High Performance Computing

Pratvi Shah, 201801407
Arkaprabha Banerjee,201801408

CS-301
Prof. Bhaskar Chaudhry
Due Date: 20/02/2021

Lab Date: 10/02/2021

Lab 3

Hardware Details:

- CPU = 4
- 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

1 Introduction

In this report we shall be comparing serial and parallel implementations for various algorithms and compare their performance as well as their shortcomings. The broad structure of the report would be as follows :

- Implement the serial version of the algorithm and analyze the relevant metrics.
- In order to make the code parallel, we shall try to divide the computations equally between the specified number of threads and at the end of the computations we shall be synchronizing all the values in order to get our final answer. We have employed a host of methods and we shall be comparing their performance in a relative sense. We shall be comparing the parallel implementations via 3 main categories :
    1. Overall Parallel Implementation (i.e How are we actually parallelising the core algorithm)
    2. Comparing the Locking Mechanisms
    3. Comparing various scheduling algorithms

Question 1: Calculate value of π using trapezoidal method

Implementation Details:

1. Description about the Serial implementation

In serial implementation a single loop runs for all the N(problem size) steps and gives the value of the integral using trapezoidal method. The output of the code gave us the desired value of π.

2. Description about the Parallel implementation

- Core Algorithms :
  1. In the first method we used a simple *pragma omp parallel* directive and manually divided the iterations by virtue of the thread id that is running the section at a particular instance.
  2. We then used the **for** directive to implicitly divide the computations of the entire loop between all the available threads.
  3. Lastly in order to solve the problem of false sharing, we employed the **Padding Method**, which essentially uses a 2D matrix to store the relevant values so that each of our required values are stored in different cache lines and the overall synchronization overhead is slightly reduced.
- Locking Mechanisms :
  1. **Critical :** Since the integrated value (via trapezoidal rule) needs to be added to the global variable, hence the critical clause has been used. It essentially makes a particular section accessible to a single thread at a time and in essence serializes that section of the code.
  2. **Atomic :** This is a fine tuned version of the Critical Clause. It basically ensures that the next instruction will be performed by only one thread at a time. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using critical.
  3. **Reduction :** The reduction clause is also implemented to avoid race condition or data inconsistency. It makes a copy of the global variable and update the value there and once our parallel section ends, it is updated to the global variable. The reduction algorithm is significantly efficient as compared to its counterparts.
- Scheduling Mechanisms :
  1. **Static :** It divides the iterations into chunks of size chunk-size and then distributes the chunks to threads in a circular order. The entire allocation is decided beforehand hence the overhead is slightly low as compared to other methods. The default value of chunk size was considered for our implementation.
  2. **Dynamic :** It divides the iterations into chunks of size chunk-size. Each thread then executes its share of iterations and then requests another chunk until there are no more chunks available.There is no particular order in which the chunks are distributed to the threads.The default value of chunk-size(=1) was utilized.

3. **Guided :** It is similar to Dynamic Scheduling except the way the chunks are allocated. The Chunk size is not same and is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases as we proceed.The default value of chunk size is taken.
4. **Auto :** In this method, the compiler decides the best possible form of scheduling.

Complexity

We consider complexity in terms of the number of computations done by each thread or each processor.

Complexity of serial code:

In serial code total number of computations are same as the problem size so, complexity of serial code is $O(N)$.

Complexity of parallel code:

In methods which explicitly define the for loops the number of computations are equally divided amongst all the threads. For the implicit methods we assume that similar condition will arise. Hence, the complexity of any parallelisation will be $O(N/P)$.

Cost of Parallel algorithm

We define the cost of any implementation in terms of its theoretical speedup w.r.t., the serial implementation.

$$cost = speedup = \frac{time\_taken\_by\_serial\_implementation}{time\_taken\_by\_parallel\_implementation} = \frac{N}{N/P} = P \tag{1}$$

We could Also define Cost in a slightly different way. One could also think of it as the overhead incurred while defining and synchronizing parallel directives. It would be directly proportional to the problem size as well as the number of threads used, K(N,P).

Theoretical Speed Up:

We use Eq. (1), the asymptotic approximation of speedup, to calculate the theoretical speedup for different number of threads.As one thread runs on one processor, we can safely approximate the speedup.

No. of threads	**Speedup**
1	P=1
2	P=2
3	P=3
4	P=4

Curve Related Analysis

For our experiments we have varied N from 10 to 10^8 in steps of powers of 10 and considered 1,2,3 and 4 threads(or processors P) for each value of N.

Time Curve Related analysis:

The time taken to complete the execution is directly proportional to the problem size(N) as expected. We can further observe that as the number of processors/threads increase for the same implementation the time reduces, with some exceptions arising due to random nature of the system.

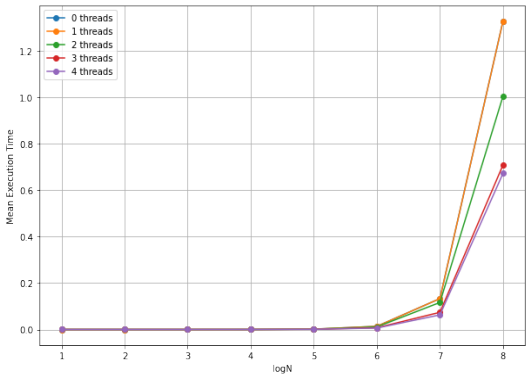


Figure 1: Mean Execution Time for basic parallelisation

We notice that in general with an increase in the number of threads the mean execution time reduces. This is because that the work is being divided to a larger number of threads. However on comparing the serial execution time and the parallel execution with 1 thread we find that the parallel implementation takes a slightly higher amount of time. This is because even though they are conceptually similar yet the parallel one requires a higher amount of overhead.

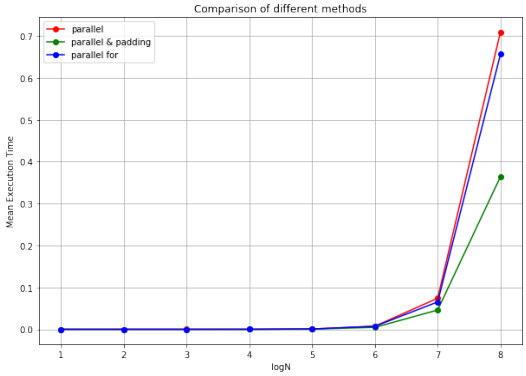


Figure 2: Mean Execution Time for parallelisation Techniques with 3 threads

We compare the parallel implementation (with manual division of work), the 'for' directive and the padding mechanism. Although theoretically , the 'for' directive while implicitly dividing work may not equally divide the work (in some cases), as compared to the standard parallel counterpart, resulting in a higher execution time, however, we observe the opposite.This is because of better scheduling policies while using 'for'. However, the least time occurs while using padding. This is because it prevents false sharing ,which takes a large overhead and is present in the other two cases.

When we don't use any locking mechanism and write the code such that we don't update

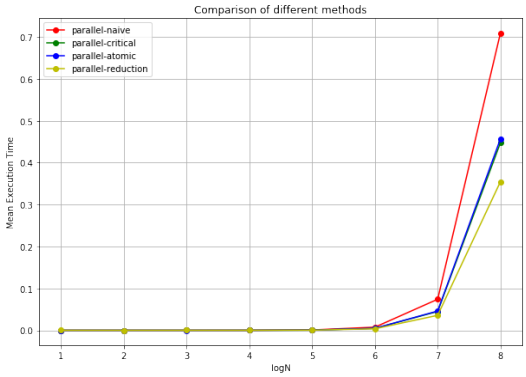


Figure 3: Mean Execution Time for Locking Mechanisms with 3 threads

anything to the common variable in the parallel region, we obtain the highest time. This is because of the extra loop at the end of the parallel region. The locking mechanisms have a significantly reduced time due to the extra loop not being present. Among them critical and atomic are almost at par with atomic having slightly lower time on account of its specificity towards particular statements. Reduction has the lowest time on account of the extremely efficient reduction algorithm which doesn't serialize the entire piece of code and instead makes copies and updates them at the end.

