Matrix Multiplication (outermost)

Nishi Doshi
Roshani

Dhirubhai Ambani Institute of Information and Communication Technology
201601408@daiict.ac.in
201601059@daiict.ac.in

1 Implementation Details

1(a) Brief and clear description about the Serial implementation

Serial implementation of matrix multiplication involves running 3 for loops for calculating the value at certain cell in matrix. The loops run on same thread.

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

In parallel implementation, we have parallelized the outermost loop which runs for iterating through rows. We have divided rows into groups of

$$\frac{number\_of\_rows}{number\_of\_processors}$$

that is $\frac{n}{p}$ sizes with adding n%p left rows to the last processor involved in running the parallel code. The inner loops run the similar way as they run in serial code.

2 Complexity and Analysis Related

2(a) Complexity of serial code

The serial code contains 3 for loops and hence has $O(n^3)$ time complexity.

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

The parallel code has inner two loops running the similar way as in serial and hence run for $O(n^2)$ time complexity. The outer loop gets its time divided into the available processors and hence overall time complexity turns out to be $O(n^2 * (\frac{n}{p}))$ which hence proves the overall time complexity of code to be $O(\frac{n^3}{p})$

2(c) Theoretical Speedup (using asymptotic analysis, etc.)

Theoretical speedup should in this case should reach equal to the number of processors available. The reason for this is that the code includes no idle time as well as no overhead and hence direct parallelization of problem happens. Thus as per the number of processors available the speedup should change accordingly. Calculation : $\frac{T_{serial}}{T_{parallel}}$ which is equal to $\frac{n^3}{\frac{n^3}{p}} = p$

2(d) Experimental Serial Fraction

The serial fraction is given by $Serial\ fraction = \frac{(end\ to\ end\ time - algo\ time)}{end\ to\ end\ time}$. Experimentally, for different problem sizes we find different values of serial fraction as shown below :

Problem Size	Serial Fraction
16	0.0002222
32	0.0015897
64	0.0129348
128	0.1030589
512	10.4093519
1024	83.00040454

2(e) Number of memory accesses

As C language is row major, while loading data from memory into cache elements present in a single row are loaded into the cache. Not all the elements of row are loaded - as per the capacity of cache elements are loaded that is 8 elements. Hence, till the time we access the next 8 elements after the first element, we find a cache hit 8 times in a row. Again a cache miss is encountered and 8 elements from row are loaded which give series of 8 cache hits and then a miss. This cycle goes on.
As in matrix multiplication we use row of one matrix and column of another matrix; the cache miss of other matrix which involves column traversal is equal to the number of elements present in the column (Reason : C is row major language).
Hence overall calculation of one data value in output matrix requires $\frac{N}{8} * N$ main memory access.
In case of parallel code same number of memory access are required. The division of work on processors to calculate a single data value in output matrix is done efficiently in parallel code. In this case as we parallelize the outermost loop which means that data is decomposed row wise. Hence, each processor access certain rows which makes the code efficient enough in terms of time required to access memory. Thus a speedup of 4 equivalent to theoretical value can be reached in this case.

2(f) Number of computations

Total number of computations is calculated by total floating point operations performed. There are two floating point operation + and $*$ in the multiplication part of matrices. So the total computation are $2n^3$.

3 Curve Based Analysis

3(a) Time Curve related analysis (as no. of processor increases)

It is observed from the graphs that the time taken for execution is the least in the case when the program runs on 4 threads i.e. on 4 processors. The maximum execution time is obtained in the case when the code works on 0 threads i.e. in the case of serial implementation. Greater the number of processors on which the code runs parallely lesser will be the execution time.

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

As the problem size increases the time taken for execution also increases. In case of serial the execution time will increase rapidly. As problem size n increases the time increases considerably for serial code because the complexity reaches n^3 but with parallel the increase is not much we get good speedup for higher values of problem size n.

