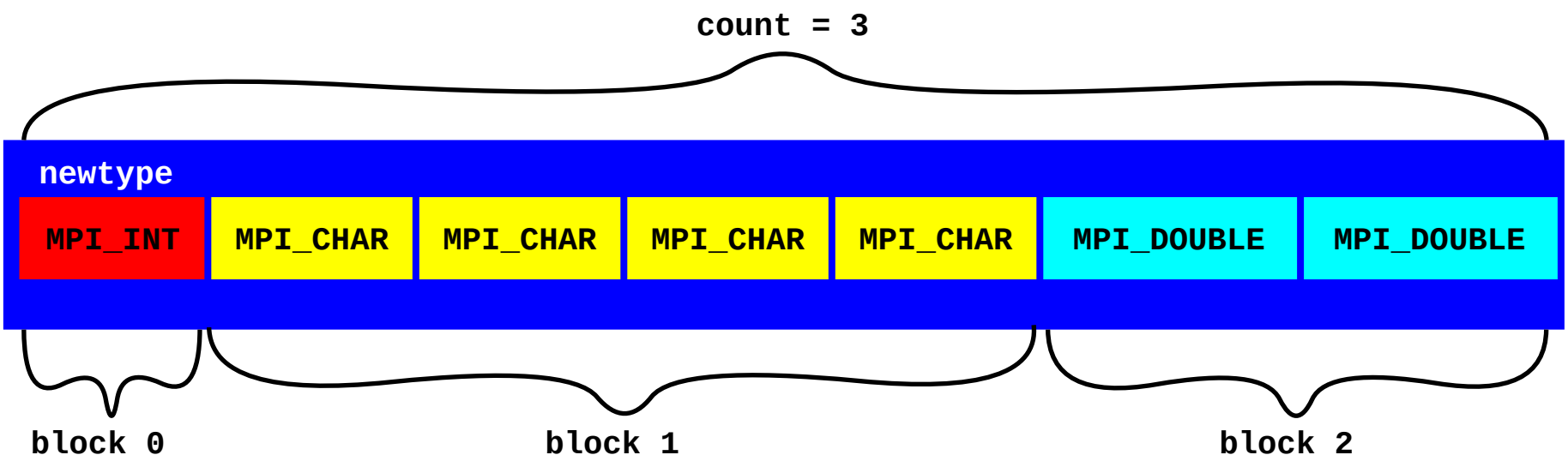
```
MPI_Type_struct(int count, int blocklengths[],
  MPI_Aint displacements[], MPI_Datatype oldtypes[],
  MPI_Datatype *newtype)
```

`MPI_Type_struct()` builds a new data type from a data structure that may include different basic data types:

- `count` is the number of data blocks in the new derived data type (it is also the number of items in `blocklengths[]`, `displacements[]` and `oldtypes[]`)
- `blocklengths[]` gives the number of contiguous elements in each block
- `displacements[]` gives the starting position, in *bytes*, of each block
- `oldtypes[]` gives the data type of the elements in each block
- `newtype` is the identifier of the new derived data type

Derived Data Types for Heterogeneous Data

```
MPI_Type_struct(int count, int blocklengths[],
  MPI_Aint displacements[], MPI_Datatype oldtypes[],
  MPI_Datatype *newtype)
```




```
blocklengths[3] = {1, 4, 2}
displacements[3] = {0, int_length, int_length + 4 * char_length}
oldtypes[3] = {MPI_INT, MPI_CHAR, MPI_DOUBLE}
```

Getting Information for Derived Data Types

A derived data type represents a collection of data items in memory that specifies both the basic types of the items and their relative locations in memory. To specify the locations, one needs to determine the **size in bytes** of a data type.

```
MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

`MPI_Type_extent()` returns in `extent` the size in bytes of `datatype`.

```
MPI_Address(void *location, MPI_Aint *address)
```

`MPI_Address()` returns in `address` the memory address of `location`.

Certifying Derived Data Types

```
MPI_Type_commit(MPI_Datatype *datatype)
```

`MPI_Type_commit()` certifies within the MPI execution environment the existence of a new derived type identified by `datatype`.

```
MPI_Type_free(MPI_Datatype *datatype)
```

`MPI_Type_free()` frees from the MPI execution environment the derived type identified by `datatype`.

Handling Columns of Matrices (mpi-typevector.c)

```
MPI_Datatype colMatrix;
int my_matrix[ROWS][COLS];
int my_vector[ROWS];
...
// build a derived datatype with ROWS blocks
// of 1 element separated by COLS elements
MPI_Type_vector(ROWS, 1, COLS, MPI_INT, &colMatrix);
MPI_Type_commit(&colMatrix);
...
// send column 1 of my_matrix
MPI_Send(&my_matrix[0][1], 1, colMatrix, dest, tag, comm);
...
// receive a given column of data in column 3 of my_matrix
MPI_Recv(&my_matrix[0][3], 1, colMatrix, src, tag, comm, &status);
...
// receive a given column of data in my_vector
MPI_Recv(&my_vector[0], ROWS, MPI_INT, src, tag, comm, &status);
...
// free the derived datatype
MPI_Type_free(&colMatrix);
```

Handling Data Structures (mpi-typestruct.c)

```
struct { int a; char b[4]; double c[2]; } my_struct;
MPI_Datatype strType, oldtypes[3];
MPI_Aint int_length, char_length, offsets[3];
int blocklengths[3];
...
// build a derived datatype representing my_struct
MPI_Type_extent(MPI_INT, &int_length);
MPI_Type_extent(MPI_CHAR, &char_length);
blocklengths[0] = 1; blocklengths[1] = 4; blocklengths[2] = 2;
oldtypes[0] = MPI_INT; oldtypes[1] = MPI_CHAR;
oldtypes[2] = MPI_DOUBLE;
offsets[0] = 0; offsets[1] = int_length;
offsets[2] = int_length + 4 * char_length;
MPI_Type_struct(3, blocklengths, offsets, oldtypes, &strType);
MPI_Type_commit(&strType);
...
// send my_struct
MPI_Send(&my_struct, 1, strType, dest, tag, comm);
...
// receive in my_struct
MPI_Recv(&my_struct, 1, strType, src, tag, comm, &status);
```

Packing Data

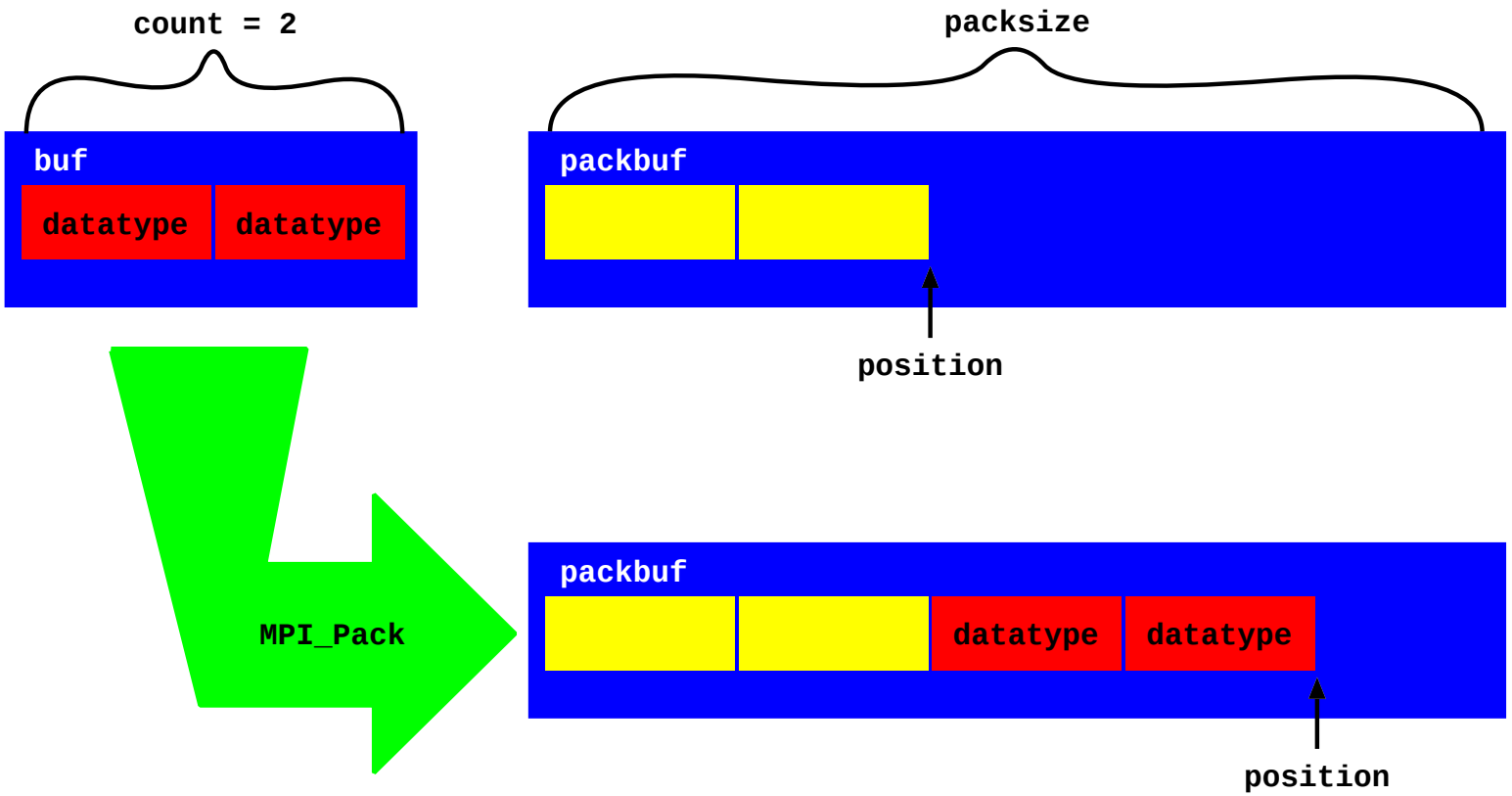
```
MPI_Pack(void *buf, int count, MPI_Datatype datatype,
  void *packbuf, int packsize, int *position, MPI_Comm comm)
```

`MPI_Pack()` packs non-contiguous data into contiguous memory positions:

- `buf` is the starting address of the data to be packed
- `count` is the number of elements of type `datatype` to be packed
- `datatype` is the type of data to be packed
- `packbuf` is the starting address of the packing buffer
- `packsize` is the size in bytes of the packing buffer
- `position` is the buffer position (in bytes) from where the data should be packed
- `comm` is the communicator for the processes involved

Packing Data

