Exercises 5:
Intro to MPI

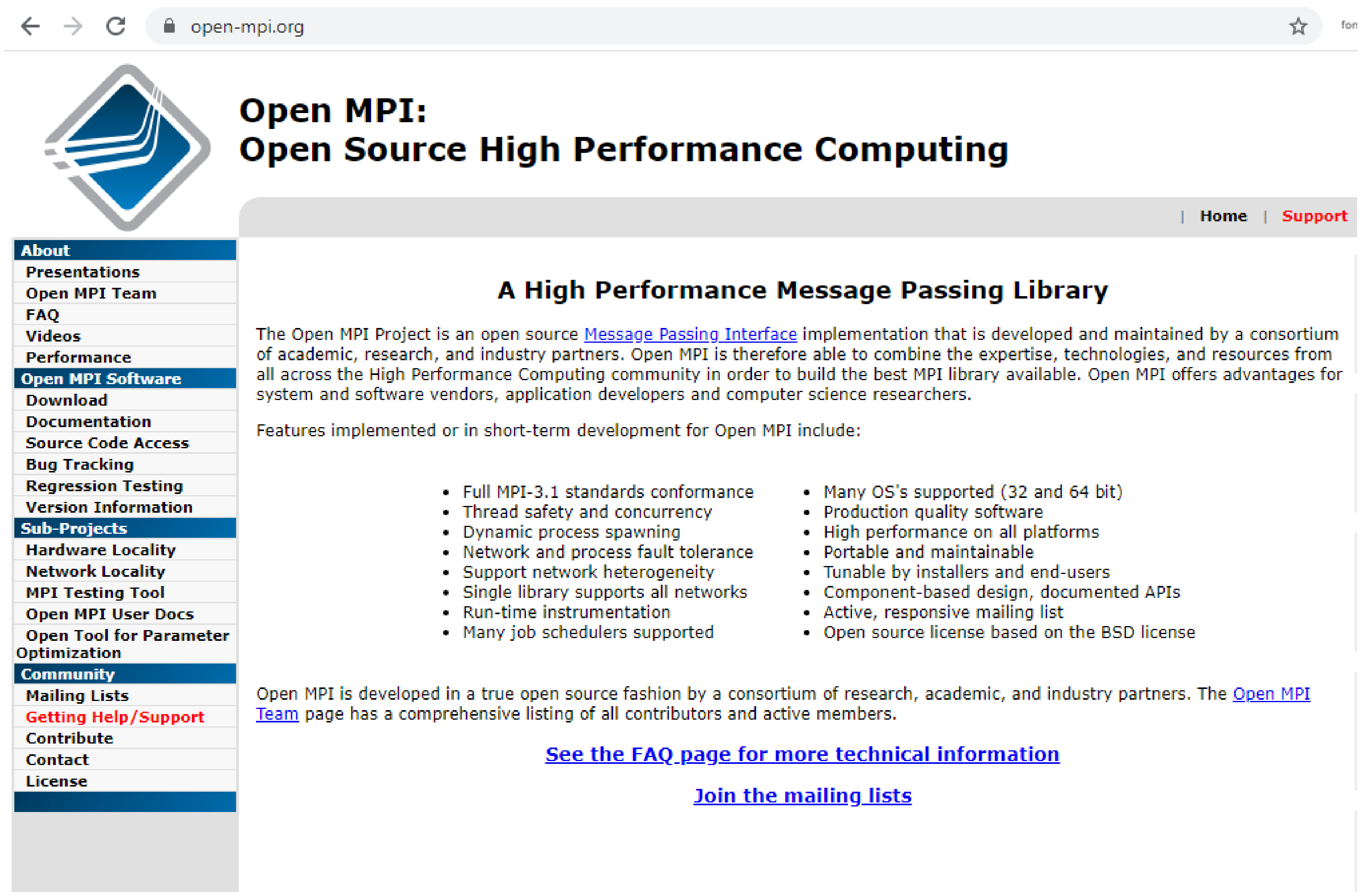
MPI on the Karlin cluster

- Open up a terminal and login to the cluster
- SCP the files for today's lab (ex5.tar on Moodle) to your home directory on the cluster

- In order to load MPI functionality on the cluster, we need to type

```
module load openmpi
```

OpenMPI Implementation



Open MPI:
Open Source High Performance Computing

A High Performance Message Passing Library

The Open MPI Project is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available. Open MPI offers advantages for system and software vendors, application developers and computer science researchers.

