CS 301
High-Performance Computing

Lab 2

Krunal Lukhi (201901449)
Dhwanil Shah (201901450)

March 3, 2022

Contents

1 Introduction

With the increase in the computational power of machines it is very important to parallelize a serial algorithm to utilize those machines in the most effective way. In this lab we would be comparing the performances of various serial and parallel algorithms visualize them graphically. For each problem we would be starting with the serial implementation, followed by the different ways to parallelize it and finally compare the algorithm on various metrics such as speedup, efficiency, execution time etc.

Speedup: It is defined as the ratio of the time taken for serial implementation and the time taken for parallel implementation of the algorithm. Mathematically it can be written as,

$$speedup = \frac{T_s}{T_p} \tag{1}$$

Efficiency: It is defined as the ratio of the speedup and the number of available cores/processors. Mathematically it can be written as,

$$efficiency = \frac{speedup}{number\_of\_cores} \tag{2}$$

2 Hardware Details

- Model - Intel(R) Core(TM) i5-4590
- CPU - 4
- Clock Speed - 3.3 GHz
- Socket - 1
- Cores per Socket - 4
- Size of L1d cache - 32K
- Size of L1i cache - 32K
- Size of L2 cache - 256K
- Size of L3 cache - 6144K

3 Calculating the value of π using trapezoidal method

3.1 Serial Implementation

For the serial implementation of the algorithm we used a for loop which calculates the area of each of the N rectangles (the region is divided into N rectangles) and finally calculates the value of π. The time complexity for the algorithm would be $O(N)$ since the for loop runs N times.

3.2 Parallel Implementation

For the parallel implementation of the algorithm we explicitly divided the total rectangles for each of the different threads and finally summed up the values obtained from each thread. Further we also added padding so that for each access we get the next 8 values as well which leads to an increase in the speedup. The theoretical time complexity for the algorithm is $O(N/P)$, where P is the number of cores/threads.
Using Eq. 1 we can calculate the theoretical speedup as,

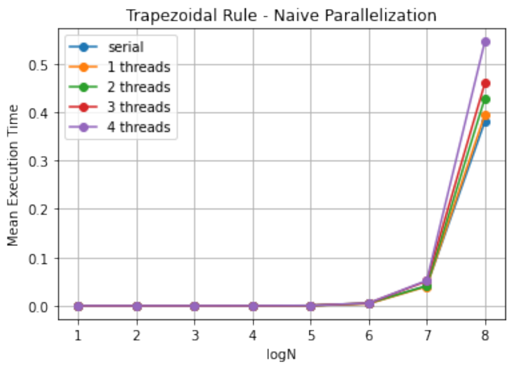
$$\text{speedup} = \frac{N}{N/P} = P$$

Hence from the above equation the theoretical value of the speedup should be equal to the number of cores used for eg. if 3 cores are used then the speedup should be 3. However as we would see in the next section, practically the speedup would be lesser then the number of cores used.

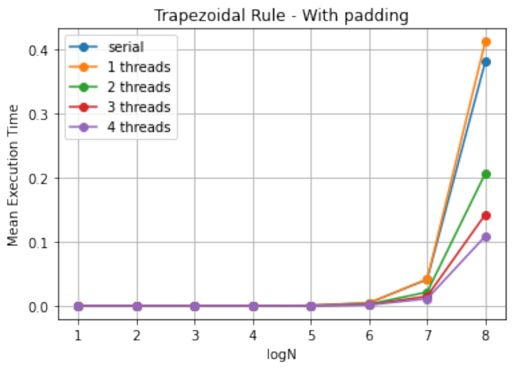
3.3 Graphical Analysis

For our experiments we have varied the values of N from 10 to 10^8 in powers of 10 and considered 4 different values of threads i.e. 1, 2, 3 and 4.

1. **Mean Execution Time:** The following figures show the mean execution time for two different variations of the parallel algorithm, the first being the naive method i.e. we explicitly dividing the work and the second being the same thing but with padding.



