CS 301 : HIGH PERFORMANCE COMPUTING

LAB2 - OPENMP

PURVIL MEHTA (201701073)
BHARGEY MEHTA (201701074)

Dhirubhai Ambani Institute of Information and Communication Technology
Gandhinagar

FEBRUARY 17, 2020

Contents

1 Hardware Details

- CPU model - Intel® Core™ i5-4590 CPU @ 3.30GHz
- Socket(s) - 1
- Core(s) per socket - 4
- Thread(s) per core - 1
- Cache sizes - L1: 32K, L2: 256K and L3: 6144K

2 Basic Vector Operations using Critical

2.1 Implementation Details

2.1.1 Brief and clear description about the Serial implementation

The components are looped over serially and added to the final result. The for loops runs N times

2.1.2 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

The problem has been parallelised by distributing the subspaces of the vector space amongst the threads. The total of each subspace is then added to the final result using the manual critical pragma.

2.2 Complexity and Analysis Related

2.2.1 Complexity of serial code

The serial code runs over the for loop and each iteration takes 1 addition and 1 multiplication operation. Serial code time complexity is $O(N)$.

2.2.2 Complexity of parallel code (split as needed into work, step, etc.)

The complexity of the parallel algorithm is $O(\frac{N}{p})$.

2.2.3 Cost of Parallel Algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N}{p}) = O(N)$.

2.2.4 Theoretical Speedup (using asymptotic analysis, etc.)

Since the entire code is parallelisable, the theoretical speedup would be $\frac{1}{p}$.

2.3 Curve Based Analysis

2.3.1 Time Curve related analysis (as problem size increases, also for serial)

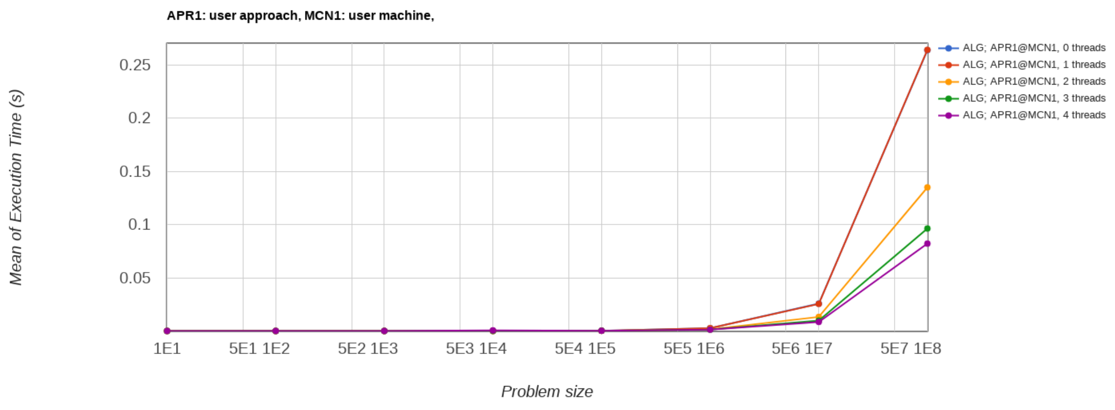


Figure 1: Mean Execution Time

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size.Hence any time saved by parallelising the problem is then offset by the computations regarding parallelisation. For higher problem sizes, the time saved is comparatively more so as number of threads increases, the execution time decreases

2.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

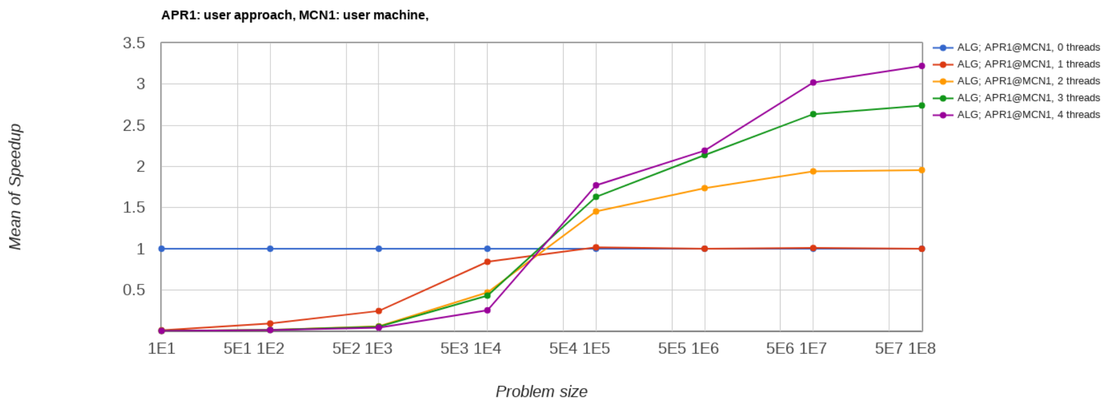


Figure 2: Mean Execution Time

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size.Any speedup achieved is then offset by the overhead of parallelisation. The speedup is more noticeable for higher problem sizes since the speedup is more as compared to the overhead of parallelisation.

2.3.3 Efficiency Curve related analysis

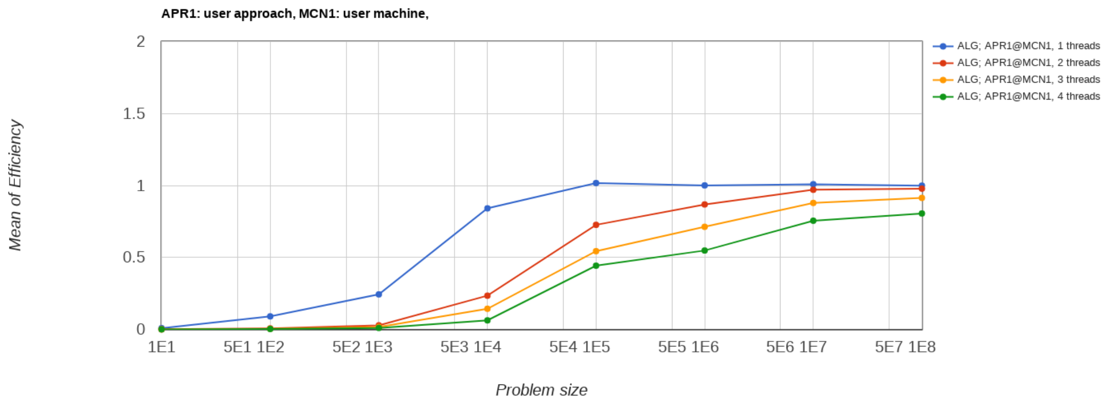


Figure 3: Mean Execution Time

In serial code, the processor is only performing the computations regarding the problem. As we increase the number of threads, each processor also has to perform overhead calculations for synchronisation. So time spent in computation decreases and so efficiency decreases. This behaviour is same as the previous approach.

3 Basic Vector Operations using Reduction

3.1 Implementation Details

3.1.1 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

The problem has been parallelised by distributing the subspaces of the vector space amongst the threads. The total of each subspace is then added to the final result using the manual critical pragma.

