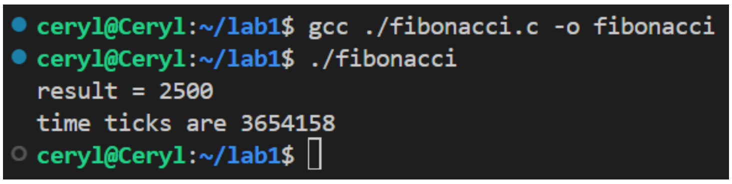
Lab1

学号： N/A

姓名： N/A

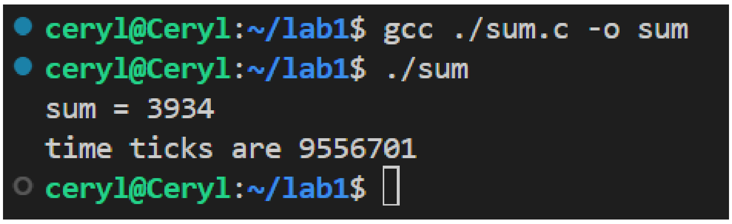
专业： 计算机科学与技术

Question 1



```
ceryl@Ceryl:~/lab1$ gcc ./fibonacci.c -o fibonacci
ceryl@Ceryl:~/lab1$ ./fibonacci
result = 2500
time ticks are 3654158
ceryl@Ceryl:~/lab1$
```

Figure 1: result of code (a)



```
ceryl@Ceryl:~/lab1$ gcc ./sum.c -o sum
ceryl@Ceryl:~/lab1$ ./sum
sum = 3934
time ticks are 9556701
ceryl@Ceryl:~/lab1$
```

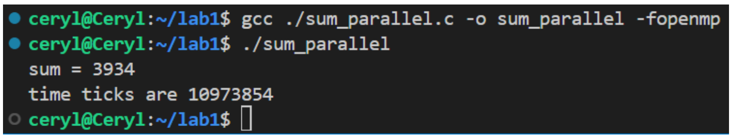
Figure 2: result of code (b)

Question 2

Code (b) is more appropriate for multi-threading. Calculation of $i^3 + i^2$ is independent of each other, so it is possible to calculate them in parallel. Sum of the results can also be accelerated by dividing the array into several parts and summing them in parallel.

The tasks in code (a) are dependent on each other. $\mathrm{fib}(n)$ is dependent on $\mathrm{fib}(n-1)$ and $\mathrm{fib}(n-2)$, so it is not suitable for multi-threading.

Question 3



```
ceryl@Ceryl:~/lab1$ gcc ./sum_parallel.c -o sum_parallel -fopenmp
ceryl@Ceryl:~/lab1$ ./sum_parallel
sum = 3934
time ticks are 10973854
ceryl@Ceryl:~/lab1$
```

Figure 3: result of parallel code

The parallel code accelerate the calculation by dividing the range of i into 4 parts and calculating them in parallel. The thread-local sums are then summed up atomically. The result is the same as the serial code.

