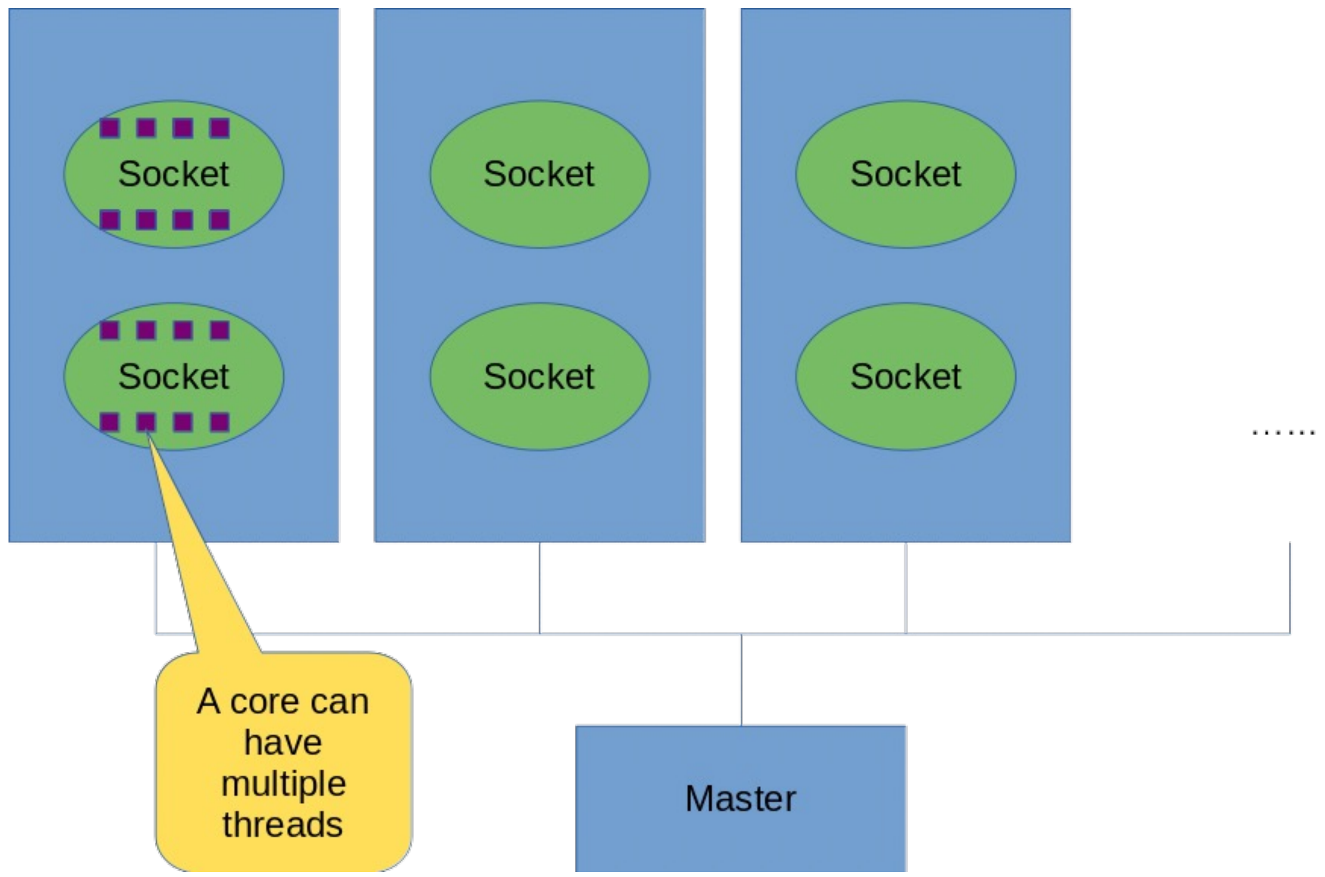
CSC4005 Parallel Programming Tutorial 3

Introduction to MPI Programming

Yangzhixin Luo, 119010224@link.cuhk.edu.cn

Outline of Tutorial 3

Cluster Topology



MPI: Blocking vs Non-blocking

- Blocking communication

Blocking communication is done using **MPI_Send()** and **MPI_Recv()**.

These functions do not return (i.e., they block) until the communication is finished.

The buffer passed to **MPI_Send()** can be reused, either because MPI saved it somewhere, or because it has been received by the destination. Similarly, **MPI_Recv()** returns when the receive buffer has been filled with valid data.

- Non-blocking communication

Non-blocking communication is done using **MPI_Isend()** and **MPI_Irecv()**.

These function return immediately (i.e., they do not block) even if the communication is not finished yet. You must call **MPI_Wait()** or **MPI_Test()** to see whether the communication has finished.