Among scheduling techniques, dynamic scheduling performs the worst. This is justified as

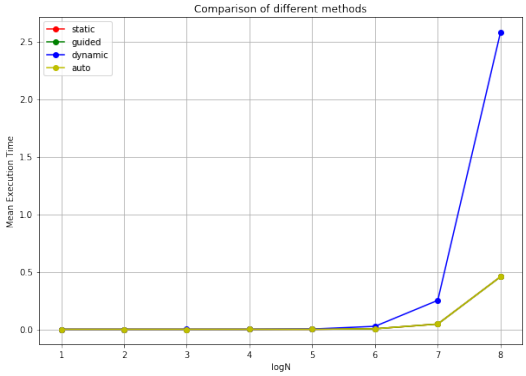


Figure 4: Mean Execution Time for scheduling Techniques with 3 threads

dynamic scheduling allocates one computation(default chunk-size) to one thread in arbitrary order which makes scheduling random and thus introduces delay due to synchronisation at the end. Guided Scheduling, Static Scheduling and Auto Scheduling perform equally better on account of the nature of their algorithms and the low amount of overhead as compared to dynamic scheduling.

Speedup Curve Related analysis

We observe that by increasing the number of threads increases the performance but the maximum performance remains within the theoretical limit.

We observe the speedup is a direct consequence of the mean execution time, i.e higher the

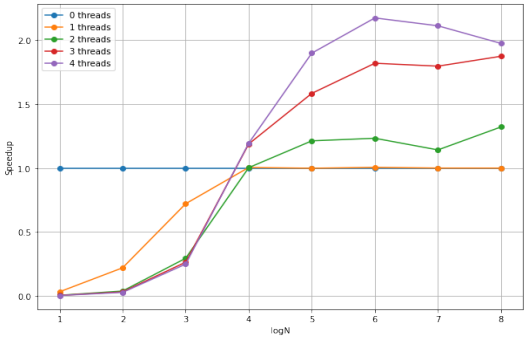
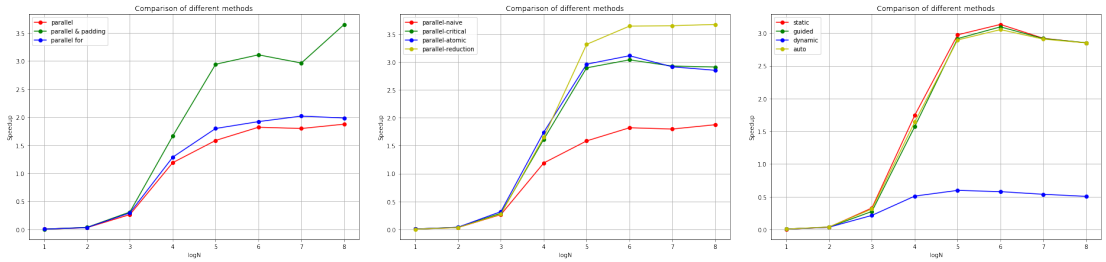


Figure 5: Speedup for simple parallelisation with varying threads

mean execution time, lower the speedup. Although theoretically the speedup shouldn't exceed the number of processors (3 in this case), however we observe a super-linear speedup in some cases (> 3). This is because we have a different cache for each thread in the parallel domain, hence their speedup with respect to the standard serial code (which has a single cache for computation) exceeds the theoretical value.Compiler optimization could also be another reason for this.



(a) (b) (c)

Figure 6:
(a) Speedup for parallelisation Techniques with 3 threads
(b) Speedup for Locking Mechanisms Techniques with 3 threads
(c) Speedup for scheduling Techniques with 3 threads

Efficiency Curve Related analysis

Although the net speedup increases with an increase in the number of processors , however, we can notice that the efficiency reduces as the number of threads/processors increase for a given problem size. So we could conclude from our experiment that the efficiency of an algorithm depends on the level of ease of synchronisation amongst the threads/processors and the parallel overhead.Even if the net speedup is higher however the work done per processor maybe inefficient.

The trend for various scheduling, locking mechanisms etc still remain the same (similar to speedup) in a relative sense.

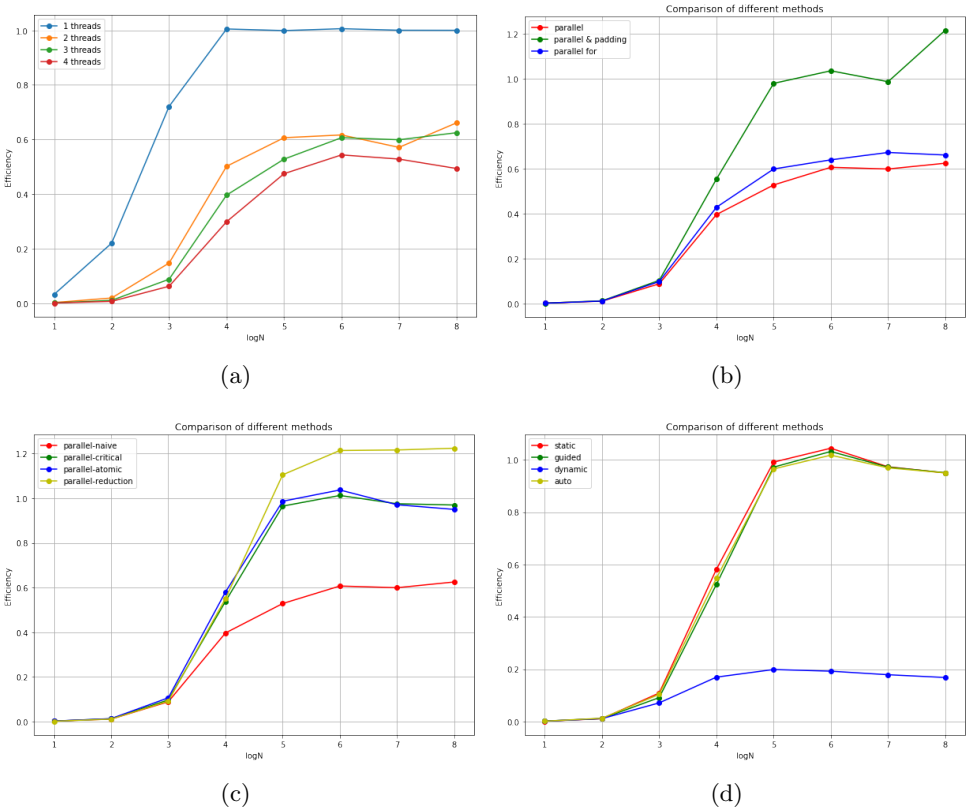


Figure 7:
(a) Efficiency for simple parallelisation with varying threads
(b) Efficiency for parallelisation Techniques with 3 threads
(c) Efficiency for Locking Techniques with 3 threads
(d) Efficiency for scheduling Techniques with 3 threads

Speedup v/s Number of cores analysis

.

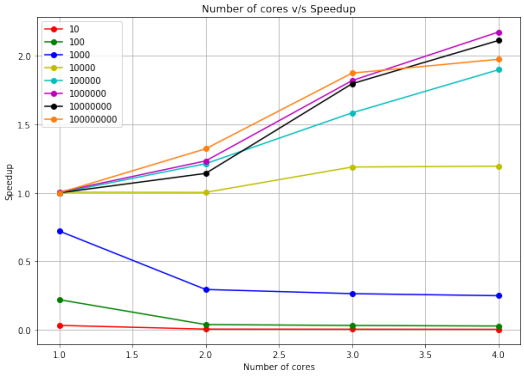


Figure 8: Speedup v/s Number of cores for a given parallelisation Technique

We can see from the plots that the speedup increases with increase in the number of cores for most problem sizes.For very low problem size we observe the reverse trend. That is perhaps due to the fact that the overhead exceeds the speedup gained by optimization. We could also observe

that the maximum speedup that can be achieved for a given number of threads could exceed the number of threads used(P).We observe that for a higher problem size, with the increase in number of threads, the speedup increases. This is because the parallel overhead increases with a lower rate as compared to the problem size hence after a particular size it becomes negligible and we see a really good speedup. For lower values we see the opposite because the parallel overhead , overweighs the speedup. We also see that they tend to saturate after a point. This is because no more speedup via parallelization can be achieved for this particular implementation. In some of the scheduling methods and while using reduction clause we observe that the speedup goes beyond the maximum expected theoretical speedup.

