Supercomputing in Plain English
Exercise #6: MPI Point to Point

In this exercise, we'll use the same conventions and commands as in Exercises #1, #2, #3, #4 and #5. You should refer back to the Exercise #1, #2, #3, #4 and #5 descriptions for details on various Unix commands.

In the exercise, you'll again be parallelizing and benchmarking, but this time you'll be parallelizing with MPI instead of with OpenMP. Then you'll benchmark various numbers of MPI processes using various compilers and various levels of compiler optimization.

NOTE: This exercise is **VERY DIFFICULT!**

Here are the steps for this exercise:

1. Log in to the Linux cluster supercomputer (`sooner.oscer.ou.edu`).
2. Choose which language you want to use (C or Fortran90), and `cd` into the appropriate directory:

   ```
   % cd ~/SIPE2011_exercises/NBody/C/
   ```

   OR:

   ```
   % cd ~/SIPE2011_exercises/NBody/Fortran90/
   ```

3. Copy the `Serial` directory to a new `MPI` directory:

   ```
   % cp -r Serial/ MPI_p2p/
   ```

4. Copy the MPI Point-to-Point batch script into your `MPI_p2p` directory.

   ```
   % cp -r ~hneeman/SIPE2011_exercises/NBody/C/ nbody_mpi_p2p.bsub MPI_p2p/
   ```

5. Go into your `MPI` point to point directory:

   ```
   % cd MPI_p2p
   ```

6. Edit the `makefile` to change the compiler to `mpicc` (regardless of the compiler family).
7. Parallelize the code using MPI, specifically using point-to-point communication with `MPI_Send` and `MPI_Recv`.

   HINTS

   a. Be sure to include `mpi.h` (C) at the top of the file, or include `mpif.h` (Fortran) at the beginning of the main program.

   b. In the declaration section of the `main` function (C) or main program (Fortran), declare the appropriate variables (my rank, the number of processes and the mpi error code that's returned by each MPI routine)

   c. In the `main` function (C) or the main program (Fortran), put the startup of MPI (`MPI_Init`, `MPI_Comm_rank` and `MPI_Comm_size`) as the very first executable statements in the body of the `main` function, and put the shutdown of MPI (`MPI_Finalize`) as the very last statement in the body of the `main` function.

   d. Identify the routine where the bulk of the calculations take place, as in Exercise #5.

e. You can get the information about my rank and the number of processes into that routine in either of the following ways:

   i. you can pass, as arguments, those variables from the `main` function (C) or the main program (Fortran) to the routine that does the bulk of the calculations, OR

   ii. inside the routine that does the bulk of the calculations, you can declare the same variables, and then call `MPI_Comm_rank` and `MPI_Comm_size` inside that routine.

f. At the beginning of that routine, determine the subset of interactions that will be performed by each MPI process.

   i. A simple way to do this is to split up the iterations of the outer loop (which correspond to the list of particles) into roughly equal sized subregions, with each subregion calculated by a different MPI process.

   ii. It's a good idea to have a pair of integer arrays, each of length equal to the number of processes, with one array containing the first iteration of the outer loop to be calculated by each MPI process, and the other array containing the last iteration of the outer loop for each MPI process.

   So, be sure to declare dynamically allocatable arrays for the local first iteration and local last iteration, and at the beginning of the routine, allocate those arrays.

   In C, it would look something like this:

```
...
int* local_first_iteration = (int*)NULL;
int* local_last_iteration  = (int*)NULL;
...
local_first_iteration =
    (int*)malloc(sizeof(int) * number_of_processes);
if (local_first_iteration == (int*)NULL) {
    fprintf(stderr,
    "%d: ERROR: can't allocate local_first_iteration of length %d.\n",
        my_rank, number_of_processes);
    mpi_error_code = MPI_Abort(MPI_Comm_world, 0);
} /* if (local_first_iteration == (int*)NULL) */
local_last_iteration  =
    (int*)malloc(sizeof(int) * number_of_processes);
if (local_last_iteration == (int*)NULL) {
    fprintf(stderr,
    "%d: ERROR: can't allocate local_last_iteration of length %d.\n",
        my_rank, number_of_processes);
    mpi_error_code = MPI_Abort(MPI_Comm_world, 0);
} /* if (local_last_iteration == (int*)NULL) */
...
```

And in Fortran90, it'd look like this:

```
...
INTEGER,PARAMETER :: memory_success = 0
...
INTEGER,DIMENSION(:),ALLOCATABLE :: local_first_iteration
INTEGER,DIMENSION(:),ALLOCATABLE :: local_first_iteration
...
INTEGER :: memory_status
...
ALLOCATE(local_first_iteration(number_of_processes),STAT=memory_status)
IF (memory_status /= memory_success) THEN
    WRITE(0,*) my_rank,  &
&       "ERROR: can't allocate local_first_iteration of length",  &
&       number_of_processes
    CALL MPI_Abort(MPI_Comm_world, mpi_error_code, mpi_error_code)
END IF !! (memory_status /= memory_success)
ALLOCATE(local_last_iteration(number_of_processes),STAT=memory_status)
IF (memory_status /= memory_success) THEN
    WRITE(0,*) my_rank,  &
&       "ERROR: can't allocate local_last_iteration of length",  &
&       number_of_processes
    CALL MPI_Abort(MPI_Comm_world, mpi_error_code, mpi_error_code)
END IF !! (memory_status /= memory_success)
...
```

iii. For the first MPI process, the first iteration to be calculated is the first iteration of the outer loop (i.e., the first particle).