Blocking Point-wise Communication

```
MPI_Send(
        void* data,
        int count,
        MPI_Datatype datatype,
        int destination,
        int tag,
        MPI_Comm communicator);
```

Blocking Point-wise Communication

```
MPI_Recv(
        void* data,
        int count,
        MPI_Datatype datatype,
        int source,
        int tag,
        MPI_Comm communicator,
        MPI_Status* status);
```

MPI Supported Datatypes

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

Blocking Point-wise Communication Example 1

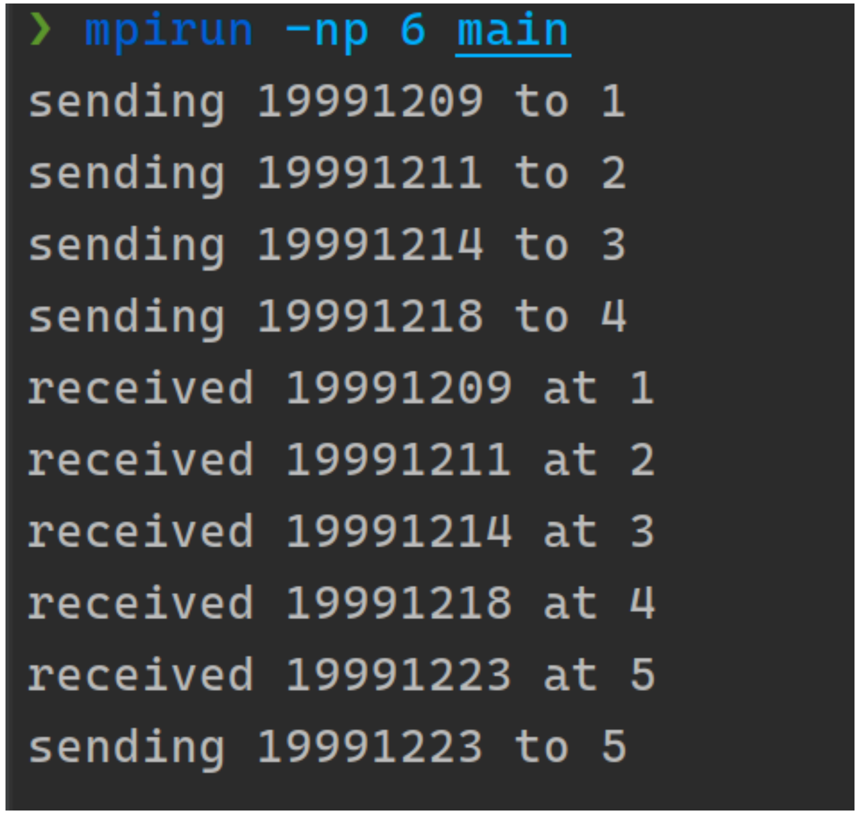
```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n", number);
}
```

Blocking Point-wise Communication Example 2

```
if (0 == rank) {
    data = 1999'12'08;
    for (int i = 1; i < size; ++i) {
        data = data + i;
        std::cout << "sending " << data << " to " << i << std::endl;
        MPI_Send(&data, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
} else {
    MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    std::cout << "received " << data << " at " << rank << std::endl;
}
```

Blocking Point-wise Communication Example 2 Output

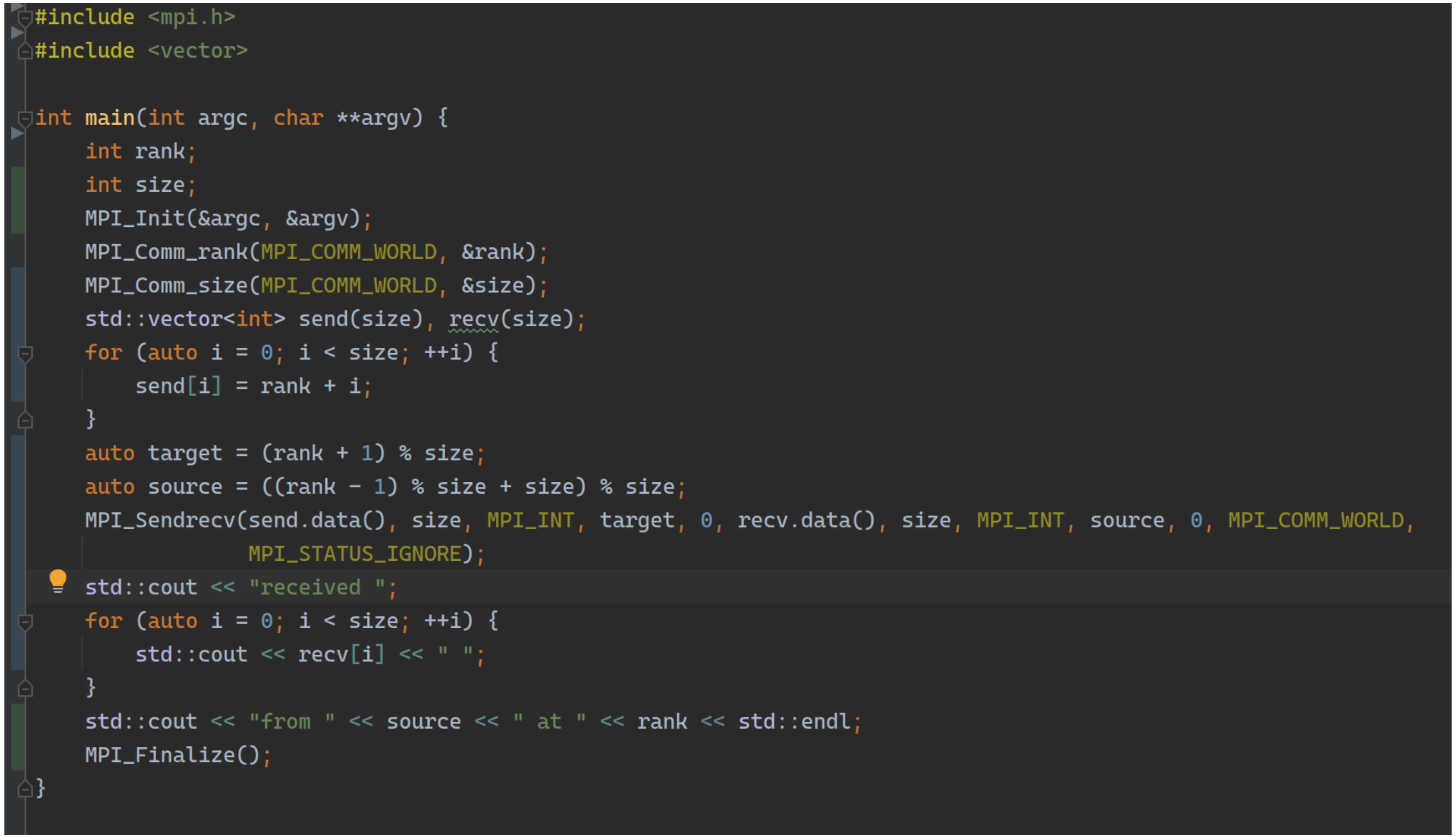


```
❯ mpirun -np 6 main
sending 19991209 to 1
sending 19991211 to 2
sending 19991214 to 3
sending 19991218 to 4
received 19991209 at 1
received 19991211 at 2
received 19991214 at 3
received 19991218 at 4
received 19991223 at 5
sending 19991223 to 5
```