Question 2a: Summation of two vectors

Implementation Details:

1. Description about the Serial implementation

In serial implementation, one single master thread will run through the whole problem size(N) and in single loop all element wise addition will take place.

2. Description about the Parallel implementation

- Core Algorithms :
    1. In the first method we used a simple *pragma omp parallel* directive and manually divided the iterations into equal consecutive P number of chunks and then each chunk was taken up by a particular thread.
    2. We then used the **for** directive to implicitly divide the computations of the entire loop between all the available threads.
    3. As in this problem we divided the computations in a consecutive manner we do not see a need to perform padding due to the cache locality.
- Locking Mechanisms : In this part we only access one element of the output vector only once hence, we do not need any locking mechanism.
- Scheduling methods:
    1. **Static**: In this scheduling technique we considered two cases one with default chunk size(which appears when we do not provide any value to that attribute) and for chunk size of 8.
    2. **Dynamic**: In this scheduling technique we considered two cases one with default chunk size which is 1 and for chunk size of 8.

Complexity

We consider complexity in terms of the number of computations done by each thread or each processor.

Complexity of serial code:

In serial code total number of computations are same as the problem size so, complexity of serial code is $O(N)$.

Complexity of parallel code:

In parallel codes, two factors play a major role in determining the complexity one is the complexity due to the overhead and the other is computational complexity. While computational complexity has a definite upper bound(= P) we can not accurately calculate the complexity or the time involved in the parallel overhead $K(N, P)$.

Cost of Parallel algorithm

We could define Cost in two ways. One could also think of it as the overhead incurred while defining and synchronizing parallel directives. It would be directly proportional to the problem size as well as the number of threads used, K(N,P).Here, we neglect the parallel overhead cost and only consider the computational cost. So, the cost is same as the theoretical speedup given by:

$$cost = speedup = \frac{time\_taken\_by\_serial\_implementation}{time\_taken\_by\_parallel\_implementation} = \frac{N}{N/P} = P \tag{2}$$

Theoretical Speed Up:

We use Eq. (2), the asymptotic approximation of speedup, to calculate the theoretical speedup for different number of threads.As one thread runs on one processor, we can safely approximate the speedup.

No. of threads	**Speedup**
1	P=1
2	P=2
3	P=3
4	P=4

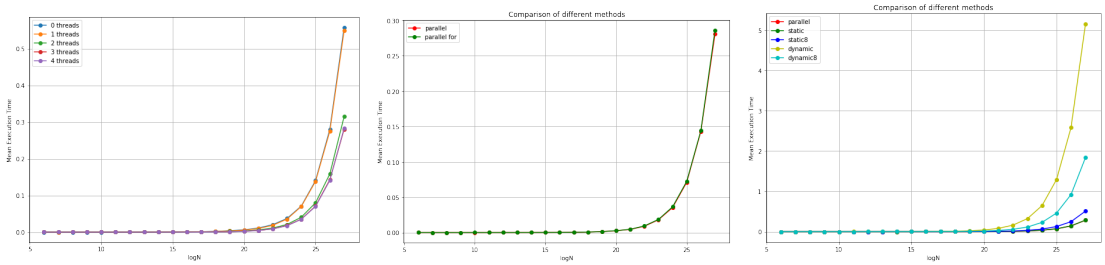
Curve Related Analysis

For our experiments we have varied N from 2^6 to 2^{28} in steps of powers of 2 and considered 1,2,3 and 4 threads(or processors P) for each value of N.

Time Curve Related analysis:

The time taken to complete the execution is directly proportional to the problem size(N) as expected. We can further observe that as the number of processors/threads increase for the same implementation, the time reduces, with some exceptions arising due to random nature of the system.

The rest of the observations for various directives remain the same (Ref Q1) with some notable points. We observe that with an increase in the number of chunk size in dynamic scheduling, it performs slightly better although it still takes more time than static scheduling. Furthermore, we also observe that '**pragma parallel for** directive' performs poorly and takes more time than the naive parallelization technique. This is in contrast to Q1 and happens because of poor division of work which its better scheduling policies can't make up for.

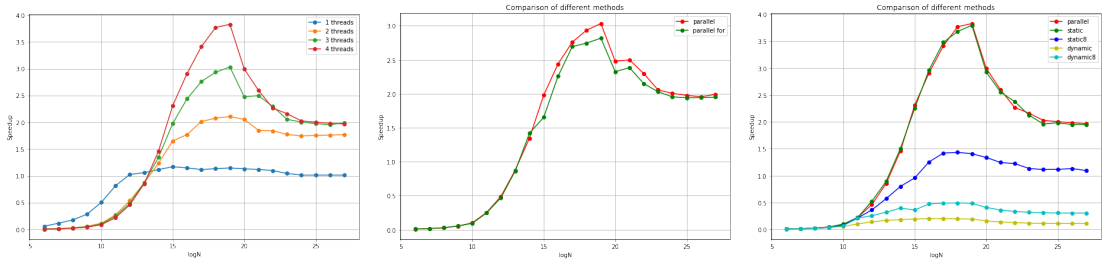


(a) (b) (c)

Figure 9:
(a) Mean Execution Time for basic parallelisation
(b) Mean Execution Time for parallelisation Techniques with 3 threads
(c) Mean Execution Time for scheduling Techniques with 3 threads

Speedup Curve Related analysis

We observe the speedup is a direct consequence of the mean execution time, i.e higher the mean execution time, lower the speedup. Although theoretically the speedup shouldn't exceed the number of processors (3 in this case for (b) and (c)), however we observe a super-linear speedup in some cases (> 3). This is because we have a different cache for each thread in the parallel domain, hence their speedup with respect to the standard serial code (which has a single cache for computation) exceeds the theoretical value.Compiler optimization could also be another reason for this. We also notice that the static scheduling with chunk-size=8 performs worse than the one with default chunk size as in default case each thread gets a single chunk of same size whereas when chunk-size is explicitly mentioned the same size of chunk is allocated to each thread.



(a) (b) (c)

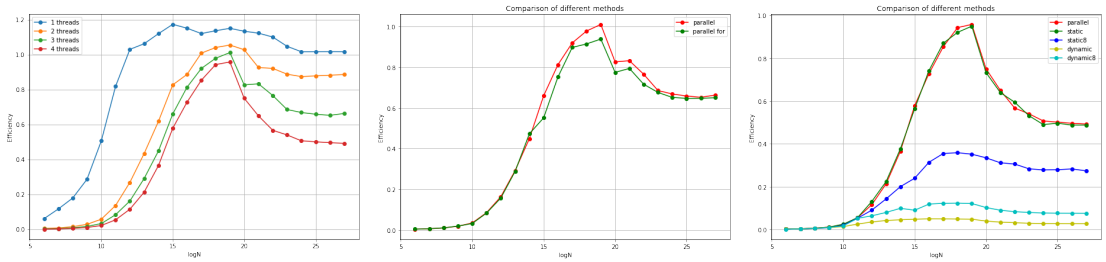
Figure 10:
(a) Speedup for basic parallelisation with varying threads
(b) Speedup for parallelisation Techniques with 3 threads
(b) Speedup for scheduling Techniques with 3 threads

Efficiency Curve Related analysis

Although the net speedup increases with an increase in the number of processors , however, we can notice that the efficiency reduces as the number of threads/processors increase for a given problem size. So we could conclude from our experiment that the efficiency of an algorithm depends on the level of ease of synchronisation amongst the threads/processors and the parallel

overhead.Even if the net speedup is higher however the work done per processor maybe inefficient. We also noticed that the efficiency of thread 1 is higher than the other cases, this might be due to efficient synchronisation due to communication between lesser number of threads.

