1. Hello World in Parallel

   Write a parallel version of the "Hello World" program using MPI and run it to observe the output. The basic elements of this simple program are: initialize MPI, determine the number of each processor and total number of processors, have each process write out the message "Hello World. I am processor 3 of 4 processors" to the screen (standard output), and finalize MPI. Run the program interactively using 16 processors to observe the output.

   (a) In what order do the messages appear on the screen?

   (b) Modify the program using the appropriate logic and MPI calls to ensure that the messages on the screen appear in order.

2. Monte Carlo Determination of the Value of π

   Using the Monte Carlo method of numerical integration, determine the value of π. This is accomplished by choosing N random points within a box $-1 \le x \le 1$ and $-1 \le y \le 1$, and calculating the integral using

   $$\pi = 4*n/N$$

   where n is the number of the random points that fall within a circle of radius r = 1.

   (a) First, write a serial code that calculates this value of π.

   (b) Next, use MPI to make the code parallel and verify that it gives you the same results. (Keep the working serial version intact for the next step).

   (c) Calculate the time it takes to perform this calculation with $N = 10^8$ points using 1, 2, 4, 8, and 16 processors. I suggest using the `MPI_WTime` call to calculate the CPU time on each processor (see `HYDRO` for an example of how to use this call), but you are free to use other C or Fortran calls to calculate the CPU time, or use the unix command `time` before the run, for example,

   Note that not all of the methods will work on every architecture (although `MPI_WTime` should), so it is worth-while testing the timing calls to be sure they give an acceptable output before committing to one approach. NOTE: Consider whether or not you want to include the initialization time in these performance tests.

3. Write a code that sets a real variable on each of N processors equal to the MPI rank (Task ID) of the task. Then write your own routine to perform a reduction operation over all processors to sum the values using only `MPI_Send` and `MPI_Recv` calls. Do this global reduction operation using the following communication algorithms:

   (a) Communications in a ring.

   (b) Hypercube communications.

   Put in timing calls using `MPI_WTime` (see `HYDRO` for an example of how to use this call) to test the timing of your routines compared to using the MPI routine `MPI_Allreduce`.to do the same computation.

4. Array Processing Using a Master-Slave Strategy:

   Consider a problem where you have a very large N × N array of data, where N = 65536, and you want to perform the same computation on each element of the array. Take the initial conditions of the array such that A_{ij} = `real(i)*real(j)`. The operation to perform on each element is the function $f(x) = \sqrt{x}$. After the operation has been performed on all array elements, we want to calculate the sum of all of the array elements.

   (a) First, write a serial code that calculates one element at a time in sequential order. Try looping through the two dimensions in different orders (row major vs. column major) to see if it changes the computation time. Why might different orders take different wallclock times?

   (b) Next, write a parallel code using a Master-Slave scheme where the Master initializes the array, distributes chunks of the array to the slave workers to do the computation, and then gathers the computed data to ultimately produce the sum. Consider the following issues when writing this parallel code:

      i. Should the Master keep part of the array to do some of the computation itself?

      ii. How should you decompose the array to send it to different processors? Consider the answer to the question in the serial part of this problem.

      iii. How will you achieve a load balance between processors?