Blocking Point-wise Communication

```
int MPI_Sendrecv(
        const 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);
```

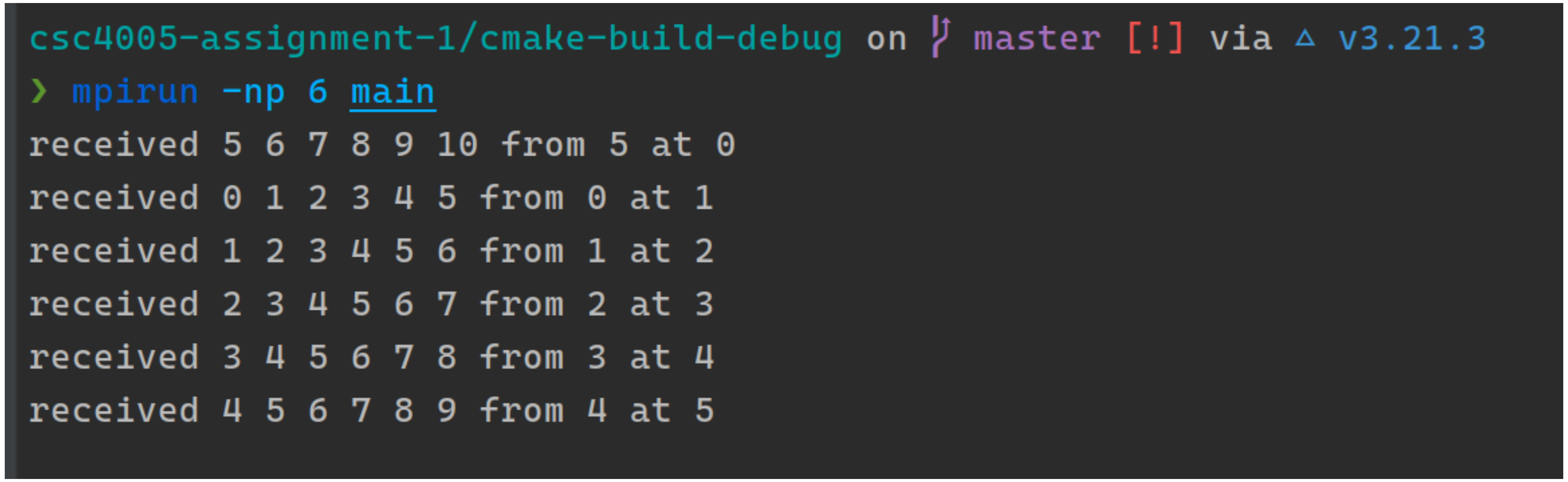
Blocking Point-wise Communication Example 3



```cpp
#include <mpi.h>
#include <vector>

int main(int argc, char **argv) {
    int rank;
    int size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    std::vector<int> send(size), recv(size);
    for (auto i = 0; i < size; ++i) {
        send[i] = rank + i;
    }
    auto target = (rank + 1) % size;
    auto source = ((rank - 1) % size + size) % size;
    MPI_Sendrecv(send.data(), size, MPI_INT, target, 0, recv.data(), size, MPI_INT, source, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    std::cout << "received ";
    for (auto i = 0; i < size; ++i) {
        std::cout << recv[i] << " ";
    }
    std::cout << "from " << source << " at " << rank << std::endl;
    MPI_Finalize();
}
```

Blocking Point-wise Communication Example 3 Output



```
csc4005-assignment-1/cmake-build-debug on  master [!] via △ v3.21.3
❯ mpirun -np 6 main
received 5 6 7 8 9 10 from 5 at 0
received 0 1 2 3 4 5 from 0 at 1
received 1 2 3 4 5 6 from 1 at 2
received 2 3 4 5 6 7 from 2 at 3
received 3 4 5 6 7 8 from 3 at 4
received 4 5 6 7 8 9 from 4 at 5
```

Probing

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

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

MPI_Status

If we pass an **MPI_Status** structure to the **MPI_Recv** function, it will be populated with additional information about the receive operation after it completes. The three primary pieces of information include:

- **The rank of the sender**

The rank of the sender is stored in the **MPI_SOURCE** element of the structure. That is, if we declare an **MPI_Status** stat variable, the rank can be accessed with **stat.MPI_SOURCE.**

- **The tag of the message**

The tag of the message can be accessed by the **MPI_TAG** element of the structure (similar to **MPI_SOURCE**).

