1 Background

This lab assumes that the previous assignment has already been completed to set up a Raspberry Pi cluster. It also assumes familiarity with a terminal program and how to SSH into each Raspberry Pi cluster node.

In the first lab, we constructed a Raspberry Pi cluster computer and benchmarked it using an existing tool, HPL, that relied on MPI for communication between cluster nodes. This lab will build on that progress by presenting several C programs using the MPI interface to show how to develop parallel programs across a cluster using MPI.

To aid development and better simulate a realistic cluster environment, a shared directory will be created so that programs written on one cluster node are automatically accessible on the other nodes via the file system. This is in contrast to the previous lab, where separate copies of all software was needed.

2 Create a shared directory for the cluster

The goal of this section is to set up an NFS (Network File System) server on the router node to create a directory that is shared with the other nodes in the cluster. Programs placed in the shared directory can then be run on the whole cluster without needing to manually transfer copies to other nodes.

Note: An NFS server can be run from any cluster node, and this lab only runs the NFS server from the router node for simplicity's sake.

2.1 Configure the NFS server on router node

- Establish an SSH connection to the router node. The remainder of Section 2.1 assumes the following commands are run on the router node.
- Run the following commands on the router node:

```
# install the NFS server software
sudo apt install nfs-kernel-server

# create a directory to share among all cluster nodes
sudo mkdir /home/shared

# allow any user to read, write, and execute the shared files
sudo chmod 777 /home/shared

# change the owner of the directory to the "pi" user.
# this is necessary since the "pi" user is who shares the directory
sudo chown -R pi:pi /home/shared
```

- Run `sudo nano /etc/exports` to open the configuration file that controls how the file system is exported to remote hosts (in this case, the other nodes in the cluster).

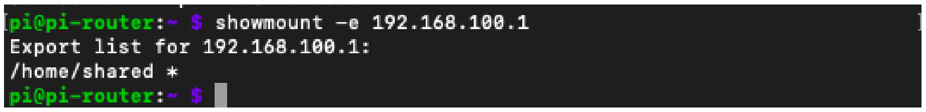
- Add the following line to the end of the file, which exports the `/home/shared` directory:

  ```
  /home/shared *(rw,all_squash,nohide,insecure,no_subtree_check,async)
  ```

- Save and close the file.

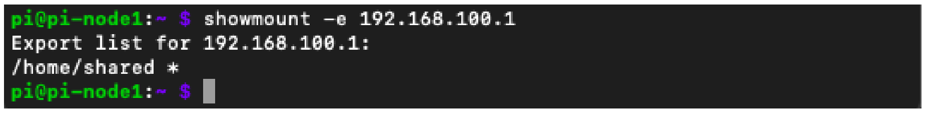
- Run `sudo reboot` and reconnect after the node reboots. At that point, it should be automatically exporting the `/home/shared` directory.

- To test that the directory is mounted, run the command `showmount -e 192.168.100.1` (relying on the router node's static IP address of `192.168.100.1`):



```
pi@pi-router:~ $ showmount -e 192.168.100.1
Export list for 192.168.100.1:
/home/shared *
pi@pi-router:~ $
```

- You can also test the command from a different cluster node, which should yield the same results:



```
pi@pi-node1:~ $ showmount -e 192.168.100.1
Export list for 192.168.100.1:
/home/shared *
pi@pi-node1:~ $
```

2.2 Mount shared directory on non-router nodes

These steps will make the `/home/shared` directory on the router node available from another cluster node. This will need to be completed on each non-router node.

- SSH into the non-router node.

- Run the following commands on the non-router node:

  ```
  # install the NFS client software
  sudo apt install nfs-common

  # create a directory on which to mount the shared directory
  sudo mkdir /home/shared

  # allow any user to read, write, and execute the shared files
  sudo chmod 777 /home/shared
  ```

- Run `sudo nano /etc/fstab` to open the file systems table for editing.

- At the end of the file, add the following on a single line (it's broken up into two lines here to fit the page; use a space instead of a new line):

```
192.168.100.1:/home/shared /home/shared nfs
    rw,hard,intr,noauto,x-systemd.automount 0 0
```

- Save and close the file.
- Run `sudo reboot` and reconnect after the node reboots. When it does, it should have automatically mounted the shared directory that was exported from the router node.
- To test that the shared directory is working as expected, run `touch /home/shared/ping` from the non-router node to write an empty file to the shared directory.
- On the router node, run `ls /home/shared` to check that the empty file has appeared there as well.

