

[Sample Solution]

Lab 03
Git URL

Contribution {To be completed by Students}:

Student ID	Student Name	Contribution

1. POSIX Threads

- a) Write a serial C program to search for prime numbers which are less than an integer, n , which is provided by the user. The expected program output is a list of all prime numbers found, which is written into a single text file (e.g., primes.txt).

For instance, if the user inputs n as 10 on the terminal, the prime numbers written to the text file are: 2, 3, 5, 7.

Hint: It is known that for a given number, prime numbers would only exist for values less than or equal to the square root of the given number. As such, you can make use of the `sqrt()` of `math.h` to optimize your code.

To answer this question, a program namely the `q1.c` is developed to search for all prime numbers within the range from 0 until $n - 1$ in a serial manner. In particular, the process of the `q1.c` are as follows:

- The program takes user input and store it into the integer variable `pUpperLimit`
- After that, a for loop is employed to loop the integer value i from 2 until `pUpperLimit - 1` to perform prime number searching. In each iteration of loop, the prime number searching mechanism first passes the i into a user-defined function `isPrime(...)` where the `isPrime(...)` returns the Boolean value false when i is a non-prime number. In the case that true is returned from the `isPrime(...)`, the i is added into the dynamic array `*primeNumbers`. To be noted, an integer variable `primeCounter` is used to track the total number of prime numbers filled into the array.
- Lastly, iterate through the `*primeNumbers` and write the value of `primeNumbers[i]` into the "primes.txt" where $i = 0, 1, \dots, \text{primeCounter} - 1$.

```

/* Function Definition */
bool isPrime(int n){
    // Exclude the case when n <= 1
    if(n<=1){
        return false;
    }

    // Exclude the case for n is even number and larger than 2
    if(n!=2 && n%2==0){
        return false;
    }

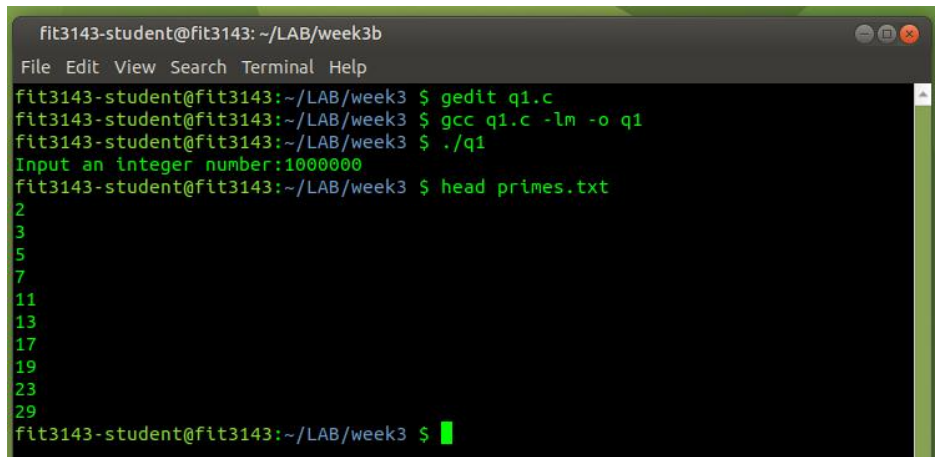
    // If a factor can be found between 3 - sqrt(n), the given number n is a non-prime number
    for(int i=3; i<=sqrt(n); i++){
        if(n%i==0){
            return false;
        }
    }

    return true;
}

```

Fig. 1. isPrime(...) function

Figure above shows the function definition of the isPrime(...) that is used by the prime number searching mechanism to check whether the given number n is a prime or non-prime number. Particularly, isPrime(...) is a Boolean function that takes n as input and return the value of true (when n is prime number) or false (when n is non-prime number). The function starts by excluding the case where n is smaller equal than 1 or n is an even number that is larger than 2. After that, the function use a for loop to find any factor for n within the range of 3 until square root of the n (i.e., \sqrt{n} in the figure). If there is a factor, the n is identified as a non-prime number and the function returns false.



```

fit3143-student@fit3143: ~/LAB/week3b
File Edit View Search Terminal Help
fit3143-student@fit3143:~/LAB/week3 $ gedit q1.c
fit3143-student@fit3143:~/LAB/week3 $ gcc q1.c -lm -o q1
fit3143-student@fit3143:~/LAB/week3 $ ./q1
Input an integer number:1000000
fit3143-student@fit3143:~/LAB/week3 $ head primes.txt
2
3
5
7
11
13
17
19
23
29
fit3143-student@fit3143:~/LAB/week3 $

```

Fig. 2. Screenshot of executed program and output text file

Fig. 2 shows the procedure to compile and execute the program, and to display the content of the primes.txt.