Features implemented or in short-term development for Open MPI include:

- Full MPI-3.1 standards conformance
- Thread safety and concurrency
- Dynamic process spawning
- Network and process fault tolerance
- Support network heterogeneity
- Single library supports all networks
- Run-time instrumentation
- Many job schedulers supported
- Many OS's supported (32 and 64 bit)
- Production quality software
- High performance on all platforms
- Portable and maintainable
- Tunable by installers and end-users
- Component-based design, documented APIs
- Active, responsive mailing list
- Open source license based on the BSD license

Open MPI is developed in a true open source fashion by a consortium of research, academic, and industry partners. The Open MPI Team page has a comprehensive listing of all contributors and active members.

See the FAQ page for more technical information

Join the mailing lists

Recall: Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **MPI_Comm_size** reports the number of processes.
  - **MPI_Comm_rank** reports the *rank*, a number between 0 and size-1, identifying the calling process

Example 1: Hello World in MPI

1. Open the file helloworld_mpi.c
2. Compile the code

   compiling on Karlin cluster:

   ```
   mpicc -o helloworld_mpi helloworld_mpi.c
   ```

3. Run the file

   Recall: The MPI Standard does not specify how to run an MPI program

   On the Karlin cluster (with 4 MPI processes):

   ```
   mpirun -n 4 ./programname
   ```

helloworld_mpi.c output

```
carson@r3d3:[~/ex5]: mpirun -n 3 ./helloworld_mpi
Hello World from process 0 of 3
Hello World from process 2 of 3
Hello World from process 1 of 3
```

```
carson@r3d3:[~/ex5]: mpirun -n 4 ./helloworld_mpi
Hello World from process 2 of 4
Hello World from process 1 of 4
Hello World from process 3 of 4
Hello World from process 0 of 4
```

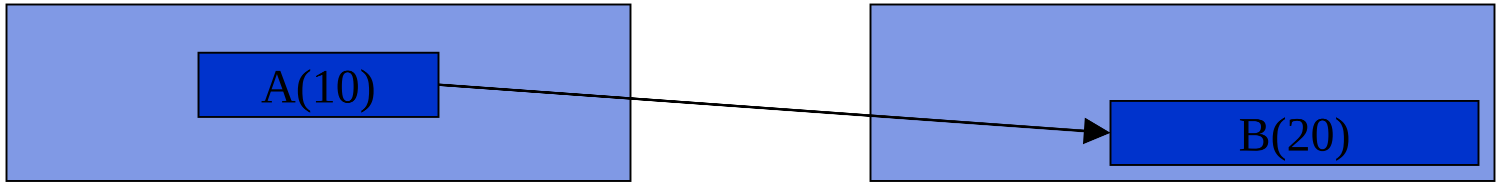
How many MPI processes to use?

- 1 process per node?
- 1 process per core?

- Option 1: Can treat the system as "flat" and use only MPI both intra-node and inter-node (i.e., even though memory is physically shared, treat it as distributed)
- Option 2: Hybrid approach: 1 MPI process per node, use shared memory programming for multiple cores within the node (e.g., MPI+OpenMP)

- Recall: If you want to actually run MPI across multiple nodes on the Karlin cluster, you will need to use the queuing system
  - Either write a batch script and submit the job, or use an interactive shell (see exercises 1)

Recall: MPI Basic (Blocking) Send



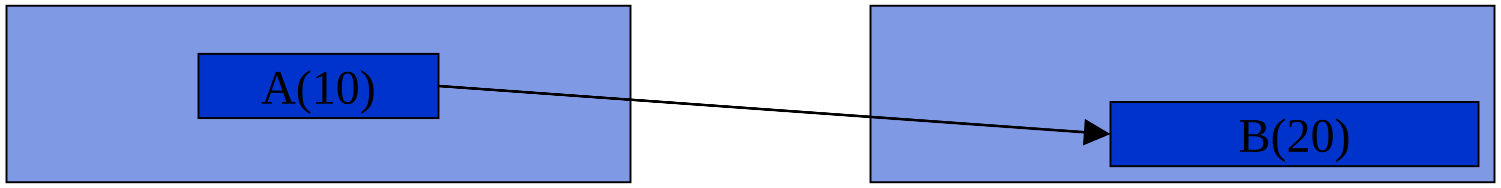
MPI_Send(A, 10, MPI_DOUBLE, 1, ...)