```
MPI_Pack(void *buf, int count, MPI_Datatype datatype,
  void *packbuf, int packsize, int *position, MPI_Comm comm)
```



Unpacking Data

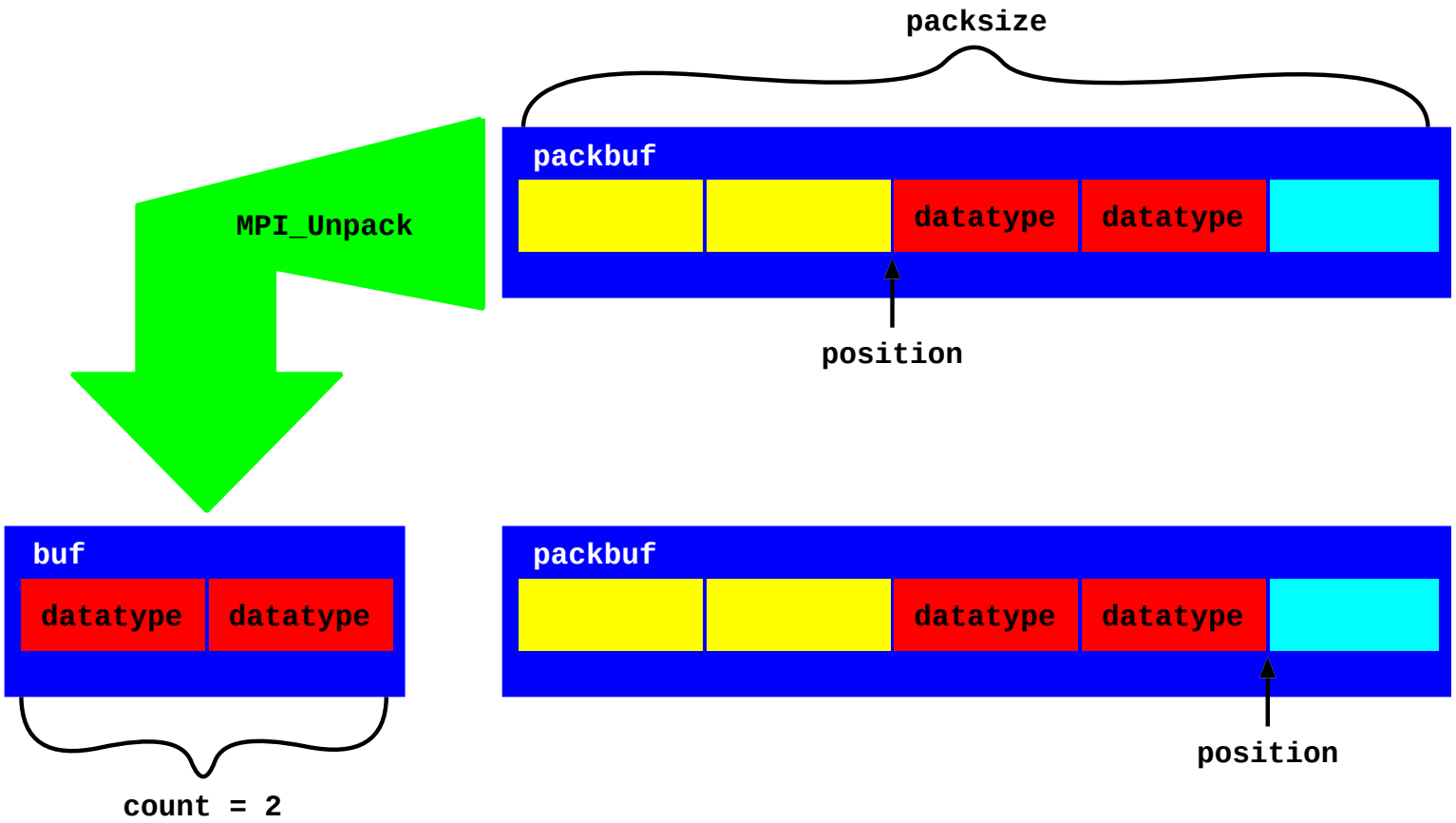
```
MPI_Unpack(void *packbuf, int packsize, int *position,
  void *buf, int count, MPI_Datatype datatype, MPI_Comm comm)
```

`MPI_Unpack()` unpacks contiguous data into non-contiguous positions in memory:

- `packbuf` is the starting address of the packing buffer
- `packsize` is the size in bytes of the packing buffer
- `position` is the buffer position (in bytes) from where the data should be unpacked
- `buf` is the starting address to where the data should be unpacked
- `count` is the number of elements of type `datatype` to be unpacked
- `datatype` is the type of data to be unpacked
- `comm` is the communicator for the processes involved

Unpacking Data

```
MPI_Unpack(void *packbuf, int packsize, int *position,
  void *buf, int count, MPI_Datatype datatype, MPI_Comm comm)
```



Matrix of Variable Size (mpi-pack.c)