3 "Hello World" in MPI

In this section, you will run a simple program on the cluster using MPI's C interface. The program will be written on the router node in the shared directory created in Section 2, where it will be visible to all other nodes.

- SSH into the router node. The commands in this section will be run on the router node.
- Run `which mpicc` to verify the presence of the MPI C compiler (it should have been installed with MPICH and added to the system path in a previous section). If `mpicc` is correctly installed, the `which` command will print its location in the file system.
- Run the following commands to prepare for writing the Hello World program:

```
# create a directory for the program's files
mkdir /home/shared/helloworld

# move to the created directory
cd /home/shared/helloworld

# copy the previously used hostfile into this directory
cp ~/hpl-2.3/bin/rpi/hostfile-all.txt .

# start editing the program source code
nano helloworld.c
```

- Write the following program to `helloworld.c`:

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

int main(int argc, char** argv) {
  int process_rank, number_of_processes, hostname_length;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &number_of_processes);
  MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

  MPI_Get_processor_name(hostname, &hostname_length);

  printf("Hello world from process %d of %d on node %s\n",
    process_rank, number_of_processes, hostname);

  MPI_Finalize();
  return 0;
}
```

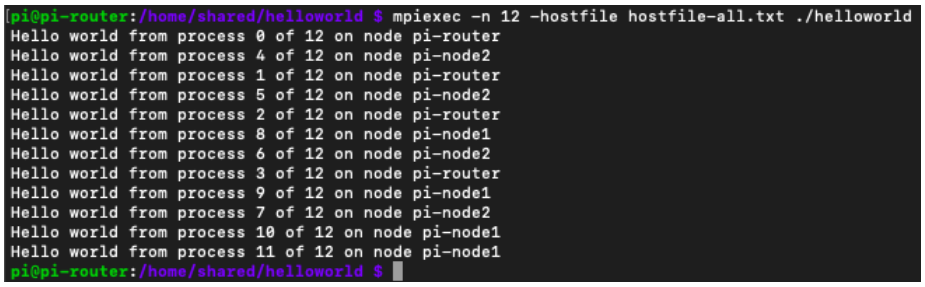
- Run the command `mpicc helloworld.c -o helloworld` to compile the program into the executable `helloworld`.
- Use `mpiexec` to run the program with various numbers of processes as follows:

```
# by default, mpiexec uses 1 process on the current node
mpiexec ./helloworld

# use the -n argument to assign different numbers of processes:
mpiexec -n 3 ./helloworld
mpiexec -n 12 ./helloworld

# use the -hostfile argument to assign processes on the other cluster nodes:
mpiexec -n 3 -hostfile hostfile-all.txt ./helloworld
mpiexec -n 12 -hostfile hostfile-all.txt ./helloworld
```

- The following screenshot shows the expected output when the program is run using 12 processes on the 3-node cluster.



