Homework 3: Questions

(20 points) We have discussed how to convert the sequential program into parallel program in our class. We also identified a few ways to improve the algorithm. In this exercise, three versions of sieve program are given, i.e., sieve1.c, sieve2.c, sieve3.c. Go through the code and answer the following questions:

a. (5 points) Read the code and briefly explain how each version works.

Assume all steps are executed by each process, unless otherwise noted

i. sieve1.c

- Pre-Algorithm
  1. Call parallelization initiation
     1. Initialize MPI execution
     2. Blocks caller until all processes have called
     3. Determine process rank (PID) and group size
  2. Start Timer
  3. Parse and verify user input for n (sieve's upper limit)
     1. Convert string to integer
  4. Determine contiguous block values and size
  5. Verify acceptable block size
     1. Calculate size of process 0
     2. If $\text{process } 0\text{'s largest value}^{\,2}$ is less than n
        2. Process 0 prints error message (as the algorithm will not work due to too many processes)
        3. Terminate MPI execution
        4. Exit, returning 1 – Execution Failure
        - Note: the actual function calculates if $(proc0size + 2) < \sqrt{n}$
  1. Allocate memory for Process's share of array
     1. Allocates $\text{blockSize} \times 4$ (size of int)
        - Note: Book/slides say it should be set to $\text{blockSize} \times 1$ (size of char), but the code uses 4 (size of int)
     2. Error Check: valid malloc return value
        5. If malloc returns NULL
           a. Process 0 prints error message
           b. Terminate MPI execution
           c. Exit, returning 1 – Execution Failure
- Algorithm Step 1: Create list of unmarked natural numbers 2, 3, …, n
  1. For all values 0 to block size
     1. create array of 'unmarked' values
- Algorithm Step 2: Set 'Prime'
  1. Set array index to 0 for process 0 only

2. Set current prime being sieved to 2

- Algorithm Step 3: Repeat until $prime^2 > n$:
  A. Mark all multiples of prime between $prime^2$ and n
     1. Set index to first integer needing marked
        1. if `prime` ^{2} is > lowest value in array
           - `first = prime` ^{2} `- lowestValue`
        2. if `prime` ^{2} is ≤ lowest value in array
           a. if $lowestValue$ is multiple of $prime$
              - `first = 0`
           b. if $lowestValue$ is NOT multiple of $prime$
              - `first = prime - lowestValue % prime`
     2. Sieve Data: Mark multiples of `prime` from `first` to n
  B. Locate next unmarked location
     3. Process 0 increments index by 1 until it finds `marked[index] = 0`
     4. Process 0 sets `prime = index + 2`
  C. Process 0 broadcasts new `prime` to rest of processes
  - *Repeat steps A, B, C until* `prime` $^2 \leq n$
- Algorithm Step 4: Return all unmarked numbers as primes
  1. Initialize a counter variable
  2. Increment index to parse each value in array
     1. Increment counter if value is unmarked
        - marked[index] = 0
  3. Reduce and sum local count value on all processes to global count
- Post Algorithm
  1. Calculate elapsed time
  2. Process 0 prints results
  3. Terminates MPI execution environment
  4. Returns 0 (execution success)

ii. sieve2.c

- Pre-Algorithm *(identical to sieve1.c, therefore condensed below)*
  1. Call parallelization initiation
  2. Start Timer
  3. Parse, verify, and convert user input for n (sieve's upper limit)
  4. Determine contiguous block values and size
  5. Verify acceptable block size

1. Checks that the square of the largest value in process 0 is greater than the upper limit of the sieve
    - Note: the actual function calculates if $(proc0size + 2) < \sqrt{n}$
2. If the verification fails, the program terminates MPI execution and exits, returning 1 (Execution Failure)

6. Allocate memory for Process's share of array
    1. If the allocation fails, the program terminates MPI execution and exits, returning 1 (Execution Failure)