```
// initially ROWS and COLS are not known to process 1
int *my_matrix, ROWS, COLS, pos;
char buf[BSIZE]; // packing buffer
...
if (my_rank == 0) {
  // pack and send ROWS, COLS and my_matrix
  pos = 0;
  MPI_Pack(&ROWS, 1, MPI_INT, buf, BSIZE, &pos, comm);
  MPI_Pack(&COLS, 1, MPI_INT, buf, BSIZE, &pos, comm);
  MPI_Pack(my_matrix, ROWS * COLS, MPI_INT, buf, BSIZE, &pos, comm);
  MPI_Send(buf, pos, MPI_PACKED, 1, tag, comm);
} else if (my_rank == 1) {
  // receive and unpack ROWS, COLS and my_matrix
  pos = 0;
  MPI_Recv(&buf, BSIZE, MPI_PACKED, 0, tag, comm, &status);
  MPI_Unpack(&buf, BSIZE, &pos, &ROWS, 1, MPI_INT, comm);
  MPI_Unpack(&buf, BSIZE, &pos, &COLS, 1, MPI_INT, comm);
  // allocate space to represent my_matrix
  my_matrix = (int *) malloc(ROWS * COLS * sizeof(int));
  MPI_Unpack(&buf, BSIZE, &pos, my_matrix, ROWS * COLS, MPI_INT, comm);
}
```

Which Data Types Should I Use?

If data is of the same type and found in contiguous memory positions then just use the `count` argument of the sending and receiving functions.

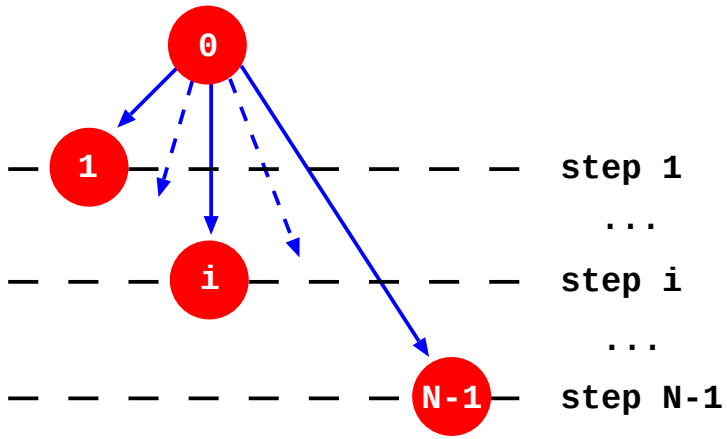
If data is of the same type but not found contiguously in memory, then one should create a derived type using `MPI_Type_vector()` (for data separated by regular intervals) or `MPI_Type_indexed()` (for data separated by irregular intervals).

If data is heterogeneous and shows a regular pattern, then one should create a derived type using `MPI_Type_struct()`.

If data is heterogeneous but not shows any regular pattern, then one should use `MPI_Pack()` and `MPI_Unpack()`. `MPI_Pack()` and `MPI_Unpack()` can also be used to exchange heterogeneous data once (or just a few times).

Collective Communications

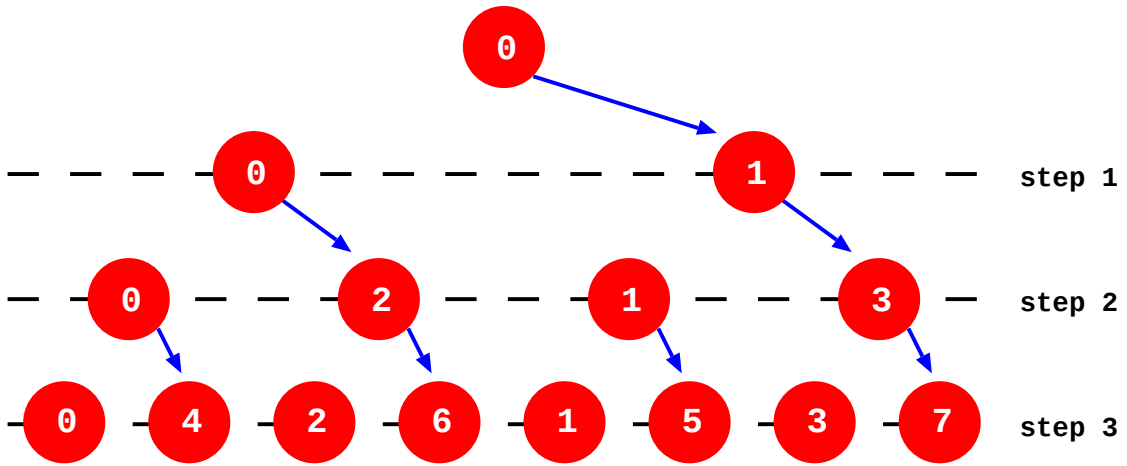
In parallel programming, it is common that, in certain moments during execution, a process wants to communicate the same set of data with the remaining processes (e.g. to initialize data or tasks).




```
...
if (my_rank == 0)
  for (dest = 1; dest < n_procs; dest++)
    MPI_Send(data, count, datatype, dest, tag, comm);
else
    MPI_Recv(data, count, datatype, 0, tag, comm, &status);
...
```

Collective Communications

The structure of communication shown in the previous example is inherently sequential since all communications are done from process 0. If **other processes collaborate in disseminating information**, one may significantly reduce the total communication time.



Using a tree structure for communication, we can distribute data in $\lceil \log_2 N \rceil$ steps instead of $N - 1$ as happened before.

Collective Communications

To implement a tree structure, in each step, each process needs to determine if it is a sender/receiver process and find what is the destination/source of data to be sent/received:

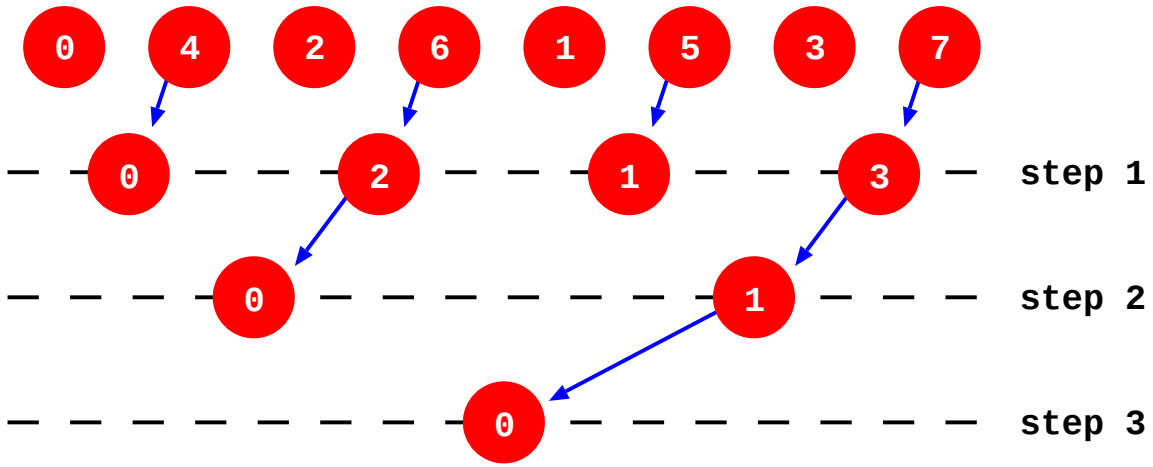
- If $my\_rank < 2^{step-1}$, then send to $my\_rank + 2^{step-1}$
- If $2^{step-1} \leq my\_rank < 2^{step}$, then receive from $my\_rank - 2^{step-1}$

A possible implementation is:

```
...
for (step = 1; step <= upper_log2(n_procs); step++)
  if (my_rank < pow(2, step - 1))
    send_to(my_rank + pow(2, step - 1));
  else if (my_rank >= pow(2, step - 1) && my_rank < pow(2, step))
    receive_from(my_rank - pow(2, step -1));
...
```

Collective Communications

In parallel programming, it is also common that a process, often process 0, receives partial results from other processes to calculate intermediate or final results. If we invert the tree structure for communication, we can apply the same idea to resume data in $\lceil \log_2 N \rceil$ steps.



Collective Messages

To free the programmer from the details of efficiently implementing collective communications, MPI defines a set of functions that deal specifically with that. We can thus classify the messages in:

- **Point-to-point** – the message is sent by one process and received by another process (e.g. every type of messages that we saw before)
- **Collective** – consist of many point-to-point concurrent messages involving all processes in a communicator

Collective messages are variants or combinations of the following primitives:

- **Broadcast**
- **Reduce**
- **Scatter**
- **Gather**

Collective messages **must be called by all processes in a communicator**

Broadcast

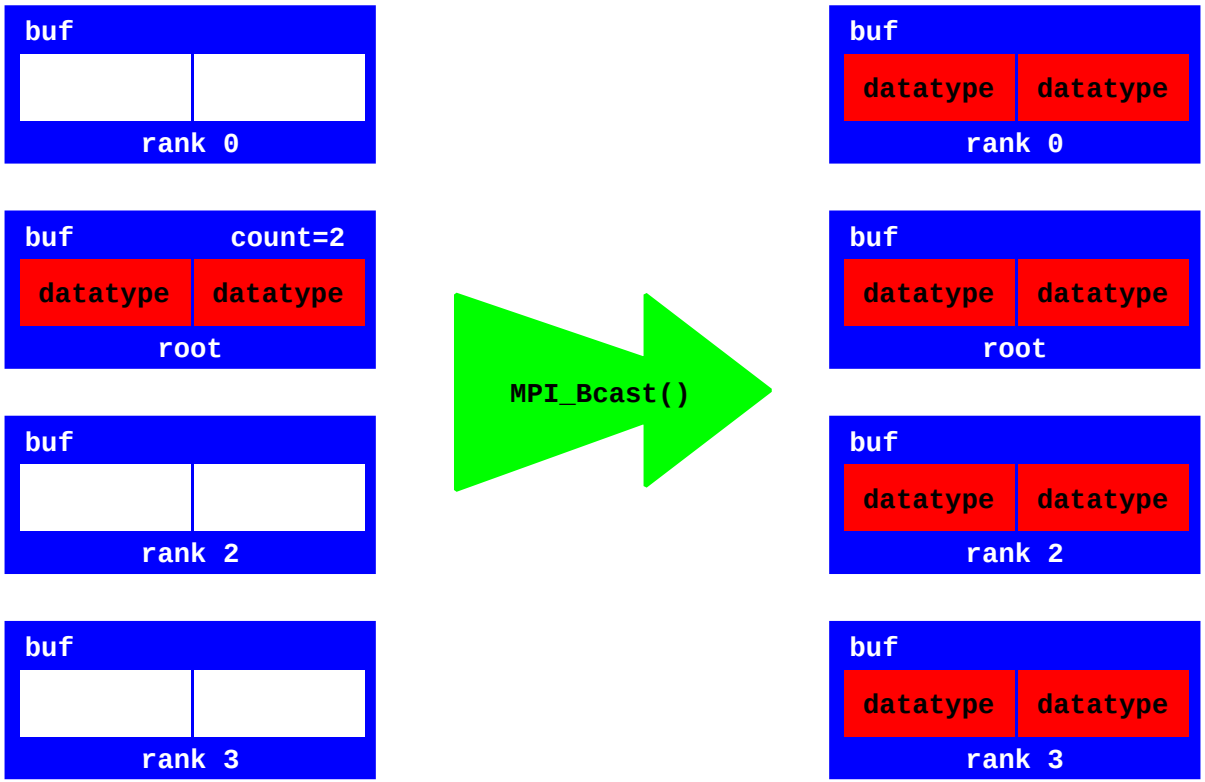
```
MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
  int root, MPI_Comm comm)
```

`MPI_Bcast()` propagates a message from one process to all the processes in a communicator:

- `buf` is the starting address of the data to be sent/received
- `count` is the number of elements of type `datatype` to be sent/received
- `datatype` is the type of data to be sent/received
- `root` is the rank of the process in communicator `comm` that holds the message to be propagated
- `comm` is the communicator to which all processes belong

Broadcast

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



Reduce

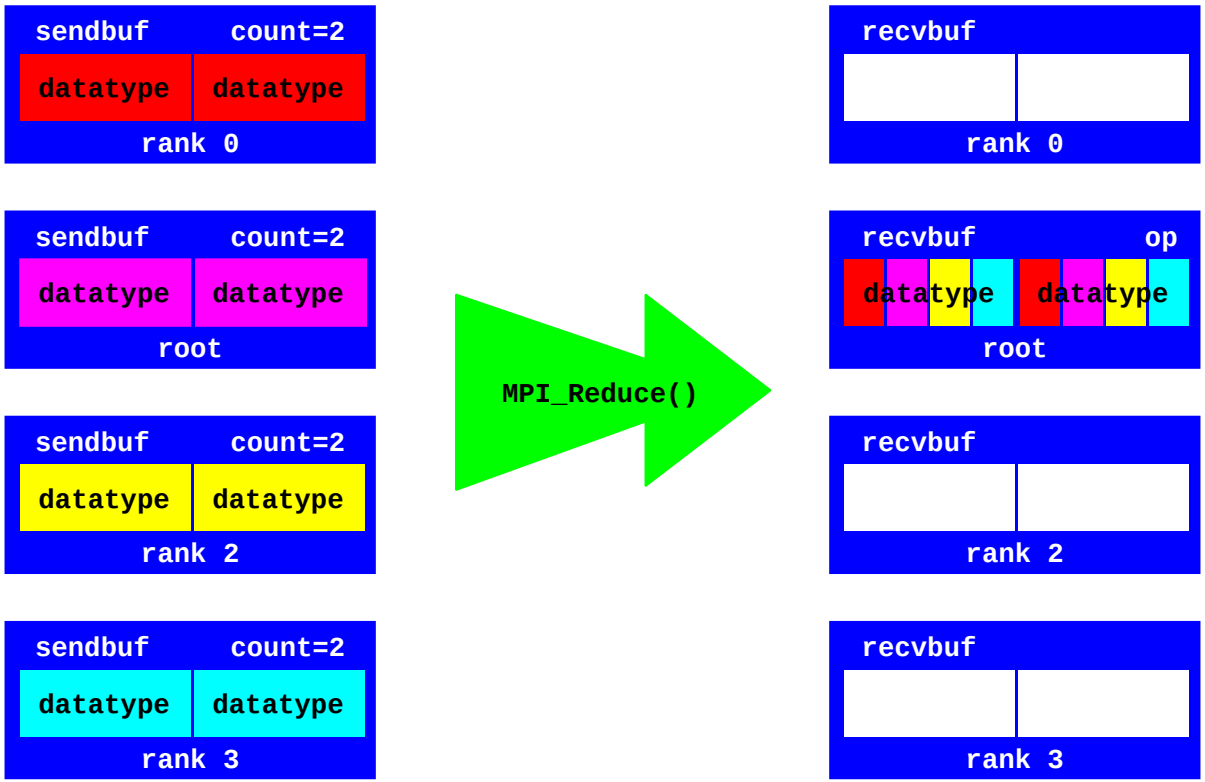
```
MPI_Reduce(void *sendbuf, void* recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

`MPI_Reduce()` resumes data from all the processes in a communicator into a unique process:

- `sendbuf` is the starting address of the data to be sent
- `recvbuf` is the starting address where received data must be resumed (only relevant for process `root`)
- `count` is the number of elements of type `datatype` to be sent
- `datatype` is the type of data to be sent
- `op` is the reduction operation to be applied to the received data
- `root` is the rank of the process in communicator `comm` that receives and resumes the data
- `comm` is the communicator to which all processes belong

Reduce

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

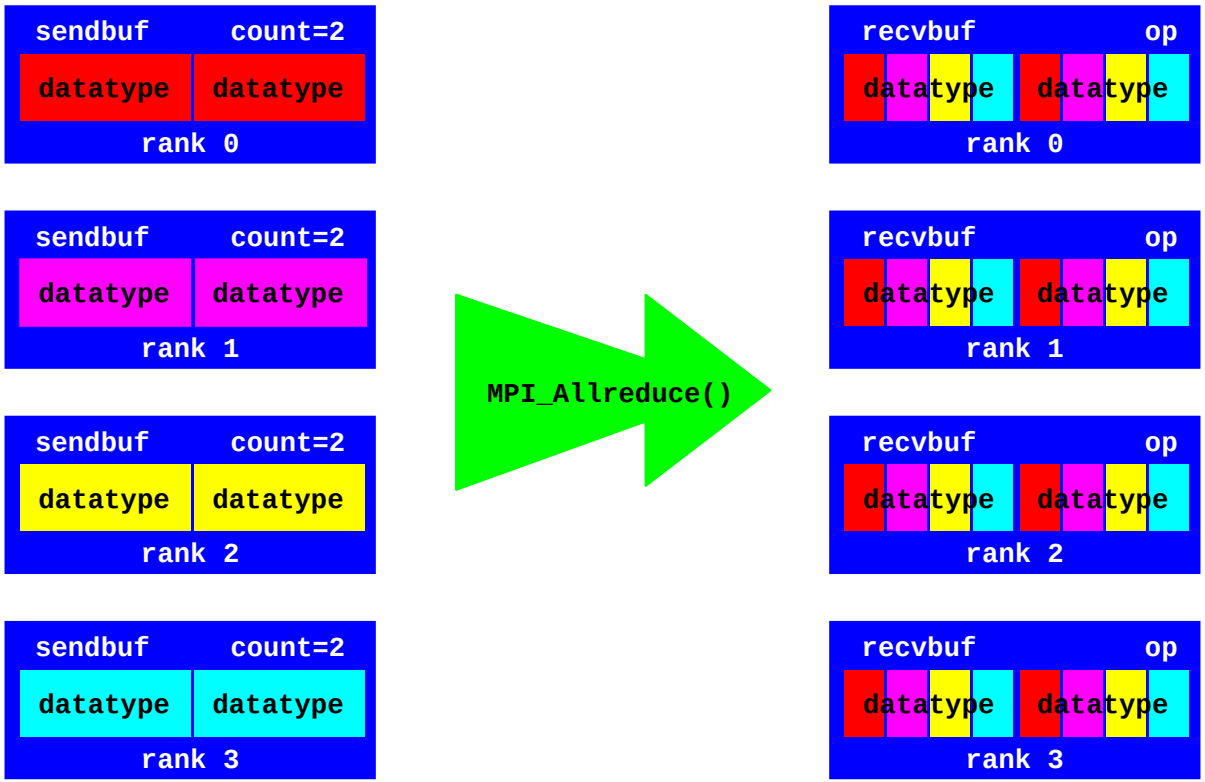


Reduction Operations

Operation	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical exclusive-or
MPI_BXOR	bit-wise exclusive-or

Allreduce

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



Scatter

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

`MPI_Scatter()` divides data from one process in equal parts and distributes it orderly to all the processes in a communicator:

- `sendbuf` is the starting address of the data to be sent (only relevant to process `root`)
- `sendcount` is the number of elements of type `sendtype` to be sent to each process (only relevant to process `root`)
- `sendtype` is the type of data to be sent (only relevant to process `root`)

