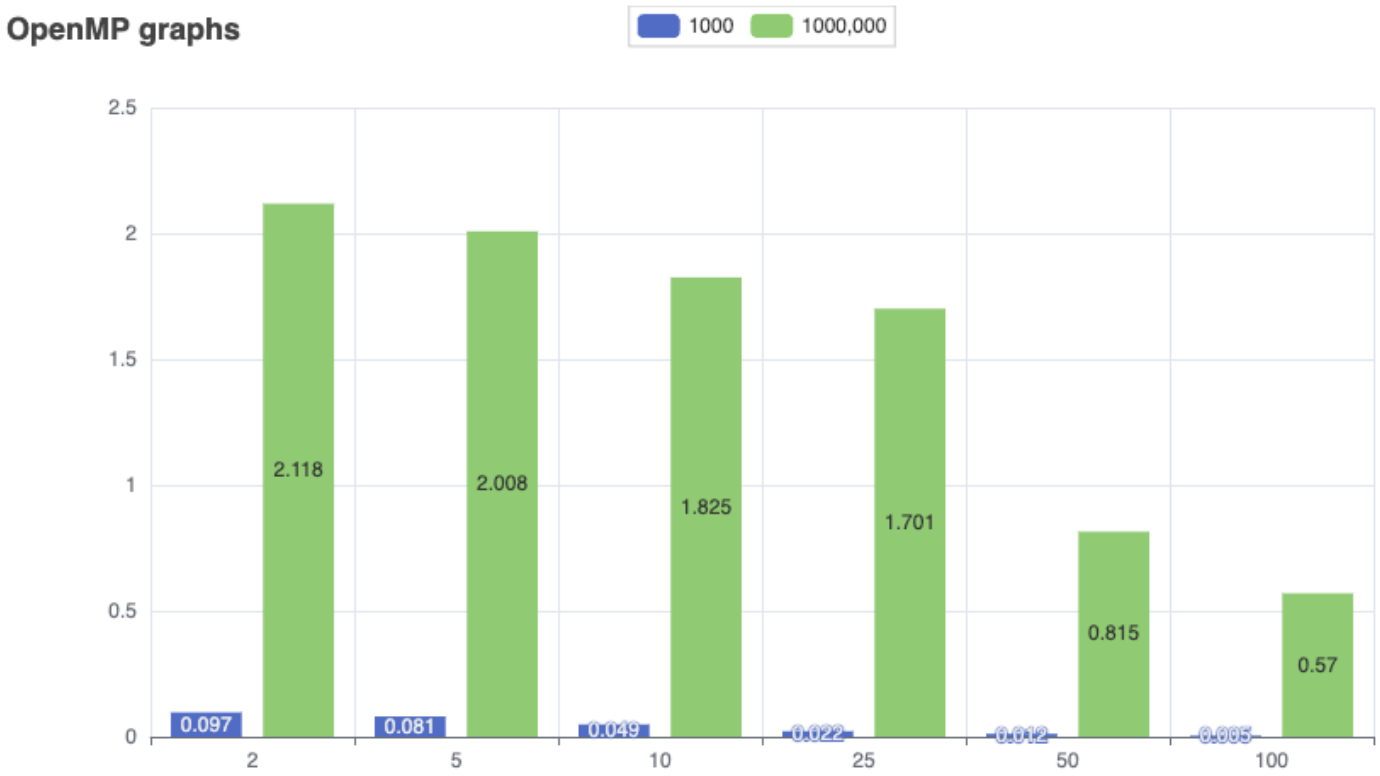
Lab2 report

Graph 1:



Graph 1 (N = 1000), y-axis is the speedup relative to the running time with 1thread; and x-axis is the number of threads: 2, 5, 10, 25, 50, and 100.

> When N=1000, it is fast to just use the sequential version of the algorithm because creating and determining threads have extra overhead and it increases with the number of threads. Therefore, the speedup continues decreasing.

Graph 2 (N = 1000,000), y-axis is the speedup relative to the running time with 1thread; and x-axis is the number of threads: 2, 5, 10, 25, 50, and 100.

> Basically, because of the optimization of the sequential codes, it performs better but still does much more delicate work when it comes to multithreading.
>
> We can use take less time with do these for loops:

```
#pragma omp for nowait
for (int i = 3; i <= N; i+=2) {
    isPrime[i] = true;
}
#pragma omp for
for (int i = 4; i <= N; i += 2) {
    isPrime[i] = false;
}
```

but when doing this loop, the advantage of prior work decreases.

```
#pragma omp for
for (int i = 3; i <= (N + 1) / 2 ; i +=2) {
    if (isPrime[i]&& i<=N/i) {
        for (int j = i * i; j <= N; j += i)
            isPrime[j] = false;
    }
}
```

For example, when i=0, we make a large number of fake primes to be false. Then the following for loops can do less work. But if the for loop is divided by threads, many fake primes haven't been made to false, so basically, the performance is largely affected by the span, i.e. when i=0. Hence, it causes the speedup to be the largest when the number of threads equals to 2.