Introduction

The purpose of this assignment is to become familiar with MPI by parallelizing the quicksort algorithm.

You need to write a program, qs_mpi that take as input an integer **N**, generates **N** *32-bit non-negative integers* randomly and saves to a file the generated numbers in non-decreasing order (ascending order).

Parallelization strategy

Design your program so that it follows the following steps:

1. If there are **P** workers, each worker generates **N/P** *32 bit non-negative integers* randomly. Each worker should seed its random number generator with its rank. You can assume that **N** is divisible by **P**.
2. All the workers are working together to do the sorting (this means no serial operations of size O(N)). At the end, each worker should have **N/P** *integers*. You need to follow a hybrid of data and recursive decomposition. The data decomposition will be used to perform the partition step. Once partitioned, the less and greater than the pivot subarrays will be divided among the available processors working on that partition step, the processors will be split into two groups, and each group will be responsible for performing the divide step (one for the less than the pivot and one for the greater than the pivot subarrays). For pivot selection, do the following: Each processor randomly selects one of its elements, and the median of these selected elements will become the overall pivot of the divide step. For pivot selection, another option is to select the median at each worker and the median of these medians will become the overall pivot.
3. One worker is responsible for doing the output.

You only need to time step 2. Step 2 must be parallel.

Command line arguments and output format

Your programs should take arguments as follows:

Two workers to sort 5 integers, the output file is *output.txt*:

```
mpirun -np 2 ./qs_mpi 5 output.txt
```

The first line in the output file should be the total number of elements (**N**). Next **N** lines should contain **N** integers in non-decreasing order, one integer per line. For example, if the numbers in sorted order are 1,2,4,6 and 6; *output.txt* should look as below:

```
5
1
2
4
6
6
```

You should only run your code on one of the plate machines. You will use the last digit of your student ID to select the machine. ID's ending in {0, 4, 8} should use plate01, ID's ending in {1, 5, 9} should use plate02, ID's ending in {2, 6} should use plate03, and ID's ending in {3, 7} should use plate04.

Here are some IO routines you should use. Additionally, you should use the timing routines from the previous assignment.

```
/**
* @brief Write an array of integers to a file.
*
* @param filename The name of the file to write to.
* @param numbers The array of numbers.
* @param nnumbers How many numbers to write.
*/
static void print_numbers(
    char const * const filename,
    uint32_t const * const numbers,
    uint32_t const nnumbers)
{
  FILE * fout;

  /* open file */
  if((fout = fopen(filename, "w")) == NULL) {
    fprintf(stderr, "error opening '%s'\n", filename);
    abort();
  }

  /* write the header */
  fprintf(fout, "%d\n", nnumbers);

  /* write numbers to fout */
  for(uint32_t i = 0; i < nnumbers; ++i) {
    fprintf(fout, "%d\n", numbers[i]);
  }

  fclose(fout);
}
```

```
/**
* @brief Output the seconds elapsed while sorting. This excludes input and
*        output time. This should be wallclock time, not CPU time.
*
* @param seconds Seconds spent sorting.
*/
static void print_time(
    double const seconds)
{
  printf("Sort Time: %0.04fs\n", seconds);
}
```

What you need to turn in

1. The source code of your program.
2. A short write-up describing how you went about parallelizing the quicksort algorithm. Make sure to describe how you ensured that at the end of the operation each worker had **N/P** elements. The write-up must be a .pdf file named report.pdf and should include the timing results (#3).
3. Timing results on 1, 2, 4, 8, and 16 processors for 3 different values of **N:** 1,000,000; 10,000,000 and 100,000,000 (1M, 10M and 100M). Include these in the PDF. **When timing, ensure all ranks are executed on the same host machine.** You can use a hostfile to achieve this and the function *MPI_Get_processor_name* can be used ensure you are running on a single host.
4. The source file must be named qs_mpi.c.
5. A makefile must be provided to compile and generate the executable. The executable should be named qs_mpi.
6. Do not print any debug statements to stdout (as this is what will be re-directed to a file for correctness testing).
7. All files (Code + Report) must be in a single directory and the directory's name should be your University ID (your email is <ID>@umn.edu).
8. Submission must be gzip compressed tar balls (ending in .tar.gz). To create this, run 'tar -czvf <id>.tar.gz <id>/'. Your directory structure should look like:

```
<studentid>/Makefile
<studentid>/qs_mpi.c
<studentid>/report.pdf
```

Evaluation criteria

Evaluation will be based on:

1. solving the problem correctly,
2. do so in parallel,
3. achieved speedup,
4. following directions (correctly packaging your submission, using our output functions, etc.),
5. using reasonable programming practices (do not allocate 100MB of space regardless of input size, etc.).

The speedups obtained will depend on the size of the input. I do not expect you to get good speedups for small input, but you should be able to get speedups for large input. **Programs that fail to do all five of these will lose significant points.**

It does not matter how fast a program is if it does not produce 100% correct output. This is a parallel programming class. Submissions which do not solve the problem in parallel will not receive credit. Submissions which implement a sorting algorithm other than quick sort will also not receive credit.

Finally, late submissions will NOT be graded.