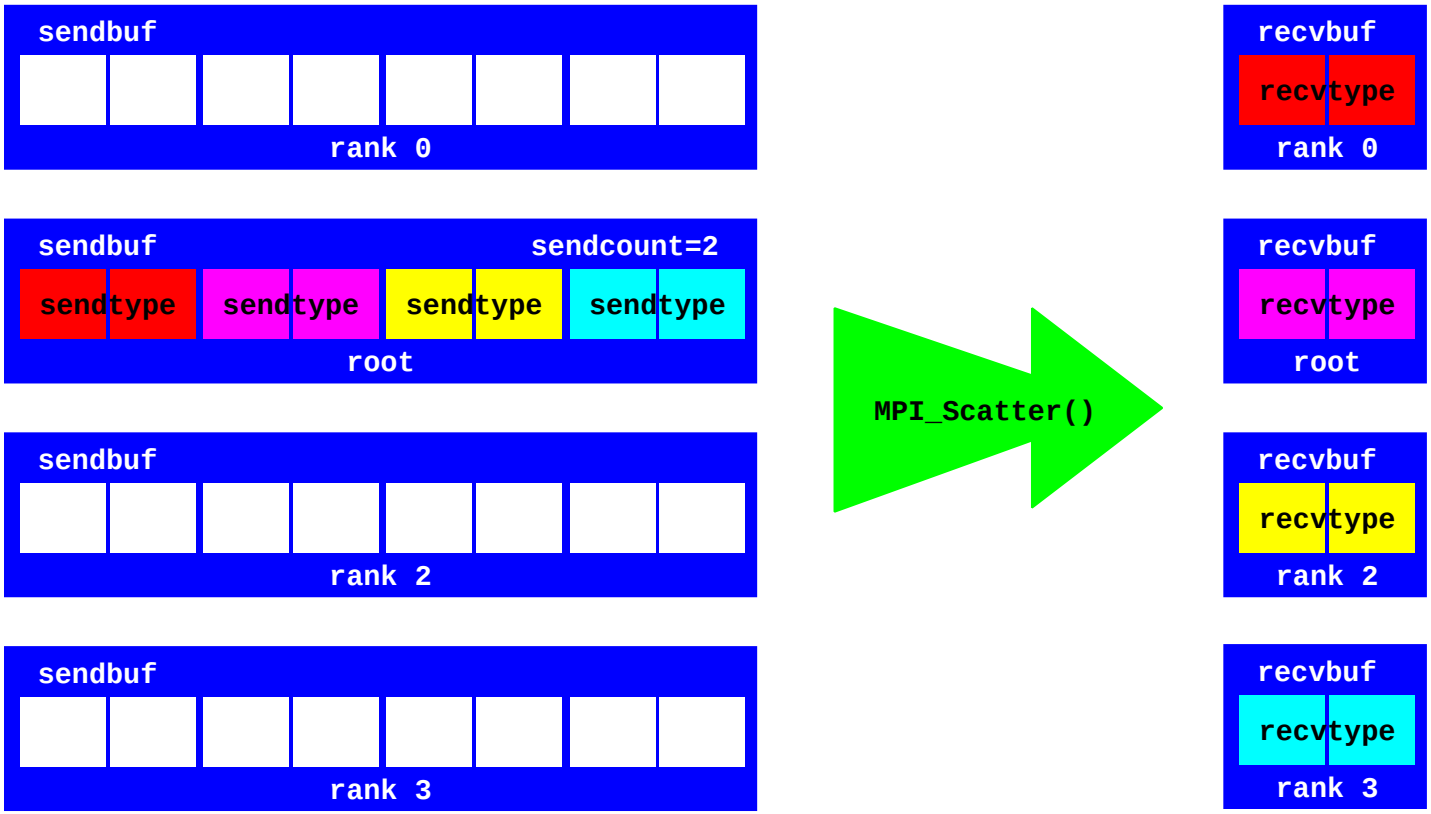
Scatter

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

- **recvbuf** is the starting address where received data must be placed
- **recvcount** is the number of elements of type **recvtype** to be received (usually the same as **sendcount**)
- **recvtype** is the type of data to be received (usually the same as **sendtype**)
- **root** is the rank of the process in communicator **comm** that holds the data to be distributed
- **comm** is the communicator to which all processes belong

Scatter

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



Gather

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

`MPI_Gather()` receives orderly in a unique process data from all the processes in a communicator:

- **`sendbuf`** is the starting address of the data to be sent
- **`sendcount`** is the number of elements of type **`sendtype`** to be sent by each process
- **`sendtype`** is the type of data to be sent

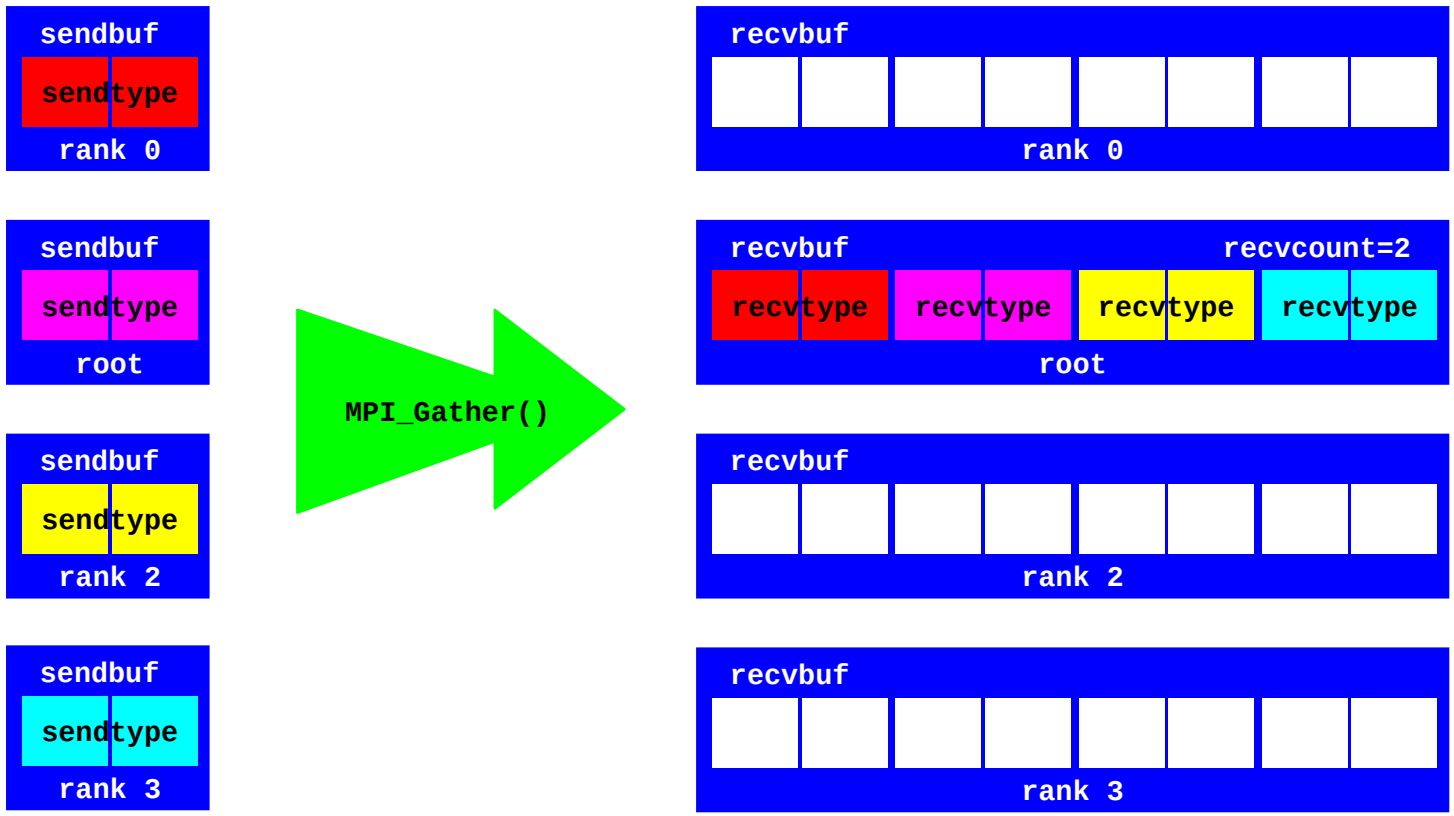
Gather

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

- `recvbuf` is the starting address where received data must be placed (only relevant to process `root`)
- `recvcount` is the number of elements of type `recvtype` to be received from each process (only relevant to process `root` and usually the same as `sendcount`)
- `recvtype` is the type of data to be received (only relevant to process `root` and usually the same as `sendtype`)
- `root` is the rank of the process in communicator `comm` that receives the data
- `comm` is the communicator to which all processes belong