The trend for various scheduling, locking mechanisms etc still remain the same (similar to speedup) in a relative sense.



(a) (b) (c)

Figure 11:
(a) Efficiency for simple parallelisation with varying threads
(b) Efficiency for parallelisation Techniques with 3 threads
(c) Efficiency for scheduling Techniques with 3 threads

Speedup v/s Number of cores analysis

We can see from the plots that the speedup increases with increase in the number of cores for a most problem sizes.For very low problem size we observe the reverse trend. That is perhaps due to the fact that the overhead exceeds the speedup gained by optimization. We could also observe that for most cases, the maximum speedup that can be achieved for a given number of threads, is same as the number of threads used(P). This was expected based on the theoretical speedup calculated previously. We could also verify that for each case the maximum speedup remains within 3.5.

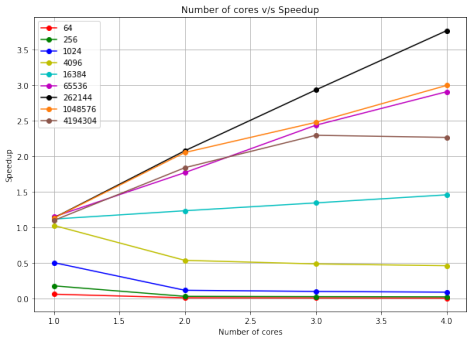


Figure 12: Speedup for a given parallelisation technique

Question 2b: Dot Product of two vectors

Implementation Details:

1. Description about the Serial implementation

In serial implementation, one single master thread will run through the whole problem size(N) and in single loop all element wise multiplication and their addition to the final answer will take place.

2. Description about the Parallel implementation

- Core Algorithms :
    1. In the first method we used a simple *pragma omp parallel* directive and manually divided the iterations into equal consecutive P number of chunks and then each chunk was taken up by a particular thread.
    2. We then used the **for** directive to implicitly divide the computations of the entire loop between all the available threads.
    3. The Padding method has also been employed to rectify false sharing in case it is happening. We basically convert our 1D vector into a 2D vector with the first element of each row containing our actual data and we use the same for our calculations.
- Locking Mechanisms : A locking mechanism has been employed because the global answer variable needs to avoid race condition and data inconsistency. The explanation of the methods has been provided in Q1.
    1. Critical
    2. Reduction
- Scheduling methods
    1. **Static**: In this scheduling technique we considered default chunk size(which appears when we do not provide any value to that attribute).
    2. **Dynamic**: In this scheduling technique we considered a chunk size of 8.

Complexity

We consider complexity in terms of the number of computations done by each thread or each processor.

Complexity of serial code:

In serial code total number of computations are same as the problem size so, complexity of serial code is $O(N)$.

Complexity of parallel code:

In parallel codes, two factors play a major role in determining the complexity one is the complexity due to the overhead and the other is computational complexity. While computational complexity has a definite upper bound($= P$) we can not accurately calculate the complexity or the time involved in the parallel overhead $K(N, P)$.

Cost of Parallel algorithm

We could define Cost in two ways. One could also think of it as the overhead incurred while defining and synchronizing parallel directives. It would be directly proportional to the problem size as well as the number of threads used, K(N,P).Here, we neglect the parallel overhead cost and only consider the computational cost. So, the cost is same as the theoretical speedup given by:

$$cost = speedup = \frac{time\_taken\_by\_serial\_implementation}{time\_taken\_by\_parallel\_implementation} = \frac{N}{N/P} = P \tag{3}$$

Theoretical Speed Up:

We use Eq. (3), the asymptotic approximation of speedup, to calculate the theoretical speedup for different number of threads.As one thread runs on one processor, we can safely approximate the speedup.

No. of threads	**Speedup**
1	P=1
2	P=2
3	P=3
4	P=4

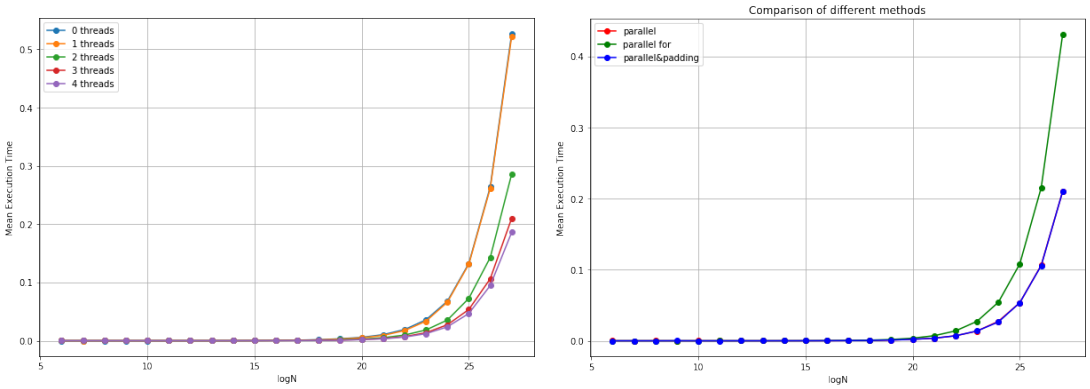
Curve Related Analysis

For our experiments we have varied N from 10 to 10^8 in steps of powers of 10 and considered 1,2,3 and 4 threads(or processors P) for each value of N.

Time Curve Related analysis:

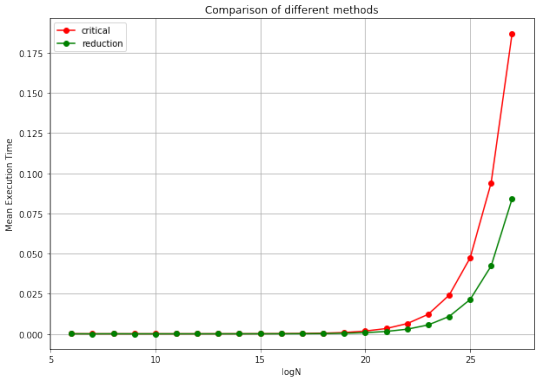
The time taken to complete the execution is directly proportional to the problem size(N) as expected. We can further observe that as the number of processors/threads increase for the same implementation, the time reduces, with some exceptions arising due to random nature of the system.

The rest of the observations for various directives remain the same (Ref Q1) with some notable points. We observe that with the reduction clause performs significantly better. Furthermore, we also observe that '**pragma parallel for** directive' performs poorly and takes more time than the naive parallelization technique. This is in contrast to Q1 and happens because of poor division of work which its better scheduling policies can't make up for. However, the padding mechanism still has the lowest time in this domain.