- Algorithm Step 1: Create list of unmarked natural numbers 2, 3, …, n
    1. Create List and 'mark' all even numbers
        1. For all values 0 to block size
            1. If lowest value is even, then `marked[even] = even`
            2. If lowest value is odd, then `marked[odd] = even`
    - *Note: Essentially, the function determines if starting value is even or odd, then uses that to mark all even numbers*
- Algorithm Step 2: Set 'Prime'
    1. If Process 0
        1. Set array index to 1
        2. Unmark first value in array
    2. Set current prime being sieved to 3
        - Since the prime 2 was accounted for above, the sieving can start at prime 3
- Algorithm Step 3: Repeat until $prime^2 > n$:
    A. Mark all multiples of prime between $prime^2$ and n *(identical to sieve1.c)*
        1. Set index to first integer needing marked
            3. if `prime` ^{2} is $>$ lowest value in array
                - `first = prime` ^{2} `- lowestValue`
            4. if `prime` ^{2} is $\leq$ lowest value in array
                a. if $lowestValue$ is multiple of $prime$
                    - `first = 0`
                b. if $lowestValue$ is NOT multiple of $prime$
                    - `first = prime - lowestValue % prime`
        2. Sieve Data: Mark multiples of `prime` from `first` to n
    B. Locate next unmarked location

1. Process 0
    1. increments index by 2 until it finds `marked[index] = 0`
    2. sets `prime = index + 2`
    3. increments `index` by 2 to match new prime

C. Process 0 broadcasts new `prime` to rest of processes

- *Repeat steps A, B, C until* $prime^2 \leq n$

- Algorithm Step 4: Return all unmarked numbers as primes *(identical to sieve1.c)*
    1. Initialize a counter variable
    2. Parse each value in array and count all unmarked numbers
    3. Reduce and sum local count value on all processes to global count
- Post-Algorithm *(identical to sieve1.c)*
    1. Calculate elapsed time
    2. Process 0 prints results
    3. Terminates MPI execution environment
    4. Returns 0 (execution success)

ii. sieve3.c

- Pre-Algorithm
    1. Call parallelization initiation
    2. Start Timer
    3. Parse, verify, and convert user input for n (sieve's upper limit)
    4. Determine contiguous block values and size
    5. Verify acceptable block size
        1. Checks that the square of the largest value in process 0 is greater than the upper limit of the sieve
            - Note: the actual function calculates if $(proc0size + 2) < \sqrt{n}$
        2. If the verification fails, the program terminates MPI execution and exits, returning 1 (Execution Failure)
        - *Note: while the process for this verification step is the same as sieve1.c and sieve2.c, the code is slightly different as sieve3.c calculates and saves* $\sqrt{n}$ *prior to computation*
    6. Allocate memory for Process's share of array
        - `marked = block_size × int (4)`
        - `sieve =` $\sqrt{n}$ `× int (4)`
        - If either allocation fails, the program terminates MPI execution and exits, returning 1 (Execution Failure)
- Algorithm Step 1: Create list of unmarked natural numbers 2, 3, …, n
    1. Create Marked List and 'mark' all even numbers *(identical to sieve2.c)*

        - `marked[i]`
        - *determines if starting value is even or odd, then uses that to mark all even numbers*
    2. Create Sieves List and Set all Even Indexes
        - `sieves[even]=1`
- Algorithm Step 2: Set 'Prime'
    1. If Process 0
        1. Unmark first value in `marked` array
    2. Set current prime being sieved to 3
        - *Since the prime 2 was accounted for above, the sieving can start at prime 3*
- Algorithm Step 3-1: Repeat until $prime^2 > \sqrt{n}$:
    A. Create list of sieving primes from 2 to $\sqrt{n}$
        1. Unset first value in `sieves` array
        2. Set array index to 1
        3. Sieve Data: Set value for non-primes from 4 to $\sqrt{n}$
    B. Locate next Unset Sieve Value
        4. loop until `sieves[index]=0`
        5. increment `prime` and `index`
    ~~C. Process 0 broadcasts new `prime` to rest of processes~~
        - *This step is skipped in this instance, as each process is able to calculate primes using `sieves[]`*
    - *Repeat steps A, B, C until* `prime` $^2 \leq \sqrt{n}$
- Algorithm Step 3-2: Repeat until $prime^2 > n$:
    A. Mark all multiples of prime between $prime^2$ and n *(identical to sieve1.c, sieve2.c)*
        1. Set index to first integer needing marked
            4. if `prime` ^{2} is $>$ lowest value in array
                - `first = prime` ^{2} `- lowestValue`
            5. if `prime` ^{2} is $\leq$ lowest value in array
                a. if $lowestValue$ is multiple of $prime$
                    - `first = 0`
                b. if $lowestValue$ is NOT multiple of $prime$
                    - `first = prime - lowestValue % prime`
        2. Sieve Data: Mark multiples of `prime` from `first` to n
    B. Locate next Unmarked Sieves Location to determine new Prime
        3. loop until `sieves[index]=0`
        4. increment loop by 2
        5. increment prime and index
    ~~C. Process 0 broadcasts new `prime` to rest of processes~~

