The above send and receive are point-to-point messages, namely one source and one destination.
MPI
Multiple participant messaging

The above send and receive are point-to-point messages, namely one source and one destination.

MPI provides many more general kinds of messaging.
MPI
Multiple participant messaging

The above send and receive are point-to-point messages, namely one source and one destination.

MPI provides many more general kinds of messaging.

Point-to-point turns out to be much less useful than you might think.
Broadcast:
MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

The buffer of data is sent from the process with rank root to all processes in the communicator.
Note: all processes, including the receivers, should call MPI_Bcast with the same value for root.
Note: all processes, including the receivers, should call `MPI_Bcast` with the same value for `root`.

The destination buffer can vary as you see fit, but is typically the “same” buffer.
int n[2];
if (myrank == 1) {
    n[0] = 23;
    n[1] = 42;
}
...
MPI_Bcast(n, 2, MPI_INT, 1, MPI_COMM_WORLD);

All processes will now have the same values for their versions of n
MPI

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

This takes the data sendbuf, an array, in processor with rank root, and sends sendcount items from the array to each other processor (and to itself) to end up in recvbuf

Scattering single values
The processor with rank 0 (in the specified communicator) gets the first `sendcount` items from `sendbuf`; processor 1 gets the next `sendcount` items; and so on.

Note: `recvtype` can be different from `sendtype`, but you had better be sure you understand what you are doing.

`recvcount` can be different from `sendcount`, but you had better be sure you understand what you are doing.

Don't do that!
The processor with rank 0 (in the specified communicator) gets the first `sendcount` items from `sendbuf`; processor 1 gets the next `sendcount` items; and so on.

Just as in broadcast, every processor executes `SCATTER` with the same `root`.
The processor with rank 0 (in the specified communicator) gets the first `sendcount` items from `sendbuf`; processor 1 gets the next `sendcount` items; and so on.

Just as in broadcast, every processor executes `SCATTER` with the same root.

Note: `recvtype` can be different from `sendtype`, but you had better be sure you understand what you are doing.
The processor with rank 0 (in the specified communicator) gets the first sendcount items from sendbuf; processor 1 gets the next sendcount items; and so on.

Just as in broadcast, every processor executes SCATTER with the same root.

Note: recvtype can be different from sendtype, but you had better be sure you understand what you are doing.

recvcount can be different from sendcount, but you had better be sure you understand what you are doing.
The processor with rank 0 (in the specified communicator) gets the first `sendcount` items from `sendbuf`; processor 1 gets the next `sendcount` items; and so on.

Just as in broadcast, every processor executes `SCATTER` with the same root.

Note: `recvtype` can be different from `sendtype`, but you had better be sure you understand what you are doing.

`recvcount` can be different from `sendcount`, but you had better be sure you understand what you are doing.

Don’t do that!
MPI

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

Takes sendcount elements of data sendbuf from each processor and puts them in the array recvbuf on processor root

Gathering single values
MPI

MPI_Gather is the “opposite” of MPI_Scatter
MPI Gather is the “opposite” of MPI Scatter

The recvbuf on the root processor is filled, in order, with the specified number of items from processors rank 0, 1, etc.
MPI

MPI_Gather is the “opposite” of MPI_Scatter

The recvbuf on the root processor is filled, in order, with the specified number of items from processors rank 0, 1, etc.

Type and counts can vary across processors
MPI

MPI_Gather is the “opposite” of MPI_Scatter

The recvbuf on the root processor is filled, in order, with the specified number of items from processors rank 0, 1, etc.

Type and counts can vary across processors

But don’t do that
MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

Applies a reduction of operation op to each value in sendbuf, putting the result(s) into recvbuf on processor root.
Operations include
MPI_MAX, MPI_MIN, MP_SUM, MPI_PROD, MPI_LAND (logical AND), MPI_LOR (logical OR)
amongst others
Operations include MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND (logical AND), MPI_LOR (logical OR) amongst others.

You can also define your own reduction operators.
MPI

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

A prefix scan of the source sendbuf. Processor of rank $i$ gets the reduction of values from processors 0...$i$ stored in its recvbuf

Prefix scans turn out to be a very useful tool in parallel algorithms
As usual with MPI, there are many other combinations of blocking and non-blocking messages possible.
As usual with MPI, there are many other combinations of blocking and non-blocking messages possible.

Note these functions are not cheap: they hide quite a bit of messaging, which you should be aware of when you are using them.
As usual with MPI, there are many other combinations of blocking and non-blocking messages possible.

Note these functions are not cheap: they hide quite a bit of messaging, which you should be aware of when you are using them.

For example, a MPI_Bcast of a large datastructure can be very slow.
For timing, MPI_Wtime() returns a “high precision” elapsed time in seconds on the calling processor.
For timing, MPI_Wtime() returns a “high precision” elapsed time in seconds on the calling processor.

It returns a double, with precision as given by MPI_Wtick().
For timing, `MPI_Wtime()` returns a “high precision” elapsed time in seconds on the calling processor.

It returns a double, with precision as given by `MPI_Wtick()`.

This might be, say, 0.000001 (1 microsecond).
MPI

MPI also provides

- defining new datatypes including arrays and structures;
- means of creating communicators;
- processor groups (communicators contain one or more groups);
- processor topologies (ways of arranging processors into particular geometric shapes that might fit a certain problem or hardware);
- more kinds of scatter/gather/reduce/scan;
- all-to-all broadcasts;
- and so on