Parallelization: Conway's Game of Life
By Aaron Weeden, Shodor Education Foundation, Inc.

Exercise 2

This exercise will take you through writing a small piece of C code, outfitting it with MPI and OpenMP directives, and running it on a cluster.

To complete this exercise, you will need access to a Linux cluster with a Portable Batch System (PBS) scheduler, a `gcc` or `gfortran` compiler, and an MPI binding with support for C or Fortran 90. Other compilers may be used, with possible differences in performance.

The example cluster used in this module is al-salam, part of the Earlham College Cluster Computing Group. If you wish to use al-salam for this exercise, you can contact the Cluster Computing Group to obtain accounts by sending email to ccg@cs.earlham.edu.

You will make use of the **vi** text editor, which is provided by default on most Linux-based operating systems, such as the one used by Earlham's cluster. For an overview of vi, see http://www.eng.hawaii.edu/Tutor/vi.html.

In this exercise, any line with a dollar sign (`$`) in front of it is a command to be entered in a shell (a command line utility used by the operating system to interact with the user).

Part I: Write, compile, and run a serial program

1. Log into the cluster. The example here is al-salam – note that you must first log into hopper, which is al-salam's gateway.

```
$ ssh <yourusername>@cluster.earlham.edu
$ ssh as0
$
```

2. Create a small "Hello, World" program in C or Fortran 90:
   a. Open a new file called `hello.c` or `hello.F90` in vi:

```
$ vi hello.c
$
```

OR

```
$ vi hello.F90
$
```

   b. Enter vi's "insert mode" by pressing the **i** key.

c. Write a small C or Fortran 90 code that will print "`Hello, World!`" on the screen:

C

```
#include <stdio.h>

int main(int argc, char** argv)
{
     printf("Hello, World!\n");

     return 0;
}
```

Fortran 90

```
PROGRAM hello
IMPLICIT NONE

PRINT *, "Hello, World!"

END PROGRAM hello
```

d. Press **Escape (esc)** to exit vi's insert mode.
e. Save the file and exit vi by entering **<Shift>-Z-Z**.

3. Compile the code with GNU's compiler. This will produce an executable file called `hello`:

```
$ gcc -o hello hello.c
$
     OR
$ gfortran -o hello hello.F90
$
```

If any errors are listed, make sure there are no typos in `hello.c` or `hello.F90` (go back through step 2).

4. Create a script to run the program on the cluster.
   a. Open a new file called `hello.qsub` in vi:
      `$ vi hello.qsub`
   b. Enter insert mode (as you did in step 2b) and write a small Portable Batch System (PBS) script:

```
#PBS -q ec
```

```
#PBS -o hello.out
#PBS -e hello.err

cd $PBS_O_WORKDIR

./hello
```

Each line of this script tells the scheduler to do something:
`#PBS -q ec` says to use the "ec" queue. Change this value to the name of the queue on the cluster you are using.
`#PBS -o hello.out` says to save the output of standard out to a file called `hello.out` rather than to print it on the terminal.
`#PBS -e hello.err` says to save the output of standard error to a file called `hello.err` rather than to print it on the terminal.
`cd $PBS_O_WORKDIR` tells the scheduler to change directories to the directory from which the job is submitted.
`./hello` says to run the `hello` executable.

c. Save the file and exit vi (as you did in steps 2d and 2e).

5. Submit a job to the scheduler:

```
$ qsub hello.qsub
19098.as0.al-salam.loc
$
```

6. This will submit a job and output its job ID, `19098` in this example. Your job will now be waiting in the queue, running, or finished. You can monitor it at any time by entering `qstat 19098` (or whatever your Job ID is) in the shell.

   You may see something like the following:

```
qstat 19098
qstat: Unknown Job Id 19098.as0.al-salam.loc
```

   This means the job is complete.

   If the job were instead still running, you would see something like the following table:

```
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
19098.as0                 STDIN            amweeden06             0 R ec
```

In this output, the S column is the status column. The letter under this column tells you the status of the job; Q means it is waiting in the queue and R means it is running.

7. Once the job is complete, show the contents of `hello.out` with the `cat` command:
```
$ cat hello.out
Hello, World!
$
```

8. `hello.err` should be empty if there were no errors in running the program. Show the contents of `hello.err` with the `cat` command:
```
$ cat hello.err
$
```

   If this command returns just a prompt (`$`), then the file is empty and there were no errors. Otherwise, the errors will be listed.