        - *This step is skipped in this instance, as each process is able to calculate primes using* `sieves[]`
      - *Repeat steps A, B, C until* `prime` $^2 \leq n$
    - Algorithm Step 4: Return all unmarked numbers as primes *(identical to sieve1.c, sieve2.c)*
      1. Initialize a counter variable
      2. Parse each value in array and count all unmarked numbers
      3. Reduce and sum local count value on all processes to global count
    - Post-Algorithm *(identical to sieve1.c, sieve2.c)*
      1. Calculate elapsed time
      2. Process 0 prints results
      3. Terminates MPI execution environment
      4. Returns 0 (execution success)

b. (5 points) What are the differences between sieve1.c and seive2.c? How does sieve2.c improve on sieve1.c?
   i. The first difference between sieve1.c and sieve2.c appears during step 1, while creating a list of unmarked natural numbers. Sieve1.c simply creates an array of unmarked values, whereas sieve2.c creates the array while simultaneously marking any even values. Since even numbers (other than 2) cannot be prime, due to divisibility by 2, the program will be less computationally expensive by automatically eliminating half of the values.
   ii. Since sieve2.c marks all even numbers, despite 2 being prime, the programs also differ on step 2. Since the value 2 will only appear at index 1 in process 0, sieve2.c creates a conditional statement for process 0 which unmarks `marked[0]`. Another difference in step 2 is the initial value prime is set to. In sieve1.c, prime is set to 2; however, this is unnecessary in sieve2.c as prime value 2 was already accounted for. As such, the sieving can start at `prime = 3`.
   iii. The last difference between the programs is in step 3. Since all even values were already located and marked, there is no reason to increment by 1 when locating the next prime. Since every other index is marked, sieve2.c can increment by 2.
   iv. The differences in sieve2.c should theoretically decrease computation time as half of the data (even values) is eliminated early in the program.

c. (5 points) What are the differences between sieve1.c and seive3.c? How does sieve3.c improve on sieve1.c?
   i. Other than some basic initialization, the first difference between sieve1.c and sieve2.c appears in step 1, while creating a list of unmarked natural numbers. Sieve3.c begins by following the same technique as sieve2.c and creates an array while simultaneously marking any even values. This has the added benefit of decreasing computational costs when parsing the data array.
   ii. Also during step 1, sieve3.c creates a sieve list using the same technique as above – creating the array while marking unnecessary even values. The benefit of this array is mentioned shortly.
   iii. Similar again to sieve2.c, since sieve3.c marks all even numbers, it creates a conditional statement for process 0 which unmarks the value 2 (`marked[0]=0`).

As mentioned, prime is set to 2 in sieve1.c, which is unnecessary in sieve3.c as prime value 2 was already accounted for. As such, the sieving can start at `prime = 3`.

iv. The major difference between the programs is during step 3. In sieve1.c, a global 'prime' value is calculated by process 0 and broadcasted to the remaining processes. This prime value is then used by each process to mark non-prime values within their block. However, sieve3.c does not use this broadcast method, instead using `sieve` array mentioned previously. By creating a list of sieving primes from 2 to $\sqrt{n}$, and looping through until all non-primes are marked, each process is able to calculate the current prime necessary. Once each process has calculated `prime`, it executes the same function as sieve1.c to mark all non-prime values within their block. Only, instead of relying on a broadcast from process 0, it uses `sieve` to Locate the next Unmarked Sieves Location and determine `prime`. This is a major benefit as each process is able to compute primes completely independently, without relying on the root process to update a global variable. I would expect this to reduce computational costs considerably.

d. (5 points) Benchmark the performance of the three versions on the Rushmore cluster using different number of processors and fill the table below. Does the Benchmark results meet your expectation? Why?