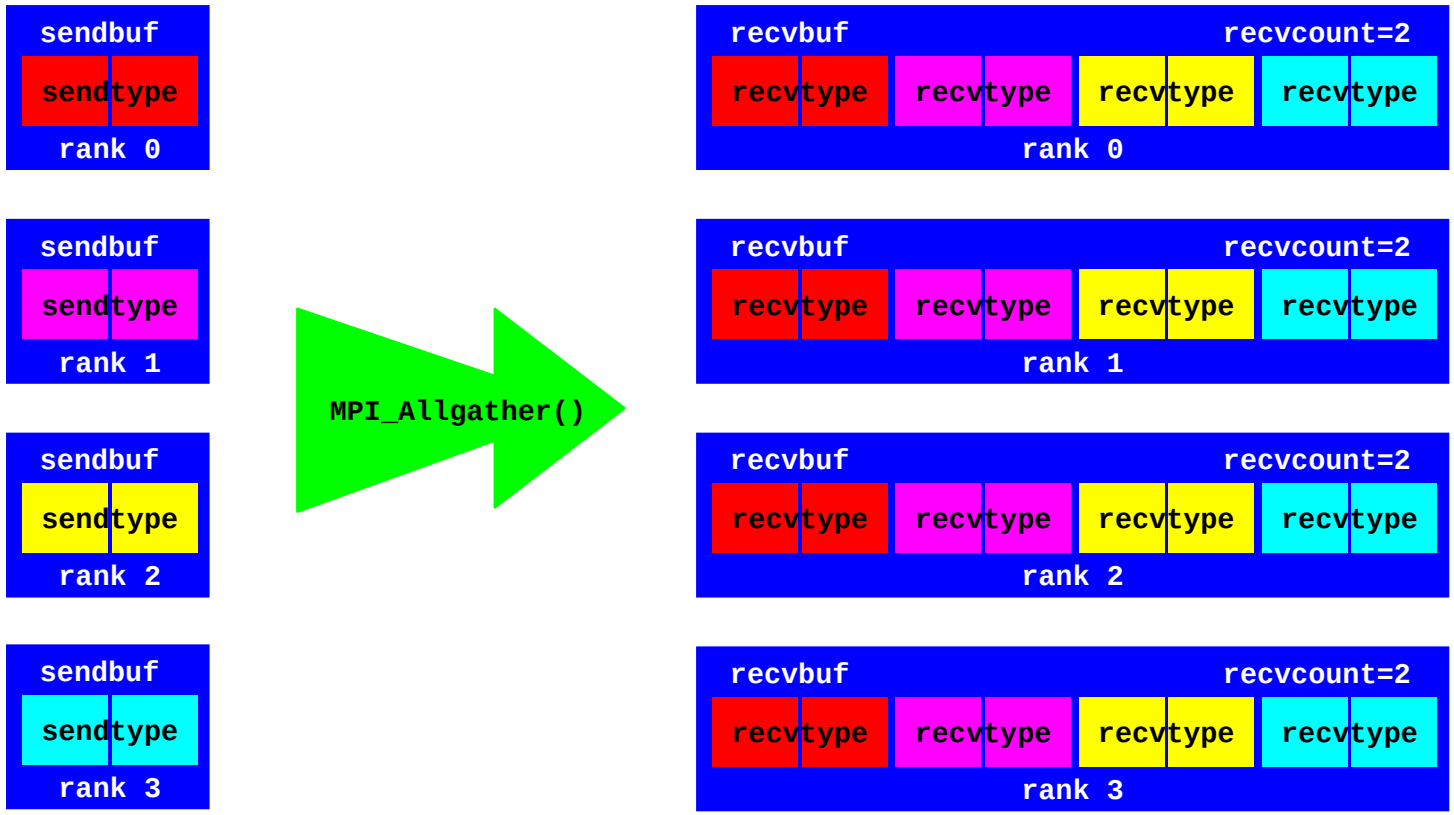
Gather

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



Allgather

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



Scalar Product

The scalar product of 2 vectors with dimension N is given by:

$$x \cdot y = x_0 y_0 + x_1 y_1 + \cdots + x_{n-1} y_{n-1}$$

A possible implementation is:

```
int scalar_product(int x[], int y[], int n) {
  int i, sp = 0;
  for (i = 0; i < n; i++) sp = sp + x[i] * y[i];
  return sp;
}
```

If we have P processes, then each one can calculate K (N/P) components of the scalar product:

	Components
Process 0	$x_0 y_0 + x_1 y_1 + \cdots + x_{k-1} y_{k-1}$
Process 1	$x_k y_k + x_{k+1} y_{k+1} + \cdots + x_{2k-1} y_{2k-1}$
...	...
Process (P-1)	$x_{(p-1)k} y_{(p-1)k} + x_{(p-1)k+1} y_{(p-1)k+1} + \cdots + x_{n-1} y_{n-1}$

Scalar Product (mpi-scalar.c)

```
int K, sp, my_sp, *vecX, *vecY, *locX, *locY;
...
if (my_rank == ROOT) {
  ... // initialize vecX, vecY and K
}
// send K to all processes
MPI_Bcast(&K, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
// allocate space for local components of vecX and vecY
locX = (int *) malloc(K * sizeof(int));
locY = (int *) malloc(K * sizeof(int));
// distribute the components of vecX and vecY
MPI_Scatter(vecX, K, MPI_INT, locX, K, MPI_INT, ROOT, MPI_COMM_WORLD);
MPI_Scatter(vecY, K, MPI_INT, locY, K, MPI_INT, ROOT, MPI_COMM_WORLD);
// calculate the scalar product and print the result
my_sp = scalar_product(locX, locY, K);
MPI_Reduce(&my_sp, &sp, 1, MPI_INT, MPI_SUM, ROOT, MPI_COMM_WORLD);
if (my_rank == ROOT)
  printf("Scalar product = %d\n", sp);
```

Matrix-Vector Product (mpi-matrixvector.c)

The product of a matrix `mat[ROWS,COLS]` and a column vector `vec[COLS]` is a row vector `prod[ROWS]` such that each `prod[i]` is the scalar product of row `i` of the matrix by the column vector. If we have `ROWS` processes, then each one can calculate one element of the result.

```
int ROWS, COLS, *mat, *vec, *prod, sp, *row;
... // initialize ROWS, COLS and the vector
if (my_rank == ROOT) { ... } // initialize the matrix
// distribute the matrix
MPI_Scatter(mat, COLS, MPI_INT, row, COLS, MPI_INT, ROOT, MPI_COMM_WORLD);
// calculate the matrix-vector product and print the result
sp = scalar_product(row, vec, COLS);
MPI_Gather(&sp, 1, MPI_INT, prod, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
if (my_rank == ROOT) {
  printf("Matrix-vector product: ");
  for (i = 0; i < ROWS; i++) printf("%d ", prod[i]);
}
```

Communicators

A communicator is defined as a set of processes that can exchange messages among themselves. Associated to a communicator we have:

- **A group** – an ordered set of processes
- **A context** – a data-structure that uniquely identifies the communicator

The MPI execution environment allows programmers to define new communicators. MPI distinguishes 2 types of communicators:

- **Intra-communicators** – allow exchange of messages and collective communications between processes belonging to a communicator
- **Inter-communicators** – allow exchange of messages between processes belonging to disjoint intra-communicators

Communicators

The general procedure for creating new communicators is as follows:

- Obtain a group of processes from an existing communicator
- Create a new group from the previous one indicating which processes must take part in it
- Create a new communicator based on the new group
- After using them, free the groups and the communicators

In addition to this general procedure, the MPI execution environment provides specific functions to create communicators automatically.

Creating Groups

```
MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

`MPI_Comm_group()` returns in `group` the group of processes in communicator `comm`.

```
MPI_Group_incl(MPI_Group old_group, int size, int ranks[],
  MPI_Group *new_group)
```

`MPI_Group_incl()` creates a new group `new_group` from `old_group` by including the `size` processes referenced in `ranks[]`.

Creating Groups

```
MPI_Group_excl(MPI_Group old_group, int size, int ranks[],
  MPI_Group *new_group)
```

`MPI_Group_excl()` creates a new group `new_group` from `old_group` by excluding the `size` processes referenced in `ranks[]`.

```
MPI_Group_free(MPI_Group *group)
```

`MPI_Group_free()` frees from the MPI execution environment the group identified by `group`.

Creating Communicators

```
MPI_Comm_create(MPI_Comm old_comm, MPI_Group group,
  MPI_Comm *new_comm)
```

`MPI_Comm_create()` creates a new communicator `new_comm` made by the group of processes in `group` of communicator `old_comm`.
`MPI_Comm_create()` is a collective communication, thus it must be called by all processes, including those not adhering to the new communicator. If more than one communicator is created, the order of creation must be the same in all processes.

```
MPI_Comm_free(MPI_Comm *comm)
```

`MPI_Comm_free()` frees from the MPI execution environment the communicator identified by `comm`.

Even Processes (mpi-even.c)

```
MPI_Group world_group, even_group;
MPI_Comm even_comm;
...
for (i = 0; i < n_procs; i += 2)
  ranks[i/2] = i;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
MPI_Group_incl(world_group, (n_procs + 1)/2, ranks, &even_group);
MPI_Comm_create(MPI_COMM_WORLD, even_group, &even_comm);
MPI_Group_free(&world_group);
MPI_Group_free(&even_group);
if (my_rank % 2 == 0) {
  MPI_Comm_rank(even_comm, &even_rank);
  printf("World %d Even %d\n", my_rank, even_rank);
  MPI_Comm_free(&even_comm);
}
```

Creating Communicators

```
MPI_Comm_dup(MPI_Comm old_comm, MPI_Comm *new_comm)
```

`MPI_Comm_dup()` creates a new communicator `new_comm` identical to `old_comm`.