(a)



(b)

Figure 1: Mean execution time for with and without padding implementation

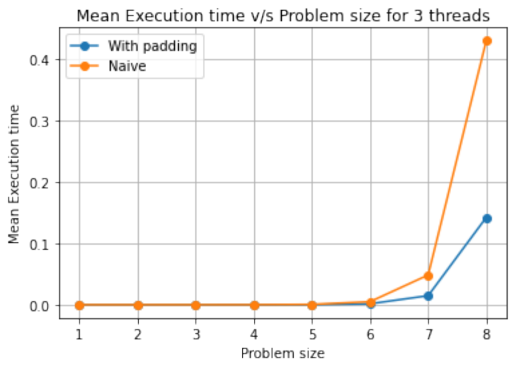
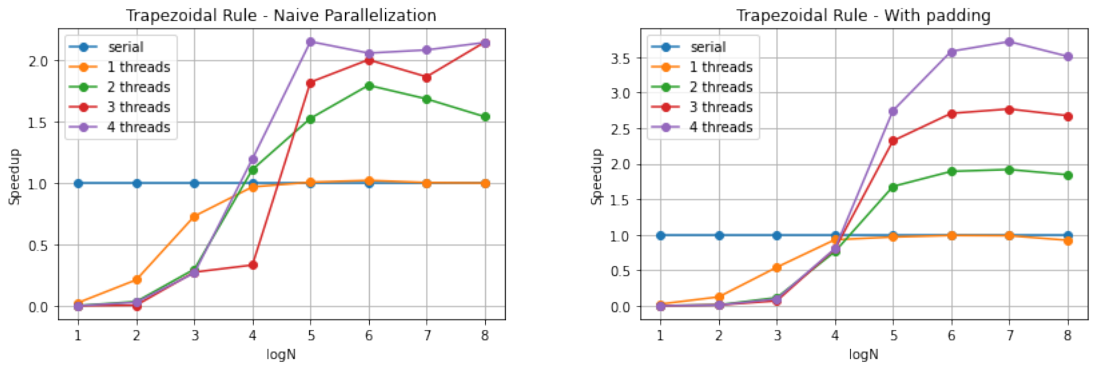


Figure 2: Mean execution time for 3 threads

We can see that the mean execution time with padding is much lesser as compared to without padding especially when number of cores used is 2, 3, 4. In fig (b) we can also see that the execution time for 1 thread is more than the serial implementation, however they mean the same thing. This is because of the overhead introduced in the parallel execution.
We also implemented the parallel algorithm using the 'for' directive which implicitly divides the work among the threads. In that case the execution time was larger as compared to the above two because of the overhead involved in dividing the work.

2. **Speedup:** The following figures show the speedup analysis for two different variations of the parallel algorithm as specified in the above subsection.



(a) (b)

Figure 3: Speedup analysis for with and without padding implementation

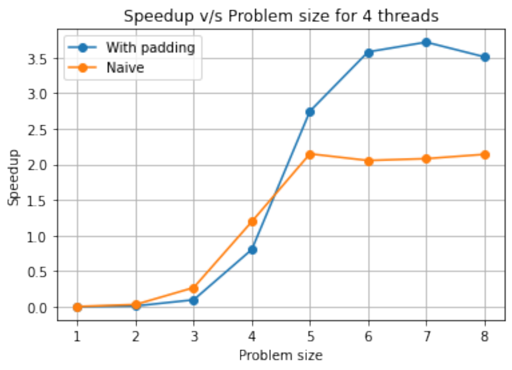


Figure 4: Speedup analysis for 4 threads

Here we can see that the speedup for the naive parallel implementation is much lesser than what should have been as per the theoretical speedup. This is because of false sharing which occurs when the cache is shared among various threads and it requires considerably large overhead leading to an inefficient algorithm. However in the case of padding the problem of false sharing is resolved and hence the speedup is just slightly lesser than the theoretical speedup.

3. **Number of cores vs Speedup:** The following figures show the speedup analysis in terms of no of cores for fixed input size.

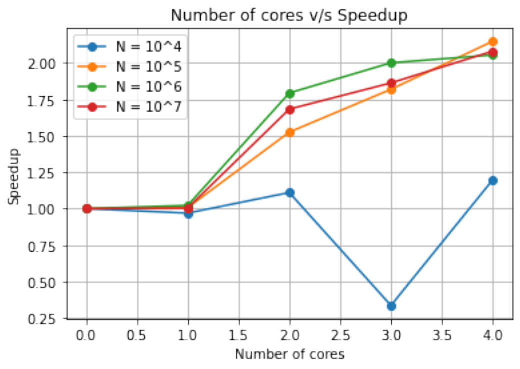


Figure 5: Number of cores vs Speedup

Here, we can see that speedup is high when no of cores are more. For lower values of input speed decreases as cores increases. This is due to the fact that overhead nullifies the speedup gained by parallelization. When input size is large, overhead becomes insignificant after some point. So, we can observe better speed up.