3.2 Complexity and Analysis Related

3.2.1 Complexity of parallel code (split as needed into work, step, etc.)

Assuming we have p processors. We parallelise the task by assigning the subarray of size $\frac{N}{p}$ to each processor. Next we add these partial results in a logarithmic fashion which results in an additional $O(\log p)$ computations. For small p this can be ignored. So again we have a complexity of $O(\frac{N}{p})$.

3.2.2 Cost of Parallel Algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N}{p}) = O(N)$.

3.2.3 Theoretical Speedup (using asymptotic analysis, etc.)

Since the entire code is parallelisable, the theoretical speedup would be $\frac{1}{p}$.

3.3 Curve Based Analysis

3.3.1 Time Curve related analysis (as problem size increases, also for serial)

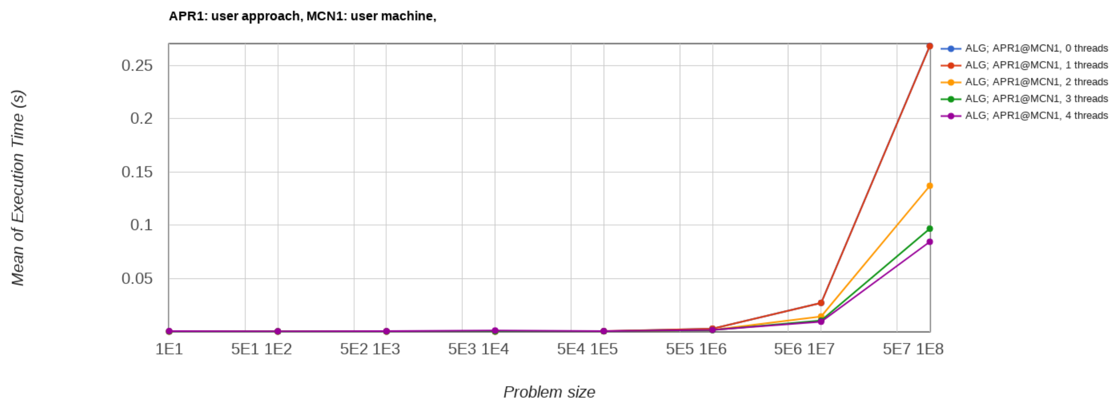


Figure 4: Mean Execution Time

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size.Hence any time saved by parallelising the problem is then offset by the computations regarding parallelisation. For higher problem sizes, the time saved is comparatively more so as number of threads increases, the execution time decreases. This behaviour is the same as the previous approach.

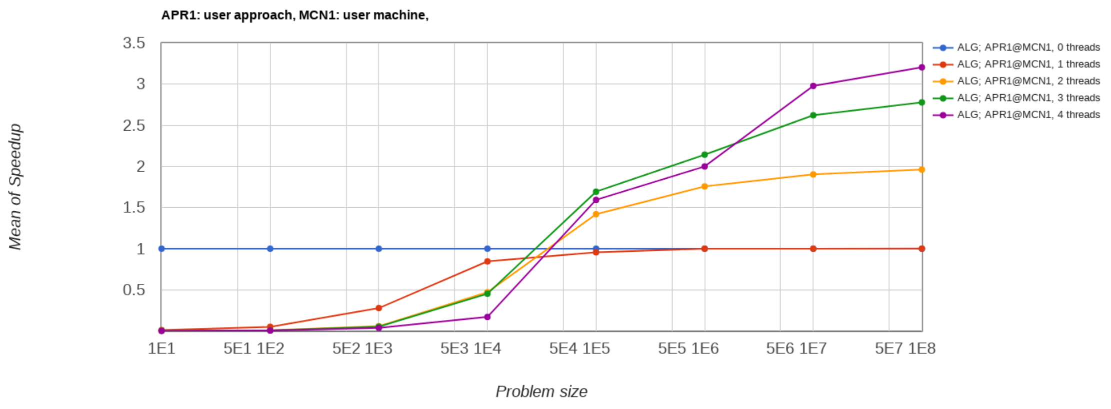


Figure 5: Mean Execution Time

3.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size.Any speedup achieved is then offset by the overhead of parallelisation. The speedup is more noticeable for higher problem sizes since the speedup is more as compared to the overhead of parallelisation. This behaviour is the same as the previous approach.

3.3.3 Efficiency Curve related analysis

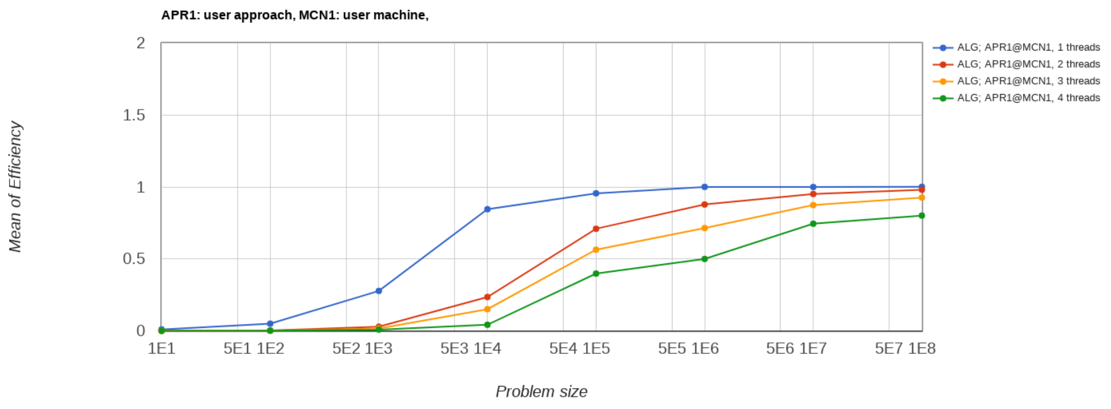


Figure 6: Mean Execution Time

In serial code, the processor is only performing the computations regarding the problem. As we increase the number of threads, each processor also has to perform overhead calculations for synchronisation. So time spent in computation decreases and so efficiency decreases. This behaviour is same as the previous approach. This behaviour is the same as the previous approach.

4 Trapezoidal Technique using Critical

4.1 Implementation Details

4.1.1 Brief and clear description about the Serial implementation

This technique is used to perform numerical integration. We divide the interval into n parts and approximate each strip as a rectangular bar. The area calculated in this way would be area = height × width = $f(x)\mathrm{d}x$. The products are calculated and added to the final result serially.

4.1.2 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

Area of each strip can be calculated independently of each other. Then the results can be added to get the final result. One possible parallelization would be to divide the points among the processors so that each processor can calculate a portion of the final result. Finally all the p portions can be serially added to get the final result.

4.2 Complexity and Analysis Related

4.2.1 Complexity of parallel code (split as needed into work, step, etc.)

Assuming we have p processors. We parallelise the task by assigning some portion of the rectangular strips to each processor. Thus in effect proc-A calculates some subset of the area to be calculated. The final result is obtained by adding these to a final sum. This has to be done serially so an additional $O(p)$ operations. Hence the final complexity becomes $O(\frac{N}{p}+p)$. Assuming $p << N$, this becomes $O(\frac{N}{p})$.

