Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing **2022**

Student: Qianbo Zang Discussed with: Volodymyr Karpenko

Solution for Project 2

Due date: 26.10.2022 (midnight)

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP [10 points]

1.1. Implement a parallel version

1.1.1. Using reduction pragma

The reduction specifics the whole parallel region.

```
for (int iterations = 0; iterations < NUM_ITERATIONS; ++iterations) {
    alpha_parallel = 0.0;
    #pragma omp parallel for reduction(+: alpha_parallel)
    for (int i = 0; i < N; ++i) {
        alpha_parallel += a[i] * b[i];
    }
}
```

1.1.2. Using a critical pragma

The critical specifies that code is executed by one thread at a time.

```
for (int iterations = 0; iterations < NUM_ITERATIONS; ++iterations) {
    alpha_parallel = 0.0;
    #pragma omp parallel
    {
        long double tmp = 0;
        #pragma omp for
        for (int i = 0; i < N; ++i) {
            tmp += a[i] * b[i];
        }
        #pragma omp critical
        alpha_parallel += tmp;
    }
}
```

1.2. plot the parallel efficiency for all these dot product benchmarks

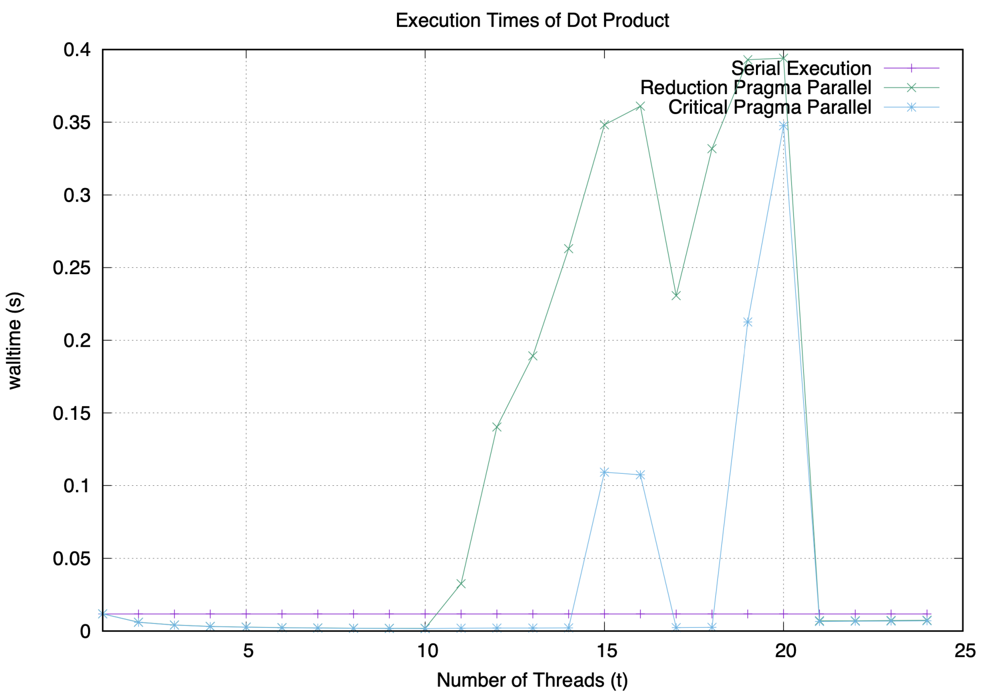


Figure 1: Performance of N = 100000

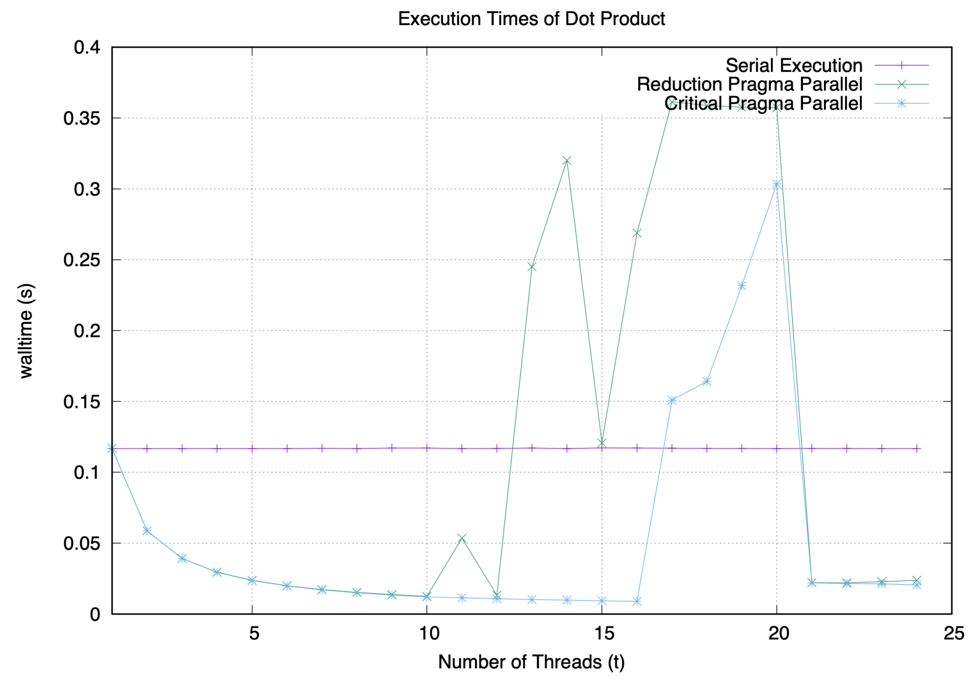


Figure 2: Performance of N = 1000000

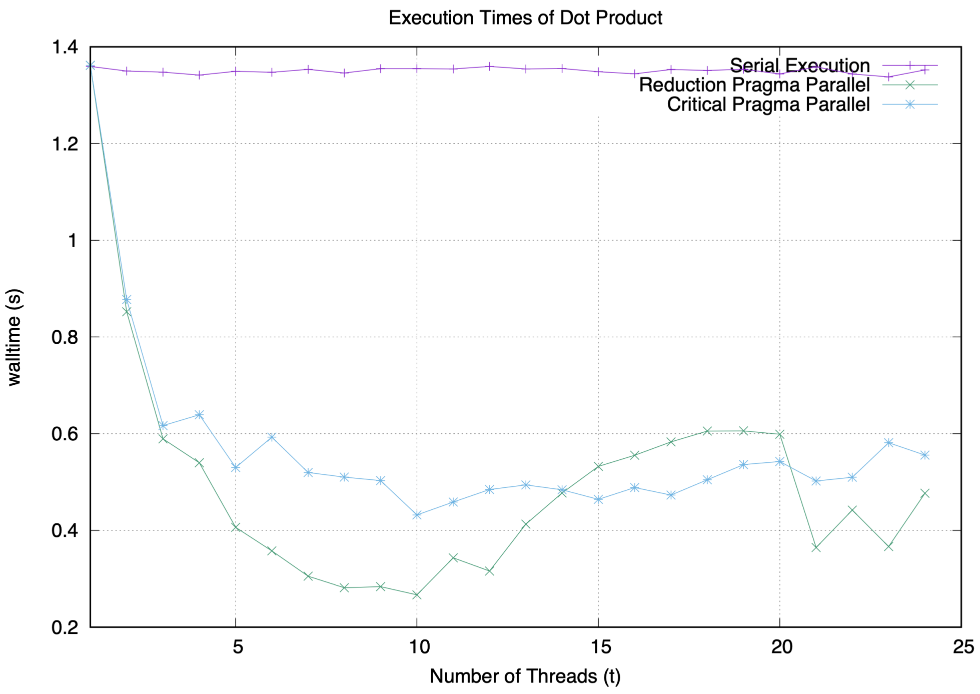


Figure 3: Performance of N = 10000000

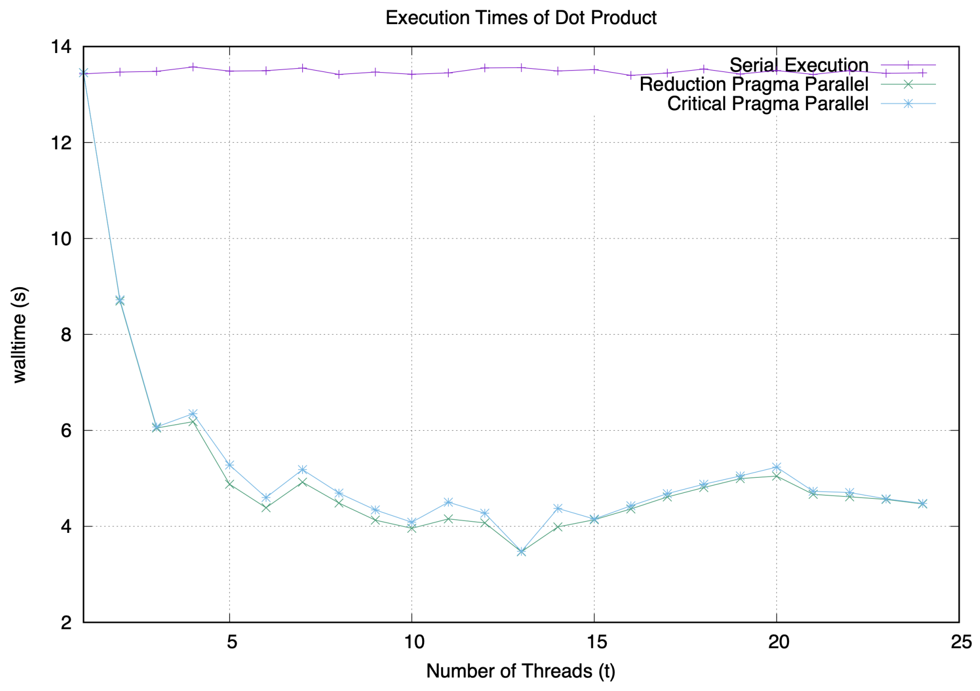


Figure 4: Performance of N = 100000000

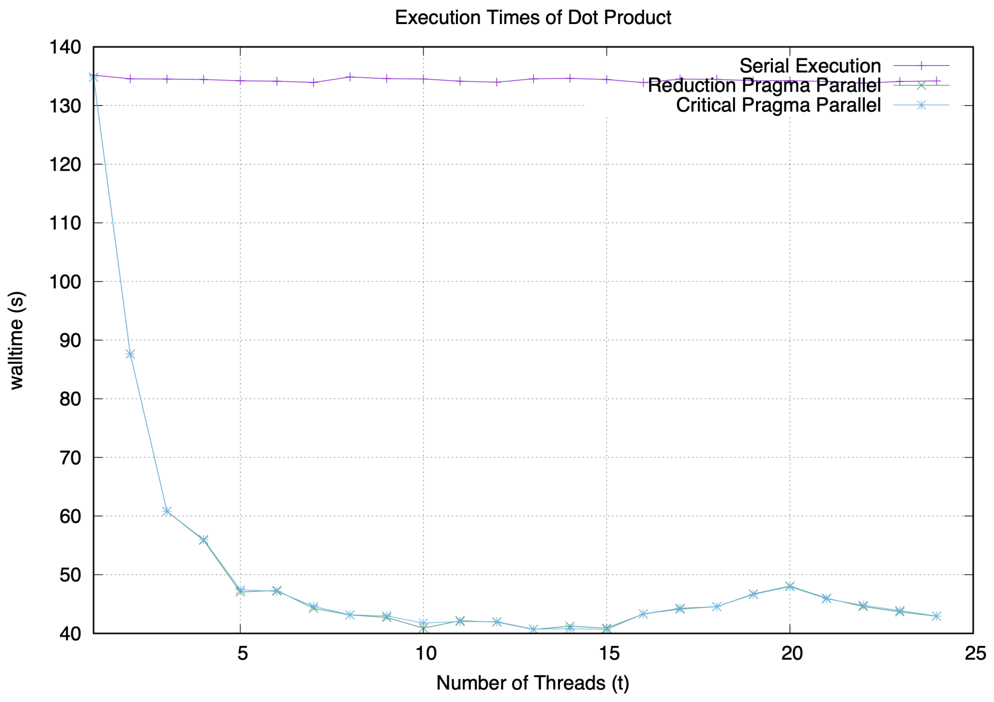


Figure 5: Performance of N = 1000000000

1.3. Summarize my observations

From Figures, I find that when the number of threads is less than 10, the performances of "Reduction" and "critical" are almost equal. And when the number of threads is from 10 to 21, the performance of "Reduction" is obviously better than "critical". I think the reason is the data synchronization happens on every iteration by modifying variables, which makes "critical" slower. But after there are enough threads, the performances of "Reduction" and "critical" become equal again.

As for the comparison of serial and parallel, when N = 100000 and N = 1000000, as the increase of threads, the walltimes of parallel decrease at first but increase very fast soon. From threads = 11 to thread = 10, the walltimes of parallel are even longer than the serial. In this case, I think the thread load is duplicated. The threads are likely to be switched frequently, resulting in slower times. When n > 10000000, as the increase of threads, the walltimes of parallel become only $\frac{1}{3}$ of serial.

2. The Mandelbrot set using OpenMP [30 points]

Reference: `https://github.com/skeeto/mandel-simd/blob/1e9cf34e93cfaa60fba487e722dec2cc33e89ca` `mandel.c#L20-L46`

2.1. Implement the computation kernel in mandel_seq.c

```
for (n = 0; (x2 + y2 < 4) && (n < MAX_ITERS); ++n, ++nTotalIterationsCount) {
    zx = x2 - y2 + cx;
    zy = 2 * x * y + cy;
    x = zx, y = zy;
    x2 = x * x, y2 = y * y;
}
```