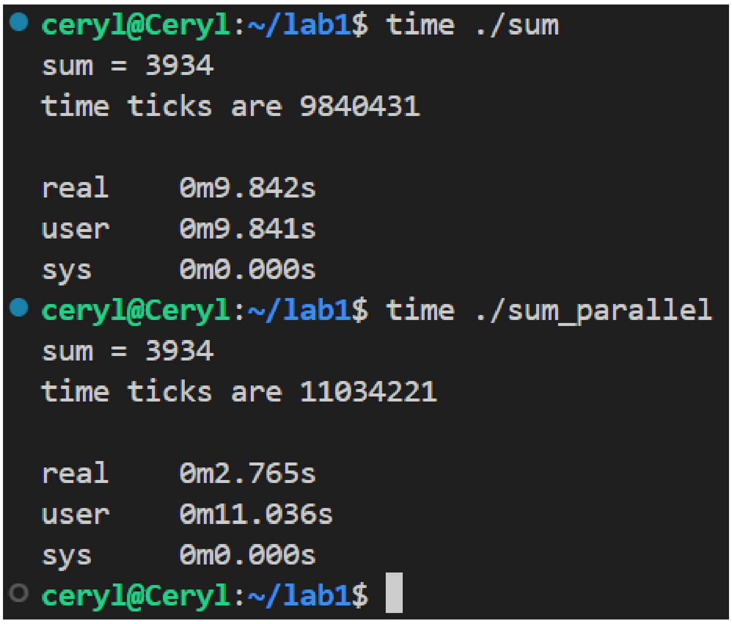


Figure 4: time comparison of serial and parallel code

The cpu time ticks are slightly more (12.1%) than the serial code, but the real time is significantly less (71.9%).

The extra cpu time ticks are caused by the overhead of creating and joining threads, calculating thread-local range of i, and atomic operations.

The real time is significantly less because the calculation is done in parallel (4 threads), resulting in a 4x speedup. The real speedup can be calculated as $\text{Speedup} = \frac{\text{user time}}{\text{real time}} = \frac{11.036}{2.765} = 3.9913$.

Simply adding `#pragma omp parallel for` will yield a wrong result because of the non-atomic operation `sum = (sum + square + cube) % mod` . There are two ways to solve this problem:

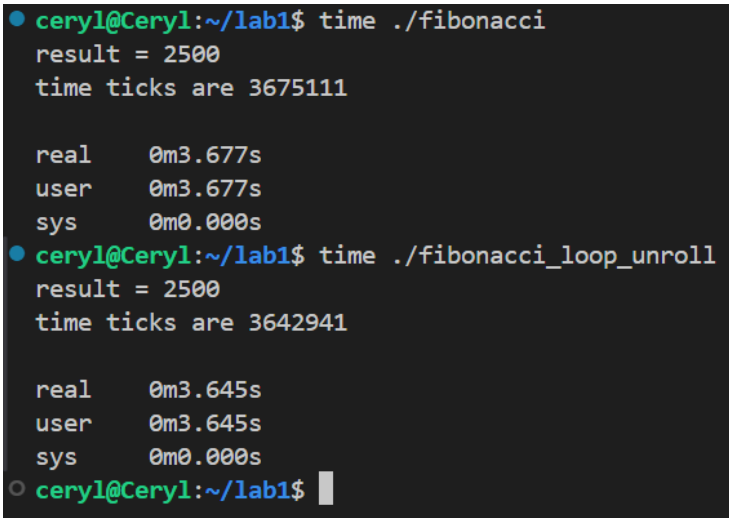
1. Use `#pragma omp critical` to lock the value "sum". But this will significantly slow down the calculation due to mutex locks.
2. Use `#pragma omp atomic` to make the operation atomic. This will not slow down the calculation, but OpenMP does not support atomic operations with two binary operators ($+$ and $\%$ in this case). `#pragma omp parallel for reduction(+:sum)` will encounter the same problem.

Therefore, the best way is dividing the range of i into 4 parts and calculating them in parallel, and summing them up atomically. This will not slow down the calculation and will yield the correct result.

Question 4

In my code, $i = 35$ will be calculated in thread t0 as it is in the first quarter of the range of i. However, if `#pragma omp parallel for num_threads(4) schedule(static, 2)` is used, $i = 35$ will be calculated in thread t1 as $35 \equiv 3 \bmod 8$.

Question 5



```
ceryl@Ceryl:~/lab1$ time ./fibonacci
result = 2500
time ticks are 3675111

real    0m3.677s
user    0m3.677s
sys     0m0.000s
ceryl@Ceryl:~/lab1$ time ./fibonacci_loop_unroll
result = 2500
time ticks are 3642941

real    0m3.645s
user    0m3.645s
sys     0m0.000s
ceryl@Ceryl:~/lab1$
```

Figure 5: result of unrolled code