4.2.2 Theoretical Speedup (using asymptotic analysis, etc.)

The theoretical speedup is $\frac{1}{p}$ where p is the number of processors. This happens because the serial part is order of p and it is very small compared to the problem size.

4.2.3 Cost of parallel algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N}{p}) = O(N)$.

4.3 Curve Based Analysis

4.3.1 Time Curve related analysis (as problem size increases, also for serial)

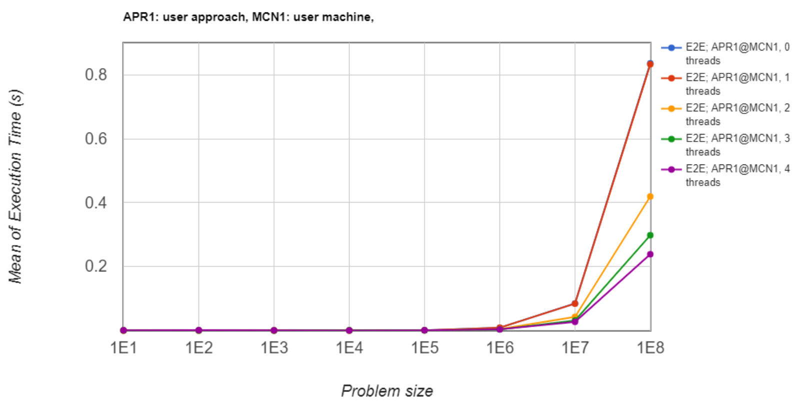


Figure 7: Mean Execution Time

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size. Hence any time saved by parallelising the problem is then offset by the computations regarding parallelisation. For higher problem sizes, the time saved is comparatively more so as number of threads increases, the execution time decreases.

4.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

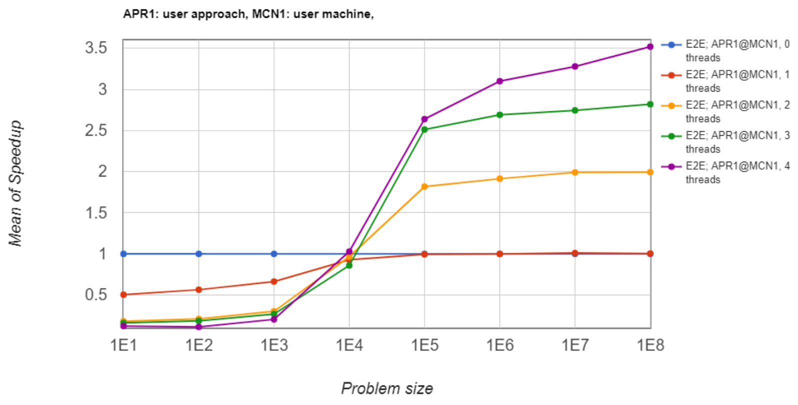


Figure 8: Mean Speedup

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size. Any speedup achieved is then offset by the overhead of parallelisation. The speedup is more noticeable for higher problem sizes since the speedup is more as compared to the overhead of parallelisation.

4.3.3 Efficiency Curve related analysis

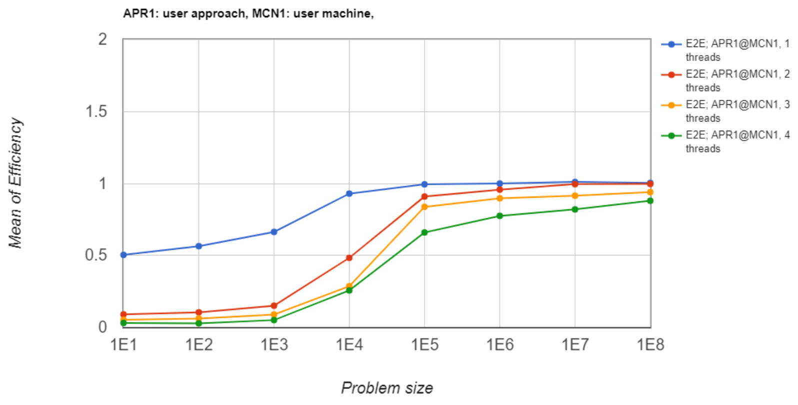


Figure 9: Mean Efficiency

In serial code, the processor is only performing the computations regarding the problem. As we increase the number of threads, each processor also has to perform overhead calculatations for synchronisation. So time spent in computation decreases and so efficiency decreases.

5 Trapezoidal Technique using Reduction

5.1 Implementation Details

5.1.1 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

In the previous approach, we were parallelising some subarray to be calculated and then serially add the results. In this approach, the result of each subarray would also be calculated parallely. This results in a logarithmic combination of partial results as compared to the serial in the previous approach.

5.2 Complexity and Analysis Related

5.2.1 Complexity of parallel code (split as needed into work, step, etc.)

Assuming we have p processors. We parallelise the task by assigning the subarray of size $\frac{N}{p}$ to each processor. Next we add these partial results in a logarithmic fashion which results in an additional $O(\log p)$ computations. For small p this can be ignored. So again we have a complexity of $O(\frac{N}{p})$.

5.2.2 Theoretical Speedup (using asymptotic analysis, etc.)

The theoretical speedup is $\frac{1}{p}$ where p is the number of processors. This happens because the serial part is order of p and it is very small compared to the problem size.

5.2.3 Cost of parallel algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N}{p}) = O(N)$.

5.3 Curve Based Analysis

5.3.1 Time Curve related analysis (as problem size increases, also for serial)

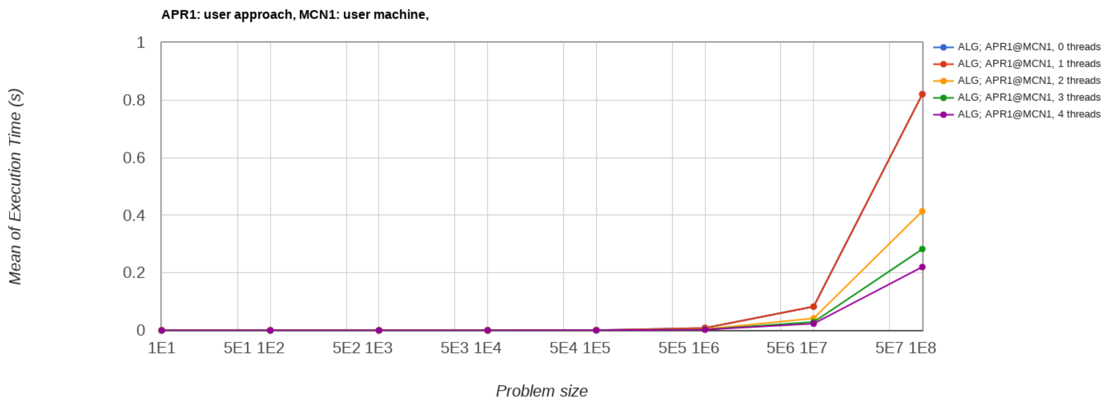


Figure 10: Mean Execution Time

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size. Hence any time saved by parallelising the problem is then offset by the computations regarding parallelisation. For higher problem sizes, the time saved is comparatively more so as number of threads increases, the execution time decreases. This behaviour is same as the previous approach.