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

The speedup of any given code is calculated as :

$$Speed\ up = \frac{Time\ taken\ by\ the\ Serial\ code}{Time\ taken\ by\ the\ Parallel\ code}$$

Thus the time taken for a code to run on 4 processors parallely will be the least and so the speedup for 4 threads will be the maximum. The following trend can be observed in the graphs. It is noted that the speedup when the code runs on 4 threads is the highest as compared to when the code runs on 3 threads, 2 threads and 1 thread.
As is observed from the graph the speedup for the case when the code runs on 4 threads is slightly greater than 4. This indicates that the serial code works slower as compared to the parallel code and hence such a speedup is obtained. The speedup obtained for 3 threads is slightly greater than 3. The speedup obtained for 2 threads is lesser than 2 and approximately near to 1.5. The speedup of 1 thread is obtained as 1.

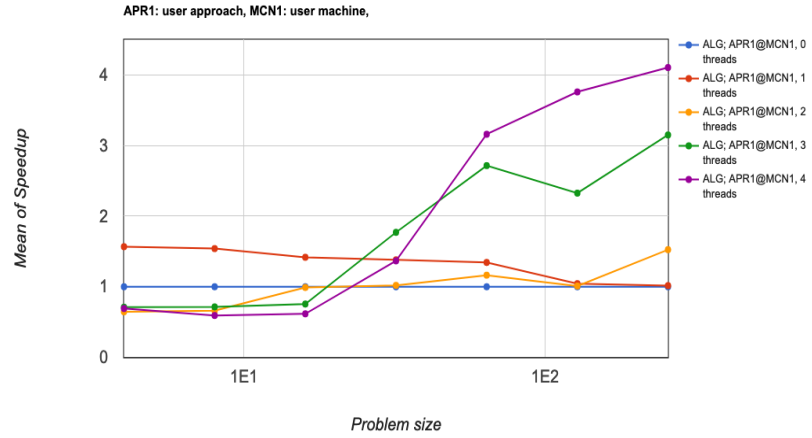
3(d) Efficiency Curve related analysis

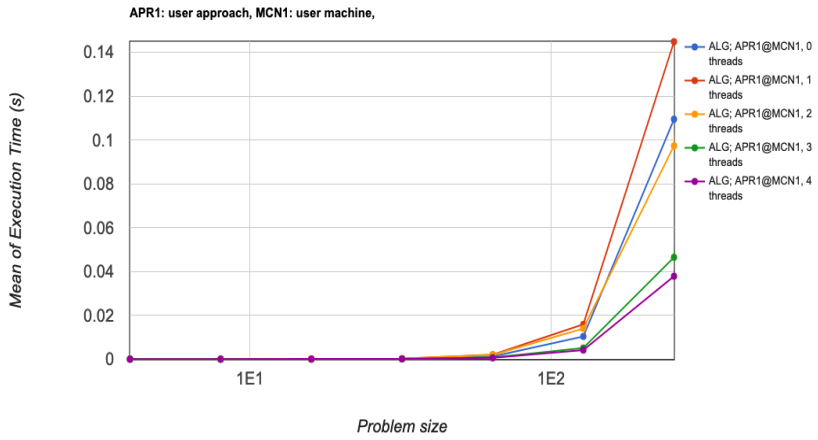
The efficiency of any code is calculated as :