- **The length of the message**

The length of the message does not have a predefined element in the status structure. Instead, we have to find out the length of the message with **MPI_Get_count**.

MPI_Get_count

Why would this information be necessary?

The **MPI_Get_count** function is used to determine the actual receive amount.

It turns out that **MPI_Recv** can take **MPI_ANY_SOURCE** for the rank of the sender and **MPI_ANY_TAG** for the tag of the message. For this case, the **MPI_Status** structure is the only way to find out the actual sender and tag of the message. Furthermore, **MPI_Recv** is not guaranteed to receive the entire amount of elements passed as the argument to the function call. Instead, it receives the amount of elements that were sent to it (and returns an error if more elements were sent than the desired receive amount).

MPI_Get_count Example

```c
const int MAX_NUMBERS = 100;
int numbers[MAX_NUMBERS];
int number_amount;
if (world_rank == 0) {
    // Pick a random amount of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

    // Send the amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
    MPI_Status status;
    // Receive at most MAX_NUMBERS from process zero
    MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0, MPI_COMM_WORLD,
             &status);

    // After receiving the message, check the status to determine
    // how many numbers were actually received
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Print off the amount of numbers, and also print additional
    // information in the status object
    printf("1 received %d numbers from 0. Message source = %d, "
           "tag = %d\n",
           number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```

MPI_Probe

Instead of posting a receive and simply providing a really large buffer to handle all possible sizes of messages, you can use **MPI_Probe** to query the message size before actually receiving it.

MPI_Probe looks quite similar to **MPI_Recv**. In fact, you can think of **MPI_Probe** as an **MPI_Recv** that does everything but receive the message. Similar to **MPI_Recv**, **MPI_Probe** will block for a message with a matching tag and sender. When the message is available, it will fill the status structure with information. The user can then use **MPI_Recv** to receive the actual message.

MPI_Probe Example

```c
int number_amount;
if (world_rank == 0) {
    const int MAX_NUMBERS = 100;
    int numbers[MAX_NUMBERS];
    // Pick a random amount of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

    // Send the random amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
} else if (world_rank == 1) {
    MPI_Status status;
    // Probe for an incoming message from process zero
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

    // When probe returns, the status object has the size and other
    // attributes of the incoming message. Get the message size
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Allocate a buffer to hold the incoming numbers
    int* number_buf = (int*)malloc(sizeof(int) * number_amount);

    // Now receive the message with the allocated buffer
    MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("1 dynamically received %d numbers from 0.\n",
           number_amount);
    free(number_buf);
}
```

Immediate (Non-blocking) Point-wise Communication

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

Immediate (Non-blocking) Point-wise Communication

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

Immediate (Non-blocking) Point-wise Communication

MPI_Wait

MPI_Wait is a blocking call that returns only when a specified operation has been completed (e.g., the send buffer is safe to access). This call should be inserted at the point where the next section of code depends on the buffer, because it forces the process to block until the buffer is ready.

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

Immediate (Non-blocking) Point-wise Communication

MPI_Test

MPI_Test is the nonblocking counterpart to **MPI_Wait**. Instead of blocking until the specified message is complete, this function returns immediately with a flag that says whether the requested message is complete (true) or not (false). **MPI_Test** is basically a safe polling mechanism, and this means we can again emulate blocking behavior by executing **MPI_Test** inside of a while-loop.

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

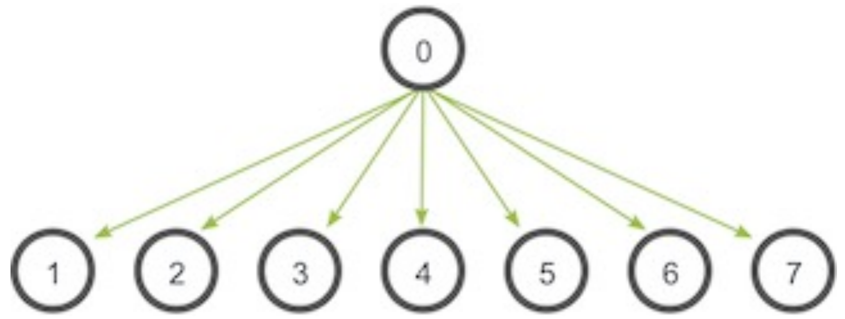
Immediate (Non-blocking) Point-wise Communication Example

```
MPI_Request send_req, recv_req;
MPI_Isend(send.data(), size, MPI_INT, target, 0, MPI_COMM_WORLD, &send_req);
MPI_Irecv(recv.data(), size, MPI_INT, source, 0, MPI_COMM_WORLD, &recv_req);
int sflag = 0, rflag = 0;
do {
    MPI_Test(&send_req, &sflag, MPI_STATUS_IGNORE);
    MPI_Test(&recv_req, &rflag, MPI_STATUS_IGNORE);
} while (!sflag || !rflag);
```

Broadcast

A broadcast is one of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.

In this example, process zero is the root process, and it has the initial copy of data. All of the other processes receive the copy of data.



Broadcast

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator);
```