5.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

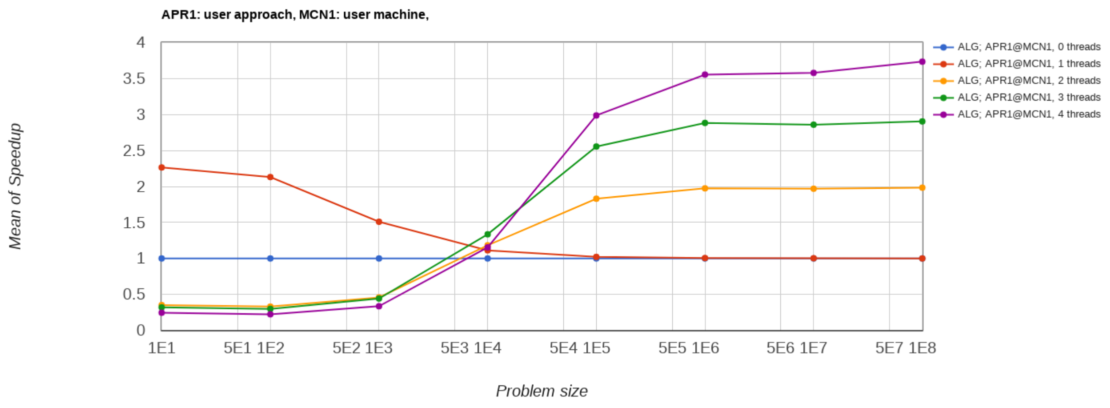


Figure 11: Mean Speedup

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size. Any speedup achieved is then offset by the overhead of parallelisation. The speedup is more noticeable for higher problem sizes since the speedup is more as compared to the overhead of parallelisation. This behaviour is same as the previous approach.

5.3.3 Efficiency Curve related analysis

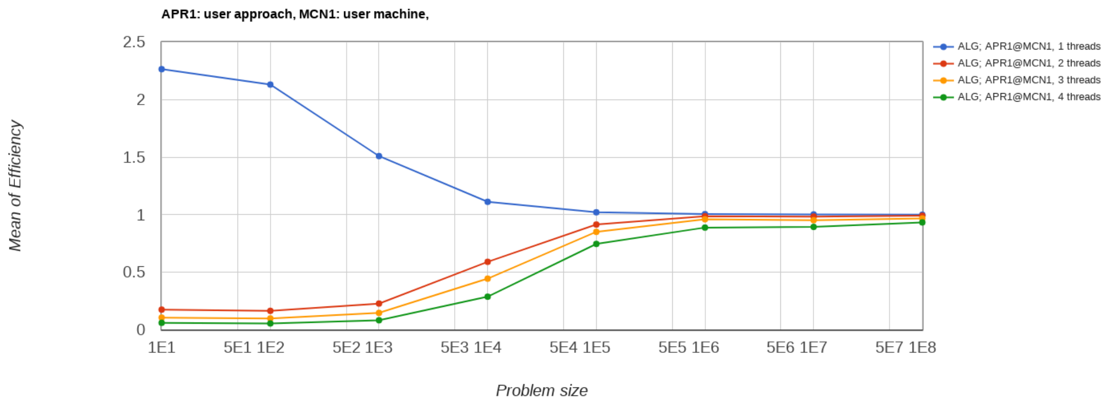


Figure 12: Mean Efficiency

In serial code, the processor is only performing the computations regarding the problem. As we increase the number of threads, each processor also has to perform overhead calculatations for synchronisation. So time spent in computation decreases and so efficiency decreases. This behaviour is same as the previous approach.

6 Calculation of π using critical

6.1 Implementation Details

6.1.1 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

This approach just uses the previous code but with a sqrt routine added because of the mathematical expression.

6.2 Complexity and Analysis Related

6.2.1 Complexity of parallel code (split as needed into work, step, etc.)

Assuming we have p processors. We parallelise the task by assigning the subarray of size $\frac{N}{p}$ to each processor. Next we add these partial results serially which takes $O(p)$ computations. For small ps this can be ignored. So again we have a complexity of $O(\frac{N}{p})$.

6.2.2 Theoretical Speedup (using asymptotic analysis, etc.)

The theoretical speedup is $\frac{1}{p}$ where p is the number of processors. This happens because the serial part is order of p and it is very small compared to the problem size.

6.2.3 Cost of parallel algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N}{p}) = O(N)$.

6.3 Curve Based Analysis

6.3.1 Time Curve related analysis (as problem size increases, also for serial)

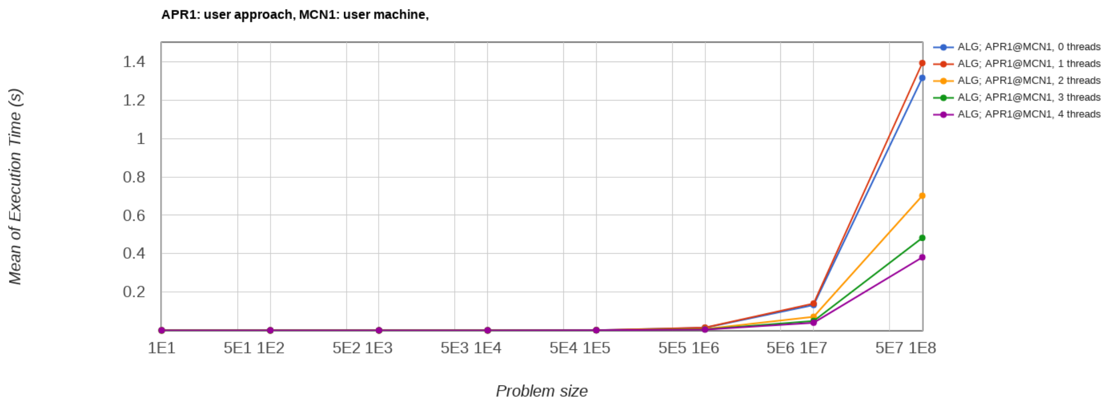


Figure 13: Mean Execution Time

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size. Hence any time saved by parallelising the problem is then offset by the computations regarding parallelisation. For higher problem sizes, the time saved is comparatively more so as number of threads increases, the execution time decreases.

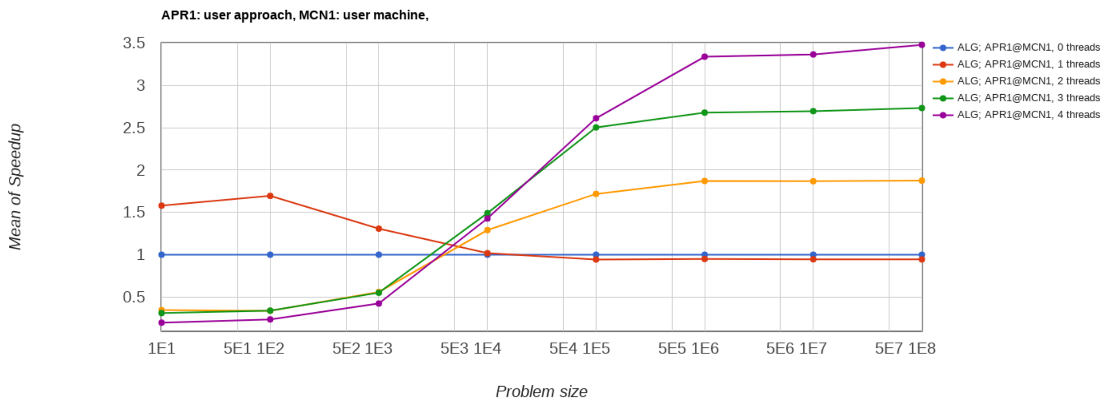


Figure 14: Mean Speedup

6.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size. Any speedup achieved is then offset by the overhead of parallelisation. The speedup is more noticeable for higher problem sizes since the speedup is more as compared to the overhead of parallelisation. This behaviour is same as the previous approach.