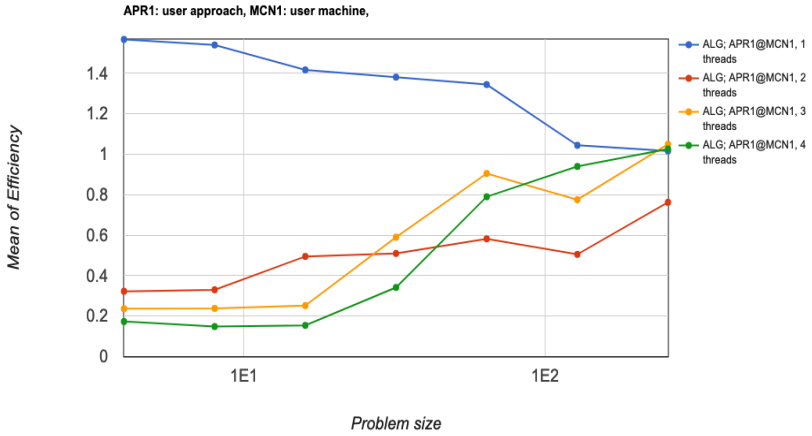
$$Efficiency = \frac{Speed\_up}{Number\_of\_Processors}$$

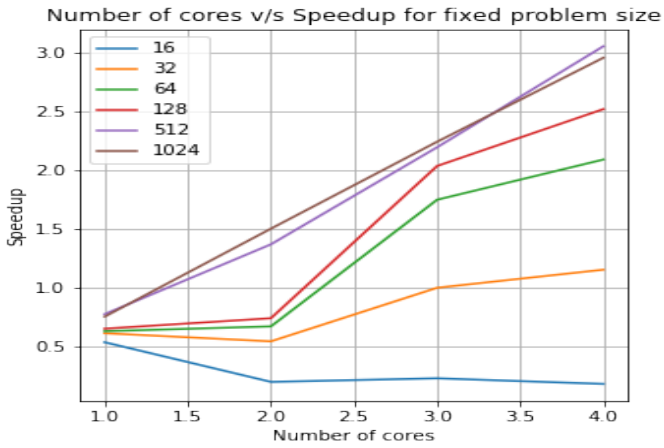
Thus in this case it is noted that the since the speedup of 1 thread was obtained to be around 1 the efficiency will also be obtained around 1. Therefore, as observed in the graph we see that the efficiency when the code runs on 1 thread, on 3 threads and 4 threads, is almost equal. Whereas, in the case when the code runs on 2 threads it is seen that the speedup obtained was less than 2 and around 1.5. Thus, the efficiency obtained in this case will be less then 1. Thus, the efficiency of 2 threads is obtained to be the least as compared to the other cases.

4 Graphs


APR1: user approach, MCN1: user machine,
Mean of Speedup
0
1
2
3
4
1E1
1E2
Problem size
ALG; APR1@MCN1, 0 threads
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads


APR1: user approach, MCN1: user machine,
Mean of Execution Time (s)
0
0.02
0.04
0.06
0.08
0.1
0.12
0.14
1E1
1E2
Problem size
ALG; APR1@MCN1, 0 threads
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads


APR1: user approach, MCN1: user machine,
Mean of Efficiency
Problem size
1E1
1E2
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads


Number of cores v/s Speedup for fixed problem size
Speedup
Number of cores
16
32
64
128
512
1024

Matrix Multiplication (Middle)

Nishi Doshi
Roshani

Dhirubhai Ambani Institute of Information and Communication Technology
201601408@daiict.ac.in
201601059@daiict.ac.in

1 Implementation Details

1(a) Brief and clear description about the Serial implementation

Serial implementation of matrix multiplication involves running 3 for loops for calculating the value at certain cell in matrix. The loops run on same thread.

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

In this part of code, we have parallelized the computations of 2^{nd} loop involved in matrix multiplication. The parallelization occurs in such a way that computations are equally divided on different processors as they were done in previous approach taking into account number of processors available. Division of rows into groups of sizes $\frac{n}{p}$ and adding the last $n\%p$ left rows in the last processor computation.

2 Complexity and Analysis Related

2(a) Complexity of serial code

The serial code contains 3 for loops and hence has $O(n^3)$ time complexity.