Program	***Np=1***	***Np=2***	***Np=3***	***Np=4***
sieve1.c	0.042112	0.013157	0.006709	0.005614
sieve2.c	0.016337	0.000543	0.001914	0.002434
sieve3.c	0.000074	0.000697	0.000707	0.000877

- *Note, for all benchmarks I used* $n = 1000$

i. Yes, overall the benchmark results meet my expectations. Since sieve2.c and sieve3.c created their `marked[]` arrays while simultaneously marking any even values, thus eliminating the need to parse half of the data, it makes sense that their times were faster than sieve1.c. Furthermore, the elimination of the `broadcast` function saved additional computation time for sieve3.c. As mentioned earlier, this is beneficial as each process is able to compute primes completely independently, without relying on the root process to update a global variable.

2. (10 points) Compare the differences of MPI and openMP programming. Which one do you like better and why?

- MPI is message based parallelism, where openMP is directive based. In other words, in MPI, the program is split into multiple processes which run independently and concurrently; the only communication between them is passed by sending asynchronous messages/ API calls. On the other hand, openMP uses directives (`#pragma`) to instruct compiler behavior and does not require message passing as the threads access a shared memory. It is important to note that the use of shared memory in openMP does require critical sections; however, this is easily implemented using the directed previously mentioned. Furthermore, instead of running parallelization across a network (as we do in our Rushmore cluster with MPI), openMP enables the running threads across multiple

processors. Therefore, MPI requires a network while openMP can be run in isolation. In terms of programming, it can be very time consuming to transfer a sequential program into a parallel program with MPI because the programmer must ensure that the program is split correctly, each process has access to the necessary information, and that the final data is correct resolved to the root process. On the other hand, openMP uses compiler directives which automatically allocates data and generates code that forks/joins threads. Also, since openMP incorporates incremental parallelization, the program introduces parallelization "as a sequence of incremental changes, parallelizing one loop at a time." As a result, openMP can require significantly less effort to implement.

- While I think both MPI and openMP have their benefits. I enjoy the logic and control over MPI better. I feel it is easier to debug since I can see each step that is happening, with openMP, I struggle to know what the compiler handles versus what I need to handle. This could stem from the fact that I currently have had more practice with MPI at this point. However, once I practice with openMP more, I could see my preference leaning more towards it as a result of the simplicity.

3. (30 points) gprof is a type of tool called a profiler. Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing. This information can show you which pieces of your program are slower than you expected and might be candidates for rewriting to make your program execute faster. gprof can be made available in the VMs in the Rushmore virtual cluster.

    - (15 points) For linpack_bench.cpp, read the profile tool gprof user manual (see the html guide), run the profiling tool and report the percentage of running times for each function.

% time	*calls*	*name*
80.80	501499	`daxpy(int, double, double*, int, double*, int)`
13.17	2000000	`r8_random(int*)`
2.63	1	`dgefa(double*, int, int, int*)`
1.76		`main`
0.88	1003000	`r8_max(double, double)`
0.88	2	`r8_matgen(int, int)`
0.00	509110	`r8_abs(double)`
0.00	999	`dscal(int, double, double*, int)`
0.00	999	`idamax(int, double*, int)`
0.00	11	`std::setw(int)`
0.00	4	`cpu_time()`
0.00	2	`timestamp()`
0.00	1	`_GLOBAL__sub_I_main`
0.00	1	`r8_epsilon()`
0.00	1	`__static_initialization_and_destruction_0(int, int)`
0.00	1	`dgesl(double*, int, int, int*, double*, int)`

- (15 points) For linepack_bench.cpp, based on the percentage of running times from gprof, briefly describe your strategy to modify the program if openmp is chosen to optimize the program.
    - If I were to use openMP to modify and optimize the program, I would focus my attention on parallelizing `daxpy()` as it consumes 80.80% of the entire execution time. With 501,499 calls, splitting the workload between multiple threads would greatly minimize the computation time. Also, according to the call graph, `daxpy()` has no children which further confirms it is responsible for a majority of the runtime.

4. (40 points) Programming Assignment 4: For the 4 sequential c programs, p1.c, p2.c, p3.c, p4.c, using openmp to parallelize them as much as possible and do the following profiling.
    - Add timestamp functions to the code to count the running time of the sequential programs and your openmp programs and show the result in the following table.
    - Copy your openmp program running outputs to a report file running.txt. You need to make sure openmp programs should generate the same results as the sequential codes.

	P1	***P2***	***P3***	***P4***
Sequential code running time	0.0122	0.0040	1.6557	0.1002
openMP running time	0.0010	0.0164	1.2115	0.1368
Speed up `(omp÷seq)`	0.0820	4.1000	0.7317	1.3653
No. of threads	5	5	5	5