6.3.3 Efficiency Curve related analysis

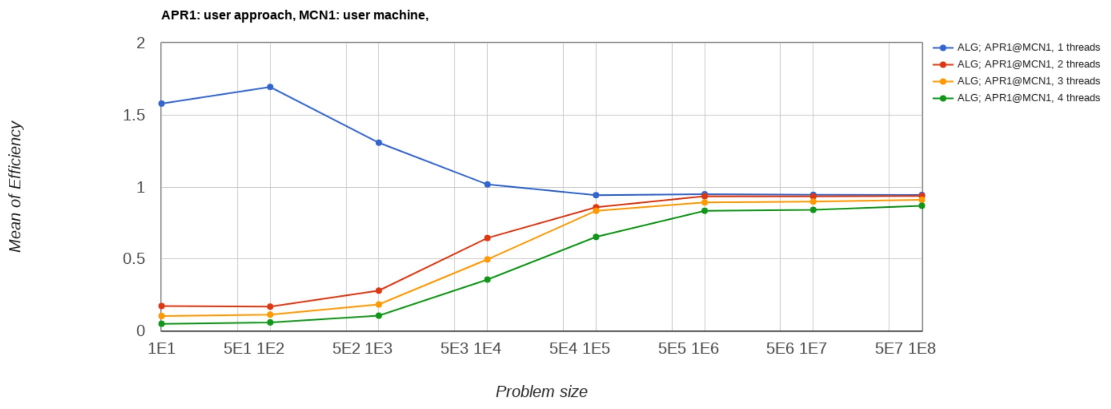


Figure 15: Mean Efficiency

In serial code, the processor is only performing the computations regarding the problem. As we increase the number of threads, each processor also has to perform overhead calculatations for synchronisation. So time spent in computation decreases and so efficiency decreases.

7 Calculation of π using Reduction

7.1 Implementation Details

7.1.1 Brief and clear description about the implementation of the approach (Parallelization Strategy, Mapping of computation to threads)

This approach uses the previous idea but with a reduction applied instead of a manual use of the critical pragma.

7.2 Complexity and Analysis Related

7.2.1 Complexity of parallel code (split as needed into work, step, etc.)

Assuming we have p processors. We parallelise the task by assigning the subarray of size $\frac{N}{p}$ to each processor. Next we add these partial results in a logarithmic fashion which results in an additional $O(\log p)$ computations. For small p this can be ignored. So again we have a complexity of $O(\frac{N}{p})$.

7.2.2 Theoretical Speedup (using asymptotic analysis, etc.)

The theoretical speedup is $\frac{1}{p}$ where p is the number of processors. This happens because the serial part is order of p and it is very small compared to the problem size.

7.2.3 Cost of parallel algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N}{p}) = O(N)$.

7.3 Curve Based Analysis

7.3.1 Time Curve related analysis (as problem size increases, also for serial)

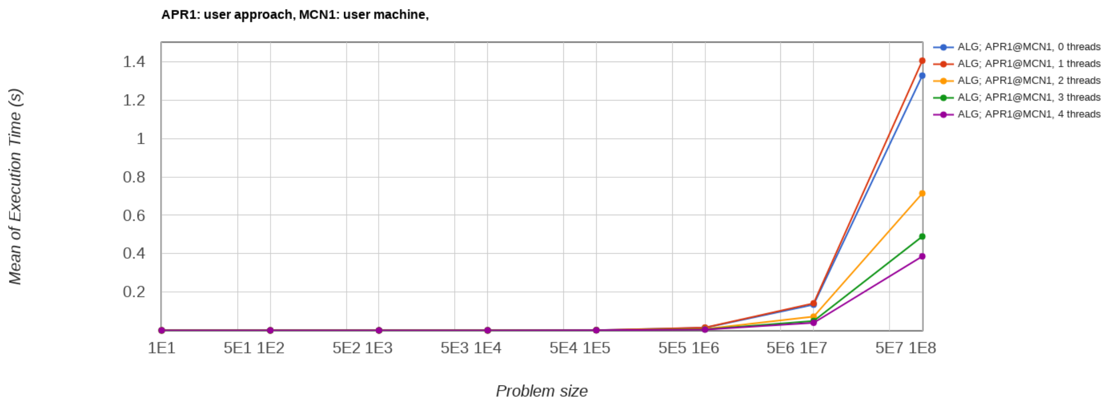


Figure 16: Mean Execution Time

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size. Hence any time saved by parallelising the problem is then offset by the computations regarding parallelisation. For higher problem sizes, the time saved is comparatively more so as number of threads increases, the execution time decreases. This behaviour is same as the previous approach.

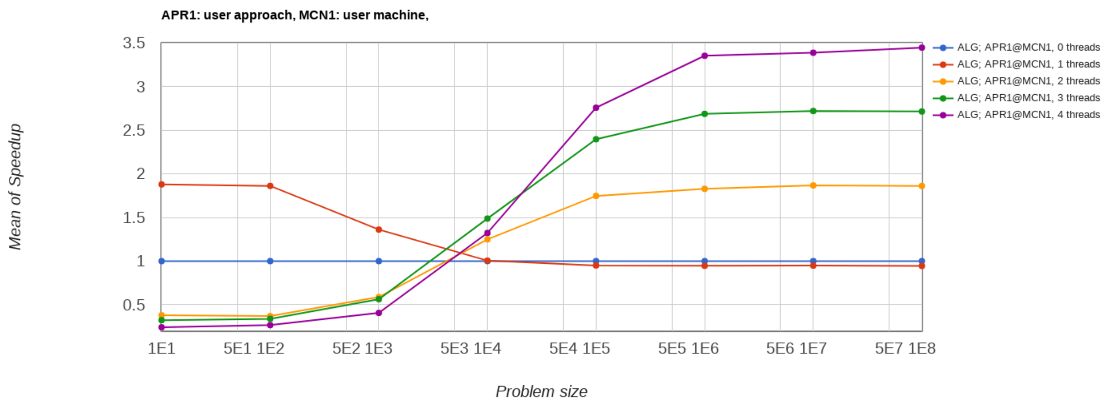


Figure 17: Mean Speedup

7.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

For smaller problem sizes the cost of parallelisation is very high as compared to the actual problem size. Any speedup achieved is then offset by the overhead of parallelisation. The speedup is more noticeable for higher problem sizes since the speedup is more as compared to the overhead of parallelisation. This behaviour is same as the previous approach.