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

The parallel code has inner two loops running the similar way as in serial and hence run for $O(n^2)$ time complexity. The outer loop gets its time divided into the available processors and hence overall time complexity turns out to be $O(n^2 * (\frac{n}{p}))$ which hence proves the overall time complexity of code to be $O(\frac{n^3}{p})$

2(c) Cost of Parallel Algorithm

Cost of parallel algorithm is calculated by multiplying number or processors with the time taken for the code to run. Hence, cost of parallel algorithm here is $p * \frac{n^3}{p}$ which is equivalent to cost of serial code that is n^3.

2(d) Theoretical Speedup (using asymptotic analysis, etc.)

The theoretical speedup is given by $\frac{T_{serial}}{T_{parallel}}$ which is equal to $\frac{n^3}{\frac{n^3}{p}} = p$

2(e) Experimental Serial Fraction

The serial fraction is given by $Serial\ fraction = \frac{(end\ to\ end\ time - algo\ time)}{end\ to\ end\ time}$
Experimentally, for different problem sizes we find different values of serial fraction as shown below :

Problem Size	Serial Fractionn
16	0.000205
32	0.0015822
64	0.0126284
128	0.1066274
512	10.6792131
1024	82.3950517

2(f) Number of memory accesses

As C language is row major, while loading data from memory into cache elements present in a single row are loaded into the cache. Not all the elements of row are loaded - as per the capacity of cache elements are loaded that is 8 elements. Hence, till the time we access the next 8 elements after the first element, we find a cache hit 8 times in a row. Again a cache miss is encountered and 8 elements from row are loaded which give series of 8 cache hits and then a miss. This cycle goes on.
As in matrix multiplication we use row of one matrix and column of another matrix; the cache miss of other matrix which involves column traversal is equal to the number of elements present in the column (Reason : C is row major language).
Hence overall calculation of one data value in output matrix requires $\frac{N}{8} * N$ main memory access.
In case of parallel code same number of memory access are required. The division of work on processors to calculate a single data value in output matrix is not done efficiently in this parallel code. We observe that data decomposition is done in terms of column. Hence, time to access memory increases and we observe that a speedup less than 4 is obtained in this case compared to when the outermost loop of compuations was parallelized.

3 Curve Based Analysis

3(a) Time Curve related analysis (as no. of processor increases)

It is observed from the graphs that the time taken for execution is the least in the case when the program runs on 4 threads i.e. on 4 processors. The maximum execution time is obtained in the case when the code works on 0 threads i.e. in the case of serial implementation. Greater the number of processors on which the code runs parallely lesser will be the execution time.

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

As the problem size increases the time taken for execution also increases. In case of serial the execution time will increase rapidly.

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

The speedup of any given code is calculated as :

$$Speed\ up = \frac{Time\ taken\ by\ the\ Serial\ code}{Time\ taken\ by\ the\ Parallel\ code}$$

It is observed that initially the speedup of all the cases start from approximately the same point except for the case of 1 thread. It is noted that the case when the code runs on 4 threads, displays the highest speedup as the execution time is the least in this case. This case shows a speedup of slightly greater than 4 and then reduces to 3.5. This may happen because as the problem size increases there may be some overhead that comes into being. The speedup obtained in the case of 3 threads is approximately equal to 3. The speedup obtained for 2 threads is about 1.5. And similarly the speedup obtained for the case of 1 thread is near 1.