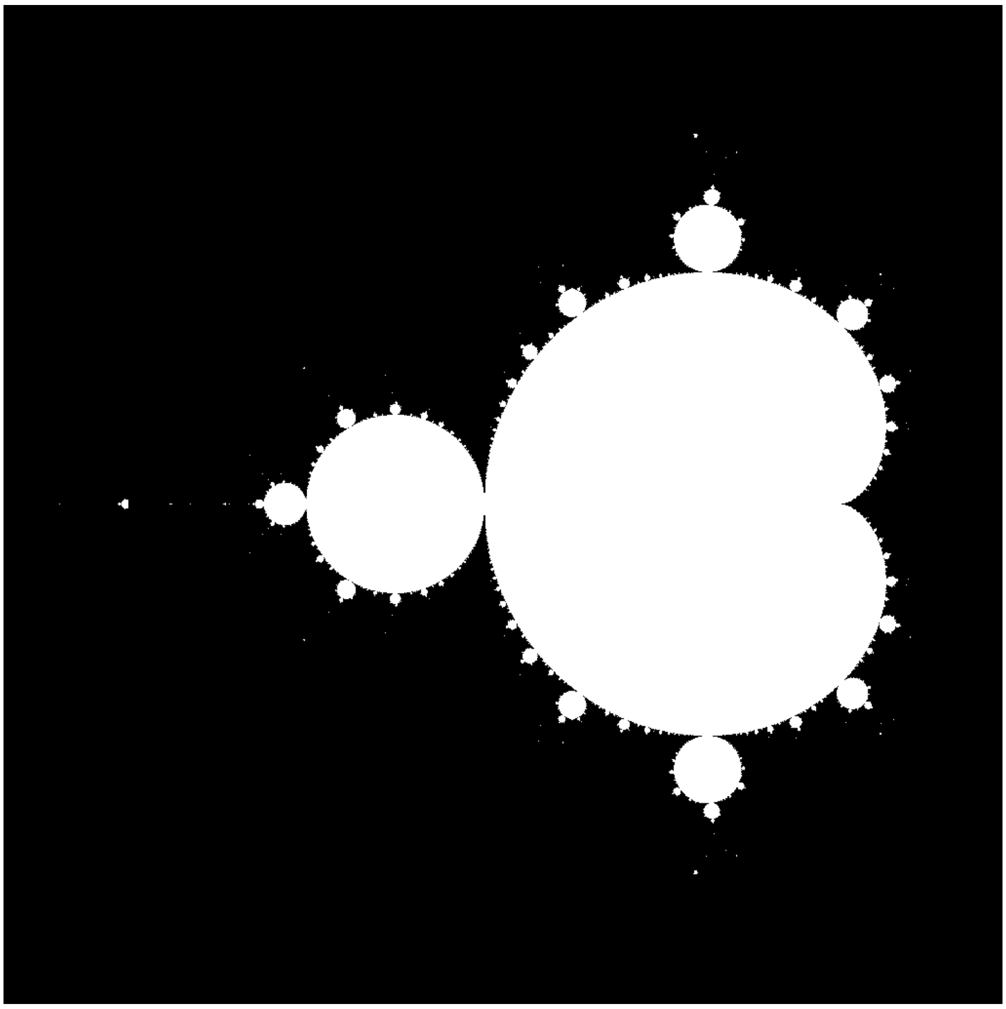
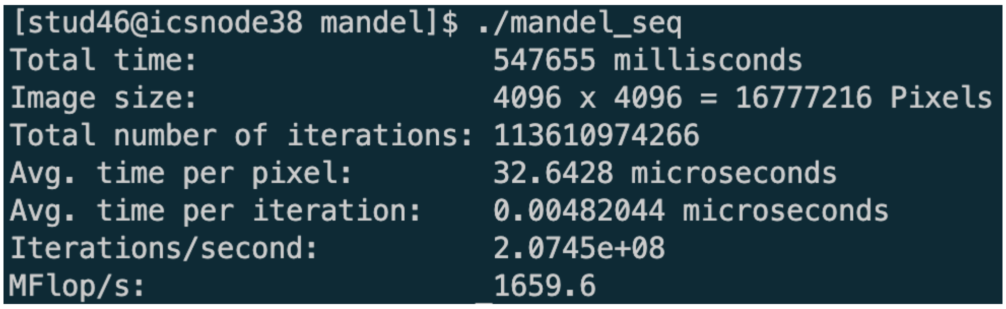


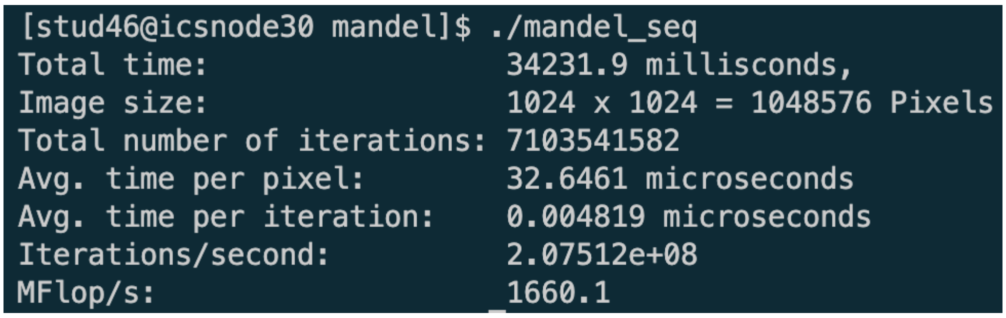
Figure 6: Mandel

From Figures 7 and 8, we know how many iterations it performs per second, and what the performance is in MFlop/s. I choose Image size (4096 x 4096) and (1024 x 1024).



```
[stud46@icsnode38 mandel]$ ./mandel_seq
Total time:                  547655 millisconds
Image size:                  4096 x 4096 = 16777216 Pixels
Total number of iterations: 113610974266
Avg. time per pixel:         32.6428 microseconds
Avg. time per iteration:     0.00482044 microseconds
Iterations/second:           2.0745e+08
MFlop/s:                     1659.6
```

Figure 7: Serial, Image Size 4096 x 4096



```
[stud46@icsnode30 mandel]$ ./mandel_seq
Total time:                  34231.9 millisconds,
Image size:                  1024 x 1024 = 1048576 Pixels
Total number of iterations: 7103541582
Avg. time per pixel:         32.6461 microseconds
Avg. time per iteration:     0.004819 microseconds
Iterations/second:           2.07512e+08
MFlop/s:                     1660.1
```

Figure 8: Serial, Image Size 1024 x 1024

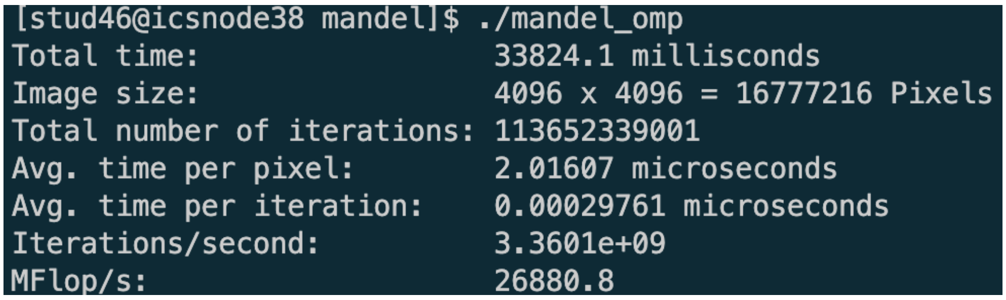
2.2. Count the total number of iterations

```
for (n = 0; (x2 + y2 < 4) && (n < MAX_ITERS); ++n, ++nTotalIterationsCount)
```