7.3.3 Efficiency Curve related analysis

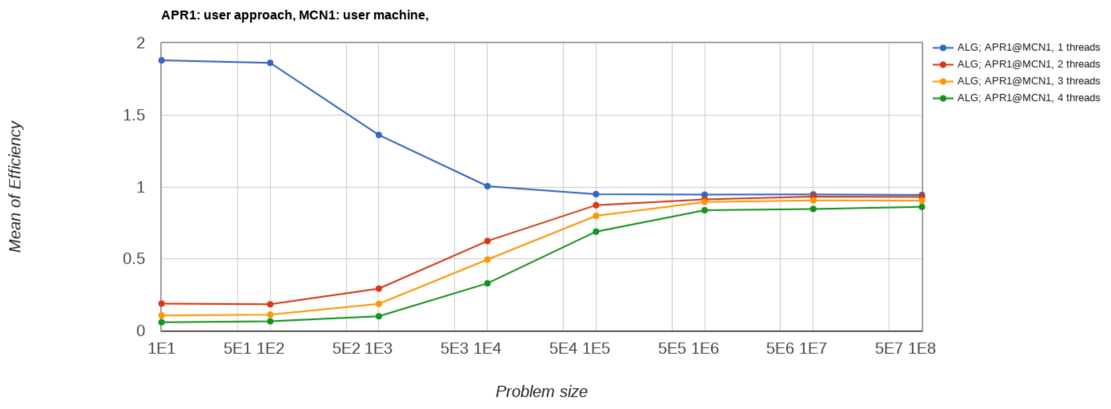


Figure 18: Mean Efficiency

In serial code, the processor is only performing the computations regarding the problem. As we increase the number of threads, each processor also has to perform overhead calculatations for synchronisation. So time spent in computation decreases and so efficiency decreases. This behaviour is same as the previous approach.

8 QR Decomposition

8.1 Implementation Details

8.1.1 Brief and clear description about the Serial implementation

There are just 3 nested loops, each iterating over N. The time complexity is $O(N^3)$.

8.1.2 Brief and clear description about the implementation of the approach (Parallelisation Strategy, Mapping of computation to threads)

There are 3 loops of i, j and k. The first loop has an internal dependency so it can't be parallelised. The internal loops operate on the columns of the matrices and are independent so can be easily parallelised using reduction method.

8.2 Complexity and Analysis Related

8.2.1 Complexity of parallel code (split as needed into work, step, etc.)

The outer loop can't be parallelised. For each run of the outer loop, the internal loops operate on the columns and are independent. The first and second loops of j are parallelised using the reduction method. The nested loops inside the main loop can also be nested parallelised. Since the number of processors are just 4, creating 4 threads inside each of these 4 threads results in a total of 16 threads. We parallelise just the inner loop in these nested loops to achieve true parallelism. The time complexity would be $O(\frac{N^3}{p})$.

8.2.2 Theoretical Speedup (using asymptotic analysis, etc.)

The theoretical speedup is $\frac{1}{p}$ where p is the number of processors. This happens because the entire code is parallelisable so theoretical speedup would be $O(p)$.

8.2.3 Cost of parallel algorithm

Since the entire code is parallelisable, the total parallel cost would be $p \times O(\frac{N^3}{p}) = O(N^3)$.

8.3 Curve Based Analysis

8.3.1 Time Curve related analysis (as problem size increases, also for serial)

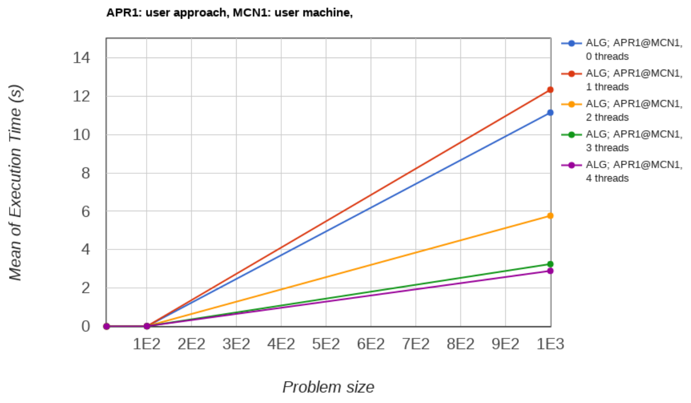


Figure 19: Mean Execution Time

We can see that the execution time decreases as number of threads increases. This happens because the parallelisation overhead is very low. We also note that the execution time of a single thread parallelisation approach is more than the serial approach since it is basically the serial code getting executed plus the overhead.

8.3.2 Speedup Curve related analysis (as problem size and no. of processors increase)

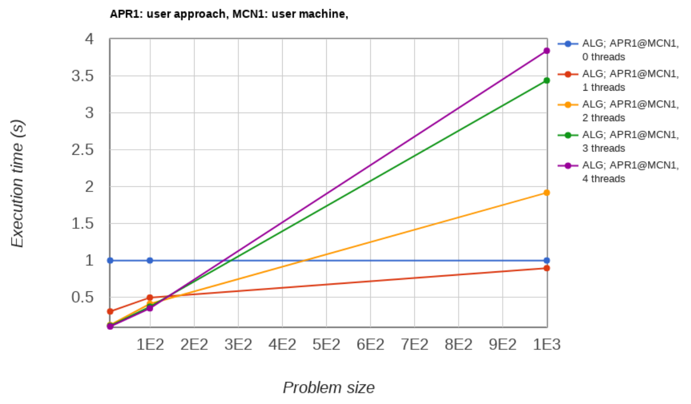


Figure 20: Mean Speedup

We can see that the speedup approaches the theoretical speedup since the entire code is parallelisable. We get a speedup of 3.8 for the case of 4 processors.

8.3.3 Efficiency Curve related analysis

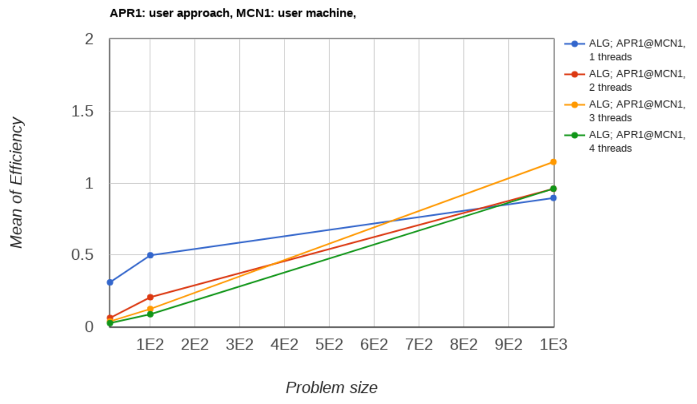
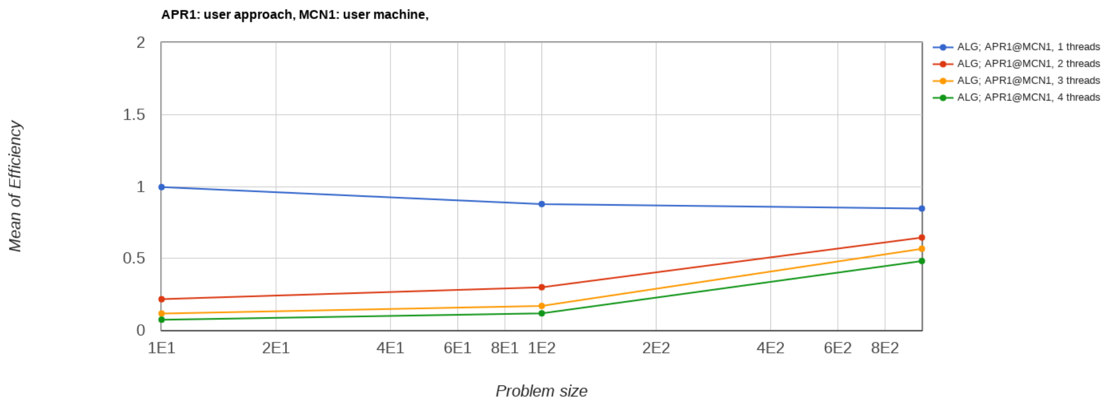