MPI_Recv(B, 20, MPI_DOUBLE, 0, ...)

`MPI_Send(address, count, datatype, dest, tag, comm)`

- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the memory in **`address`** can be reused. The message may not have been received by the target process.

Recall: MPI Basic (Blocking) Receive



MPI_Send(A, 10, MPI_DOUBLE, 1, ...)

MPI_Recv(B, 20, MPI_DOUBLE, 0, ...)

`MPI_Recv(address, count, datatype, source,tag, comm, status)`

- Waits until a matching (both **`source`** and **`tag`**) message is received from the system
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g., size of message)

Example 2: Basic Send and Recv

Send_recv.c is a simple program: send the number 123456 from process 0 to process 1

1. Open send_recv.c and read the file, try to understand it

A Simple MPI Program: send_recv.c

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {

        // Initialize the MPI environment
        MPI_Init(NULL, NULL);
        // Find out rank, size
        int mpirank, mpisize;
        MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);
        MPI_Comm_size(MPI_COMM_WORLD, &mpisize);

        MPI_Status status; int message;

        //Process 0 sends and process 1 receives
        if (mpirank == 0) {
               message = 123456;
                MPI_Send(&message, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        } else if (mpirank == 1) {
                MPI_Recv(&message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                               &status);
                printf("Process 1 received message %d from process
                      0\n", message);
        }
        MPI_Finalize();
}
```

Need an if/else since every process is running the same code

Example 2: Basic Send and Recv

Send_recv.c is a simple program: send the number 123456 from process 0 to process 1

1. Open send_recv.c and read the file, try to understand it
2. Compile it and run it with 2 processes

Task 1: MPI Ping Pong Program

- The next example is a ping pong program.
- In this example, processes use MPI_Send and MPI_Recv to continually bounce messages off of each other until they decide to stop.
- Take a look at ping_pong.c.

Task 1: MPI Ping Pong Program

- This example is meant to be executed with only two processes. The processes first determine their partner with some simple arithmetic.
- A ping_pong_count is initiated to zero and it is incremented at each ping pong step by the sending process.
- As the ping_pong_count is incremented, the processes take turns being the sender and receiver.
- Finally, after the limit is reached (10 iterations in the code), the processes stop sending and receiving.

- **Your task: Add an MPI_Send() call and an MPI_Recv() call to make the program work as described**
  - **Don't look at the solution on the next slide before you try it yourself!**

Solution

```
MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank,
         0, MPI_COMM_WORLD);
```

```
MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank,
         0, MPI_COMM_WORLD, &status);
```

or

```
MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank,
         MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

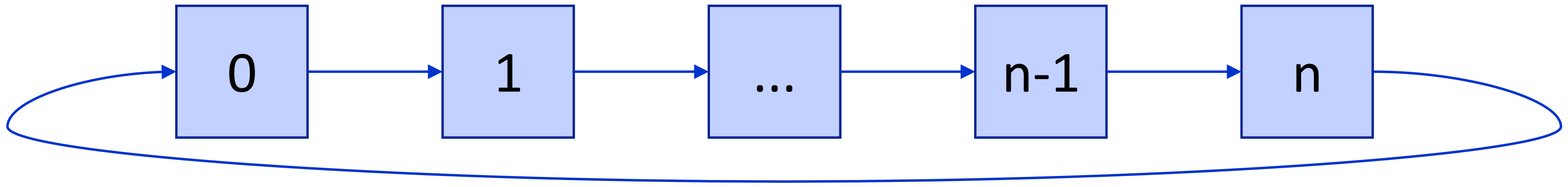
Task 2: MPI Ring communication

- Now we will look at an example of MPI_Send and MPI_Recv using more than two processes
- In this example, a value is passed around by all processes in a ring-like fashion.
- Take a look at ring.c.

MPI Ring Communication

- The ring program initializes a value from process zero, and the value is passed around every single process. (see pseudocode on next slide)
- The program should terminate when process zero receives the value from the last process.