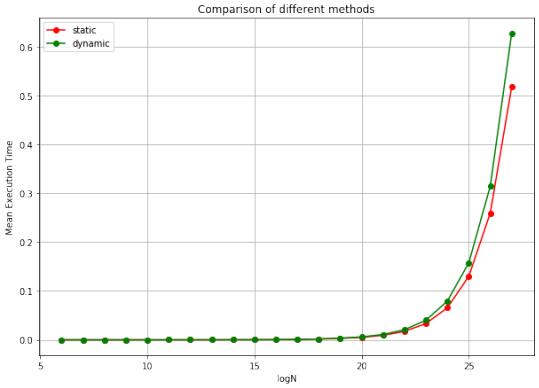


(a)

(b)



(c)

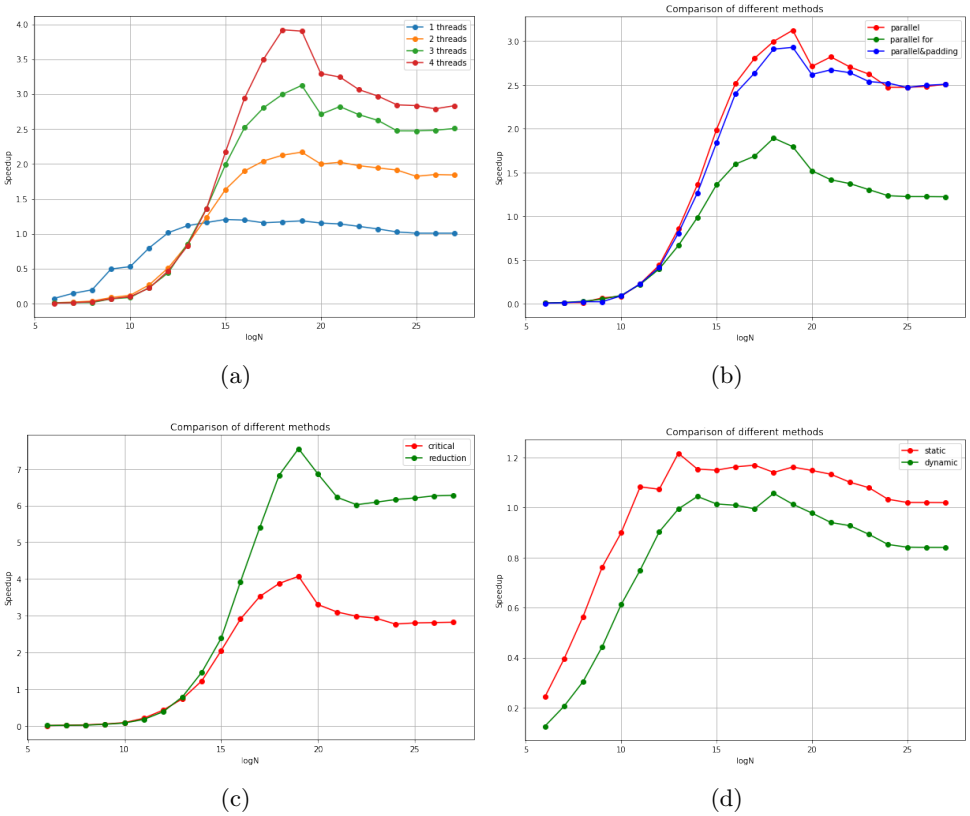


(d)

Figure 13:
(a) Mean Execution Time for basic parallelisation
(b) Mean Execution Time for parallelisation Techniques with 3 threads
(c) Mean Execution Time for Locking Mechanisms with 3 threads
(d) Mean Execution Time for scheduling Techniques with 3 threads

Speedup Curve Related analysis

We observe the speedup is a direct consequence of the mean execution time, i.e higher the mean execution time, lower the speedup. Although theoretically the speedup shouldn't exceed the number of processors (3 in this case), however we observe a super-linear speedup in some cases (> 3). This is because we have a different cache for each thread in the parallel domain, hence their speedup with respect to the standard serial code (which has a single cache for computation) exceeds the theoretical value.Compiler optimization could also be another reason for this.



(a) (b)

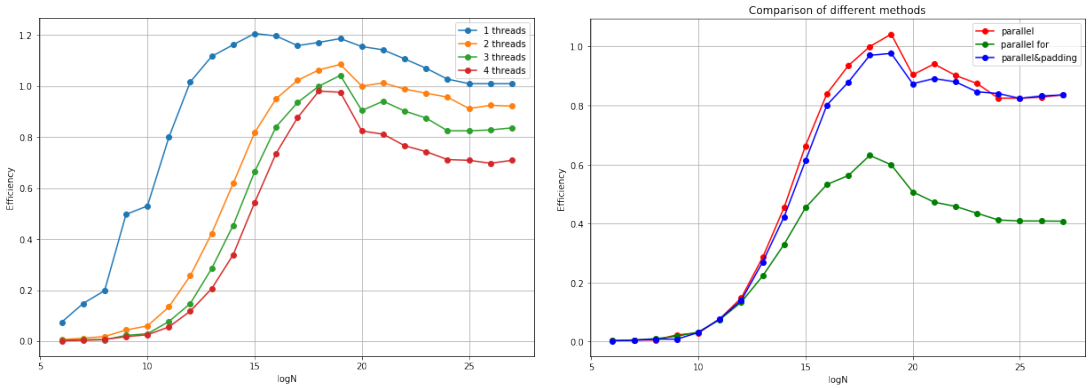
(c) (d)

Figure 14:
(a) Speedup for basic parallelisation with varying threads
(b) Speedup for parallelisation Techniques with 3 threads
(c) Speedup for Locking Mechanisms Techniques with 3 threads
(d) Speedup for scheduling Techniques with 3 threads

Efficiency Curve Related analysis

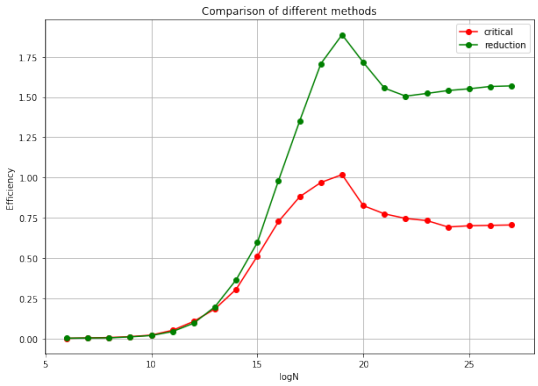
Although the net speedup increases with an increase in the number of processors , however, we can notice that the efficiency reduces as the number of threads/processors increase for a given problem size. So we could conclude from our experiment that the efficiency of an algorithm depends on the level of ease of synchronisation amongst the threads/processors and the parallel overhead.Even if the net speedup is higher however the work done per processor maybe inefficient.

We also see the significant difference between critical and reduction clause as mentioned before while comparing the efficiency. Also while comparing the scheduling clauses we see that in static scheduling, the default mode has higher efficiency than with a chunk size of 8. This is because in the former cases the chunk size is N/4 and in the latter is 8. With decreasing chunk size the overhead increases. The exact same trend is observed for dynamic scheduling as well. Static scheduling again performs better than dynamic scheduling.

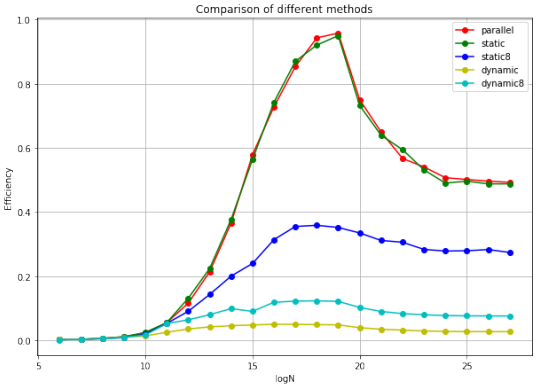


(a)

(b)



(c)



(d)

Figure 15:
(a) Efficiency for simple parallelisation with varying threads
(b) Efficiency for parallelisation Techniques with 3 threads
(c) Efficiency for Locking Mechanisms Techniques with 3 threads
(d) Efficiency for scheduling Techniques with 3 threads

Speedup v/s Number of cores analysis

We can see from the plots that the speedup increases with increase in the number of cores for most problem sizes.For very low problem size we observe the reverse trend. That is perhaps due to the fact that the overhead exceeds the speedup gained by optimization. We could also observe that the maximum speedup that can be achieved for a given number of threads could exceed the number of threads used(P). This was expected based on the fact that each thread has its own local cache. Hence while using multiple threads and comparing it against the serial code which only has a single cache could lead to such speedups.Compiler optimization also plays a role in this. However, for most cases it remained within the theoretical bound.