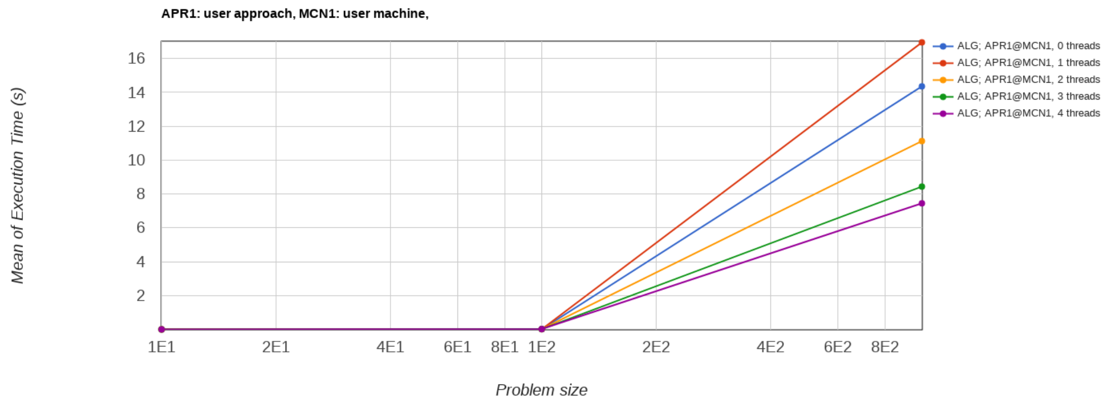
Figure 21: Mean Efficiency

The efficiency is less for smaller problem sizes since the parallelisation overhead is more compared to the actual computations for the problem. As it increases, the overhead remains same since it is proportional to the number of processors but the problem size increases and the overhead becomes insignificant leading to an efficiency close to 1 for higher problem sizes.

8.4 Scheduling Strategies (Dynamic Vs Static)

8.4.1 Results





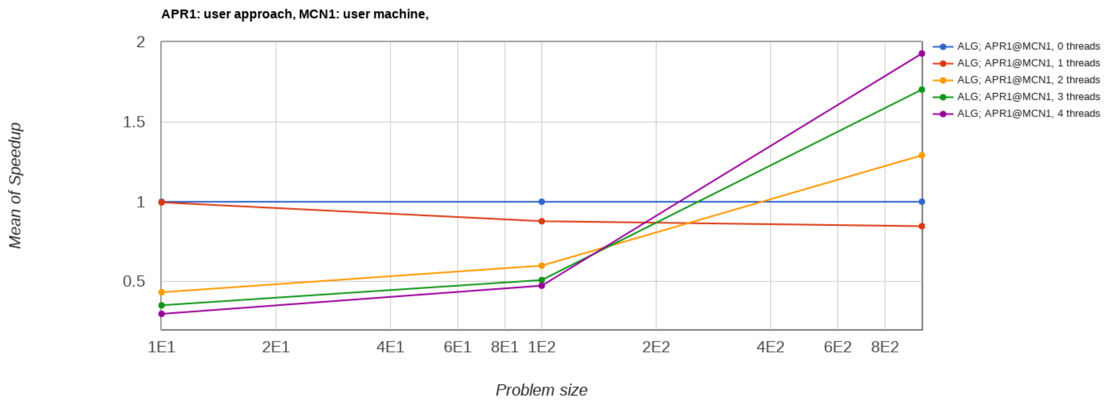
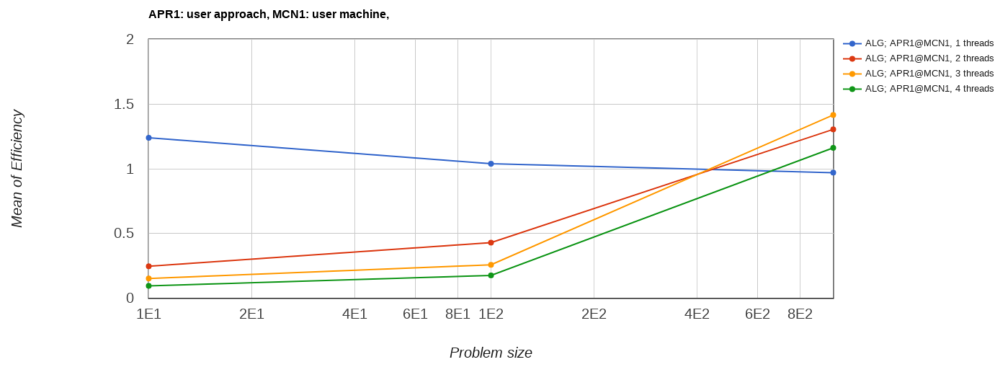
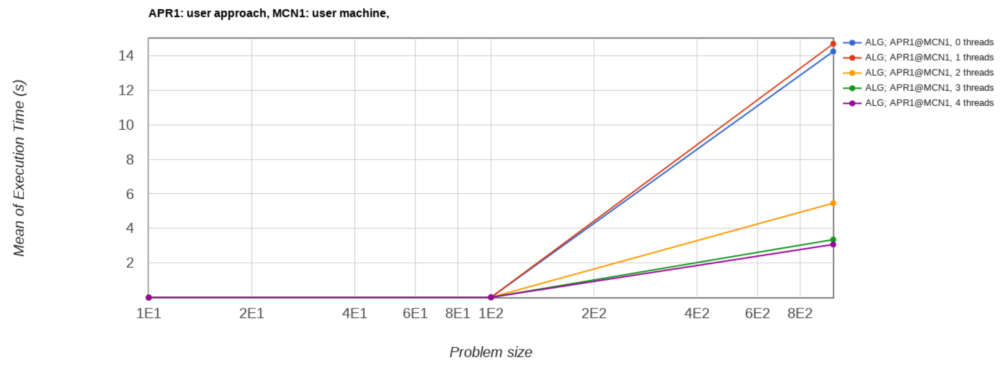