4 Vector addition and Vector multiplication followed by addition

4.1 Serial Implementation

In the serial implementation of vector addition we used a for loop to add the corresponding values for each of the vectors and storing them in the third vector. Similarly, for the implementation of vector multiplication followed by addition also we used a for loop which multiplies the corresponding values and adds it to a variable 'sum'. The time complexity for both the algorithms would be $O(N)$ (where N is the size of the vectors used).

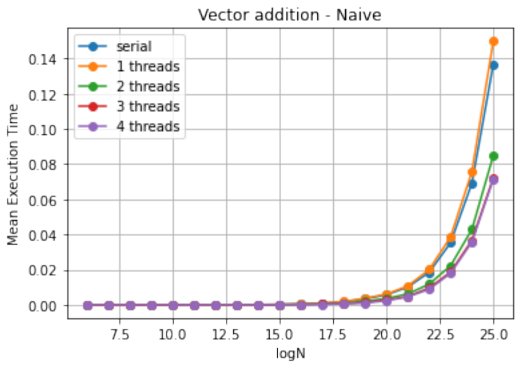
4.2 Parallel Implementation

In the naive parallel implementation we divided the vectors for each of the threads where the start and end values for a portion are given as $(N/P) * id$ and $(N/P) * (id + 1)$ respectively (where id the thread's id). Further we also used the 'for' directive which would implicitly divide the work among the threads. The theoretical complexity for the parallel implementation would be $O(N/P)$ since the problem of size N is divided among P threads.

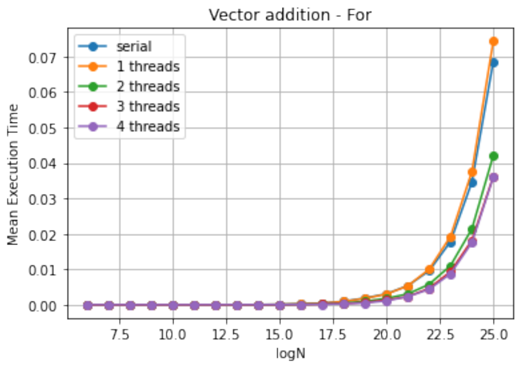
4.3 Graphical Analysis

For our experiments we have varied the values of N from 2^6 to 2^{25} in powers of 2 and considered 4 different values of threads i.e. 1, 2, 3 and 4.

1. **Mean execution time:** The following figures show the mean execution time (both vector addition and vector multiplication followed by addition) for two different variations of the parallel algorithm, the first being the naive method i.e. we explicitly dividing the work and the second being the one using the 'for' directive.



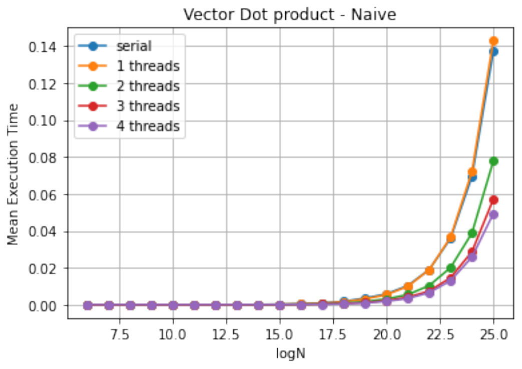
(a)



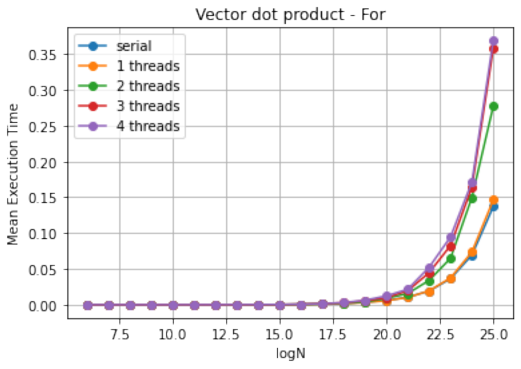
(b)

Figure 6: Mean execution time for vector addition

We can see that for the case of naive implementation the mean execution time is much larger as compared to that obtained using the 'for' directive which means that the implicit division of work is better in case of vector addition as compared to explicit division.



