Multicore Computing

Assignment Three (Practical)

Department of Computer Engineering

Sharif University of Technology

Spring 2022

Lecturer:
Dr. Falahati

Name - Student Number:
Saba Hashemi - 97100581
Amirmahdi Namjoo - 97107212

1 Question One

- Team member No.1: Saba Hashemi,97100581
- Team member No.2: Amirmahdi Namjoo,97107212

2 Question Two

3 Question Three

a. The code is included in `Q3.c`.

b. The results of running the code for different parameters are included in `results3.xlsx`. Note that it has multiple sheets and the name of each sheet represents the parameters given to the program. The column with title 1 shows the results of running the unparalleled version of the code. All other columns represent the results of running the Parallelized version of the code with OpenMP.

c. Plots for different input parameters:

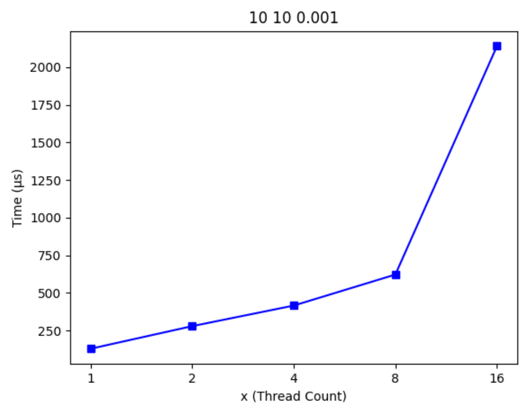

Figure 1: Results of running the code with parameters 10 10 0.001

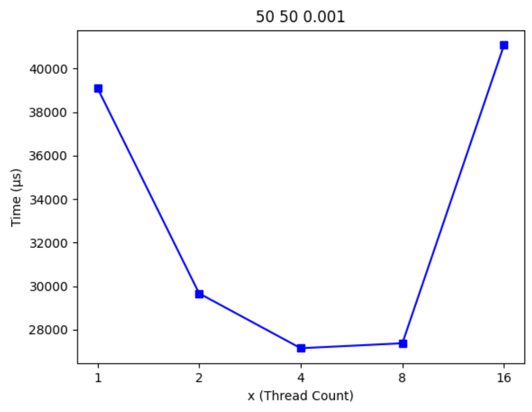

Figure 2: Results of running the code with parameters 50 50 0.001

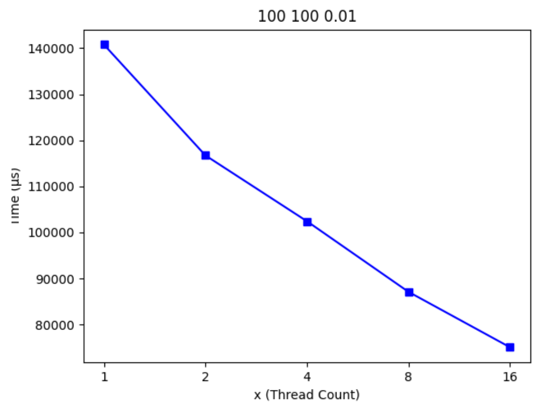

Figure 3: Results of running the code with parameters 100 100 0.01

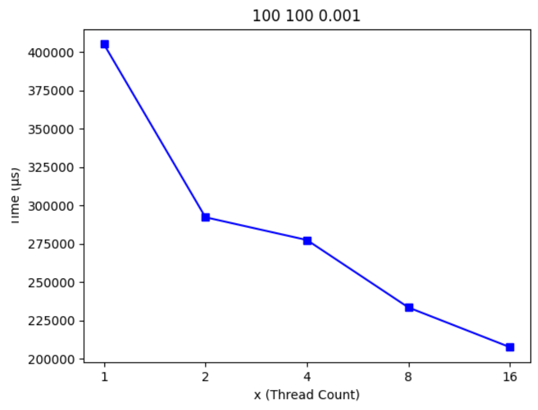

Figure 4: Results of running the code with parameters 100 100 0.001

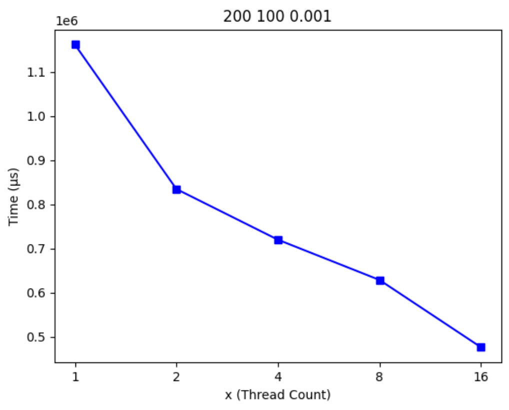

Figure 5: Results of running the code with parameters 200 100 0.001

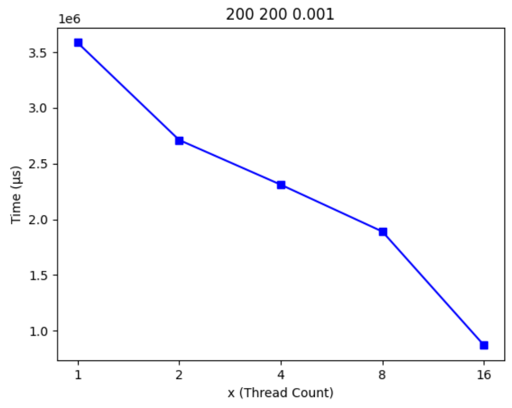

Figure 6: Results of running the code with parameters 200 200 0.001

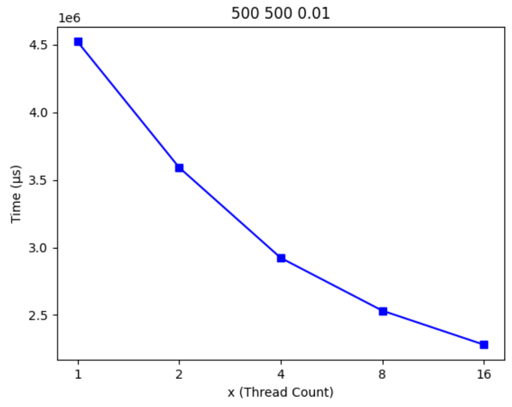

Figure 7: Results of running the code with parameters 500 500 0.01

d. With very small inputs like $n = 10, m = 10$ (Figure 1), we see that the overhead of creating new threads is too high that it causes the parallelized code to perform worse than the serial code.

   For medium inputs like $n = 50, m = 50$ (Figure 2), we see that at first, parallelization has a positive effect. Still, after increasing the number of threads from 8 to 16, the overhead of creating new threads overcomes the positive effects of parallelization.