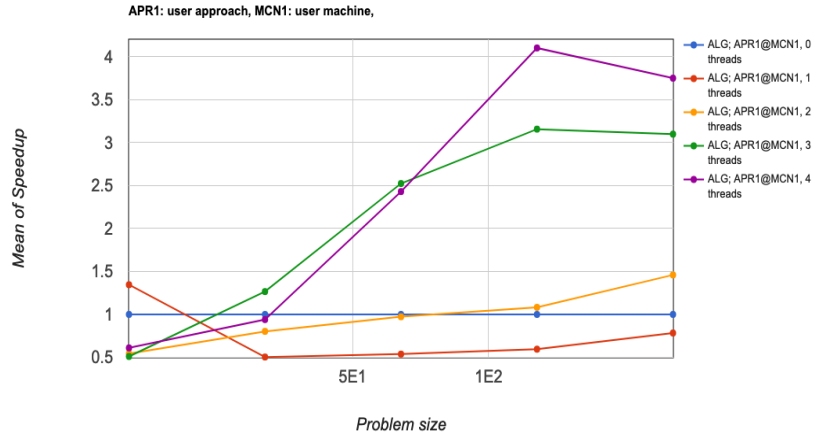
3(d) Efficiency Curve related analysis

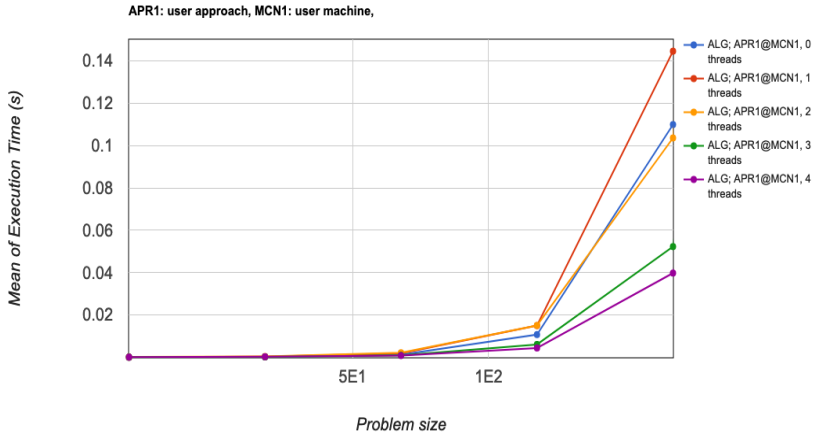
The efficiency of any code is calculated as :

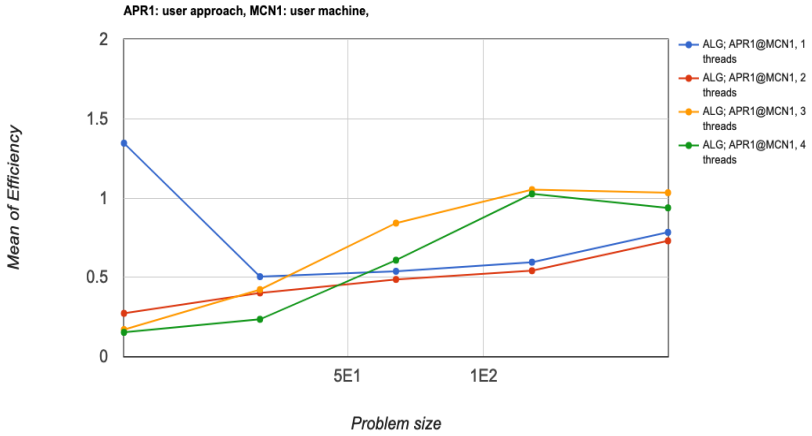
$$Efficiency = \frac{Speed\ up}{Number\ of\ Processors}$$

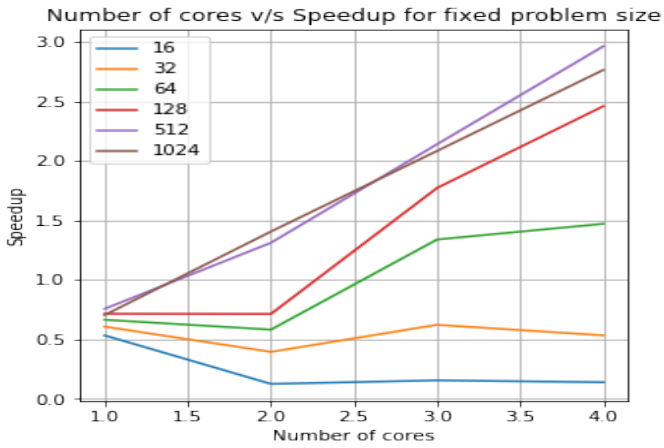
In this case it is observed that the speedup of the case when the code runs on 3 threads is the maximum followed by 4 threads, 1 thread and then 2 threads. This behaviour is observed because the speedup of 3 threads is always above 3 so the efficiency obtained will always be greater than 1. In case of 4 threads it is seen that the speedup first goes slightly above 4 but then decreases to 3.5. So the efficiency also decreases. Whereas, in the case when the code runs on 2 threads it is seen that the speedup obtained was less than 2 and around 1.5. Thus, the efficiency obtained in this case will be less then 1. Thus, the efficiency of 2 threads is obtained to be the least as compared to the other cases.