```
pi@pi-router:/home/shared/helloworld $ mpiexec -n 12 -hostfile hostfile-all.txt ./helloworld
Hello world from process 0 of 12 on node pi-router
Hello world from process 4 of 12 on node pi-node2
Hello world from process 1 of 12 on node pi-router
Hello world from process 5 of 12 on node pi-node2
Hello world from process 2 of 12 on node pi-router
Hello world from process 8 of 12 on node pi-node1
Hello world from process 6 of 12 on node pi-node2
Hello world from process 3 of 12 on node pi-router
Hello world from process 9 of 12 on node pi-node1
Hello world from process 7 of 12 on node pi-node2
Hello world from process 10 of 12 on node pi-node1
Hello world from process 11 of 12 on node pi-node1
pi@pi-router:/home/shared/helloworld $
```

Note: By default, `mpiexec` assigns processes to the nodes listed in the hostfile in a round-robin way. That is, one process is assigned to each listed node in order until the required number of processes are assigned, looping through the file if necessary. This is reflected above, where each node is assigned 4 processes.

- The Hello World program uses several key features of the MPI interface:
  - `MPI_Init` and `MPI_Finalize` must be the first and last MPI calls in any program using MPI. These functions are used to create and destroy the MPI environment, so all MPI usage must occur between their calls.
  - `MPI_COMM_WORLD` is an example of an MPI *communicator*. Each communicator refers to a group of processes, and two different processes can only communicate if they share a communicator. In this case, `MPI_COMM_WORLD` is the default communicator that contains every allocated process.
  - `MPI_Comm_size` determines the number of processes belonging to the communicator passed as an argument.
  - Each communicator maintains a list of processes belonging to it, and the *rank* of a process is its position in that list. `MPI_Comm_rank` determines the rank of the caller's process in the communicator.
  - `MPI_Get_processor_name` determines the name of the caller's processor. This can mean different things on different systems, but on the Raspberry Pi cluster it will be the hostname of the caller's node.

4 Prime counting

In this section we will write a program that counts how many prime numbers there are below $10,000,000$. The program will use a very naive method known as *trial division*, which decides if the number n is prime by checking if n is divisible by any of the numbers between 2 and $\sqrt{n}$ (if it isn't, then n is prime). The benefit of this method, however, is that it is straightforward to split up the work between any number of parallel processes without needing any interprocess communication.

Processes used	**Real-time duration (s)**
4	21.754
6	14.681
10	9.023
12	7.900

Table 1: Sample real-time durations of the prime-counting program.

- The commands in this section will once again be run on the router node.
- Run the following commands to prepare for writing the prime counting program:

  ```
  mkdir /home/shared/primes
  cd /home/shared/primes
  cp ~/hpl-2.3/bin/rpi/hostfile-all.txt .
  ```

- Run `nano primes.c` and enter the program given in Figure 1.
- Run the following command to compile the program:

  ```
  mpicc ./primes.c -o primes -lm
  ```

  Note: The argument `-lm` is required to link the math library that implements the `sqrt` function.

- Run the following command to execute the program on the cluster with 12 processes:

  ```
  time mpiexec -n 12 -hostfile hostfile-all.txt ./primes
  ```

  Note: The `time` command prefix times the following `mpiexec` command and prints the result. This is a simple way to time any command that doesn't have a built-in timer. For multithreaded programs, the `real`-time measurement is more useful than the `user` and `system` measurements.

- Modify the previous command to time the program's execution when 4, 6, and 10 processes are used instead of 12. Verify that your observed execution times are similar to Table 1.

  Note: The issue in Assignment 1 (where an incorrect number of processes were used by MPI) does not occur here. The `-n` argument to `mpiexec` seems to correctly spawn the desired number of processes across the cluster.

  Note: Each process searches for primes among the numbers of the form $(r + 1) + k(p + 1)$, where r is the process' rank and p is the total number of processes. This works well when $p + 1$ is itself a prime number, in

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

#define UPPER_LIMIT 10000000

int is_prime(int number) {
  if (number < 2) return 0;

  for (int i = 2; i <= sqrt((double)number); i++) {
    if (number % i == 0) return 0;
  }

  return 1;
}

int main(int argc, char** argv) {
  int rank, process_count;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &process_count);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int primes_found = 0;
  int a = rank + 1;
  int k = process_count + 1;
  for (int n = a; n <= UPPER_LIMIT; n += k) {
    if (is_prime(n) == 1) ++primes_found;
  }

  printf("process %d found %d primes\n", rank, primes_found);

  int primes_found_total, primes_found_max, primes_found_min;
  MPI_Reduce(&primes_found, &primes_found_total, 1, MPI_INT,
             MPI_SUM, 0, MPI_COMM_WORLD);
  MPI_Reduce(&primes_found, &primes_found_max, 1, MPI_INT,
             MPI_MAX, 0, MPI_COMM_WORLD);
  MPI_Reduce(&primes_found, &primes_found_min, 1, MPI_INT,
             MPI_MIN, 0, MPI_COMM_WORLD);

  if (rank == 0) {
    primes_found_total += is_prime(k);
    printf("found %d primes less than or equal to %d\n",
           primes_found_total, UPPER_LIMIT);
    printf("max found by one process = %d\n, primes_found_max);
    printf("min found by one process = %d\n, primes_found_min);
  }

  MPI_Finalize();
  return 0;
}
```

Figure 1: `primes.c` code listing

which case the primes will roughly be evenly distributed between processes (by Dirichlet's prime number theorem). When $p + 1$ is not prime, however, then the workload will be highly imbalanced between processes (with some processes finishing almost immediately) and the observed execution duration will go up significantly.