- **Compilation of program:** gcc q1.c -lm -o q1
- **Execution of program:** ./q1
- **Display result (first few lines):** head primes.txt

- b) Measure the time required to search for prime numbers less than an integer, n when $n = 1,000,000$ (i.e., t_s). Calculate the theoretical speed up using Amdahl's law when $p = 4$, with p = number of processes (or threads).

Amdahl's law, $s(p)$:

$$s_{theory}(p) = \frac{1}{r_s + \frac{r_p}{p}}$$

where r_p is parallel ratio (parallelizable portion) of the algorithm, r_s is the serial ratio (non-parallelizable portion) of the algorithm and p is the number of processes (or threads).

In this question, we need to calculate the theoretical speed up using the formula given. Given that p is known, we need to identify the value for r_s , r_p and t_s using the serial program we developed in the question (a). We can begin the answering by estimating the value for r_s instead of r_p since q1.c is a serial program

r_s is the serial ratio which can be estimated using following formula

$$r_s = \frac{\text{Execution time of nonparallelizable portion}}{\text{Execution time of the program}}$$

To estimate the value for r_s , we need to identify the non-parallelizable portion of the program. Recall the workflow of the program, there are three main steps:

- Read input from user and initialize variables (non-parallelizable)
- Prime number searching via for loop
- Output the result to a text file (non-parallelizable)

In the sample solution, the second step is the part that can be parallelized in the sense that we can split the for loop into different parts and have the threads to perform prime number searching in each part. After identifying the non-parallelizable portion, the next step is to obtain the execution time of the program and the non-parallelizable portion. This is done by adding the following code (see Fig. 3) into the q1.c to record the execution time.

```

1 struct timespec start, end;
2 double time_taken;
3
4 // Get current clock time.
5 clock_gettime(CLOCK_MONOTONIC, &start);
6
7 /*
8  / Process
9  /
10 /
11 /
12 */
13
14
15 // Get the clock current time again
16 // Subtract end from start to get the CPU time used.
17 clock_gettime(CLOCK_MONOTONIC, &end_s1);
18 time_taken = (end_s1.tv_sec - start_s1.tv_sec) * 1e9;
19 time_taken = (time_taken + (end_s1.tv_nsec - start_s1.tv_nsec)) * 1e-9;
20 printf("Processing time (non-parallelizable)(Read)(s): %lf\n", time_taken);

```

Fig. 3. Code to record and display the time taken for the “process”

```
fit3143-student@fit3143:~/LAB/week3 $ ./q1
Input an integer number:5000000
Processing time (non-parallelizable)(Read)(s): 0.000014
Processing time (non-parallelizable)(Write)(s): 0.064214
Overall time (Including read, search and write)(s): 10.037655
```

Fig. 4. Executed program and processing time

Fig. 4 shows the execution of the program. From the figure, we can acquire the execution time for the non-parallelizable portions and entire program. To be noted, it is okay for not to factor in the execution time for the `scanf(...)` (user input) since the time is inconsistent due to the user's delay in typing the number.

Remark: In the sample solution, the value of n is 5,000,000 instead of the 1,000,000 to show the effect of the speedup in parallel version.

Based on the Fig. 4, the execution time for non-parallelizable portions and entire program are roughly 0.06 and 10.04 seconds. Knowing that, we can perform the calculation.

$$r_s = \frac{0.06}{10.04} = 0.0059$$

$$r_p = 1 - r_s = 1 - 0.0005 = 0.9940$$

$$s_{theory}(p) = \frac{1}{r_s + \frac{r_p}{p}} = \frac{1}{0.0059 + \frac{0.9940}{4}} = 3.93$$

- c) Write a parallel version of your serial code in C utilizing POSIX Threads. Here, design and implement a parallel partitioning scheme which distributed the workload among the threads.

Recall the workflow of the program, there are three main steps:

- Read input from user and initialize variables (non-parallelizable)
- Prime number searching via for loop
- Output the result to a text file (non-parallelizable)