- **Your task: Add the specified MPI_Send and MPI_Recv calls to the program**



Pseudocode

```
if not process 0,
      Recv() from process to the left
else (if process 0)
      set the value of the token


Send() to process to the right (or to process 0 if you
are the rightmost)


if process 0
      Recv() from the rightmost process
```

MPI Ring Communication

- Note the order in which the Send's and the Recv's are placed!
  - Extra care is taken to assure that it doesn’t deadlock.
  - Process zero makes sure that it has completed its first send before it tries to receive the value from the last process.
  - All of the other processes simply call MPI_Recv (receiving from their neighboring lower process) and then MPI_Send (sending the value to their neighboring higher process) to pass the value along the ring.
  - MPI_Send and MPI_Recv will block until the message has been transmitted.

Solution

```
if (mpirank != 0) {
    MPI_Recv(&token, 1, MPI_INT, mpirank - 1, 0,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else { token = -1;}


MPI_Send(&token, 1, MPI_INT, (mpirank + 1) % mpisize, 0,
         MPI_COMM_WORLD);


if (mpirank == 0) {
    MPI_Recv(&token, 1, MPI_INT, mpisize - 1, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Task 3: Non-blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
MPI_Request request, request2;
MPI_Status status;

 MPI_Isend(start, count, datatype,
     dest, tag, comm, &request);

 MPI_Irecv(start, count, datatype,
     dest, tag, comm, &request2);

 MPI_Wait(&request, &status);

 MPI_Wait(&request2, &status);
```

(each request must be Waited on)

- One can also test without waiting:

```
 MPI_Test(&request, &flag, &status);
```

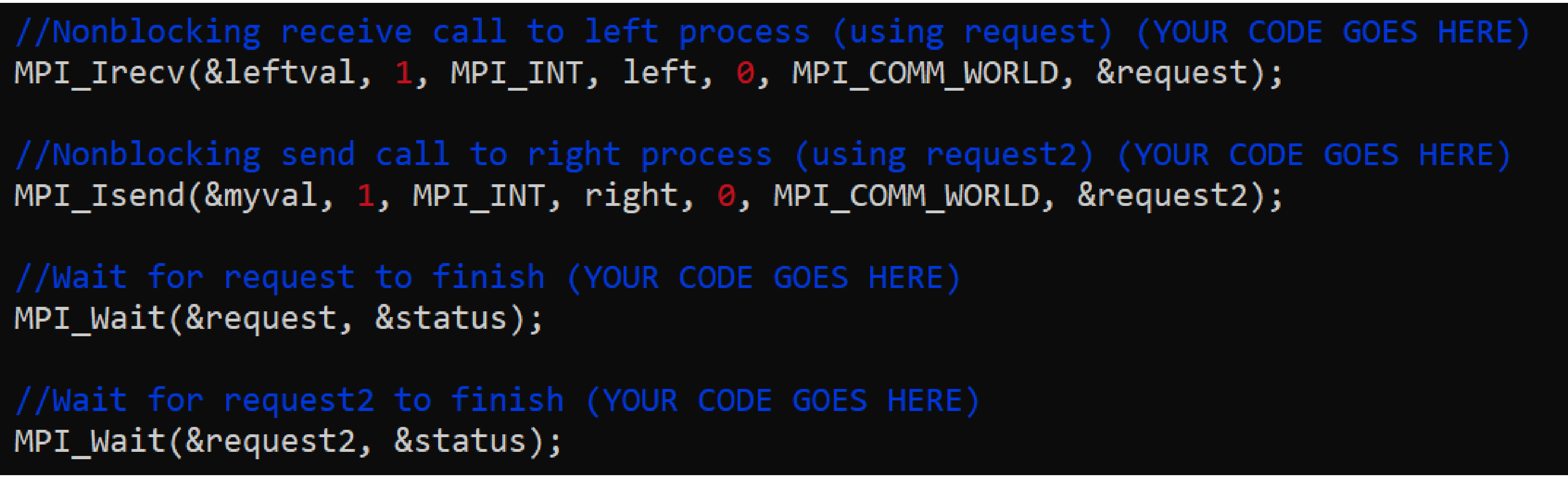
- Accessing the data buffer without waiting is undefined

Task 3: Nonblocking Operations

- The file nonblocking.c will use nonblocking sends and receives in order to do unidirectional nearest neighbor communication on a ring
  - send a value to right process, receive a value from left process