Part II. Outfit the program with MPI

9. We will now make a parallel version of the code using MPI. First we will tell the program to include the MPI library. We also tell the program that we are using MPI by putting `MPI_Init` at the top of main and `MPI_Finalize` at the bottom.
   a. Open `hello.c` or `hello.F90` and add the lines to the code as below:

C

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

int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    printf("Hello, World!\n");

    MPI_Finalize();

    return 0;
}
```

Fortran 90

```
PROGRAM hello
IMPLICIT NONE
INCLUDE 'mpif.h'
```

```
INTEGER :: ierror

CALL MPI_INIT(ierror)

PRINT *, "Hello, World!"

CALL MPI_FINALIZE(ierror)

END PROGRAM hello
```

b. Save and quit the file.

10. Compile the code with GNU's MPI compiler:

```
$ mpicc –o hello hello.c
$
```

OR

```
$ mpif90 –o hello hello.F90
$
```

If any errors are listed, make sure there are no typos in `hello.c` or `hello.F90` (go back through step 9).

11. Edit the PBS script to use the MPI run command:
    a. Open the `hello.qsub` file in vi:
       `$ vi hello.qsub`
    b. Enter insert mode and change the last line to use `mpirun` as below:

```
#PBS –q ec
#PBS –o hello.out
#PBS –e hello.err

cd $PBS_O_WORKDIR

mpirun –np 2 ./hello
```

Here `–np 2` tells MPI to use 2 processes. Both processes will run the `hello` executable.

c. Save and quit the file.

12. Submit a job to the scheduler:

```
$ qsub hello.qsub
19099.as0.al-salam.loc
$
```

13. Monitor the job with `qstat`. Once it finishes, view the contents of standard out and standard error:

```
$ cat hello.out
Hello, World!
Hello, World!
$ cat hello.err
$
```

What do you notice about hello.out this time?

14. Let's have the processes print some useful information. We will have them print their rank, the total number of processes, and the name of the processor on which they are running.
    a. Open `hello.c` or `hello.F90` in vi:

```
$ vi hello.c
$
     OR
$ vi hello.F90
$
```

    b. Add the following lines to `hello.c` or `hello.F90`:

C

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

int main(int argc, char** argv)
{
    int rank = 0;
    int size = 0;
    int len = 0;
    char name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(name, &len);

    printf("Hello, World from rank %d of %d on %s\n",
           rank, size, name);

    MPI_Finalize();

    return 0;
```

```
}
```

Fortran 90

```
PROGRAM hello
IMPLICIT NONE
INCLUDE 'mpif.h'

INTEGER :: ierror, rank, size, len
CHARACTER*(MPI_MAX_PROCESSOR_NAME) :: name

CALL MPI_INIT(ierror)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
CALL MPI_GET_PROCESSOR_NAME(name, len, ierror)

PRINT *, "Hello, World from rank ", rank, "of ", size, &
"on ", name

CALL MPI_FINALIZE(ierror)

END PROGRAM hello
```

`rank` will be the rank of the process, `size` the total number of processes, and `name` the name of the processor on which the process is running.

c. Save and quit the file.

15. Compile the code with GNU's MPI compiler:
```
$ mpicc –o hello hello.c
$
```
OR
```
$ mpif90 –o hello hello.F90
$
```

16. We don't need to change the PBS script because we will be using the same `mpirun –np 2 ./hello` command to execute the program. We will expect to see something different in `hello.out`, however. Let's submit the job and see what we get:
```
$ qsub hello.qsub
19100.as0.al-salam.loc
$
```

17. Monitor the job with `qstat`, and once it is finished check the contents of `hello.out` and `hello.err`:

```
$ cat hello.out
Hello, World from rank 0 of 2 on as1.al-salam.loc
Hello, World from rank 1 of 2 on as1.al-salam.loc
$ cat hello.err
$
```

On which processor did Rank 0 run for you? How about Rank 1?

18. Let's try running across multiple nodes instead of just one node (`as1.al-salam.loc` in the example above). Edit the `hello.qsub` file to include the following lines:

```
#PBS -q ec
#PBS -o hello.out
#PBS -e hello.err
#PBS -l nodes=2:ppn=1

cd $PBS_O_WORKDIR

mpirun -np 2 ./hello
```

The line that we added, `#PBS -l nodes=2:ppn=1`, says to run the job on 2 nodes with 1 core per node.