Suppose n as the integer number inputted by the user, second step of the `q1.c` is to perform for loop from $i = 1$ until $n - 1$, and check if the i is a prime or non-prime in each iteration. In this case, we can divide the for loop into multiple parts, and have multiple threads to perform the prime number searching in each part. For instance, given $n = 10$ and there are $p = 4$ numbers of threads, we can perform a range partition to make $0 \dots 10$ into 4 parts with interval of $\left\lfloor \frac{10}{4} \right\rfloor = 2$. As such, the first thread computes the search prime from $1 - 2$, first thread computes the search prime from $3 - 4$ and so on. The implemented thread function is shown in the figure below.

```

/* Function Definition */
void *searchPrimeFunction(void *arg){

    // Get the thread id the (int*) is to type cast the arg from (void*) to (int*)
    int pid = *(int*) arg;

    // Perform range partition so that the thread search for prime number within specific range.
    int start = pid * pUpperLimit/NUMTHREAD + 1;
    int end = (pid+1) * pUpperLimit/NUMTHREAD;
    if(pid == NUMTHREAD-1){
        end = end-1;
    }

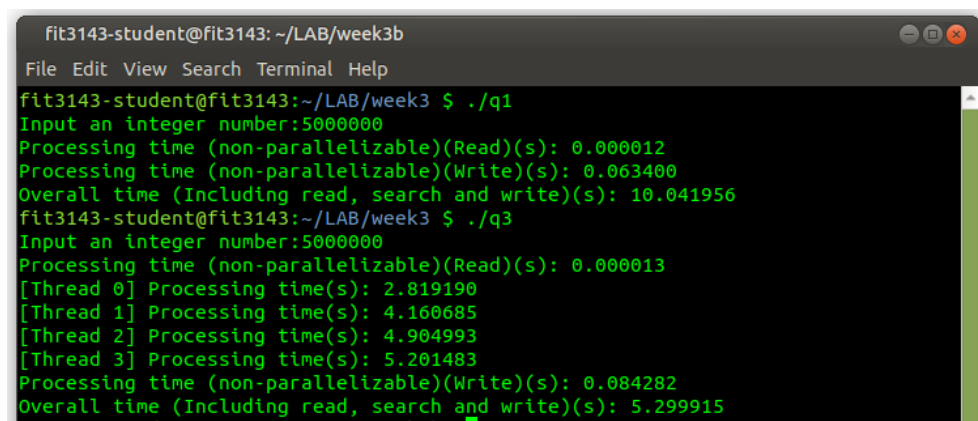
    // Prime number searching
    for(int i=start; i<=end; i++){
        if(isPrime(i)){
            // Add the prime number into the global array, mutex is implemented to prevent race condition
            pthread_mutex_lock(&g_Mutex);
            primeNumbers[primeCounter] = i;
            primeCounter++;
            pthread_mutex_unlock(&g_Mutex);
        }
    }
}

```

Fig. 5. Thread function

From the figure, the thread function first gets the thread id from the input argument. To be noted, the input argument (i.e., *arg*) is a void pointer that cannot be dereferenced and a (*int **) is used to type cast the *arg* into an integer pointer (can be dereferenced). After that, the function uses the thread id to compute the *start* and *end* points for the search prime operation. Lastly, a for loop is deployed to perform the search prime and the result is stored into the global array *primeNumbers*. Mutex lock and unlock are implemented to prevent race condition since multiple threads are accessing to the *primeNumbers* and *primeCounter*.

Compare the performance of the serial program (t_s) in part (a) with that of the parallel program (t_p) in part (c) for a fixed number of processors or threads (e.g., $p = 4$). Calculate the actual speed up, $S(p)$.



```

fit3143-student@fit3143: ~/LAB/week3b
File Edit View Search Terminal Help
fit3143-student@fit3143:~/LAB/week3 $ ./q1
Input an integer number:5000000
Processing time (non-parallelizable)(Read)(s): 0.000012
Processing time (non-parallelizable)(Write)(s): 0.063400
Overall time (Including read, search and write)(s): 10.041956
fit3143-student@fit3143:~/LAB/week3 $ ./q3
Input an integer number:5000000
Processing time (non-parallelizable)(Read)(s): 0.000013
[Thread 0] Processing time(s): 2.819190
[Thread 1] Processing time(s): 4.160685
[Thread 2] Processing time(s): 4.904993
[Thread 3] Processing time(s): 5.201483
Processing time (non-parallelizable)(Write)(s): 0.084282
Overall time (Including read, search and write)(s): 5.299915

```

Fig. 6. Executed serial and parallel programs

Fig.6 shows the execution time of the serial program (compiled from *q1.c*) and the parallel program (compiled from *q3.c*). From the figure, $t_s \approx 10.04$, $t_r \approx 5.30$ and this shows the execution time is improved after extending the serial program to the parallel version. The actual speed up $S(p)$ is the ratio between the performance of serial and parallel programs, which can be calculated using the below formula

$$S(p) = \frac{10.04}{5.30} \approx 1.89$$

The actual speed for the program to search for prime number lesser than 5000000 using 4 threads is 1.90.