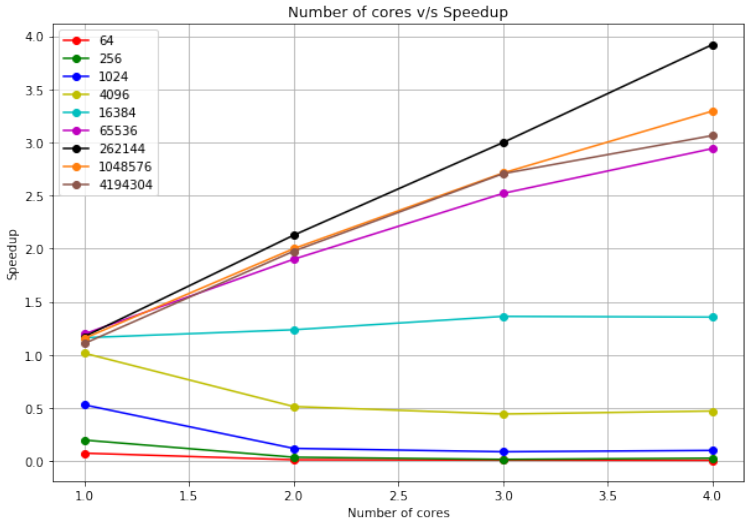


Figure 16: Speedup v/s Number of cores for a given parallelisation technique

Question 3: Parallel matrix multiplication

Implementation Details:

1. Description about the Serial implementation

In the serial implementation, we considered the ijk loop ordering for the simple matrix multiplication and for the blocked case we considered two outer loops and the similar ijk ordering. We also implemented the simple matrix multiplication for the same loop ordering via transposition.

2. Description about the Parallel implementation

In the parallel section we have 3 broad implementations. Under each of these banners we shall explore various other directive and clause. Following is the broad structure of the experiment.

- Unblocked Matrix Multiplication :
  1. Parallelise the outermost loop
  2. Parallelise the middle loop
  3. Use the **collapse** clause on all the three loops.
- Blocked Matrix Multiplication :
  1. Parallelise the outermost loop counterpart (from Unblocked Matrix multiplication).
  2. Parallelise the middle loop counterpart (from Unblocked Matrix multiplication).
- Unblocked Matrix Multiplication with Transposition :
  1. Parallelise the outermost loop
  2. Parallelise the middle loop
  3. Use the **collapse** clause on all the three loops.
- Core Algorithms :

1. Firstly, we parallelised the outermost loop with *pragma omp for* and then we moved to parallelising the middle loop in the same way. This methods of parallelisation were implemented in all unblocked, transpose(unblocked) and blocked matrix multiplication.
2. We also implemented the **collapse(3)** clause for the unblocked matrix multiplication case which converts the three loops into linear form and performs the computations. While trying to implement the same for blocked matrix multiplication the execution take a lot of time. This may be due the complex nature of the new indices of the single loop.

- Scheduling Mechanisms : We implemented given scheduling mechanisms for unblocked matrix multiplication.

  1. **Static**: In this scheduling technique we considered two cases one with default chunk size(which appears when we do not provide any value to that attribute) and for chunk size of 8.
  2. **Dynamic**: In this scheduling technique we considered two cases for chunk size 4 and 8.
  3. **Auto** : Chunk size is not required.

Complexity

We consider complexity in terms of the number of computations done by each thread or each processor.

Complexity of serial code:

In serial code total number of computations are same as the computations taking place which is $O(N^3)$ for all unblocked, blocked and transpose.

Complexity of parallel code:

Just like all other cases here also the parallel overhead is not possible to calculate but we can get an estimate for the computation part. We have considered two cases in unblocked where we parallelise first the outermost loop and then the middle loop. Same cases are considered in both transpose as well as blocked matrix multiplication(with block size B).

1. For unblocked, two loops are same as the serial one and the iterations of one loop gets divided equally in all the threads. So, complexity comes out to be $O(N^2 * \frac{N}{P})$. Same goes on for the transpose case.

2. For blocked, when we parallelise the outermost loop we divide the first loop iterations(=N/B) into the threads while when we consider the middle loop we divide the iterations(=N) amongst the threads. In every way we get a factor of P inwhich the iterations are divided hence, the complexity is $O(\frac{N}{BP} * \frac{N}{B} * N * B^2) = O(\frac{N^3}{P})$

Cost of Parallel algorithm

We could define Cost in two ways. One could also think of it as the overhead incurred while defining and synchronizing parallel directives. It would be directly proportional to the problem

size as well as the number of threads used, K(N,P).Here, we neglect the parallel overhead cost and only consider the computational cost. So, the cost is same as the theoretical speedup given by:

$$cost = speedup = \frac{time\_taken\_by\_serial\_implementation}{time\_taken\_by\_parallel\_implementation} = \frac{N^3}{N^3/P} = P \tag{4}$$

Theoretical Speed Up:

We use Eq. (4), the asymptotic approximation of speedup, to calculate the theoretical speedup for different number of threads.As one thread runs on one processor, we can safely approximate the speedup.

No. of threads	**Speedup**
1	P=1
2	P=2
3	P=3
4	P=4

Effect of varying blocksize on Blocked Matrix Multiplication:

	Serial	1 thread	2 threads	3 threads	4 threads
4	5.857410	6.967834	3.522422	2.492539	1.895459
8	5.583182	6.705107	3.406111	2.358567	1.834838
16	5.295514	6.313793	3.195902	2.249279	1.695827
32	5.222948	6.221770	3.156056	2.209203	1.666761
64	5.711172	7.006540	3.453912	2.693517	1.886155
128	6.556846	8.442380	4.269146	3.253903	2.260621
256	6.533594	8.415801	4.251565	4.277866	2.263436

Figure 17: Time taken by outer loop parallelisation and serial code while varying block size(B)

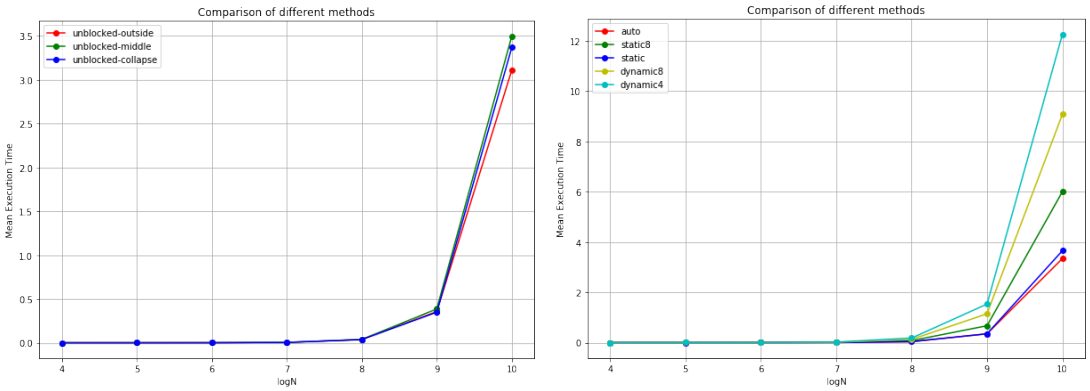
As we can see from the above table the time taken by blocksize=32 is the least for all cases(serial and parallel) so for all further experiments we considered blocksize **B=32**.

Curve Related Analysis

For our experiments we have varied N from 2^4 to 2^{11} in steps of powers of 2 and considered 1,2,3 and 4 threads(or processors P) for each value of N.