- The only new feature of MPI used by the prime-counting program is `MPI_Reduce`. Generally speaking, `MPI_Reduce` is passed a value from each process in a communicator and *reduces* those values into a single result via an operation that's also specified in the call.

  For example, given the call:

  ```
  MPI_Reduce(&primes_found, &primes_found_total, 1,
             MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  ```

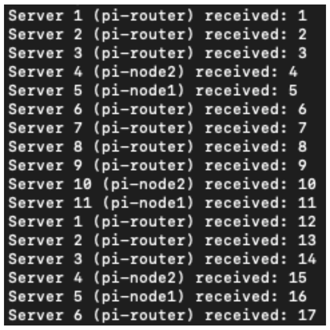
  - The call will receive the value of `primes_found` (which is of type `MPI_INT`) from each process in the `MPI_COMM_WORLD` communicator.
  - It will reduce those values via the `MPI_SUM` operation, which adds the values together as it receives them.
  - The resulting sum of all values will be stored in the variable `primes_found_total`.
  - Only the process with rank 0 (specified in the second-to-last argument) will receive the result. The process that receives the result is also known as the *root* process.
  - `MPI_Reduce` is *blocking*, which means that MPI will pause processes that reach the reduce call until every process in the communicator has reached that point in the program. This ensures that the final sum won't be printed by the root process until each process has contributed its `primes_found` value to the total sum.

- Similarly, the second call to `MPI_Reduce` uses the `MPI_MAX` operation, which stores the maximum of all `primes_found` values into `primes_found_max`. The third call uses the `MPI_MIN` operation, which stores the minimum value into `primes_found_min`.

5 Round-robin scheduler

This section will implement a minimal round-robin scheduler using MPI. In this context, *round-robin* means using processes in a circular order, returning to use the first process after the last process has been used.

Given n processes, our program will use the rank 0 process as the *scheduler* process and the remaining $n - 1$ processes (with ranks 1 through $n - 1$) as *server* processes. The scheduler will pass messages (represented by increasing numbers) to the servers in round-robin order, as shown in the following screenshot:



```
Server 1 (pi-router) received: 1
Server 2 (pi-router) received: 2
Server 3 (pi-router) received: 3
Server 4 (pi-node2) received: 4
Server 5 (pi-node1) received: 5
Server 6 (pi-router) received: 6
Server 7 (pi-router) received: 7
Server 8 (pi-router) received: 8
Server 9 (pi-router) received: 9
Server 10 (pi-node2) received: 10
Server 11 (pi-node1) received: 11
Server 1 (pi-router) received: 12
Server 2 (pi-router) received: 13
Server 3 (pi-router) received: 14
Server 4 (pi-node2) received: 15
Server 5 (pi-node1) received: 16
Server 6 (pi-router) received: 17
```

- Create the `roundrobin` directory for this section's program in the `/home/shared` directory as in the previous sections.
- Copy the `hostfile-all.txt` hostfile to the newly created `roundrobin` directory.
- Begin editing the `roundrobin.c` file inside the `roundrobin` directory and write the program given in Figure 2.
- Run the following command to compile the program:

  ```
  mpicc ./roundrobin.c -o roundrobin
  ```

- Run the following command to execute the program:

  ```
  mpiexec -n 12 -hostfile hostfile-all.txt ./roundrobin
  ```

  Note: The shortcut Control+C can be used to exit the program while it's still running and regain control of the terminal. This shortcut isn't specific to this program and should interrupt any program running in a terminal.

- This program introduces the concept of *message passing* in MPI, after which MPI is named.
  - The scheduler process sends a message with the call:

    ```
    MPI_Send(&next_data, 1, MPI_INT, next_server, 0, MPI_COMM_WORLD);
    ```

    This sends the integer variable `next_data` to the process with rank `next_server`. `MPI_Send` is *blocking*, which means that the scheduler process halts execution until the message is received by the targeted server process.
  - A server process receives a message with the call:

    ```
    MPI_Irecv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
    ```

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

void run_scheduler_process(int process_count) {
  int next_server = 1, next_data = 1;
  while (1) {
    MPI_Send(&next_data, 1, MPI_INT, next_server,
             0, MPI_COMM_WORLD);
    next_data++;
    next_server++;
    if (next_server == process_count) next_server = 1;
    usleep(300000);
  }
}

void run_server_process(int rank, char* hostname) {
  int data, request_complete;
  MPI_Request request;
  while (1) {
    MPI_Irecv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
    request_complete = 0;
    while (!request_complete) {
      MPI_Test(&request, &request_complete, MPI_STATUS_IGNORE);
      usleep(300000);
    }
    printf("Server %d received: %d\n", rank, hostname, data);
  }
}

int main(int argc, char** argv) {
  int rank, process_count, hostname_length;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &process_count);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(hostname, &hostname_length);

  if (rank == 0) {
    printf("Starting scheduler (rank 0) on %s\n", hostname);
    run_scheduler_process(process_count);
  } else {
    printf("Starting server (rank %d) on %s\n", rank, hostname);
    run_server_process(rank, hostname);
  }

  MPI_Finalize();
  return 0;
}
```

Figure 2: `roundrobin.c` code listing

This receives the integer sent by the process of rank 0 (the scheduler) and stores it in the variable `data`. `MPI_Irecv` is *non-blocking*, which means that the server process will continue execution after the call, and that the message will be received and stored in `data` asynchronously (at some unknown future time). The status of the message is maintained in `request`, and will reflect when the data has been received.

– Since `MPI_Irecv` is asynchronous, the caller needs to test the `request` variable to know when the message has been received. The following `while` loop accomplishes this using `MPI_Test`:

```
request_complete = 0;
while(!request_complete) {
    MPI_Test(&request, &request_complete, MPI_STATUS_IGNORE);
    usleep(300000);
}
```

The `MPI_Test` call checks if `request` refers to a completed message, and if so will set `request_complete` to 1, which will lead to the loop being exited.

– Using the blocking receive function `MPI_Recv` instead of `MPI_Irecv` is undesirable here since the messages are sent to the server relatively infrequently. A blocking call uses a high amount of CPU while waiting, which would be inefficient given the sparsity of the messages.

6 Broadcast benchmark

This section is adapted from the `MPI_Bcast` article of Resource 3.

The previous section analyzed a program where messages were sent from one process to another. In some cases, however, it is desirable for data to be shared throughout a larger group of processes (in MPI, processes are grouped together into *communicators*). The MPI function `MPI_Bcast` exists for this purpose: to efficiently send a message from one root process in a communicator to the rest.

Of course, this functionality could be duplicated naively by repeatedly calling `MPI_Send` from the root process, targeting each other nonroot process in the communicator. The programs in this section will compare repeated calls of `MPI_Send` to using `MPI_Bcast`, and will show that `MPI_Bcast` significantly outperforms the naive method even in a relatively small communicator.

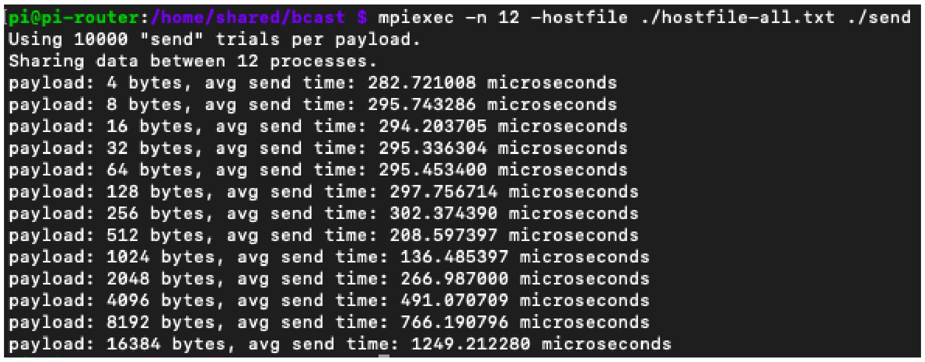
To explain the gap in performance between `MPI_Send` and `MPI_Bcast`, consider a loop such as the following, which naively sends data from the root process of rank 0 to the n nonroot processes of ranks 1 through n:

```
for (int rank = 1; rank < n; rank++) {
  MPI_Send(&data, 1, MPI_INT, rank, 0, MPI_COMM_WORLD);
}
```