2.3. Parallelize the Mandelbrot code

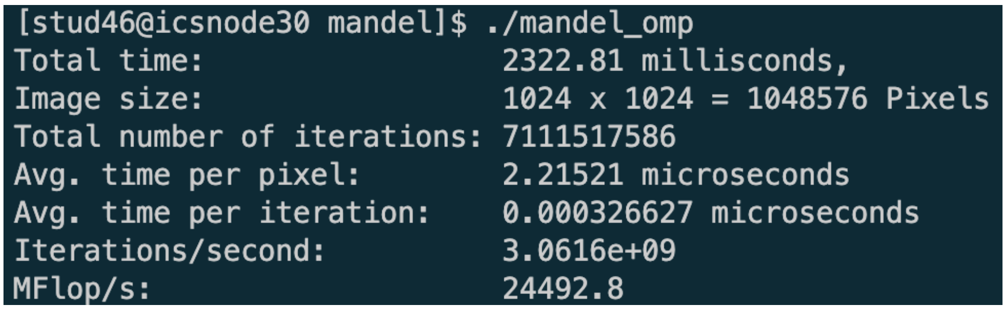
```
#pragma omp parallel for collapse(2) reduction(+: nTotalIterationsCount) schedule(
    dynamic)
for (j = 0; j < IMAGE_HEIGHT; ++j) {
    for (i = 0; i < IMAGE_WIDTH; ++i) {
        cx = MIN_X + i * fDeltaX;
        x = cx;
        y = cy;
        x2 = x * x;
        y2 = y * y;
        // compute the orbit z, f(z), f^2(z), f^3(z), ...
        // count the iterations until the orbit leaves the circle |z|=2.
        // stop if the number of iterations exceeds the bound MAX_ITERS.
        for (n = 0; (x2 + y2 < 4) && (n < MAX_ITERS); ++n, ++nTotalIterationsCount)
    {
            zx = x2 - y2 + cx;
            zy = 2 * x * y + cy;
            x = zx, y = zy;
            x2 = x * x, y2 = y * y;
        }
        cy = MIN_Y + j * fDeltaY;
        int c = ((long)n * 255) / MAX_ITERS;
        png_plot(pPng, i, j, c, c, c);
    }
}
```

The running terminals of parallel which use 30 threads are showed Figure 9 and 10. I find that the speed is 10 times larger than the serial versions. From the comparison of parallel and serial in two picture sizes, it turns out that parallelism is much better than serial even for small image sizes.



```
[stud46@icsnode38 mandel]$ ./mandel_omp
Total time:                  33824.1 millisconds
Image size:                  4096 x 4096 = 16777216 Pixels
Total number of iterations: 113652339001
Avg. time per pixel:         2.01607 microseconds
Avg. time per iteration:     0.00029761 microseconds
Iterations/second:           3.3601e+09
MFlop/s:                     26880.8
```

Figure 9: Parallel, Image Size 4096 x 4096



```
[stud46@icsnode30 mandel]$ ./mandel_omp
Total time:                  2322.81 millisconds,
Image size:                  1024 x 1024 = 1048576 Pixels
Total number of iterations: 7111517586
Avg. time per pixel:         2.21521 microseconds
Avg. time per iteration:     0.000326627 microseconds
Iterations/second:           3.0616e+09
MFlop/s:                     24492.8
```