iv. For each subsequent MPI process, the number of iterations is either (a) the total number of iterations divided by the number of MPI processes, or (b) one more than that (in case there is a remainder when dividing the number of iterations by the number of MPI processes). A good rule of thumb is to put at most one leftover (remainder) iteration on each of the MPI processes. How can you determine which MPI processes should get one of the leftover iterations?

v. For each MPI process, the last iteration to be calculated is the first iteration plus the number of iterations, minus one.

vi. Just in case, check to be sure that the last iteration of the last MPI process is the last iteration overall.

g. Dynamically allocate two "buffer" arrays of floating point (real) elements, both having length the number of spatial dimensions (in this case, 3 for X, Y and Z) times the total number of particles. These are the send buffer and the receive buffer.

h. Fill both buffer arrays with zeros.

i. In the outer loop, instead of looping over all of the particles, loop from the first particle for this MPI process to the last particle for this MPI process. That way, this MPI process will only calculate a subset of the forces.

j. After the outer loop, you'll have an array of total aggregate forces on the subset of particles corresponding to the subset of outer loop iterations calculated by this MPI process.

k. Copy those values into the appropriate portion of the send buffer.

l. Using `MPI_Send` and `MPI_Recv`, send this MPI process's subset of total aggregate forces to all other MPI processes. You can use the same basic structure as in the `greetings.c` or `greetings.f` program, except after the master process receives all of the subsets of the forces sent from each of the other processes (as the master in `greetings` receives the messages sent from the other processes), the master then sends the full force array to each of the other processes (and they each receive the full force array).

m. Copy the array of received values into the original data structure.

n. At the end of that routine, deallocate the dynamically allocated arrays (**VERY IMPORTANT!**).

In C, a deallocation would look something like this:

```
...
free(local_first_iteration);
...
```

And in Fortran90, a deallocation would look like this:

```
...
DEALLOCATE(local_first_iteration,STAT=memory_status)
IF (memory_status /= memory_success) THEN
    WRITE(0,*) my_rank,  &
&       "ERROR: can't deallocate local_first_iteration of length",  &
&       number_of_processes
    CALL MPI_Abort(MPI_Comm_world, mpi_error_code, mpi_error_code)
END IF !! (memory_status /= memory_success)
...
```

o. For outputting, only one of the MPI processes should do the output (which one?).

13. Copy the MPI point-to-point template batch script:

```
% cp  ~hneeman/SIPE2011_exercises/NBody/C/nbody_mpi_p2p.bsub  MPI_p2p/
% cp  ~hneeman/SIPE2011_exercises/NBody/Fortran90/nbody_mpi_p2p.bsub  MPI_p2p/
```

14. Edit your new batch script, following the instructions in the template batch script to make it use the appropriate files in the appropriate directories, as well as your e-mail address and so on.

15. Set the `MPI_COMPILER` and `MPI_INTERCONNECT` environment variables. Typically, you'll need to do this once per login.

    a. If your Unix shell is `tcsh`, then you would do this:

    ```
    % setenv MPI_COMPILER gnu
    % setenv MPI_INTERCONNECT ib
    ```

    b. If your Unix shell is `bash`, then you would do this:

    ```
    % export MPI_COMPILER=gnu
    % export MPI_INTERCONNECT=ib
    ```

16. Compile using `make`. You may need to do this multiple times, debugging as you go.

17. Submit the batch job and let it run to completion. If it seems to take a very long time, probably you have a bug.

18. For each run, once the batch job completes:

    a. Examine the various output files to see the timings for your runs with executables created by the various compilers under the various levels of optimization.

    b. Profile, as described above.

19. How can you debug an MPI program? The most straightforward way is to add outputs to standard error (stderr) before and after each MPI call. For example, in C:

```
fprintf(stderr, "%d: about to call MPI_Send(force_array, %d, ...)\n",
    my_rank, last_iteration[my_rank] - first_iteration[my_rank] + 1);
mpi_error_code = MPI_Send(...);
fprintf(stderr, "%d: done calling  MPI_Send(force_array, %d, ...)\n",
    my_rank, last_iteration[my_rank] - first_iteration[my_rank] + 1);
fprintf(stderr, "%d: mpi_error_code=%d\n", mpi_error_code);
```

    Or, in Fortran90:

```
WRITE (0,*) my_rank, ": about to call MPI_Send(force_array, ",  &
  & last_iteration(my_rank) - first_iteration(my_rank) + 1), "...)"
CALL MPI_Send(..., mpi_error_code)
WRITE (0,*) my_rank, ": done calling  MPI_Send(force_array, ",  &
  & last_iteration(my_rank) - first_iteration(my_rank) + 1), "...)"
WRITE (0,*) my_rank, ": mpi_error_code=", mpi_error_code
```

20. Continue to debug and run until you've got a working version of the code.

21. Use your favorite graphing program (for example, Microsoft Excel) to create graphs of your various runs, so that you can compare the various methods visually.

22. You should also run different problem sizes, to see how problem size affects relative performance.