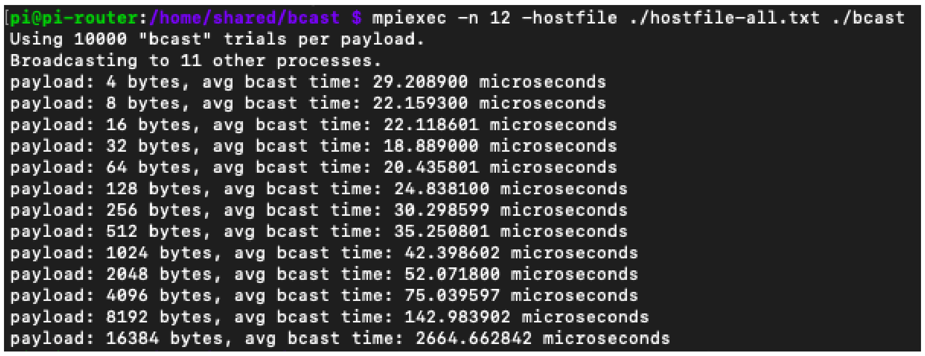
Before the loop starts, the data is only present on process 0. After one iteration, process 0 sends the data to process 1, so the data is present on those two processes. Thus, in this implementation, process 1 idles for the remainder of the loop while it waits for process 0 to send the data to the remaining processes.

This is an inefficiency that `MPI_Bcast` at least partially alleviates. When using `MPI_Bcast`, nonroot processes that have been sent the shared data will continue sending it to other processes that still need it, rather than idling until the end of the loop as in the naive method. This extra functionality is built into `MPI_Bcast` automatically, and will be used by MPI when `MPI_Bcast` is called on each process in the communicator. This is demonstrated in the `bcast.c` program in Figure 4.

- Create the `bcast` directory for this section's program in the `/home/shared` directory as in the previous sections.
- Copy the `hostfile-all.txt` hostfile to the newly created `bcast` directory.
- Write the programs given in Figures 3 and 4 into the files `send.c` and `bcast.c`, respectively.
- Use `mpicc` to compile the two programs into the `send` and `bcast` executables.
- Use `mpiexec` to execute the two programs using the usual hostfile and 12 processes.
- Run both programs. The output for `send` and `bcast` should look similar to the following screenshots, respectively:



```
pi@pi-router:/home/shared/bcast $ mpiexec -n 12 -hostfile ./hostfile-all.txt ./send
Using 10000 "send" trials per payload.
Sharing data between 12 processes.
payload: 4 bytes, avg send time: 282.721008 microseconds
payload: 8 bytes, avg send time: 295.743286 microseconds
payload: 16 bytes, avg send time: 294.203705 microseconds
payload: 32 bytes, avg send time: 295.336304 microseconds
payload: 64 bytes, avg send time: 295.453400 microseconds
payload: 128 bytes, avg send time: 297.756714 microseconds
payload: 256 bytes, avg send time: 302.374390 microseconds
payload: 512 bytes, avg send time: 208.597397 microseconds
payload: 1024 bytes, avg send time: 136.485397 microseconds
payload: 2048 bytes, avg send time: 266.987000 microseconds
payload: 4096 bytes, avg send time: 491.070709 microseconds
payload: 8192 bytes, avg send time: 766.190796 microseconds
payload: 16384 bytes, avg send time: 1249.212280 microseconds
```



```
[pi@pi-router:/home/shared/bcast $ mpiexec -n 12 -hostfile ./hostfile-all.txt ./bcast
Using 10000 "bcast" trials per payload.
Broadcasting to 11 other processes.
payload: 4 bytes, avg bcast time: 29.208900 microseconds
payload: 8 bytes, avg bcast time: 22.159300 microseconds
payload: 16 bytes, avg bcast time: 22.118601 microseconds
payload: 32 bytes, avg bcast time: 18.889000 microseconds
payload: 64 bytes, avg bcast time: 20.435801 microseconds
payload: 128 bytes, avg bcast time: 24.838100 microseconds
payload: 256 bytes, avg bcast time: 30.298599 microseconds
payload: 512 bytes, avg bcast time: 35.250801 microseconds
payload: 1024 bytes, avg bcast time: 42.398602 microseconds
payload: 2048 bytes, avg bcast time: 52.071800 microseconds
payload: 4096 bytes, avg bcast time: 75.039597 microseconds
payload: 8192 bytes, avg bcast time: 142.983902 microseconds
payload: 16384 bytes, avg bcast time: 2664.662842 microseconds
```

Note: Interestingly, the performance of `MPI_Bcast` drops sharply on the largest payload size, doing much worse than `MPI_Send`. It's not clear why this might be.