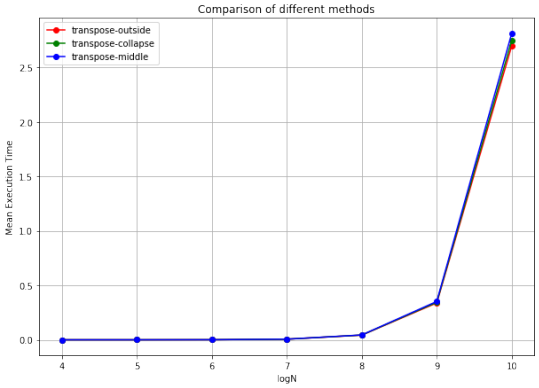
Time Curve Related analysis:

The time taken to complete the execution is directly proportional to the cube of the problem size(N^3)$as expected. We can further observe that as the number of processors/threads increase for the same implen$

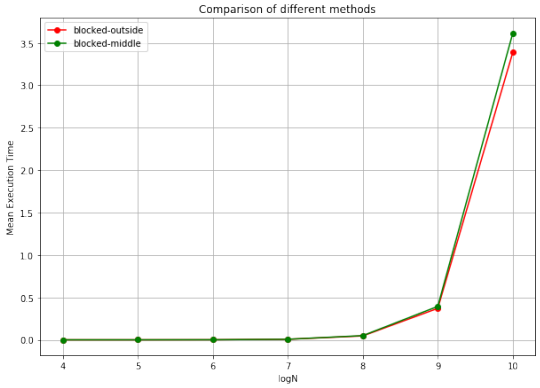


(a)

(b)



(c)



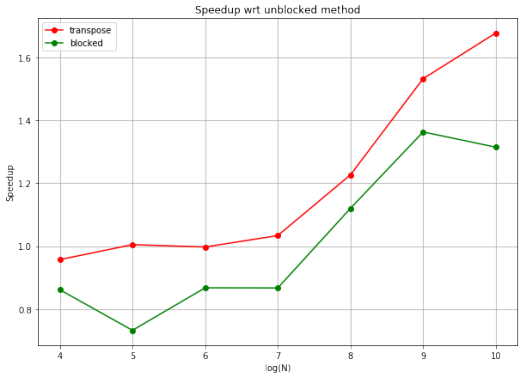
(d)

Figure 19:
(a) Mean Execution Time for different methods(simple) in unblocked matrix multiplication
(b) Mean Execution Time for different methods(scheduling) in unblocked matrix multiplication
(c) Mean Execution Time for different methods(simple) in transpose(unblocked) matrix multiplication
(d) Mean Execution Time for different methods(simple) in blocked matrix multiplication

From the above graphs we notice the following :

- On increasing the number of threads the mean execution time goes down, with the only exception being when we use 1 thread. It takes more time than the serial implementation due to reasons mentioned above.
- Matrix Multiplication via Transposition takes the least time followed by Blocked Matrix Multiplication followed by Naive Matrix Multiplication which takes the highest amount of time. This because with transposition, we hardly incur any cache miss in both of the matrices as we traverse in a row-major form in both hence making it more efficient than Blocked Matrix Multiplication.

$$speedup = \frac{time\_taken\_by\_unblocked}{time\_taken\_by\_other\_method}$$



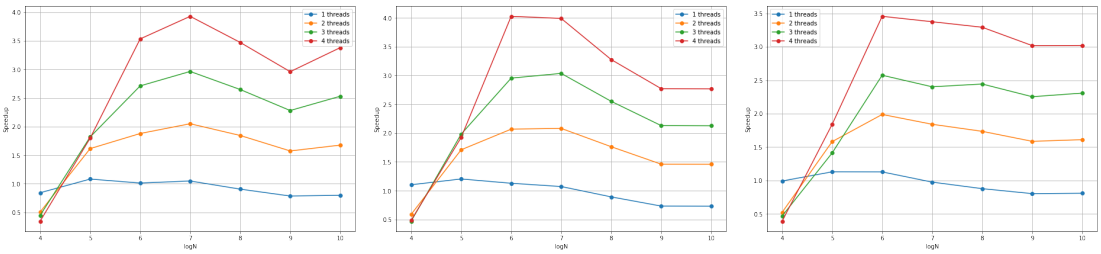
(a)

Figure 20: Speedup v/s problem size for blocked and transpose method

- For all the cases, parallelizing the outer loop (**loop contaning 'i'**), gives better performance than parallelizing the middle loop. This is because the former case is the leading reason for cache misses as we are moving to an entirely new row leading us to bring it to the fast memory again and again. When we parallelize it, the individual cache for each thread greatly helps us in this cause, hence the trend.
- The collapse clause performs the best in transposition and comes in the second position in Naive Matrix Multiplication. This is because in transposition we already took care of the cache miss problem, hence collapsing the loops will give us better performance. However,in the other case a small fragment of the problem still remains as still one of the matrix will move in column major form while calculating the value.
- The trend for scheduling mechanisms remain the same as seen in the previous questions.

Speedup Curve Related analysis

.



(a) (b) (c)

Figure 21:
(a) Speedup for unblocked basic parallelisation
(b) Speedup for transpose(unblocked) basic parallelisation
(c) Speedup for blocked basic parallelisation

We observe the speedup is a direct consequence of the mean execution time, i.e higher the mean execution time, lower the speedup. Although theoretically the speedup shouldn't exceed the number of processors (3 in this case), however we observe a super-linear speedup in some

cases (> 3). This is because we have a different cache for each thread in the parallel domain, hence their speedup with respect to the standard serial code (which has a single cache for computation) exceeds the theoretical value.Compiler optimization could also be another reason for this.

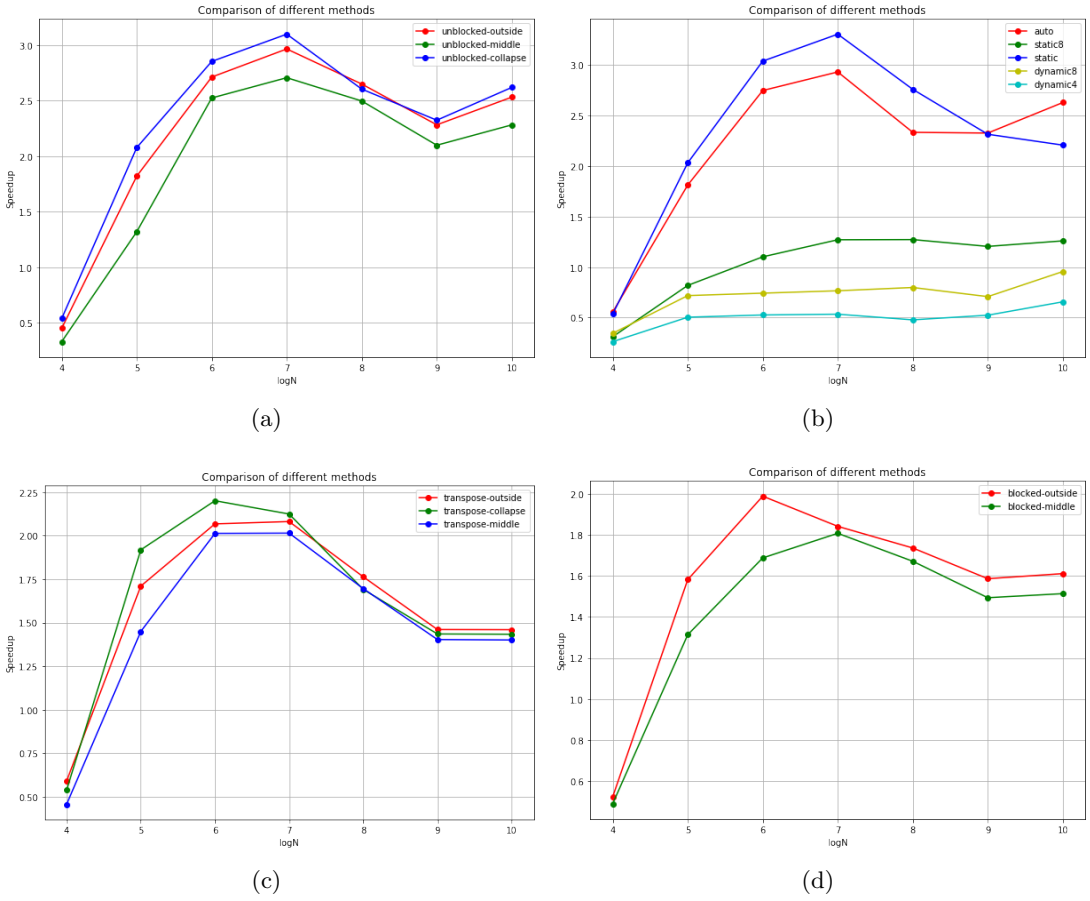
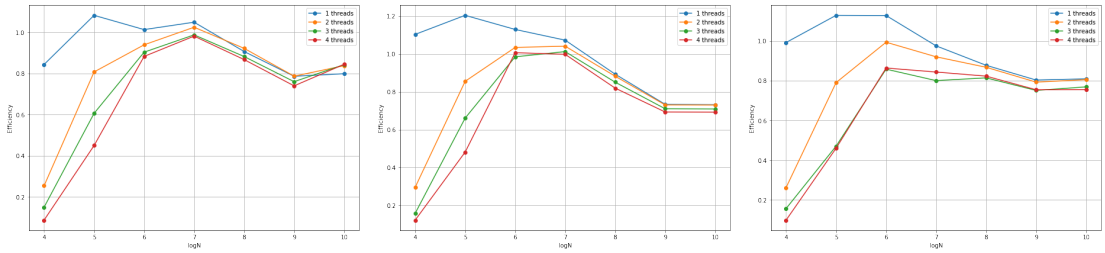