- Add the appropriate calls to MPI_Isend, MPI_Irecv, and MPI_Wait to the code to accomplish this

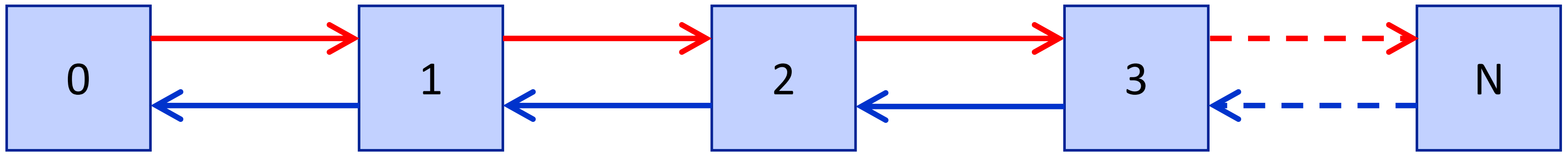
Solution



```
//Nonblocking receive call to left process (using request) (YOUR CODE GOES HERE)
MPI_Irecv(&leftval, 1, MPI_INT, left, 0, MPI_COMM_WORLD, &request);

//Nonblocking send call to right process (using request2) (YOUR CODE GOES HERE)
MPI_Isend(&myval, 1, MPI_INT, right, 0, MPI_COMM_WORLD, &request2);

//Wait for request to finish (YOUR CODE GOES HERE)
MPI_Wait(&request, &status);

//Wait for request2 to finish (YOUR CODE GOES HERE)
MPI_Wait(&request2, &status);
```

Task 4: Bidirectional Nearest Neighbor Exchange

Row of N processors. Each wants to exchange 1 double number with its neighbors.



```
//Set neighbors left and right
int left, right;
if (mpirank == 0)
        left = MPI_PROC_NULL;
else
        left = mpirank - 1;
if (mpirank == mpisize -1)
        right = MPI_PROC_NULL;
else
        right = mpirank +1;

//Sendrecv with right neighbor
MPI_Sendrecv(…);

//Sendrecv with left neighbor
MPI_Sendrecv(…);
```

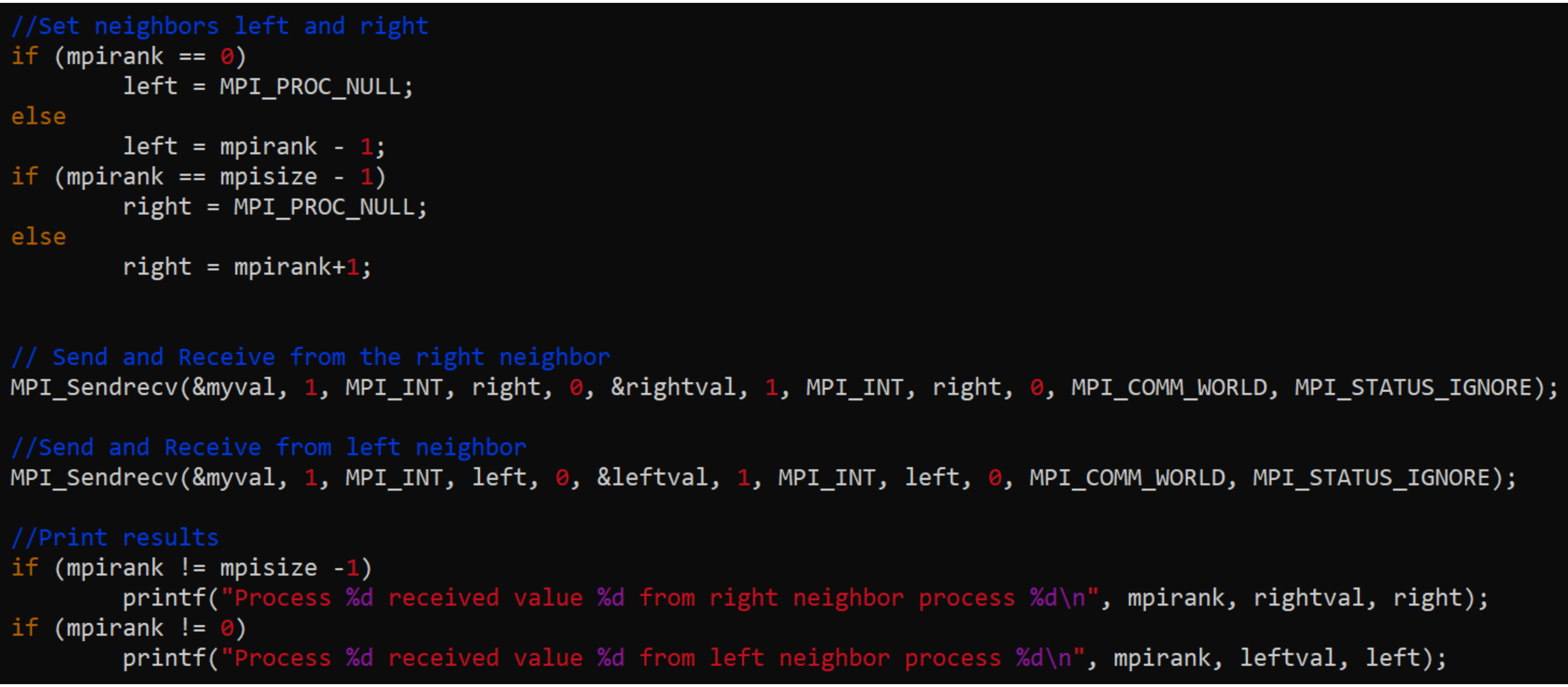
Recall: MPI_Sendrecv()

```
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  )
```