7 Resources

1. Documentation for the MPICH C language API:

   `https://www.mpich.org/static/docs/latest/`

2. Documentation for OpenMPI, another implementation of MPI:

   `https://www.open-mpi.org/doc/current/`

3. Articles explaining some introductory concepts of MPI programming:

   `https://mpitutorial.com/tutorials/`

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

#define TRIALS 10000
#define COMM MPI_COMM_WORLD

long long get_microsecond_timestamp() {
  struct timespec ts;
  clock_gettime(CLOCK_REALTIME, &ts);
  return (ts.tv_sec) * 1e6 + ts.tv_nsec / 1e3;
}

void run_root_process(int data_size, int world_size) {
  int data[data_size];
  long long start_time = get_microsecond_timestamp();
  for (int trial = 0; trial < TRIALS; trial++) {
    for (int rank = 1; rank < world_size; rank++) {
      MPI_Send(&data, data_size, MPI_INT, rank, 0, COMM);
    }
  }
  long long duration = get_microsecond_timestamp() - start_time;
  float average_duration = (float)duration / TRIALS;
  printf("payload: %d bytes, avg send time: %.6f microseconds\n",
         sizeof(int) * data_size, average_duration);
}

void run_nonroot_process(int data_size) {
  int data[data_size];
  for (int trial = 0; trial < TRIALS; trial++) {
    MPI_Recv(&data, data_size, MPI_INT, 0, 0, COMM,
             MPI_STATUS_IGNORE);
  }
}

int main(int argc, char** argv) {
  int rank, world_size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(COMM, &rank);
  if (rank == 0) {
    MPI_Comm_size(COMM, &world_size);
    printf("Using %d \"send\" trials per payload.\n", TRIALS);
    printf("Sharing data between %d processes.\n", world_size);
  }
  for (int size = 1; size < (2 << 12); size *= 2) {
    if (rank == 0) run_root_process(size, world_size);
    else run_nonroot_process(size);
  }
  MPI_Finalize();
  return 0;
}
```

Figure 3: `send.c` code listing

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

#define TRIALS 10000
#define COMM MPI_COMM_WORLD

long long get_microsecond_timestamp() {
  struct timespec ts;
  clock_gettime(CLOCK_REALTIME, &ts);
  return (ts.tv_sec) * 1e6 + ts.tv_nsec / 1e3;
}

void run_root_process(int data_size) {
  int data[data_size];
  long long start_time = get_microsecond_timestamp();
  for (int trial = 0; trial < TRIALS; trial++) {
    MPI_Bcast(&data, data_size, MPI_INT, 0, COMM);
  }
  long long duration = get_microsecond_timestamp() - start_time;
  float average_duration = (float)duration / TRIALS;
  printf("payload: %d bytes, avg bcast time: %.6f microseconds\n",
         sizeof(int) * data_size, average_duration);
}

void run_nonroot_process(int data_size) {
  int data[data_size];
  for (int trial = 0; trial < TRIALS; trial++) {
    MPI_Bcast(&data, data_size, MPI_INT, 0, COMM);
  }
}

int main(int argc, char** argv) {
  int rank, world_size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(COMM, &rank);
  if (rank == 0) {
    MPI_Comm_size(COMM, &world_size);
    printf("Using %d \"bcast\" trials per payload.\n", TRIALS);
    printf("Broadcasting to %d other processes.\n",
           world_size - 1);
  }
  for (int size = 1; size < (2 << 12); size *= 2) {
    if (rank == 0) run_root_process(size);
    else run_nonroot_process(size);
  }
  MPI_Finalize();
  return 0;
}
```

Figure 4: `bcast.c` code listing