Figure 10: Parallel, Image Size 1024 x 1024

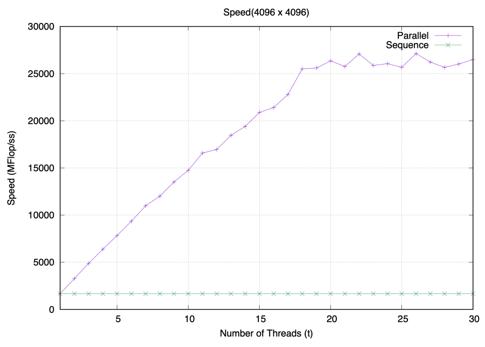


Figure 11: Speed 4096 x 4096

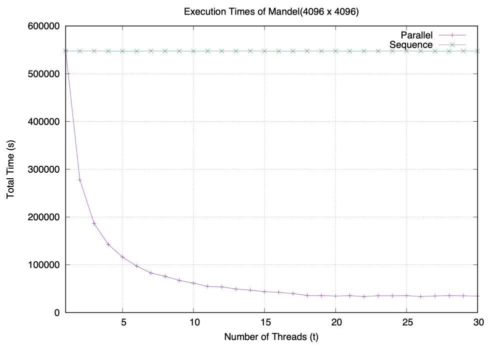


Figure 12: Total Time 4096 x 4096

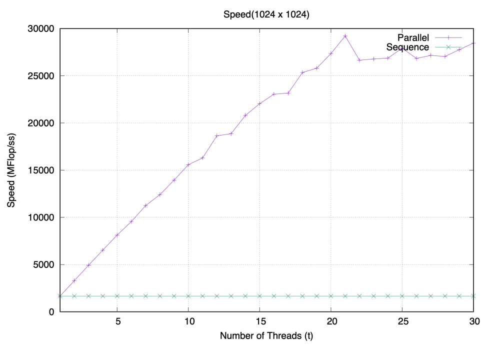


Figure 13: Speed 1024 x 1024

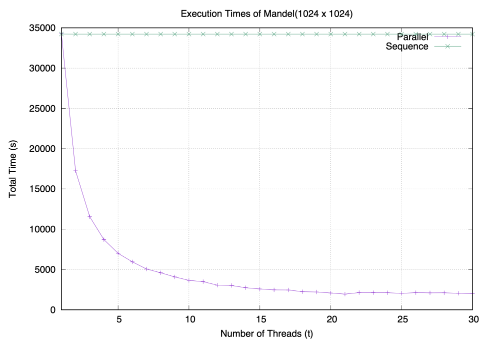


Figure 14: Total Time 1024 x 1024

3. Bug hunt [15 points]

3.1.

The 'tid' is private. So, in for loop, I must initialise variable 'tid'. I put 'tid' initialisation in the loop.

```
#pragma omp parallel for shared(a, b, c, chunk) private(i, tid) schedule(static,
    chunk)
for (i = 0; i < N; i++) {
    tid = omp_get_thread_num();
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
}
```

3.2.

The 'tid' is incorrectly shared, and I need to put it to private.

```
#pragma omp parallel private(tid) {
    /* Obtain thread number */
    tid = omp_get_thread_num();
    /* Only master thread does this */
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d is starting...\n", tid);

    #pragma omp barrier
    /* do some work */
    total = 0.0;
    #pragma omp for schedule(dynamic, 10)
    for (i = 0; i < 1000000; i++)
        total = total + i * 1.0;
    printf("Thread %d is done! Total= %e\n", tid, total);
} /*** End of parallel region ***/
```

3.3.

Only 2 threads get the task, so the other threads don't need to be synchronized after 'print_results()'. I need to delete the 'pragma omp barrier' in fuction 'print_results()'.

```c
void print_results(float array[N], int tid, int section) {
    int i, j;

    j = 1;
    /*** use critical for clean output ***/
    #pragma omp critical
    {
        printf("\nThread %d did section %d. The results are:\n", tid, section);
        for (i = 0; i < N; i++) {
            printf("%e  ", array[i]);
            j++;
            if (j == 6) {
                printf("\n");
                j = 1;
            }
        }
        printf("\n");
    } /*** end of critical ***/
}
```

3.4.

The array is too large compared with stack-size. Just increase stack-size.

```
ulimit -s unlimited
```

3.5.

The problem is deadlock. Thread 1 opens lock A conditional on opening lock B. The only thing is that lock B is held by thread 2 and is conditional on opening lock A. Without any additional information at this time, I'm not sure that one should happen first. So this question cannot be solved.

4. Parallel histogram calculation using OpenMP [15 points]

The reduction clause can be used here because all elements need to be in different threads.

```c
#pragma omp parallel for reduction(+: dist)
for (long i = 0; i < VEC_SIZE; ++i) {
    dist[vec[i]]++;
}
```

From Figure 15, I can also find that when array size is less than 10000000, parallelize this algorithm not only fails to improve the run-time significantly, but it even appears to run longer. For example, there is 20 threads. From Figure 16 and 17, when the size of the array is extremely large, the parallel speed does increase a lot and becomes stable as the number of threads increases.