```
MPI_Comm_split(MPI_Comm old_comm, int split_key, int rank_key,
  MPI_Comm *new_comm)
```

`MPI_Comm_split()` creates one or more communicators `new_comm` from `old_comm` by grouping in each new communicator the processes with identical values of `split_key` and by ordering them by `rank_key` (the process with the least `rank_key` will get rank 0, the second least gets rank 1, and so on). The processes that do not want to adhere to any new communicator must include in `split_key` a constant `MPI_UNDEFINED`.

Even and Odd Processes (mpi-split.c)

```
MPI_Comm split_comm;
...
MPI_Comm_split(MPI_COMM_WORLD, my_rank % 2, my_rank, &split_comm);
MPI_Comm_rank(split_comm, &split_rank);
printf("World %d Split %d\n", my_rank, split_rank);
MPI_Comm_free(&split_comm);
```

Executing the code with 5 processes, produces the following output:

```
World 0 Split 0
World 1 Split 0
World 2 Split 1
world 3 Split 1
World 4 Split 2
```

Topologies

The MPI programming model is independent of the physical communication topology that may exist among the processors available in the system.

Nevertheless, in order to increase the communication performance of an application, the hardware topology must coincide as much as possible with the application communication topology.

MPI allows the programmer to define the topology of a communicator with the aim that the MPI execution environment will use it to optimize communication performance within that communicator. Two topologies can be defined:

- **Cartesian Grids**
- **Arbitrary Graphs**

Creating a Cartesian Grid

```
MPI_Cart_create(MPI_Comm old_comm, int ndims, int dims[],
  int periods[], int reorder, MPI_Comm *new_comm)
```

`MPI_Cart_create()` creates a new communicator identical to `old_comm` in which the group of processes is organized as being a cartesian grid:

- `old_comm` is the communicator from which the new communicator representing the grid must be created
- `ndims` is the number of dimensions of the grid (it is also the number of items in `dims[]` and `periods[]`)
- `dims[]` specifies the number of processes in each dimension of the grid
- `periods[]` specifies if the dimensions are periodical or not, i.e., if the last process of each dimension communicates or not with the first process in the same dimension (a value of 1 indicates that it is periodical, and a value of 0 indicates that it is not)

Creating a Cartesian Grid

```
MPI_Cart_create(MPI_Comm old_comm, int ndims, int dims[],
  int periods[], int reorder, MPI_Comm *new_comm)
```

- `reorder` specifies if the positions of the processes in the grid must be equal to those in `old_comm` or if they can be re-ordered by the MPI execution environment in order to optimize performance in future communications (a value of 0 indicates that the positions must be the same, a value of 1 indicates that they may be re-ordered)
- `new_comm` is the new communicator that represents the grid

`MPI_Cart_create()` is a collective communication, thus it must be called by all processes in communicator `old_comm`.

Getting Information About Cartesian Grids

```
MPI_Cart_coords(MPI_Comm comm, int rank, int ndims,
  int coords[])
```

`MPI_Cart_coords()` returns in `coords[]` the `ndims` coordinates of the process with position `rank` in the grid represented by communicator `comm`.

```
MPI_Cart_rank(MPI_Comm comm, int coords[], int *rank)
```

`MPI_Cart_rank()` returns in `rank` the position of the process with coordinates `coords[]` in the grid represented by communicator `comm`.

Cartesian Grid (mpi-cart.c)

```
MPI_Comm grid_comm;
int up, down, right, left, reorder, dims[2], periods[2], coords[2];
...
dims[0] = 4; dims[1] = 2; periods[0] = 1; periods[1] = 1; reorder = 0;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &grid_comm);
MPI_Comm_rank(grid_comm, &grid_rank);
MPI_Cart_coords(grid_comm, grid_rank, 2, coords);
coords[0] -= 1;
MPI_Cart_rank(grid_comm, coords, &up);
coords[0] += 2;
MPI_Cart_rank(grid_comm, coords, &down);
coords[0] -= 1; coords[1] -= 1;
MPI_Cart_rank(grid_comm, coords, &left);
coords[1] += 2;
MPI_Cart_rank(grid_comm, coords, &right);
coords[1] -= 1;
printf("World %d Grid %d (%d,%d) --> up %d down %d left %d right %d\n",
      my_rank, grid_rank, coords[0], coords[1], up, down, left, right);
```

Cartesian Grid (mpi-cart.c)

Executing the code with 8 processes, produces the following output:

```
World 0 Grid 0 (0,0) --> up 6 down 2 left 1 right 1
World 1 Grid 1 (0,1) --> up 7 down 3 left 0 right 0
World 2 Grid 2 (1,0) --> up 0 down 4 left 3 right 3
World 3 Grid 3 (1,1) --> up 1 down 5 left 2 right 2
World 4 Grid 4 (2,0) --> up 2 down 6 left 5 right 5
World 5 Grid 5 (2,1) --> up 3 down 7 left 4 right 4
World 6 Grid 6 (3,0) --> up 4 down 0 left 7 right 7
World 7 Grid 7 (3,1) --> up 5 down 1 left 6 right 6
```

Creating Communicators from a Cartesian Grid

```
MPI_Cart_sub(MPI_Comm old_comm, int dims[], MPI_Comm *new_comm)
```

`MPI_Cart_sub()` creates one or more communicators `new_comm` based on the cartesian grid represented by communicator `old_comm` by grouping in each communicator the processes according to the dimensions specified in `dims[]` (a value of 1 indicates that the dimension is part of the new communicators, and a value of 0 indicates that it is not). The coordinates of the processes in the new communicators `new_comm` are the same as in `old_comm` for the dimensions that are part of the new communicators.

`MPI_Cart_sub()` is a collective communication, thus it must be called by all processes in communicator `old_comm`.

The functionality attained by `MPI_Cart_sub()` is similar to that of `MPI_Comm_split()`. The difference is that `MPI_Cart_sub()` is specific for creating communicators by combining the dimensions of cartesian grids.

Creating Communicators from a Cartesian Grid

```
MPI_Cart_sub(MPI_Comm old_comm, int dims[], MPI_Comm *new_comm)
```

Consider a tri-dimensional grid with dimensions $4 \times 2 \times 5$:

- If `dims[]` is equal to `{1,0,1}` it means that two new bi-dimensional communicators are created with dimensions 4×5. The process with coordinates (2,1,3) in the tri-dimensional grid will have coordinates (2,3) in the new communicator.
- If `dims[]` is equal to `{0,0,1}` it means that 8 new uni-dimensional communicators are created with dimension 5. The process with coordinates (2,1,3) in the tri-dimensional grid will have coordinates (3) in the new communicator.

Processes in Column (mpi-cartsub.c)

```
dims[0] = 4; dims[1] = 2; periods[0] = 1; periods[1] = 1; reorder = 0;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &grid_comm);
MPI_Comm_rank(grid_comm, &grid_rank);
dims[0] = 1; // dimension 0 is part of the new communicators
dims[1] = 0; // dimension 1 is not part of the new communicators
MPI_Cart_sub(grid_comm, dims, &col_comm);
MPI_Comm_rank(col_comm, &row_rank);
printf("World %d Grid %d Row %d\n", my_rank, grid_rank, row_rank);
```

Executing the code with 8 processes, produces the following output:

```
World 0 Grid 0 Row 0
World 1 Grid 1 Row 0
World 2 Grid 2 Row 1
World 3 Grid 3 Row 1
World 4 Grid 4 Row 2
World 5 Grid 5 Row 2
World 6 Grid 6 Row 3
World 7 Grid 7 Row 3
```

Measuring Execution Time

```
double MPI_Wtime(void)
```

```
double MPI_Wtick(void)
```

`MPI_Wtime()` returns the elapsed time, in seconds, since an arbitrary point in the past. `MPI_Wtick()` returns the precision of `MPI_Wtime()`. For example, if `MPI_Wtime()` is incremented every microsecond then `MPI_Wtick()` returns 0.000001.