Analyze and compare the actual speed up with the theoretical speed up for an increasing number of threads p and/or n . You can either tabulate your results or plot a chart when analysing the performance of the serial and parallel programs

An experiment is conducted to examine the effect of the number of threads p towards the speed up of the parallel program. In particular, the parallel program is executed with the p ranged from 2 to 8; while the input from user (n) is fixed at 5,000,000. Table 1 tabulates the experimental result in terms of the (serial and parallel) processing times and (theory and actual) speed ups. As observed from the table, the execution time for the parallel program is lesser than the serial program regardless of the number of threads. This shows the parallelization of the program is taking effect in increasing the efficiency of the program. From the table, the actual speed up is at the lowest point when $p = 2$, and starts to increase with the increment of p from 2 until 4. After that, the actual speed up levels off until $p = 8$. The curves of the theoretical speed up and actual speed up under different p are depicted in Fig. 7. From the figure, both speed ups increase when the p is in a smaller value. At the point where $p = 4$, the actual speed up starts to level off while the theoretical speed up is still increasing. Other than that, it is observed the actual speed up is lesser than the theoretical speed up under the same setting of p . The reasons for the difference between the *theoretical* and *actual* speed ups are explained as below:

- The theoretical speed up is calculated without considering the computation overhead of starting the threads. However, the actual speed up is calculated based on the parallel execution time that factors in the time of starting the threads.
- Unequal workload among threads is another problem in the implemented parallel program. In the sample solution, the task for searching prime number lesser than n is separated by using range partition to assign a range value to p number of threads. In this case, the thread with higher thread id is searching for prime number within higher value range. For example, given that $n = 1000$ and $p = 4$, the last thread search for the prime number between 751 and 1000. Recall the `isPrime(...)` function (see Fig. 1), another loop is used to determine a prime or non-prime number. Therefore, the thread that allocated with a higher value range requires higher workload. The execution time for different threads is shown in Fig. 8 to support the reasoning.

To further improve the parallel program, round robin partition strategy can be used to replace the range partition so that the workload among threads can be more equal.

Table 1: Theory speed up vs actual speed up under different setting of p

Number of Thread p	Processing Time (Serial Ver.) t_s	Processing Time (Parallel Ver.) t_p	Theoretical Speed Up $S_{theory}(p)$	Actual Speed Up $S(p)$
2	10.04	6.77	1.99	1.48
3		5.86	2.96	1.71
4		5.30	3.93	1.89
5		5.59	4.88	1.79
6		5.62	5.83	1.78
7		5.46	6.76	1.83
8		5.53	7.68	1.81

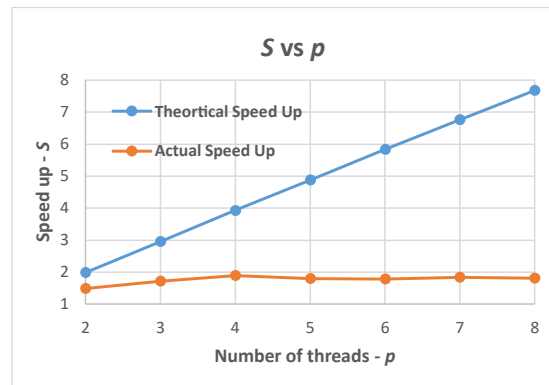


Fig. 7. Curves of speed up S vs number of threads p for the developed parallel program

```

fit3143-student@fit3143: ~/LAB/week3b
File Edit View Search Terminal Help
fit3143-student@fit3143:~$ cd LAB
fit3143-student@fit3143:~/LAB$ cd week3
fit3143-student@fit3143:~/LAB/week3 $ gedit q3.c
fit3143-student@fit3143:~/LAB/week3 $ gcc q3.c -o q3 -lm -lpthread
fit3143-student@fit3143:~/LAB/week3 $ ./q3
Input an integer number:5000000
Processing time (non-parallelizable)(Read)(s): 0.000015
[Thread 0] Processing time(s): 2.622678
[Thread 1] Processing time(s): 3.740185
[Thread 2] Processing time(s): 4.694789
[Thread 3] Processing time(s): 5.117425
[Thread 4] Processing time(s): 5.364455
[Thread 5] Processing time(s): 5.563452
[Thread 6] Processing time(s): 5.657240
[Thread 7] Processing time(s): 5.856118
Processing time (non-parallelizable)(Write)(s): 0.140034
Overall time (Including read, search and write)(s): 6.013732
  
```

Fig. 8. Executed parallel search prime program with 8 threads

Declaration

I declare that this eFolio tasks and the linked code are my individual work. I have not copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text and code, nor has any part of this submission been written for me by another person.

Signature of student 1: _____

Signature of student 2: _____