19. Submit a job:

```
$ qsub hello.qsub
19104.as0.al-salam.loc
$
```

20. Monitor the job with `qstat` until it finishes, then output the contents of `hello.out` and `hello.err`:

```
$ cat hello.out
Hello, World from rank 0 of 2 on as2.al-salam.loc
Hello, World from rank 1 of 2 on as1.al-salam.loc
$ cat hello.err
$
```

Now on which processor did Rank 0 run for you? Rank 1?

Part III. Outfit the program with OpenMP

1. If we are writing in C, we first need to tell the program to include the OpenMP library. Open `hello.c` and add a line to the top:

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

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

If we are writing in Fortran 90, we need to tell the program to use the `OMP_GET_THREAD_NUM()` and `OMP_GET_NUM_THREADS()` functions. Open `hello.F90` and add a line under the `INCLUDE 'mpif.h'` line:

```
INTEGER, EXTERNAL :: OMP_GET_THREAD_NUM, &
OMP_GET_NUM_THREADS
```

2. Compile the code with OpenMP support through the GNU compiler by using the `-fopenmp` option:
   ```
   $ mpicc -fopenmp -o hello hello.c
   $
   ```
   OR
   ```
   $ mpif90 -fopenmp -o hello hello.F90
   $
   ```

3. OpenMP does not require any special run command or arguments. We may wish to tell the program how many OpenMP threads over which to parallelize, however. Open `hello.qsub` and add a line before the `mpirun` command:

```
cd $PBS_O_WORKDIR

export OMP_NUM_THREADS=2

mpirun -np 2 ./hello
```

This line tells the program to spawn 2 OpenMP threads per process when it executes an OpenMP parallel region.

4. Submit a job with `qsub`, monitor it with `qstat` until it finishes, and then view the contents of `hello.out` and `hello.err`:
   ```
   $ qsub hello.qsub
   19105.as0.al-salam.loc
   $ qstat 19105
   qstat: Unknown Job Id 19105.as0.al-salam.loc
   $ cat hello.out
   Hello, World from rank 0 of 2 on as2.al-salam.loc
   Hello, World from rank 1 of 2 on as1.al-salam.loc
   $ cat hello.err
   $
   ```

   What do you notice about the output? You might expect to see 4 "Hello, World"s because the program is supposed to spawn 2 OpenMP threads

per process. However, OpenMP will not spawn any threads unless it is explicitly told to do so by marking an OpenMP parallel region, hence we still only get 2 "`Hello, World`"s.

5. Let's mark the print statement as part of a parallel region so each thread will print the rank of the process, the total number of processes, the thread number, the total number of threads, and the processor on which it is running. Open `hello.c` or `hello.F90` in vi and make the following change to the print statement:

C

```
#pragma omp parallel
    {
        printf("Hello, World from rank %d of %d, thread %d
of %d on %s\n", rank, size, omp_get_thread_num(),
omp_get_num_threads(), name);
    }
```

Fortran 90

```
!$OMP PARALLEL
PRINT *, "Hello, World from rank ", rank, "of ", size, &
"thread ", OMP_GET_THREAD_NUM(), "of ", &
OMP_GET_NUM_THREADS(), "on ", name
!$OMP END PARALLEL
```

Note that in the C version we have now surrounded the `printf` by `#pragma omp parallel` followed by curly braces. This indicates that the `printf` is part of a parallel region that will be executed by multiple OpenMP threads. Note that in the Fortran 90 version we have now surrounded the `PRINT` by `!$OMP PARALLEL` and `!$OMP END PARALLEL`. This indicates that the `PRINT` statement is part of a parallel region that will be executed by multiple OpenMP threads.

Note also that we added the functions `omp_get_thread_num()` and `omp_get_num_threads()`. These will return the thread number of the thread and the total number of threads, respectively.

6. Compile the code and submit a job. Monitor it with `qstat` until it finishes, then view the contents of `hello.out` and `hello.err`:

```
$ mpicc -fopenmp -o hello hello.c
$ qsub hello.qsub
19106.as0.al-salam.loc
$ qstat 19106
```

```
qstat: Unknown Job Id 19106.as0.al-salam.loc
$ cat hello.out
Hello, World from rank 0 of 2, thread 1 of 2 on
as2.al-salam.loc
Hello, World from rank 0 of 2, thread 0 of 2 on
as2.al-salam.loc
Hello, World from rank 1 of 2, thread 0 of 2 on
as1.al-salam.loc
Hello, World from rank 1 of 2, thread 1 of 2 on
as1.al-salam.loc
$ cat hello.err
$
```

What do you notice about the output now?

This completes the exercise.