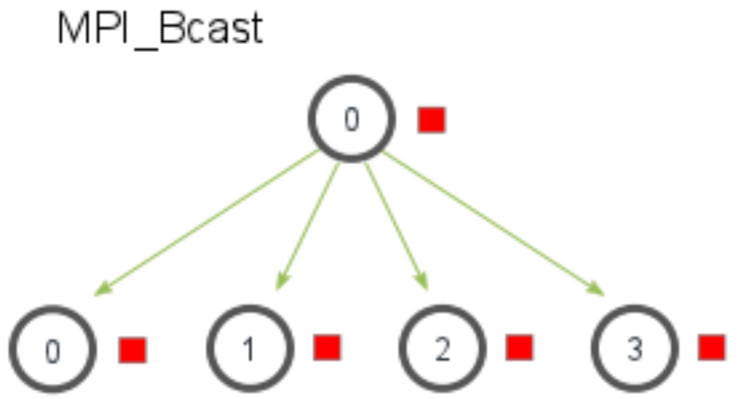
Broadcast Example

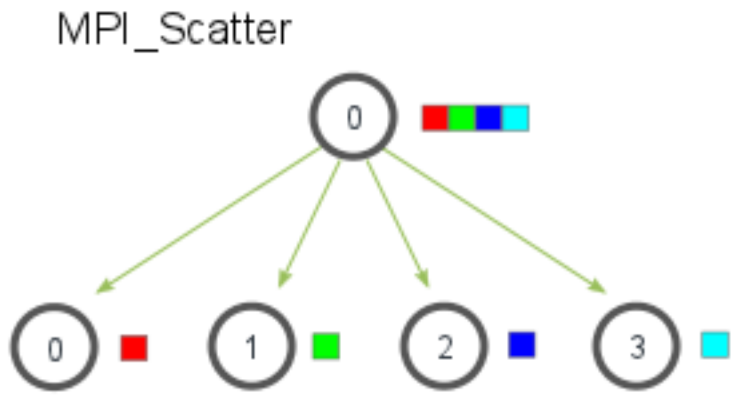
```
int main(int argc, char **argv) {
    int data;
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (0 == rank) {
        std::cin >> data;
    }
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    std::cout << data << std::endl;
    MPI_Finalize();
}
```

MPI_Scatter

MPI_Scatter is a collective routine that is very similar to **MPI_Bcast**. **MPI_Scatter** involves a designated root process sending data to all processes in a communicator.

The primary difference between **MPI_Bcast** and **MPI_Scatter** is small but important. MPI_Bcast sends the same piece of data to all processes while MPI_Scatter sends chunks of an array to different processes.





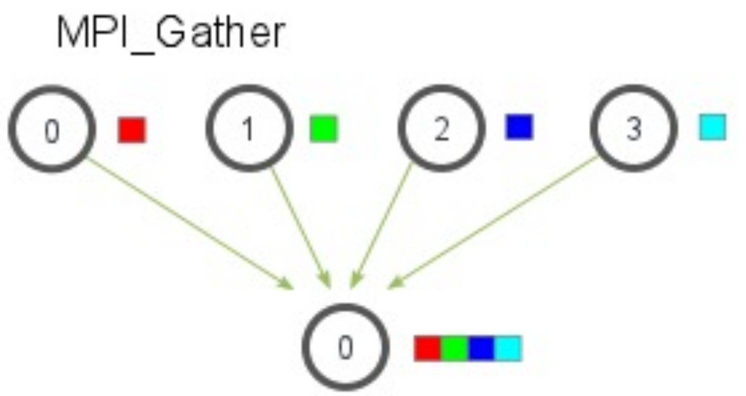
MPI_Scatter

```
MPI_Scatter(
        void* send_data,
        int send_count,
        MPI_Datatype send_datatype,
        void* recv_data,
        int recv_count,
        MPI_Datatype recv_datatype,
        int root,
        MPI_Comm communicator);
```

MPI_Gather

MPI_Gather is the inverse of **MPI_Scatter**. Instead of spreading elements from one process to many processes, **MPI_Gather** takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.

Similar to **MPI_Scatter**, **MPI_Gather** takes elements from each process and gathers them to the root process. The elements are ordered by the rank of the process from which they were received.



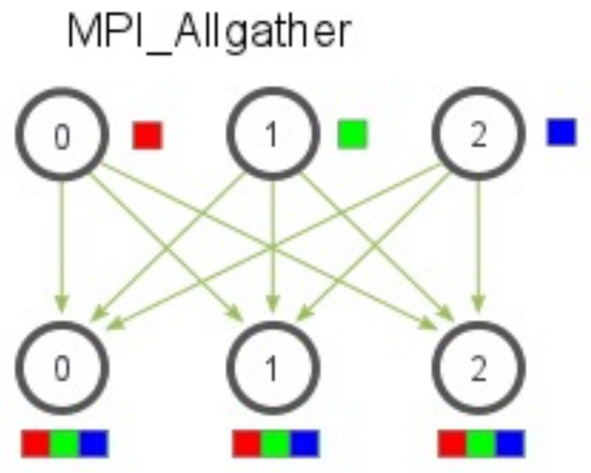
MPI_Gather

```
MPI_Gather(
        void* send_data,
        int send_count,
        MPI_Datatype send_datatype,
        void* recv_data,
        int recv_count,
        MPI_Datatype recv_datatype,
        int root,
        MPI_Comm communicator);
```

MPI_Allgather

Given a set of elements distributed across all processes, **MPI_Allgather** will gather all of the elements to all the processes. In the most basic sense, **MPI_Allgather** is an **MPI_Gather** followed by an **MPI_Bcast**.