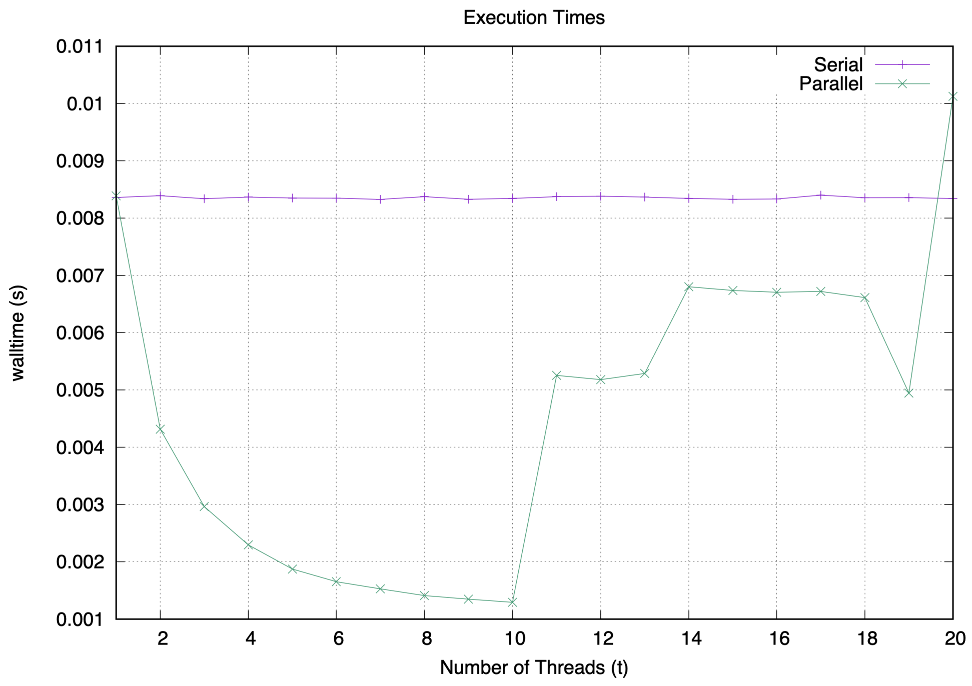


Figure 15: array size 10000000

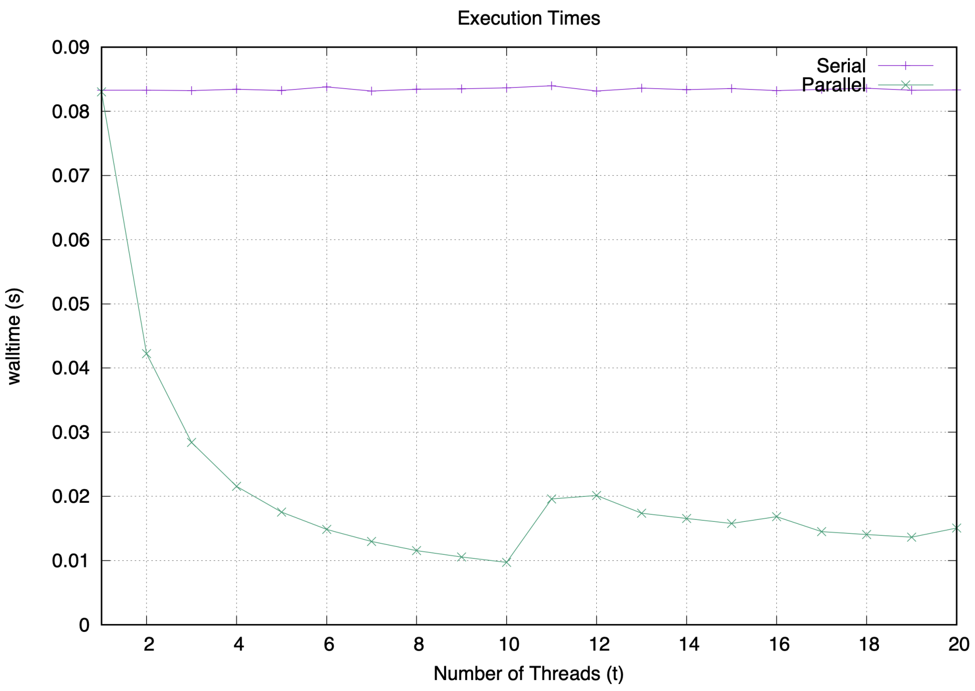


Figure 16: array size 100000000

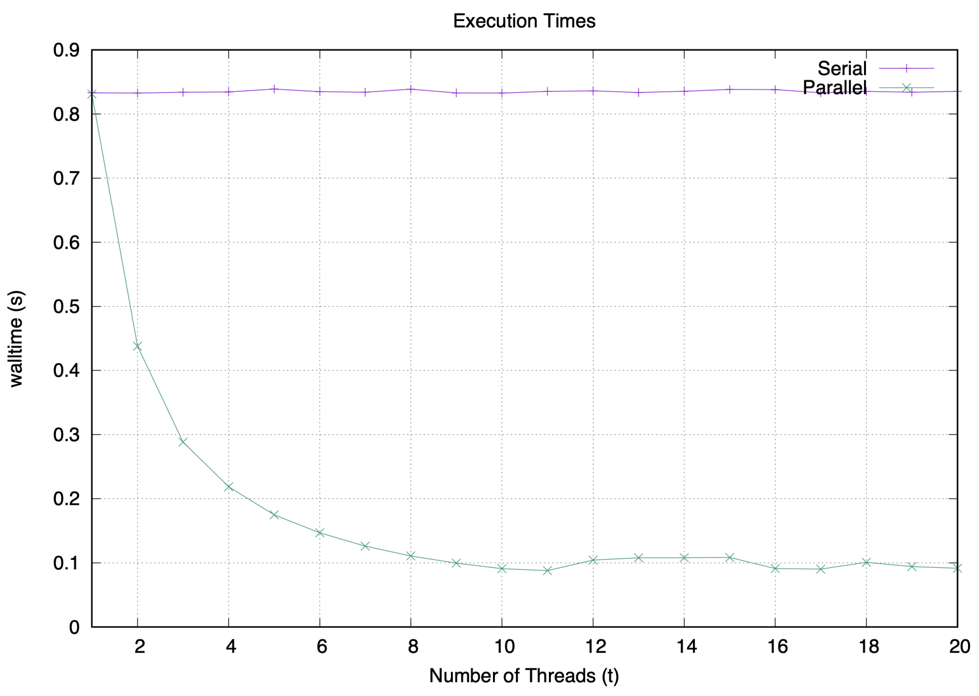


Figure 17: array size 1000000000

5. Parallel loop dependencies with OpenMP [15 points]

To check the serial version, I find that

$$V = [Sn, Sn \cdot up, Sn \cdot up^2 ...]$$

And also, Sn is equal to up. So,

$$V = [up, up^2, up^3 ...]$$

I set the variable Sn to lastprivate. because when the program ends the loop in parallel, I need to keep the Sn value from the last loop. Every thread should have access to the opt array, so I set this array to public. After the parallel loop, there will be another loop to define n=0. So, I don't need to set n to private or firstprivate.

```
#pragma omp parallel for shared(opt) lastprivate(Sn)
for (n = 0; n <= N; ++n) {
    // when n=0, Sn = up^(1) = up^(n+1)
    Sn = pow(up, n+1);
    opt[n] = Sn;
}
```

From Figure 18 and 19, the parallel code will not take less time than serial code, even when there are enough threads. This is because the second loop needs the values obtained from the first loop.