(a)

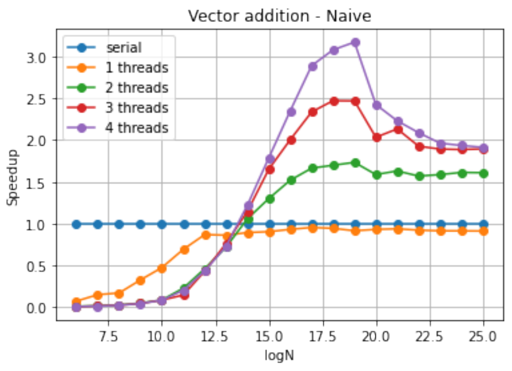


(b)

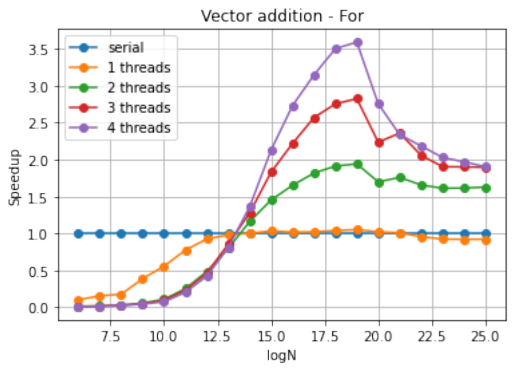
Figure 7: Mean execution time for vector multiplication followed by addition

However for vector multiplication followed by addition the mean execution time obtained when parallelizing using the 'for' directive is higher then the naive implementation. One of the reasons might be that the overhead time is increasing because of the implicit work sharing that the processor has to do.

2. **Speedup:** The following figures show the speedup analysis (both vector addition and vector multiplication followed by addition) for two different variations of the parallel algorithm as mentioned in the above section.



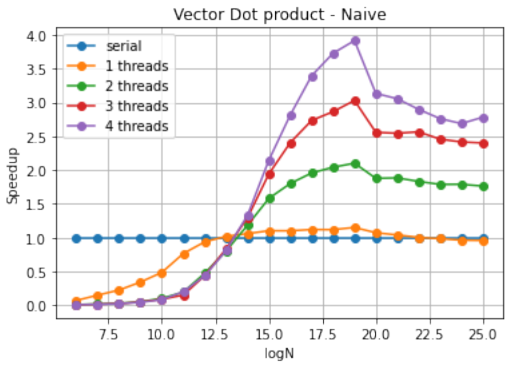
(a)



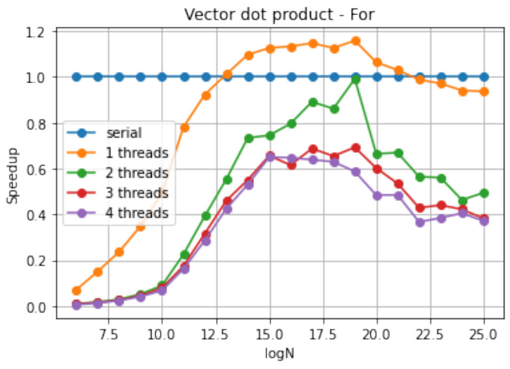
(b)

Figure 8: Speedup analysis for vector addition

Here we can see that both the variations have almost the same trend for the speedup with both being lesser than the theoretical speedup, however with the 'for' directive the speedup is a little higher as compared to the naive implementation.



(a)

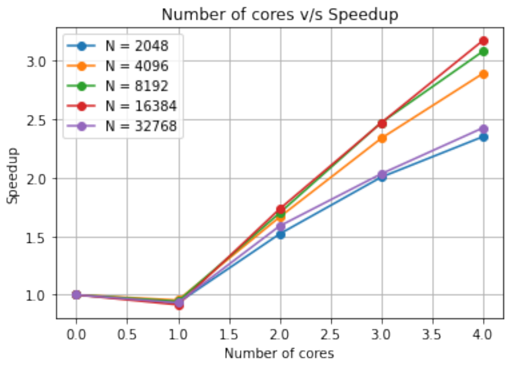


(b)

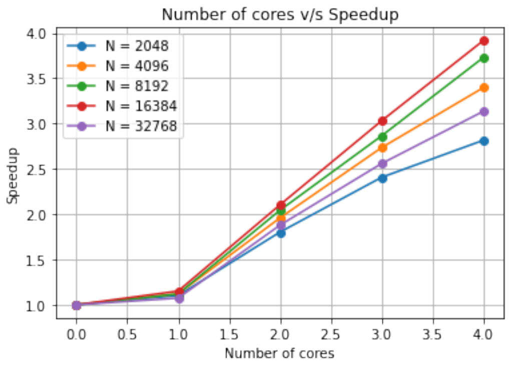
Figure 9: Speedup analysis for vector multiplication followed by addition

As was evident from the figures of mean execution time for vector multiplication followed by addition, the speedup is also less when parallelized using the 'for' directive as compared to the naive parallelization.