Just like **MPI_Gather**, the elements from each process are gathered in order of their rank, except this time the elements are gathered to all processes. The function declaration for **MPI_Allgather** is almost identical to **MPI_Gather** with the difference that there is no root process in **MPI_Allgather**.

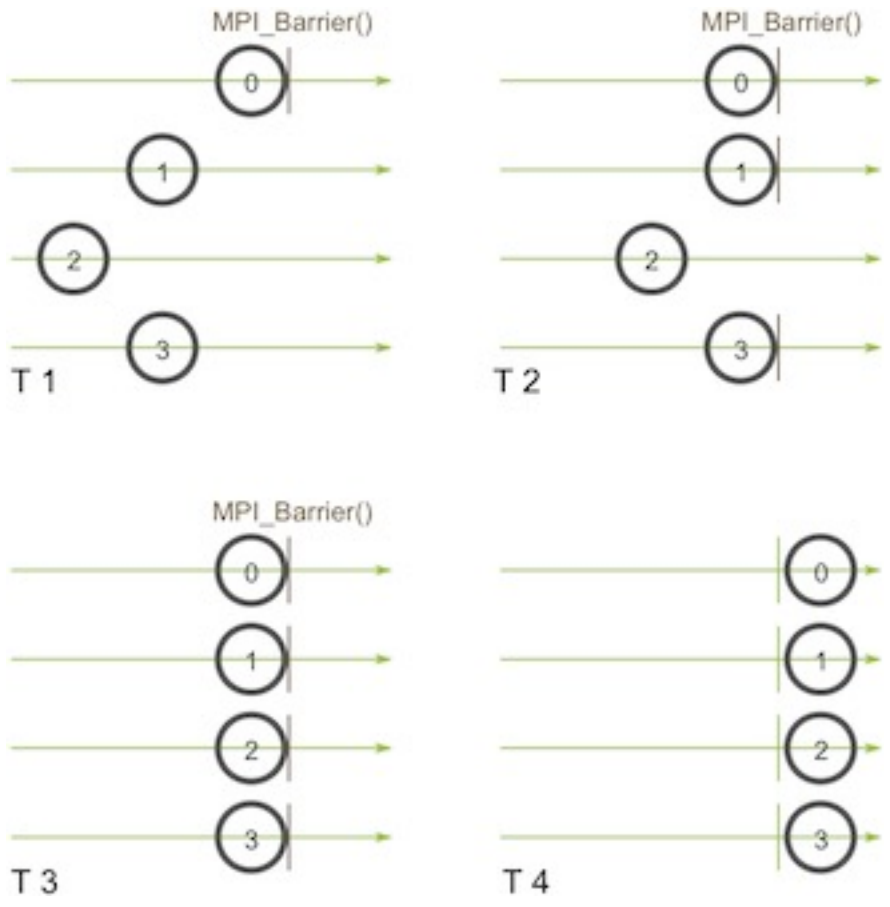


MPI_Allgather

```
MPI_Allgather(
	void* send_data,
	int send_count,
	MPI_Datatype send_datatype,
	void* recv_data,
	int recv_count,
	MPI_Datatype recv_datatype,
	MPI_Comm communicator);
```

MPI_Barrier

```
MPI_Barrier(MPI_COMM_WORLD);
```



MPI_Barrier

```
{
    std::this_thread::sleep_for(std::chrono::milliseconds{100} * rank);
    std::cout << "hi (no bar)" << std::endl;
}
MPI_Barrier(MPI_COMM_WORLD);
{
    std::this_thread::sleep_for(std::chrono::milliseconds{100} * rank);
    MPI_Barrier(MPI_COMM_WORLD);
    std::cout << "hi (bar)" << std::endl;
}
```