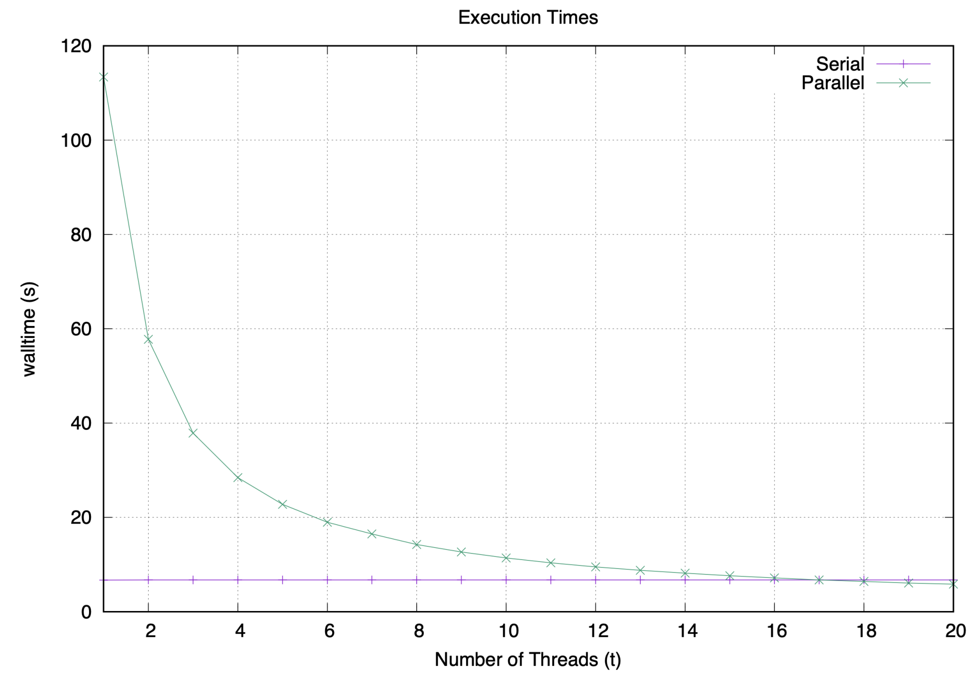


Figure 18: array size 2000000000

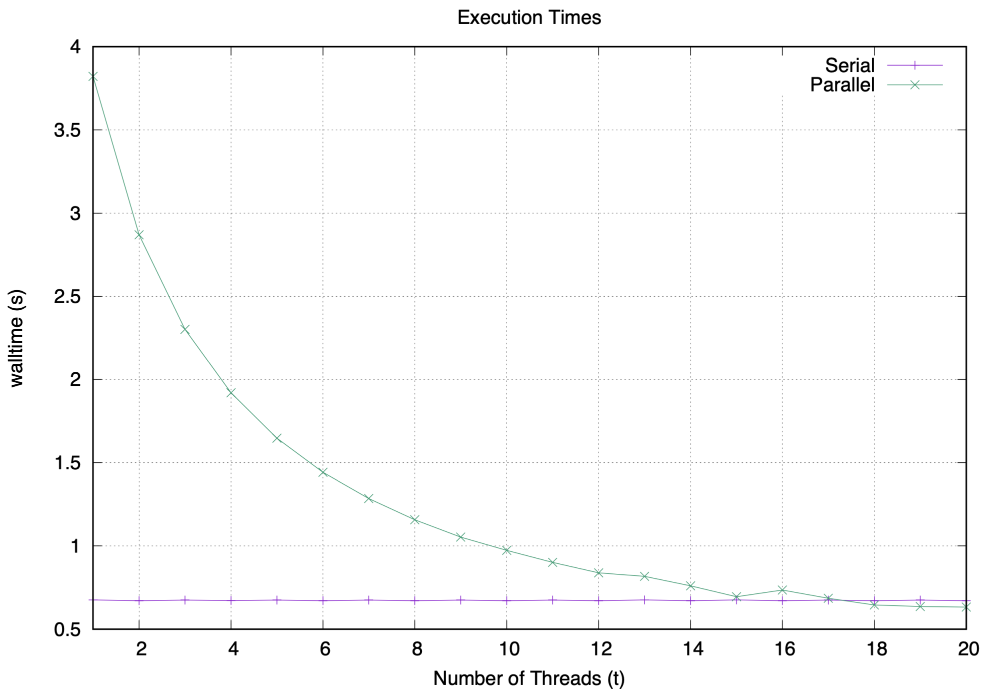


Figure 19: array size 200000000

6. Task: Quality of the Report [15 Points]