4 Graphs


APR1: user approach, MCN1: user machine,
Mean of Speedup
4
3.5
3
2.5
2
1.5
1
0.5
5E1
1E2
Problem size
ALG; APR1@MCN1, 0 threads
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads


APR1: user approach, MCN1: user machine,
Mean of Execution Time (s)
0.14
0.12
0.1
0.08
0.06
0.04
0.02
5E1
1E2
Problem size
ALG; APR1@MCN1, 0 threads
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads


APR1: user approach, MCN1: user machine,
Mean of Efficiency
Problem size
2
1.5
1
0.5
0
5E1
1E2
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads


Number of cores v/s Speedup for fixed problem size
Speedup
Number of cores
16
32
64
128
512
1024
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Block Matrix multiplication

Nishi Doshi
Roshani

Dhirubhai Ambani Institute of Information and Communication Technology
201601408@daiict.ac.in
201601059@daiict.ac.in

1 Implementation Details

1(a) Brief and clear description about the Serial implementation

Serial implementation of block matrix multiplication consists of 5 loops which divides the matrix into blocks of $Block\ size = B$ that is specified in the code. Each blocks value is calculated in the code and then finally we get the desired matrix.
We have kept block size equal to 32. The primary reason behind choosing this block size is that when block sizes were varied and time involved to solve the problem was tested, until a particular block size time consumption decreased but then it started increasing. Reason for such a behavior is that when block size was less than cache size, the time required by block size that couldn't fit all its elements in the cache increased. Hence, looking at the trend, 32 was selected as the optimal block size.

```
block,1024,2,9,98580015
block,1024,4,6,330848473
block,1024,8,5,840566966
block,1024,16,5,743673952
block,1024,32,5,639132701
block,1024,64,5,957193194
block,1024,128,7,990364764
```

Fig. 1. The above assumption of block size = 32 can be observed through this figure

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

In parallel implementation, we have parallelized the outermost loop which runs for iterating through rows. We have divided rows into groups of

$$\frac{number\ of\ rows}{number\ of\ processors}$$

that is $\frac{n}{p}$ sizes with adding n%p left rows to the last processor involved in running the parallel code. The inner loops run the similar way as they run in serial code.

2 Complexity and Analysis Related

2(a) Complexity of serial code

The serial code contains 5 for loops and hence has $O(\frac{n^3}{p})$ time complexity.

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

The complexity obtained is :

$$O(\frac{n}{\sqrt{p}} * \frac{n}{\sqrt{p}} * n) = O(\frac{n^3}{p})$$

Here we have divided both the matrices in blocks hence if we multiply block matrices in parallel then we will get comparatively less complexity.

2(c) Theoretical Speedup (using asymptotic analysis, etc.)

Theoretical speedup should in this case should reach equal to the number of processors available. The reason for this is that the code includes no idle time as well as no overhead and hence direct parallelization of problem happens. Thus as per the number of processors available the speedup should change accordingly.