Task 4: Bidirectional Nearest Neighbor Exchange

- Add the appropriate MPI_Sendrecv() calls to the file neighbors.c to implement the nearest neighbor data exchange.

Solution



```
//Set neighbors left and right
if (mpirank == 0)
        left = MPI_PROC_NULL;
else
        left = mpirank - 1;
if (mpirank == mpisize - 1)
        right = MPI_PROC_NULL;
else
        right = mpirank+1;


// Send and Receive from the right neighbor
MPI_Sendrecv(&myval, 1, MPI_INT, right, 0, &rightval, 1, MPI_INT, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

//Send and Receive from left neighbor
MPI_Sendrecv(&myval, 1, MPI_INT, left, 0, &leftval, 1, MPI_INT, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

//Print results
if (mpirank != mpisize -1)
        printf("Process %d received value %d from right neighbor process %d\n", mpirank, rightval, right);
if (mpirank != 0)
        printf("Process %d received value %d from left neighbor process %d\n", mpirank, leftval, left);
```

Note: you could also send to left and receive from right at the same time, and then send to right and receive from left

Thinking in terms of distributed memory

Problem: want to create an array of size N and set every entry a[i]=i^2

In shared memory with OpenMP, we would still allocate an array of size N, and then have threads parallelize the setting of the entries

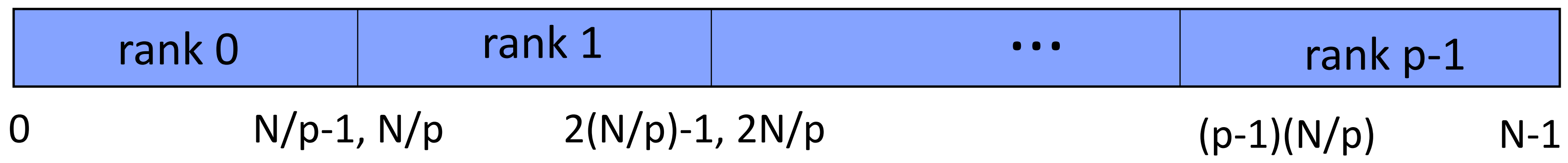
```
#pragma omp parallel for
for(int i = 0; i < N; i++)
   a[i] = i*i;
```

Now we have distributed memory. *There is no global array.*

- Each process will have a local array of size `my_N = N/mpisize`
  Each process will have to know which entries of the global array they are responsible for

Distributed Memory

Simple distribution: by contiguous chunks:



```
int my_N = N/mpisize;
int start = mpirank*my_N;

int *local_arr = (int*) calloc(sizeof(int), my_N);

for (i = 0; i < my_N; i++)
      local_arr[i] = (start+i)*(start+i);
```

Look at arrayex.c and try to understand what is happening