```
MPI_Barrier(MPI_Comm comm)
```

`MPI_Barrier()` blocks until all the processes in communicator `comm` also call `MPI_Barrier()`.

Measuring Execution Time

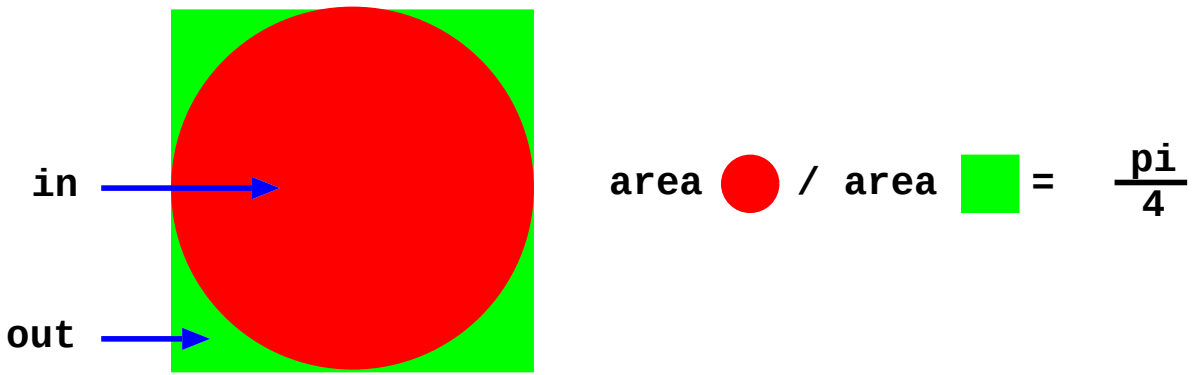
```
double start, finish;
...
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
...
// part of the execution being measured
...
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
if (my_rank == 0)
  printf("Execution time: %f seconds\n", finish - start);
```

The values returned by `MPI_Wtime()` are in real time, i.e., all the time that the process may be suspended by the operating system is also counted.

Computing π

The value of π can be calculated by approximation using the Monte Carlo method. The idea is as follows:

- Generate N random points (x, y)
- For each point (x, y) verify if $x * x + y * y < 1$ and depending on the result increment the variables **in** or **out**
- Calculate the approximate value of π as $4 * in/(in + out)$



Computing π: How To Proceed in Parallel?

Consider the following approach:

- Define one process as the server of random points
- Consider the remaining processes as clients and define a new communicator grouping them
- Clients successively ask the server for sequences of random points, count which points are in and which are out, and then propagate that info to the remaining clients
- When the total number of points processed by all clients exceeds N, the computation ends and one of the processes prints the approximate value of π

How do you comment this approach?

Computing π (mpi-pi.c)

```
...
// define a new communicator for the clients
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
ranks[0] = SERVER;
MPI_Group_excl(world_group, 1, ranks, &worker_group);
MPI_Comm_create(MPI_COMM_WORLD, worker_group, &workers_comm);
MPI_Group_free(&worker_group); MPI_Group_free(&world_group);
// read the number of points to process and broadcast it
if (my_rank == ROOT) scanf("%d", &total_points);
MPI_Bcast(&total_points, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
// initialize the time counter
MPI_Barrier(MPI_COMM_WORLD); start = MPI_Wtime();
// compute the approximate value of PI
if (my_rank == SERVER) server(); else client();
// end time counting and output result
MPI_Barrier(MPI_COMM_WORLD); finish = MPI_Wtime();
if (my_rank == ROOT) {
  printf("PI = %.20f\n", (4.0 * total_in) / (total_in + total_out));
  printf("Execution time = %f seconds\n", finish - start);
}
```

Computing π (mpi-pi.c)

```
server() {
  ...
  do {
    MPI_Recv(&req_points, 1, MPI_INT, MPI_ANY_SOURCE,
             TAG_REQUEST, MPI_COMM_WORLD, &status);
    if (req_points) {
      for (i = 0; i < req_points; i++)
        rands[i] = random();
      MPI_Send(rands, req_points, MPI_INT, status.MPI_SOURCE,
               TAG_REPLY, MPI_COMM_WORLD);
    }
  } while (req_points);
}
```

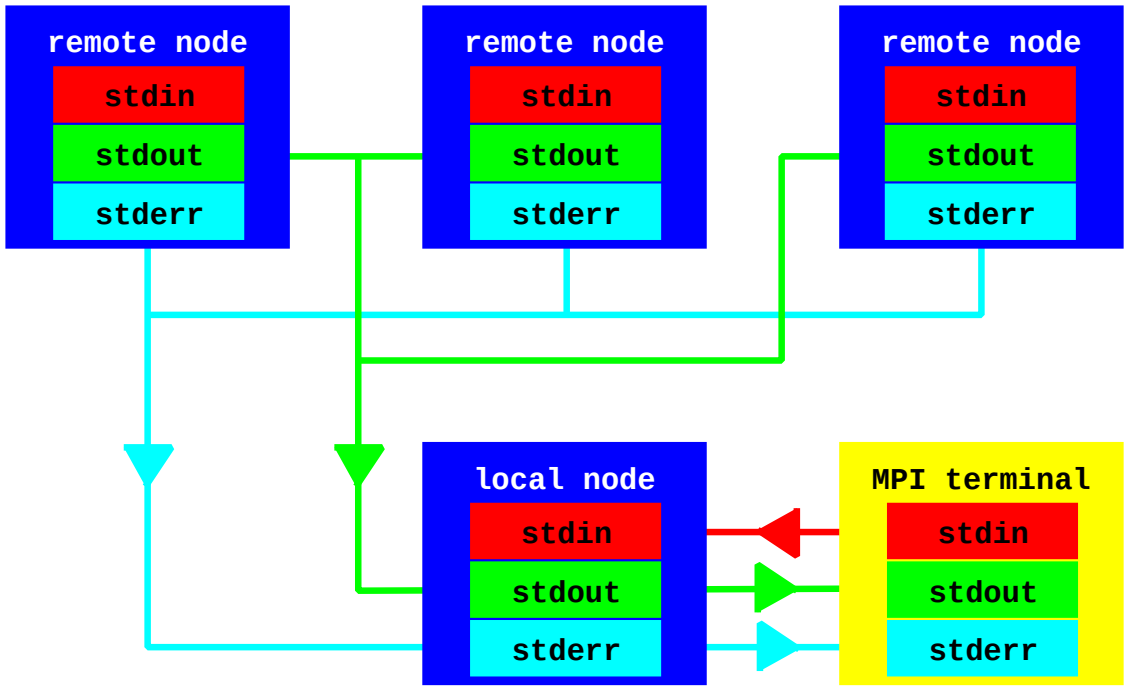
Computing π (mpi-pi.c)

```
client() {
  ...
  in = out = 0;
  req_points = REQ_POINTS;
  do {
    MPI_Send(&req_points, 1, MPI_INT, SERVER,
             TAG_REQUEST, MPI_COMM_WORLD);
    MPI_Recv(rands, req_points, MPI_INT, SERVER,
             TAG_REPLY, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (i = 0; i < req_points; i += 2) {
      x = (((double) rands[i]) / RAND_MAX) * 2 - 1;
      y = (((double) rands[i+1]) / RAND_MAX) * 2 - 1;
      (x * x + y * y < 1.0) ? in++ : out++;
    }
    MPI_Allreduce(&in, &total_in, 1, MPI_INT, MPI_SUM, workers_comm);
    MPI_Allreduce(&out, &total_out, 1, MPI_INT, MPI_SUM, workers_comm);
    more = (total_in + total_out < total_points);
    if (more == 0 && my_rank == ROOT)
      MPI_Send(&more, 1, MPI_INT, SERVER, TAG_REQUEST, MPI_COMM_WORLD);
  } while (more);
}
```

Standard I/O

In the local node (the one where the user invokes the execution command), the **standard input** is inherited from the terminal where the execution starts. In all remote nodes, it is redirected to `/dev/null`.

In all nodes, the **standard output** and the **standard error** are redirected to the terminal where the execution starts.



Compiling and Executing Programs

MPI provides a set of scripts to deal with the headers and libraries that may be necessary for compilation and execution:

- `mpicc` – compilation script for MPI programs written in C
- `mpiCC` – compilation script for MPI programs written in C++
- `mpif77` – compilation script for MPI programs written in Fortran
- `mpirun` – execution command used to start the distributed execution of a given MPI program

To allow a proper setup, the following information should be provided to command `mpirun`:

- The cluster of machines to be considered (option `--hostfile`)
- The number of executing units to be launched (option `-np`)
- The scheduling policy (option `--byslot` or `--bynode`)

Host Files and Scheduling Policies

The host file scheme must specify the name of the machines to be used and the number of slots (CPUs or cores) per machine (e.g. `slots=2`).

```
# cluster with 4 machines and 11 processing units
# one single processor machine (default slots is 1)
node01.dcc.fc.up.pt
# one dual-processor machine (default max-slots is 'infinite')
node02.dcc.fc.up.pt slots=2
# two quad-core machines with over-subscribing disallowed
node03.dcc.fc.up.pt slots=4 max-slots=4
node04.dcc.fc.up.pt slots=4 max-slots=4
```

Two scheduling policies are available:

- **By slot** – shedule processes on a node until all of its default slots are exhausted before proceeding to the next node (**default policy**)
- **By node** – schedule one process per node in round-robin (looping back to the first node as necessary) until all processes have been scheduled (nodes are skipped once their default slots are exhausted)