Figure 22: Performance Metrics for Dynamic Scheduling





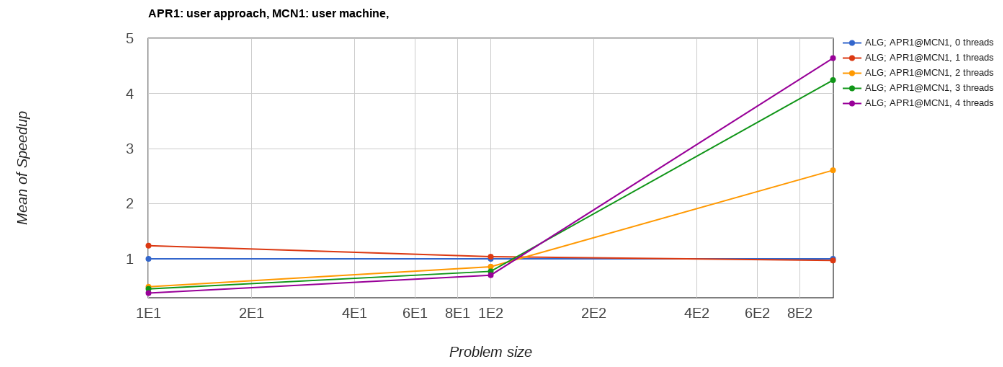


Figure 23: Performance Metrics for Static Scheduling

8.4.2 Explanation

The dynamic scheduling causes the scheduling to happen at runtime. This adds a higher overhead to the code as compared to the static scheduling where each iteration is assigned to a thread. Dynamic scheduling works better when the each iteration is of variable cost so that one thread doesn't have to do more computation as compared to the other threads. But in our case each iteration is already balanced in the sense that all iterations perform equal number of computations so there's no benefit of dynamic scheduling while there's an overhead that is incurred. Hence this overhead results in a decrease in performance.

8.5 Granularity (Coarse Vs Fine)

8.5.1 Results

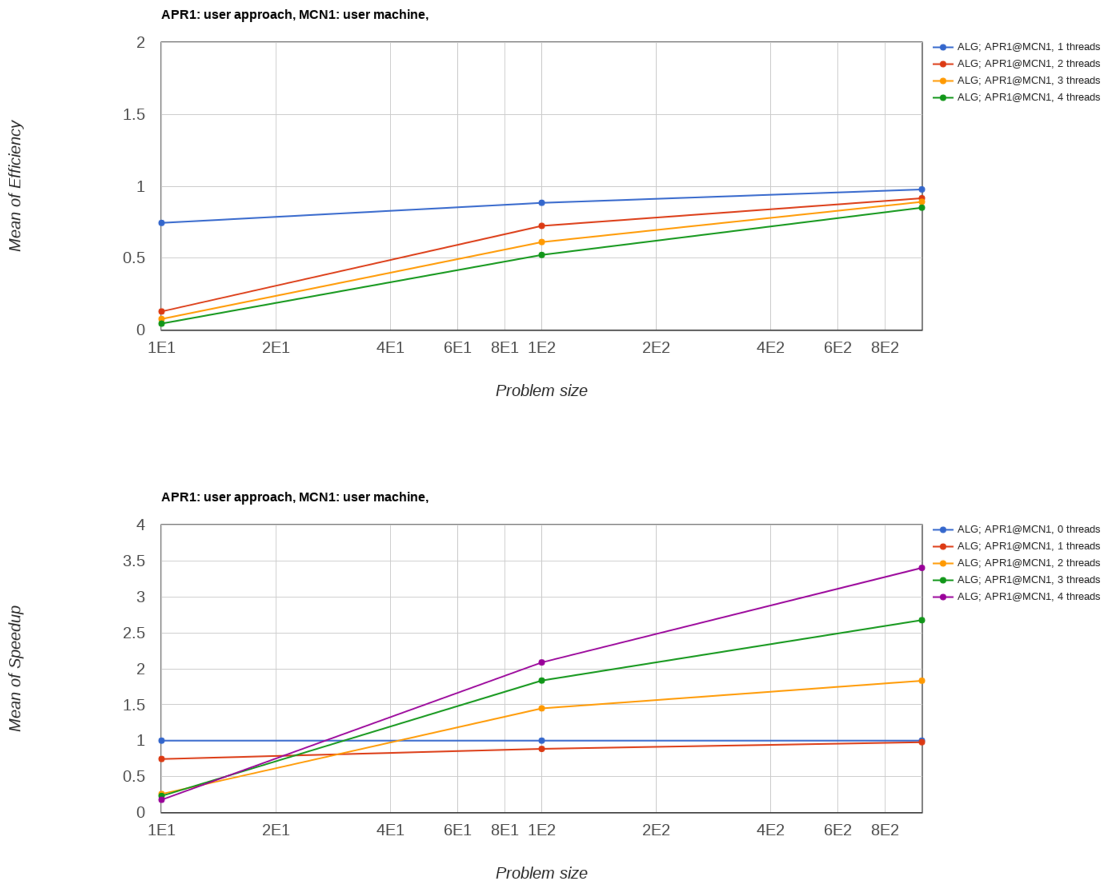


Figure 24: Performance Metrics for Coarse Grained

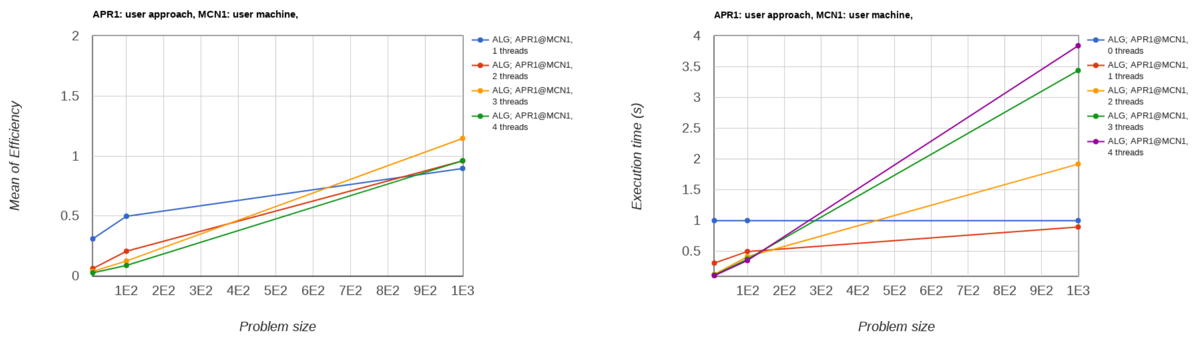


Figure 25: Performance Metrics for Fine Grained

8.5.2 Explanation

The coarse grained parallelisation results in lower speedup since the parallelisation achieved is effectively less as compared to the fine grained case. In the case of coarse grained, it might happen that a thread remains idle because some other thread is still performing it's operation. Now because the work divided is large, the thread might have to wait for say some kN^2 iterations. But in the case of fine grained parallelisation, this wait is lower i.e kN only. This results in more overlap of the cores performing computation and gives rise to a higher speedup. The fine grained case also incurs a comparatively higher overhead but in this particular case, the benefit achieved by remaining less idle is more than the loss of the increased overhead.