3. **Number of core vs Speedup:** Here, we present the graphical result of number of cores vs Speedup for parallel implementation of both vector addition and vector multiplication followed by addition. Reasons of trend is similar to reasons discussed in previous section.



(a)



(b)

Figure 10: Number of core vs Speedup analysis

5 Calculating π using random numbers

5.1 Serial Implementation

In serial implementation, we generate random points within the unit square. Then we calculate the ratio of the points that are inside the unit cycle and calculate the ratio of these points against the total. That gives us the estimated value of π. The serial implementation contains only a single for loop. Hence, time complexity of this algorithm is $O(N)$.

5.2 Parallel Implementation

The serial implementation is simple. It consists only one for loop. So, it allows us to easily parallelize serial code using reduction technique. The only problem is library function *rand()* used to generate random numbers. This function thread unsafe. We can not use it in parallel implementation. In addition to that, It might provide duplicate values for large size of input. This may affect Monte Carlo simulations results. More randomness leads to better results. So, we have used different types of random generator function in parallel implementations.

5.3 Graphical Analysis

For our experiments we have varied the values of N from 10 to 10^8 in powers of 10 and considered 4 different values of threads i.e. 1, 2, 3 and 4.

1. **Mean execution time:** Following figure shows mean execution time of parallel implementation of the algorithm.

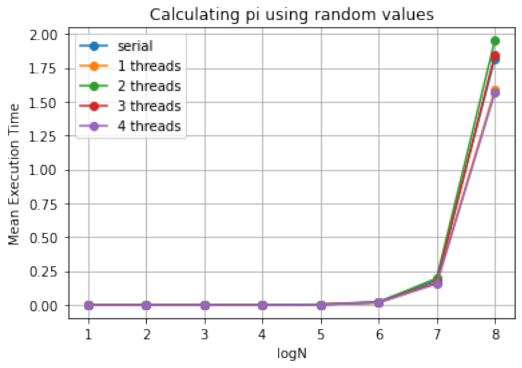


Figure 11: Mean execution time

We see obvious trend in mean execution time graph. Parallel implementation runs faster compared to serial implementation. For low values of input, speedup obtained by parallelization is neutralized by overhead. When input is larger we can see clear distinction in mean execution time as core increases.

2. **Speedup:** Following figure represents the analysis of speedup obtained by parallelization.

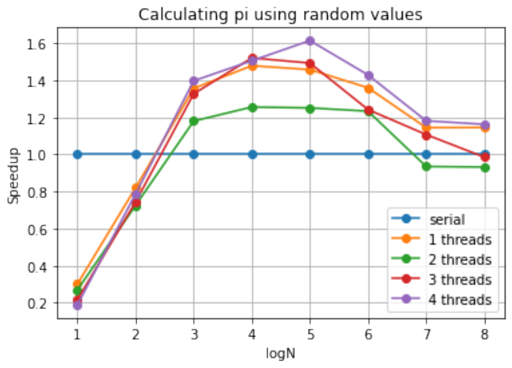


Figure 12: Speedup

Trends of speedup follows the similar reasons as we discussed in previous sections.

3. **Number of core vs Speedup:** Here we have presented the graph of speed vs number of cores. We can similar trends as observed in previous sections. Reasons also similar as discussed in previous sections.

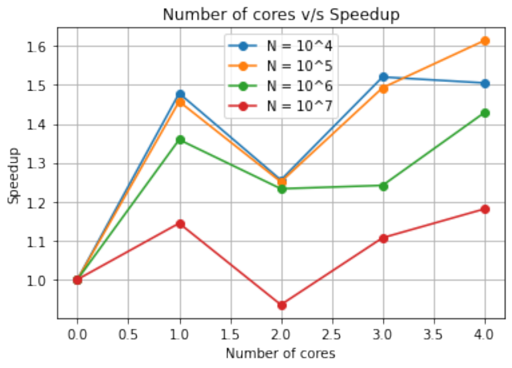


Figure 13: number of core vs Speedup

6 Conclusion

In this lab we analyzed various different algorithms using their serial and parallel implementation and compared them based on metrics such as mean execution time, speedup etc. We found that parallelization can make things better as well as worse based on the various ways in which one parallelizes the algorithm. For eg. sometimes the parallelization done using the 'for' directive can be worse than the naive parallelization because of the overhead involved. Hence, it is necessary to correctly identify the dependencies, other problems (such as that of false sharing) etc for efficient and accurate parallelization.