Figure 22:
(a) Speedup for different methods(simple) in unblocked matrix multiplication
(b) Speedup for different methods(scheduling) in unblocked matrix multiplication
(c) Speedup for different methods(simple) in transpose(unblocked) matrix multiplication
(d) Speedup for different methods(simple) in blocked matrix multiplication

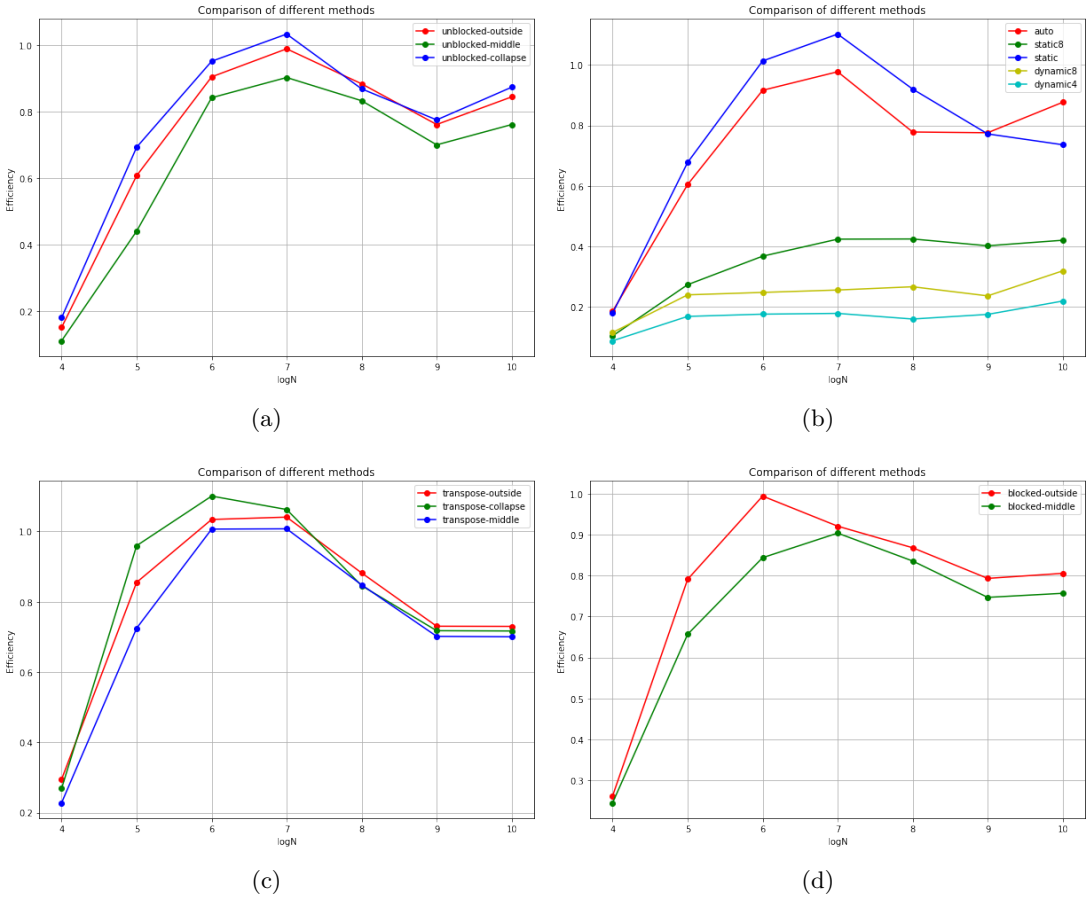
Efficiency Curve Related analysis

Although the net speedup increases with an increase in the number of processors , however, we can notice that the efficiency reduces as the number of threads/processors increase for a given problem size. So we could conclude from our experiment that the efficiency of an algorithm depends on the level of ease of synchronisation amongst the threads/processors and the parallel overhead.Even if the net speedup is higher however the work done per processor maybe inefficient. The trend for various scheduling, locking mechanisms etc still remain the same (similar to speedup) in a relative sense.



(a) (b) (c)

Figure 23:
(a) Efficiency for unblocked basic parallelisation
(b) Efficiency for transpose(unblocked) basic parallelisation
(c) Efficiency for blocked basic parallelisation



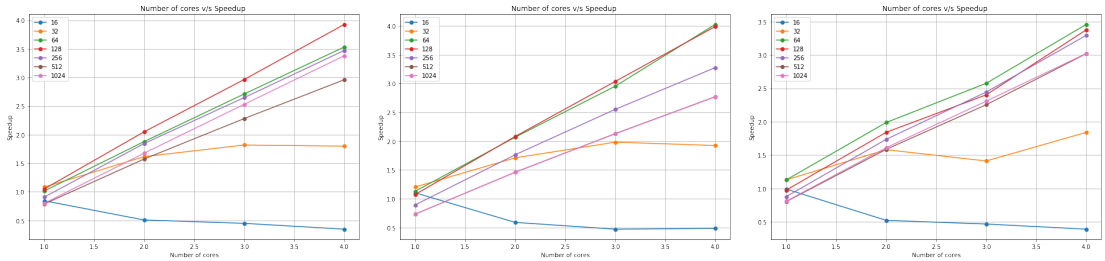
(a) (b)

(c) (d)

Figure 24:
(a) Efficiency for different methods(simple) in unblocked matrix multiplication
(b) Efficiency for different methods(scheduling) in unblocked matrix multiplication
(c) Efficiency for different methods(simple) in transpose(unblocked) matrix multiplication
(d) Efficiency for different methods(simple) in blocked matrix multiplication

Speedup v/s Number of cores analysis

We can see from the plots that the speedup increases with increase in the number of cores for most problem sizes. For low problem size like N=32 we observe the reverse trend. That is perhaps due to the fact that the overhead exceeds the speedup gained by optimization. We could also observe that the maximum speedup that can be achieved for a given number of threads is fair within the theoretical limit which is equal to P. We observed that for some of the problem size like 128 we get the maximum speedup and on further increasing the problem size the speedup decreases for the same number of threads. This could be due to proper cache utilization in case of lower problem size and increase in the parallel overhead because of the large problem size. We further observed that in blocked matrix multiplication case the best speedup was achieved for problem size 64 which is justified as it has lower parallel overhead and the blocksize of 32 makes it even better.



(a) (b) (c)

Figure 25:
(a) Speedup v/s Number of cores for unblocked basic parallelisation
(b) Speedup v/s Number of cores for transpose(unblocked) basic parallelisation
(c) Speedup v/s Number of cores for blocked basic parallelisation