More Information

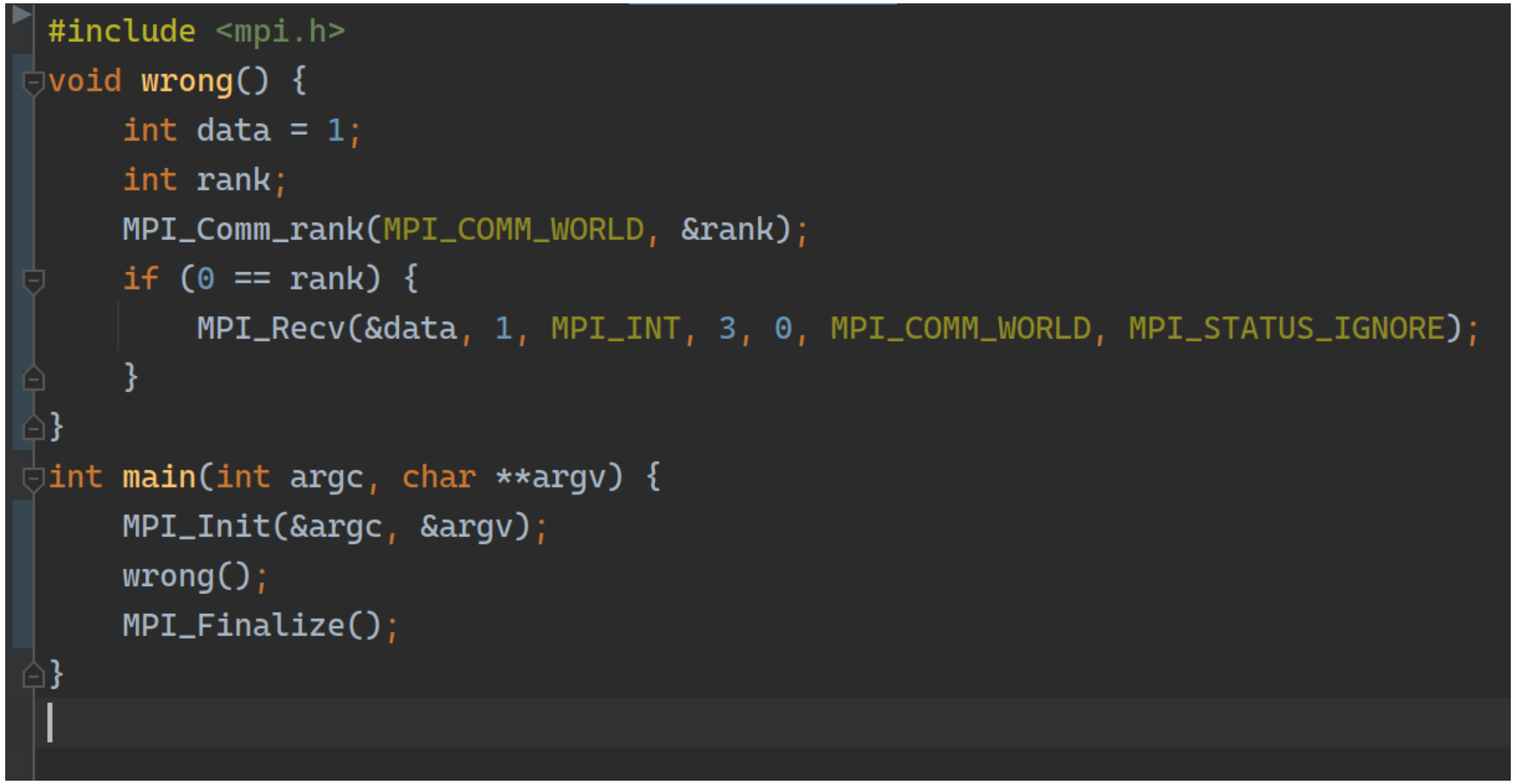
- Reduce:

https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

- Group Division:

https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/

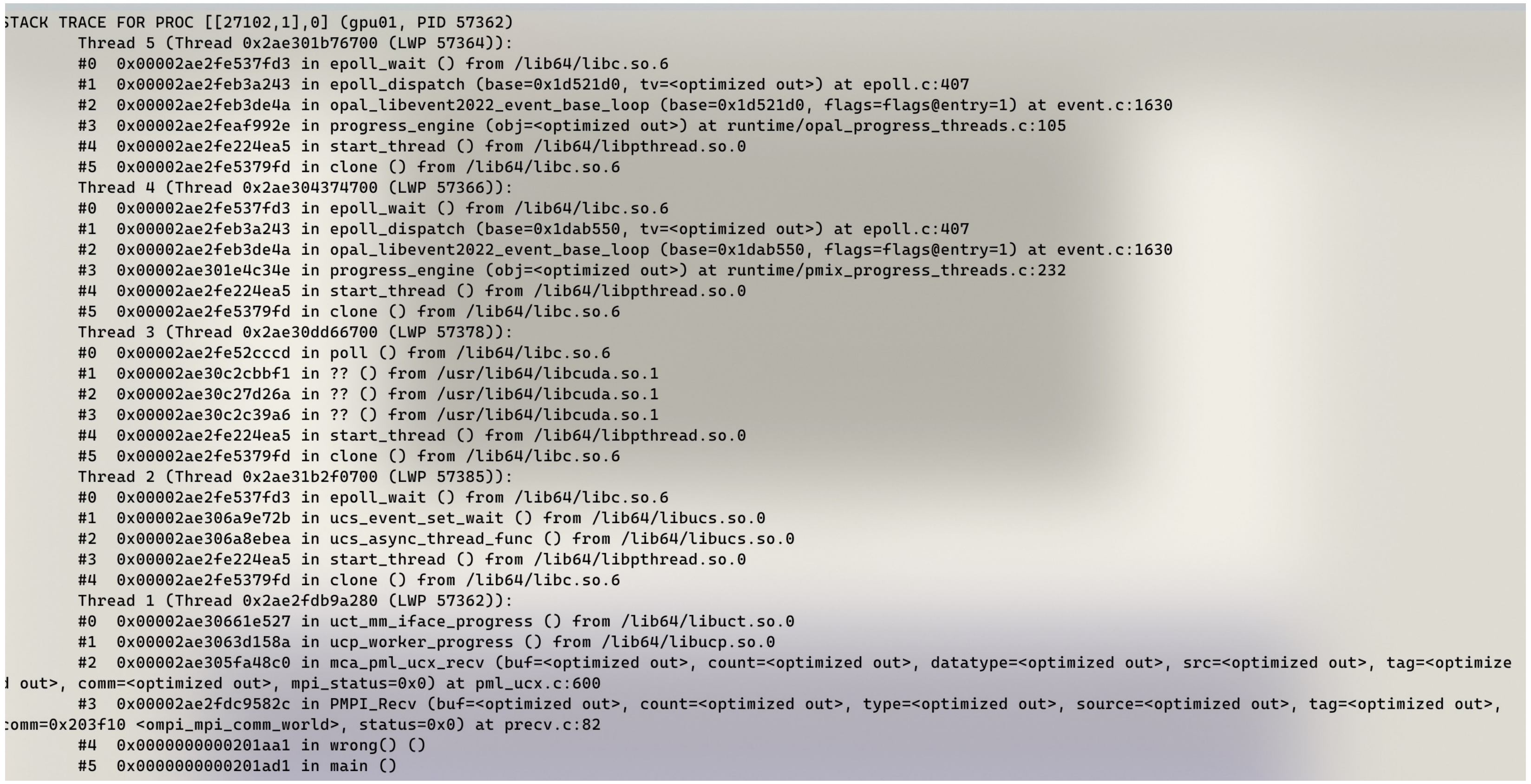
Debugging: Stacktrace



```
#include <mpi.h>
void wrong() {
    int data = 1;
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (0 == rank) {
        MPI_Recv(&data, 1, MPI_INT, 3, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    wrong();
    MPI_Finalize();
}
```

