# CPSC 457 — Principles of Operating Systems
# ASSIGNMENT 3

**Name:** Haohu Shen
**UCID:** 30063099

2. **Written question (5 marks)**

   **Solution.** The tables of the timings are shown below:

   File Name: *medium.txt*

| # threads | Observed timing | Observed speedup compared to original | Expected speedup |
|---|---|---|---|
| original program | 32.034s | 1.0 | 1.0 |
| 1 | 33.044s | 0.97 | 1.0 |
| 2 | 17.017s | 1.88 | 2.0 |
| 3 | 11.530s | 2.78 | 3.0 |
| 4 | 8.679s | 3.69 | 4.0 |
| 8 | 4.418s | 7.25 | 8.0 |
| 16 | 4.580s | 6.99 | 16.0 |

   File Name: *hard.txt*

| # threads | Observed timing | Observed speedup compared to original | Expected speedup |
|---|---|---|---|
| original program | 10.370s | 1.0 | 1.0 |
| 1 | 11.691s | 0.89 | 1.0 |
| 2 | 5.822s | 1.78 | 2.0 |
| 3 | 3.912s | 2.65 | 3.0 |
| 4 | 2.919s | 3.55 | 4.0 |
| 8 | 1.939s | 5.35 | 8.0 |
| 16 | 1.849s | 5.61 | 16.0 |

   File Name: *hard2.txt*

| # threads | Observed timing | Observed speedup compared to original | Expected speedup |
|---|---|---|---|
| original program | 9.955s | 1.0 | 1.0 |
| 1 | 11.585s | 0.86 | 1.0 |
| 2 | 5.822s | 1.71 | 2.0 |
| 3 | 3.902s | 2.55 | 3.0 |
| 4 | 2.912s | 3.42 | 4.0 |
| 8 | 1.520s | 6.55 | 8.0 |
| 16 | 1.551s | 6.42 | 16.0 |

Once you have created the tables, explain the results you obtained.
Are the timings what you expected them to be? If not, explain why they differ.

**Solution.** To parallelize the existing solution, I split the current algorithm into 2 stages without changing the logic of it, if the original algorithm takes $n$ as its input, it will cost $O(1)$ time to process $n$ if $n \leq 3$ or $2|n$ or $3|n$, and otherwise it will cost $O(\sqrt{n})$ to process and stop once a non-trivial divisor is found. Here I read all inputs and categorize them as the original program since creating and using threads on at most 10000 integers in this stage are not efficient due to the time to allocate the workload, create and join threads. If the input is categorized into the first three cases (all cost $O(1)$), then the then the result can be calculated directly, otherwise we store them into a container and handle them on the second stage using multithreads.

The strategy I use on the 2nd stage is that I process one number at a time in the container mentioned before using all threads given, from the algorithm we know we have to test factors $5 + 6i$ and $5 + 6i + 2$ such that $5 \leq 5 + 6i \leq \lfloor\sqrt{n}\rfloor$, which makes us be able to calculate and allocate roughly equal workload for each thread. Since the algorithm can stop as long as a smallest non-trivial divisor is found, we have to use a synchronization mechanism to tell other threads if one thread has get the job done. Therefore, I use an atomic variable to store the smallest divisor found so far for each test as a shared variable among all threads and cancel the current thread if a smaller non-trivial divisor is found by itself or other threads.

The observed speedups compared to original are not as much as expected, here are some reasons:

- From all three tables we can observe that when the routine takes 1 thread it is a little bit slower than the original program in all three cases, the reason is that we still have to create and terminate this 1 thread on the second stage, which is overhead in this situation since the size of inputs in all three cases are relatively small, while the original program just directly process all inputs without spawning any threads.
- If the original program takes time $T$ to complete a case, from all three cases we can observe that the routine did speed up but was still slower than $T/N$ if the number of threads $N$ given is between 1 and 8, one reason is that although I used thread barrier to re-use threads in my routine, the spawning, joining threads and barrier wait to synchronize all threads still cost time since the workload for each thread are not strictly allocated equally. The most important reason is that in order to synchronize with other

threads on the shared variable, we have to do a lot of atomic operations which are expensive especially the CAS operation can cause cache misses although it is lockless. On the other hand, if we are given $N$ threads, we cannot guarantee all $N$ CPU cores are fully utilized since it has to do with the scheduling policy of resources. Moreover, since the workload of un-parallelized tasks and parallelized tasks are different, the theoretical speedup of the execution of the whole task are not ideal due to *Amdahl's law*.

- We also notice that when $N = 16$ the observed timing is approximately the same as that when $N = 8$, and sometimes it is a little bit slower than the case when $N = 8$, that is because the Linux machine for grading only has 8 CPUs and only 1 hardware thread per core (you can use *lscpu* to check that out), when we assign more than 8 software threads, only 8 hardware threads are working at most, which is the same when $N = 8$, and frequently switching between software threads is also time-consuming.