2(d) Experimental Serial Fraction

The serial fraction is given by $Serial\ fraction = \frac{(end\ to\ end\ time - algo\ time)}{end\ to\ end\ time}$
Experimentally, for different problem sizes we find different values of serial fraction as shown below :

Problem Size	Serial Fraction
32	0.0016452
64	0.0126015
128	0.1075826
512	6.5143214
1024	58.3851918

2(e) Number of memory accesses

The number of memory accesses would be equal for three matrices. So it would be of the order of $3n^3$.

2(f) Number of computations

The number of computations is calculated by by total floating point operations performed. There are two floating point operations involved in multiplying matrices : + and $*$. So the total number of computation are $2n^3$.

3 Curve Based Analysis

3(a) Time Curve related analysis (as no. of processor increases)

It is observed from the graphs that the time taken for execution is the least in the case when the program runs on 4 threads i.e. on 4 processors. The maximum execution time is obtained in the case when the code works on 0 threads i.e. in the case of serial implementation. Greater the number of processors on which the code runs parallely lesser will be the execution time.

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

As the problem size increases the time taken for execution also increases. In case of serial the execution time will increase rapidly.

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

The speedup of any given code is calculated as :

$$Speed\_up = \frac{Time\ taken\ by\ the\ Serial\ code}{Time\ taken\ by\ the\ Parallel\ code}$$

Thus the time taken for a code to run on 4 processors parallely will be the least and so the speedup for 4 threads will be the maximum. The following trend can be observed in the graphs. It is noted that the speedup when the code runs on 4 threads is the highest as compared to when the code runs on 3 threads, 2 threads and 1 thread.
As is observed from the graph the speedup for the case when the code runs on 4 threads is around 3.5. The speedup obtained for 3 threads is slightly less than 2.5. The speedup obtained for 2 threads is lesser than 2 and approximately near to 1.5. The speedup of 1 thread remains below 1 for all times.

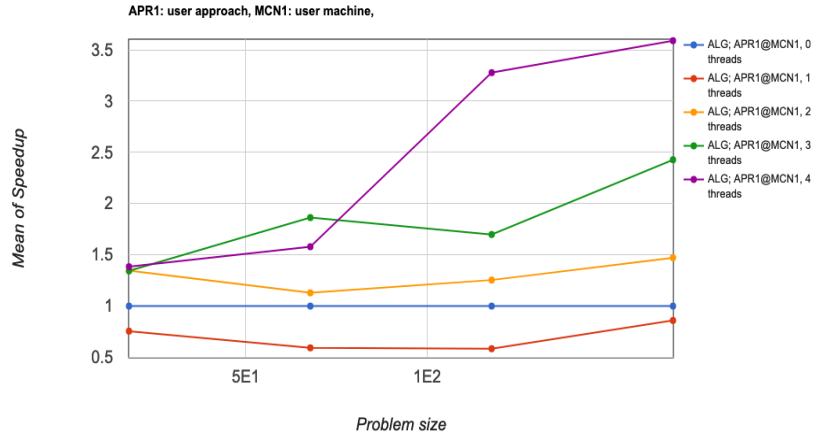
3(d) Efficiency Curve related analysis

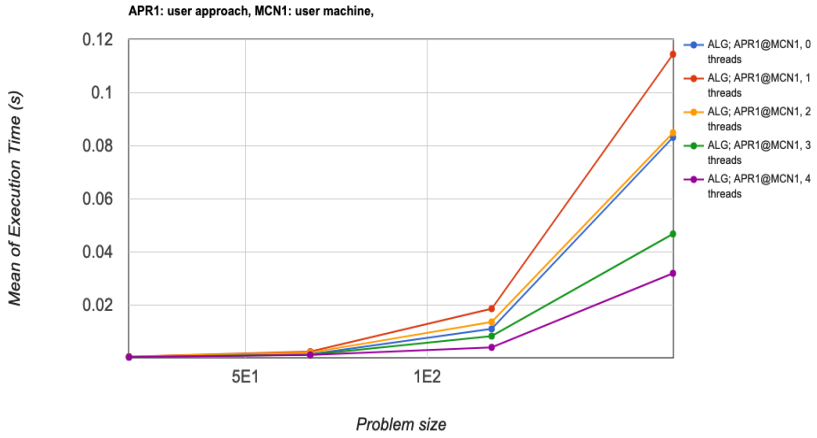
The efficiency of any code is calculated as :

