CS211 HW3

Due: 11/20/2023 23:59

Guideline:

1. The template code is on github: https://classroom.github.com/a/_MLpT1RT.
   You need to complete **sieve1**, **sieve2** and **sieve3** and write a report. Upload the report and your repository to github.
   On eLearn, you need to submit a link to your repository.
2. You can use **starter.py** to compile your code, generate sh files for required tests, and run the sh files using **sbatch**. The sh files will run your C code using **mpirun**. Using **python3 starter.py** is a suggested way to test all conditions and get the result. You can also modify the **N_list**, **ver_list** and **n** to change which tests to execute.
   N_list contains the number of nodes to use. Each node has 32 computing cores. **N_list** should contain integers between 1 and 5.
   ver_list contains the version of sieve functions to test, and should contain integers between 0 and 3.
   n indicates the largest number we want to check whether it is a prime. During actual testing, it should be 10^{10}.
3. You can also run each sh file individually using **sbatch {path_to_sh_file}**. Each executed sh file will generate an output file (end with .o) for storing the normal output of the code including the running time and total number of primes, and an error file (end with .err) for storing the abnormal output. These two files will be generated at the place you invoke the sh file. When invoking sh files with **starter.py**, output and error files will be generated in the **sh** directory in your repository.
4. You can also straightly run the compiled C code on the header node. However there are only 32 cores on the header node, so don't anticipate that the running time on the header node can be accurate. First you need to use **module load mpich-3.2.1** to activate the MPI. This command has to be executed every time you run C code straightly on a new terminal. However if you use **starter.py** or the sh files, you don't need this command. And then you compile the code using **make** and run the code using **mpirun -np {cores} ./main {function} {n}**. The result will be shown in the terminal.
5. We use **starter.py** to measure the correctness, running time and performance.
6. We use **hpc-001** to test your performance when grading.

Hint: debugging the whole MPI code can be hard. However if you can know which number is misclassified, you can also easily find the bug. If your code seems fine but there is just an expression giving an unexpected result, you can try to check the data type and the sequence of calculation. For the data type, if two integers occupying 32 bits are multiplied, the result can be an integer occupying 64 bits.

Problem 1 (10 points):

Run the parallel Sieve of Eratosthenes program given in the slides (which is also **sieve0**).

Problem 2 (30 points):
Modify the parallel Sieve of Eratosthenes program in class (**sieve0**) so that the program does NOT set aside memory for even integers. Put your code in **sieve1**.

Problem 3 (30 points):
Modify **sieve1** so that each process of the program finds its own sieving primes via local computations instead of broadcasts. Put your code in **sieve2**.

Problem 4 (30 points):
Modify **sieve2** so that the program can have effective uses of caches. Put your code in **sieve3**.

For each version of sieve, record the time and number of primes when $n = 10^{10}$ and number of cores = $32, 64, 96, 128, 160$ (which means number of nodes = $1, 2, 3, 4, 5$). Check whether your execution time is inversely proportional to the number of cores. Compare the execution time of each version of your program to see how different designs affect the execution time.