MPI Reference

General:

Initialize MPI:

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

argc and argv come from the main function input argument, can use MPI_Init(NULL, NULL) if necessary.

Cleanup:

```
int MPI_Finalize()
```

All processes need to call this before exiting.

How many processes in the communicator?:

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

Which rank in communicator am I?:

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

Check wall clock time:

```
double MPI_Wtime()
```

Datatypes:

MPI_INT	int
MPI_LONG	long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	char

Communicators:

MPI_COMM_WORLD	all processes available to the program

We can define own communicators (TBD).

Blocking Point-to-point communication:

Blocking means once the code enters the command, it waits until it is completed before moving on with execution of the rest of the program.

Send a message to one process:

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

Sends `count` elements of type `datatype` from the memory location `buf` to process `dest` in communicator `comm`. The `tag` needs to be matched by the `MPI_Recv` command.

Receive a message from one process:

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

Receives `count` elements of type `datatype` to the memory location `buf` from process source in communicator `comm`. The `tag` and `source` need to match `tag` and `dest` from the corresponding `MPI_Send` command, unless `MPI_ANY_TAG` and/or `MPI_ANY_SOURCE` wildcards are used instead. If the MPI_Recv does not match MPI_Send, the process will hang indefinitely. The status structure holds information about the number of elements received, the tag used and the sender rank. To ignore status, use `MPI_STATUS_IGNORE`.

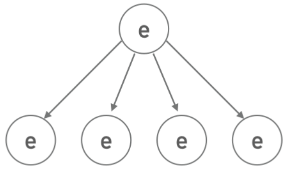
Combined Send-Receive:

```
int 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)
```

Send `sendcount` elements of type `sendtype` stored in `sendbuf` to process `dest` with tag `sendtag`, at the same time receive `recvcount` elements of `recvtype` into `recvbuf` from process `source` with tag `recvtag`. All happening within communicator `comm`.
It is possible to reuse the same buffer for `sendbuf` and `recvbuf` - see `MPI_Sendrecv_replace`.

Collective Communication #1

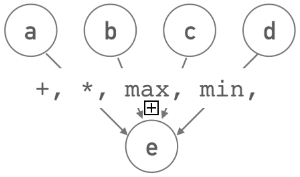
All processes in a communicator need to call the collective communication command.



Send message to all processes:

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

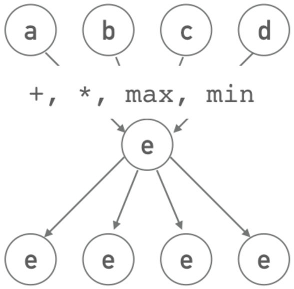
Content of `buf` (a total of `count` elements of type `datatype`) is sent from `root` to all processes in comm. All processes have the copy of data in `buf`.



Collect data from all processes and perform an operation to combine it:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
               MPI_Op operation, int root, MPI Comm comm)
```

Each process in `comm` sends the content of `sendbuf` (`count` elements of type `datatype`). An `operation` is performed on the data, and the result is stored in `recvbuf` on process `root`. Some available reduction operations: `MPI_SUM, MPI_PROD, MPI_MAX, MPI_MIN`

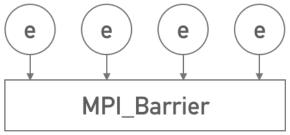


Collect data from all processes, perform reduction operation and store result on all processes:

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,

                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

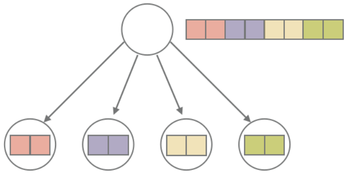
Each process in `comm` sends the content of `sendbuf` (`count` elements of type `datatype`). An `operation` is performed on the data, and the result is stored in `recvbuf` on all process in `comm`.



Wait for all processes to reach this point in the program

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

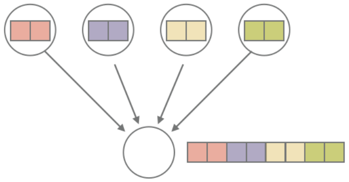
Collective Communication #2



Send equal part of a buffer to all processes:

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI Datatype sendtype,
                void *recvbuf, int recvcount, MPI Datatype recvtype,
                int root, MPI Comm comm )
```

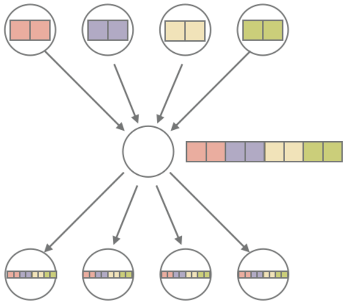
Content of sendbuf on process root is split into equal parts of size sendcount elements of type sendtype and each chunk is sent to a different process in comm, where it is stored in *recvbuf. Typically recvcount = sendcount, and recvtype = sendtype. Root process also gets its chunk of data send to self.



Collect data from all processes into a single array on one process

```
int MPI_Gather(void *sendbuf, int sendcount, MPI Datatype sendtype,
                void *recvbuf, int recvcount, MPI Datatype recvtype,
                int root, MPI Comm comm )
```

Small chunks (sendcount elements of type sendtype) of data stored in sendbuf on all processes in comm are collected into one array recvbuf on root process. Typically recvcount = sendcount, and recvtype = sendtype. Unlike MPI_Reduce, no operation is performed, but the data from each process is appended to an array, which is nproc*recvcount*sizeof(recvtype) long.



Collect data from all processes into a single array copied on all processes

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI Datatype sendtype,
                  void *recvbuf, int recvcount, MPI Datatype recvtype,
                  MPI Comm comm )
```

Small chunks (sendcount elements of type sendtype) of data stored in sendbuf on all processes in comm are collected into one array recvbuf, which is stored on all processes - each process has exact copy of recvbuf. Typically recvcount = sendcount, and recvtype = sendtype.