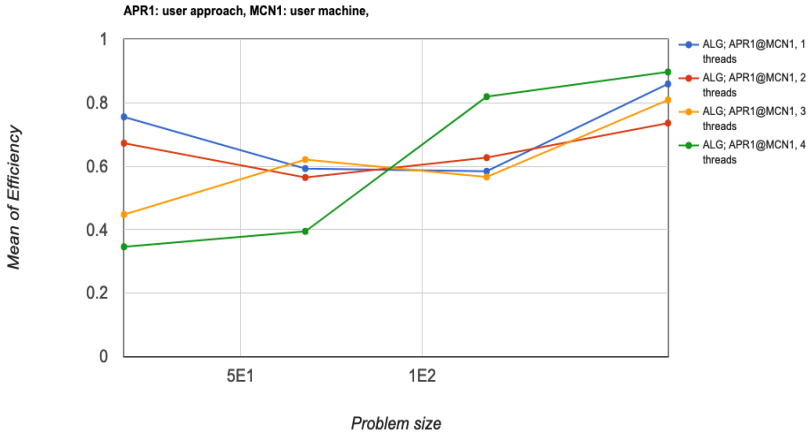
$$Efficiency = \frac{Speed\ up}{Number\ of\ Processors}$$

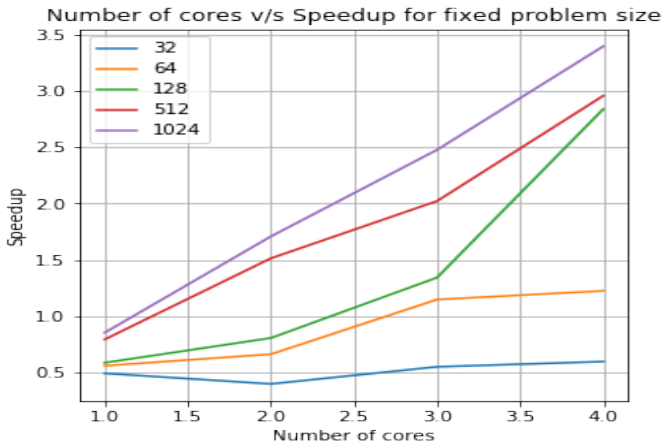
Thus in this case it is noted that the since the speedup of 1 thread was slightly less than 1 thus the efficiency obtained will also be less than 1. Whereas, in the case when the code runs on 2 threads it is seen that the speedup obtained was less than 2 and around 1.5. Thus, the efficiency obtained in this case will be less then 1. Thus, the efficiency of 2 threads is obtained to be the least as compared to the other cases. It is also observed that after a certain time the efficiency of 4 threads increases as the speedup of 4 threads increases more as compared to the other cases. Due to this reason we obtain such trends.

4 Graphs


APR1: user approach, MCN1: user machine,
Mean of Speedup
3.5
3
2.5
2
1.5
1
0.5
5E1
1E2
Problem size
ALG; APR1@MCN1, 0 threads
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads


APR1: user approach, MCN1: user machine,
Mean of Execution Time (s)
0.12
0.1
0.08
0.06
0.04
0.02
5E1
1E2
Problem size
ALG; APR1@MCN1, 0 threads
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads


APR1: user approach, MCN1: user machine,
Mean of Efficiency
Problem size
1
0.8
0.6
0.4
0.2
0
5E1
1E2
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads


Number of cores v/s Speedup for fixed problem size
Speedup
Number of cores
32
64
128
512
1024
3.5
3.0
2.5
2.0
1.5
1.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0