   For $n = 100, m = 100$, we see an increase in performance by increasing the thread count. Notably, this improvement rate decreases after a while; it is evident in Figure 4. Also, by comparing Figure 3 and Figure 4, we can find out that decreasing the tolerance decreases the overall runtime of the program (as expected). This decrease in overall runtime causes the positive effect of parallelization in Figure 3 to pale compared to Figure 4.

For larger inputs like $n = 200, m = 200$ (Figure 6) and $n = 500, m = 500$ (Figure 7), we see similar results; i.e. The parallelization has a positive effect. Also, except for the last entry in Figure 6, we see the rate of this improvement decreases by increasing the number of threads. The exception for the last entry of Figure 6 is maybe caused by 200×200 being divisible by 64 and this divisibility by a large power of two caused the 16 thread version to perform better on this case.

4 Question Four

a. The code is included in `Q4.c`.

b. The code is included in `Q4_Parallel.c`.

c. You can find the raw results for runtime (in μs) in `results4.xlsx`. The plot is shown in Figure 8. We run the the code for $N = 8$ to $N = 16$. After that, the amount of time needed to execute was very high, and it was not feasible to continue. We tested each case for thread counts of $1, 4, 8, 16$. Note that the y-axis is shown in logarithmic scale (base 10).

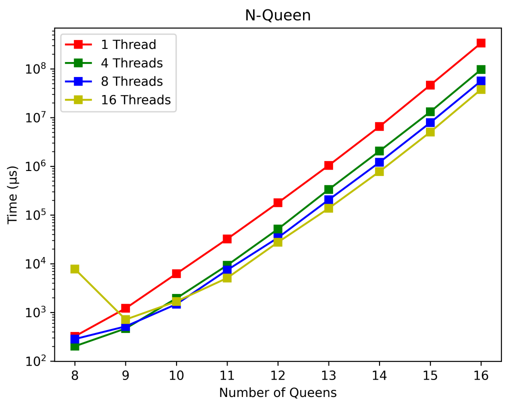

Figure 8: Results of running the N-Queen code for $N = 8$ to $N = 16$ and thread counts of $1, 4, 8, 16$

d. Yes. We got a performance boost. We used `task` construct in OpenMP. which actually puts the function specified in a queue so that each free thread picks up one task and executes it from the queue.

For the first cases like $N = 8, N = 9$, and $N = 10$, the overhead of creating 16 threads was too high, and it performed worse than the other cases; But for $N > 11$, the 16 threaded version performed better.

The linear shape of the plot on a logarithmic scale is also noteworthy. It shows that the growth rate of the running time is exponential.