Debugging: Stacktrace

mpirun –timeout 5 --get-stack-traces ./main



```
STACK TRACE FOR PROC [[27102,1],0] (gpu01, PID 57362)
        Thread 5 (Thread 0x2ae301b76700 (LWP 57364)):
        #0  0x00002ae2fe537fd3 in epoll_wait () from /lib64/libc.so.6
        #1  0x00002ae2feb3a243 in epoll_dispatch (base=0x1d521d0, tv=<optimized out>) at epoll.c:407
        #2  0x00002ae2feb3de4a in opal_libevent2022_event_base_loop (base=0x1d521d0, flags=flags@entry=1) at event.c:1630
        #3  0x00002ae2feaf992e in progress_engine (obj=<optimized out>) at runtime/opal_progress_threads.c:105
        #4  0x00002ae2fe224ea5 in start_thread () from /lib64/libpthread.so.0
        #5  0x00002ae2fe5379fd in clone () from /lib64/libc.so.6
        Thread 4 (Thread 0x2ae304374700 (LWP 57366)):
        #0  0x00002ae2fe537fd3 in epoll_wait () from /lib64/libc.so.6
        #1  0x00002ae2feb3a243 in epoll_dispatch (base=0x1dab550, tv=<optimized out>) at epoll.c:407
        #2  0x00002ae2feb3de4a in opal_libevent2022_event_base_loop (base=0x1dab550, flags=flags@entry=1) at event.c:1630
        #3  0x00002ae301e4c34e in progress_engine (obj=<optimized out>) at runtime/pmix_progress_threads.c:232
        #4  0x00002ae2fe224ea5 in start_thread () from /lib64/libpthread.so.0
        #5  0x00002ae2fe5379fd in clone () from /lib64/libc.so.6
        Thread 3 (Thread 0x2ae30dd66700 (LWP 57378)):
        #0  0x00002ae2fe52cccd in poll () from /lib64/libc.so.6
        #1  0x00002ae30c2cbbf1 in ?? () from /usr/lib64/libcuda.so.1
        #2  0x00002ae30c27d26a in ?? () from /usr/lib64/libcuda.so.1
        #3  0x00002ae30c2c39a6 in ?? () from /usr/lib64/libcuda.so.1
        #4  0x00002ae2fe224ea5 in start_thread () from /lib64/libpthread.so.0
        #5  0x00002ae2fe5379fd in clone () from /lib64/libc.so.6
        Thread 2 (Thread 0x2ae31b2f0700 (LWP 57385)):
        #0  0x00002ae2fe537fd3 in epoll_wait () from /lib64/libc.so.6
        #1  0x00002ae306a9e72b in ucs_event_set_wait () from /lib64/libucs.so.0
        #2  0x00002ae306a8ebea in ucs_async_thread_func () from /lib64/libucs.so.0
        #3  0x00002ae2fe224ea5 in start_thread () from /lib64/libpthread.so.0
        #4  0x00002ae2fe5379fd in clone () from /lib64/libc.so.6
        Thread 1 (Thread 0x2ae2fdb9a280 (LWP 57362)):
        #0  0x00002ae30661e527 in uct_mm_iface_progress () from /lib64/libuct.so.0
        #1  0x00002ae3063d158a in ucp_worker_progress () from /lib64/libucp.so.0
        #2  0x00002ae305fa48c0 in mca_pml_ucx_recv (buf=<optimized out>, count=<optimized out>, datatype=<optimized out>, src=<optimized out>, tag=<optimize
d out>, comm=<optimized out>, mpi_status=0x0) at pml_ucx.c:600
        #3  0x00002ae2fdc9582c in PMPI_Recv (buf=<optimized out>, count=<optimized out>, type=<optimized out>, source=<optimized out>, tag=<optimized out>,
comm=0x203f10 <ompi_mpi_comm_world>, status=0x0) at precv.c:82
        #4  0x0000000000201aa1 in wrong() ()
        #5  0x0000000000201ad1 in main ()
```

Pass in Arguments

```
#include <iostream>
using namespace std;

int main(int argc, char** argv)
{
    cout << "You have entered " << argc
         << " arguments:" << "\n";

    for (int i = 0; i < argc; ++i)
        cout << argv[i] << "\n";

    return 0;
}
```

- Input:

```
./main CSC4005 Parallel Programming
```

- Output:

```
You have entered 4 arguments:
./test
CSC4005
Parallel
Programming
```

Pass in Arguments

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. The value of argc should be nonnegative.

- **argv (ARGument Vector)** is array of character pointers listing all the arguments.