Dhirubhai Ambani Institute of Information and Communication Technology

CS 301

HIGH PERFORMANCE COMPUTING

Parallel Implementation of Longest Common Subsequence

May 02, 2020

Assigned by:
Prof. Bhaskar Chaudhary

Submitted by:
Ruchit Shah
201701435
Niharika Dalsania
201701438

Contents

1 ABSTRACT

The Longest Common Subsequence has found application in varied domains ranging from file matching and identifying plagiarism to bioinformatics and computational genomics. Due to enormous data size and ever increasing need of computation, it is difficult to solve problems like LCS in limited amount of time using traditional sequential algorithm. To tackle this problem, parallel algorithms are implemented for the same. In this report we have presented optimized parallel LCS algorithm using OpenMP for multi-core system architectures. We have used several optimization techniques in our parallel algorithm like removing data dependency to ensure efficient parallelisation, improve cache usage and load balancing among the threads. We conclude that the implementation of our optimized parallel LCS algorithm on HPC cluster with 2 sockets having 6 cores per socket and 2 threads per per core using OpenMP with different scheduling policies achieves approximately 3.7 speedup factor with 8 threads over the conventional sequential algorithm approach.

2 INTRODUCTION

The LCS problem deals with comparing two sequences and finding the maximum length subsequence which is common to both. Such massive computations require parallel algorithms. Because of the spread of multithreading processors and the multicore machines in the marketplace, it is now possible to create parallel programs to solve such large scale problems. To perform parallel processing, we are using a shared memory API tool (OpenMP) which provides the various constructs that can be added to sequential programs written in C thus allowing to schedule the sub problems of dynamic programming effectively to processing cores for optimal utilization of multicore processors.

Dynamic programming is based on storing all intermediate results in a tabular form, so as to utilize it for further computations. Due to its amenable computational approach, this technique has been largely adopted for solving optimization problems like LCS. Here, a score matrix is filled through a scoring mechanism. The best score is the length of the LCS and the subsequence can be found by tracing back the table. Let m and n be the lengths of two strings to be compared. The time and space complexity of this dynamic programming approach is O(mn). The aim of the parallel approach is to reduce both of these.

This report is organized as follows. In Section 3, classical dynamic programming algorithm for LCS and some related works are introduced. In Section 4, the challenges faced in parallelisation and their solutions are presented. Section 5 shows the performance analysis. In Section 6, the final conclusions are summarized and future scopes are discussed.

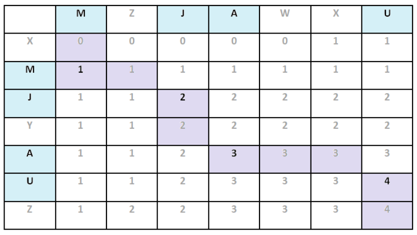
3 BACKGROUND

3.1 Classical Serial Approach

The LCS problem consists of an optimal substructure and overlapping sub problems, problems which have such properties can be solved using dynamic programming problem solving technique. The main concept of DP is to compute current state from previous state. Let X and Y be two given sequences of length m and n respectively and L[i,j] a value in the score matrix L computed using recursion equation mentioned below (check for i=0 and j=0 separately). L[i,j] denotes the length of LCS obtained so far by scanning till index i in string X and index j in string Y. Scanning through the score matrix gives us the required LCS.

$L[i,j] = L[i-1,j-1] + 1$ if $X[i-1] = Y[j-1]$
$L[i,j] = Max(L[i,j-1], L[i-1,j])$ if $X[i-1] \neq Y[j-1]$

L[m-1][n-1] is the length of required LCS. To compute the actual LCS, this algorithms does backtracking on the score matrix. Traverse the 2D array starting from L[m-1][n-1]. For every cell L[i][j], if characters in X and Y corresponding to L[i][j] are same (i.e X[i] == Y[j]), then include this character as part of LCS and move diagonally up to L[i-1][j-1] else compare values of L[i-1][j] and L[i][j-1] and go in direction of greater value. Taking an example, suppose string X is XMJYAUZ and string Y is MZJAWXU. Following is the score matrix for this case. Highlighted path shows backtracking to find LCS. Finding this LCS is of complexity O(m+n), which is lesser than the complexity O(mn) to actually calculate the entire score matrix.



	M	Z	J	A	W	X	U
X	0	0	0	0	0	1	1
M	1	1	1	1	1	1	1
J	1	1	2	2	2	2	2
Y	1	1	2	2	2	2	2
A	1	1	2	3	3	3	3
U	1	1	2	3	3	3	4
Z	1	2	2	3	3	3	4

Figure 1: Example Score Matrix

So, in DP technique main computational process to get the required LCS is the computation of the score matrix. For the two input sequence of size n, n^2 calculations are required to compute the score matrix . We aim to address this issue here. The proposed parallel algorithm speeds up the computation part of the score matrix by solving the data dependency issue hence allowing efficient parallelisation.

3.2 Potential challenges in parallelisation

While trying to parallelise the above mentioned serial algorithm, following challenges are anticipated:

- Removing **data dependency** before parallelising
- Ensuring **correctness** of result after loop skewing in parallel approach.
- Making most **optimal cache usage** for efficient data access
- Using proper scheduling constructs to ensure **work load balancing**
- **Synchronisation** among the threads

3.3 Tool used for parallelisation

In this report, the findings presented are using **OpenMP**, an Application Programming Interface used for parallelisation on the shared memory architecture falling under **Multiple Instruction Multiple Data (MIMD) category in Flynn's taxonomy**. It follows fork join model with various directives and functions for managing threads. Synchronization and work sharing via different types of scheduling are provided to manually divide the task among threads. The parallelisation approach used here explores all such features of OpenMP.

4 PARALLEL IMPLEMENTATION

4.1 Removing Data Dependency

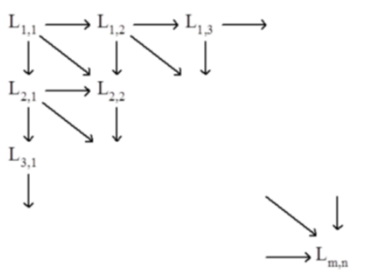


Figure 2: Data Dependency in score matrix

As mentioned in the serial approach, we see that there are data dependencies in same row and column while calculating the score matrix. While construction of score matrix using dynamic programming approach, the value of current element L[i,j] is depends on the three entries in the score matrix; L[i-1, j-1], L[i-1,j], L[i,j-1]. In other words, L[i,j] depends on data in the same column and same row which implies that we cannot compute the cells in the same row or column in parallel. To resolve this issue, we use **loop skewing technique**.

To compute the elements of score matrix in parallel, we start computing from L[1,1] then L[2,1] and L[1,2] at the same time and so on. We can see that computation of elements which are in the same diagonal can be done in parallel. To perform the computation of score matrix in parallel we are computing elements of score matrix in diagonal manner instead of row wise

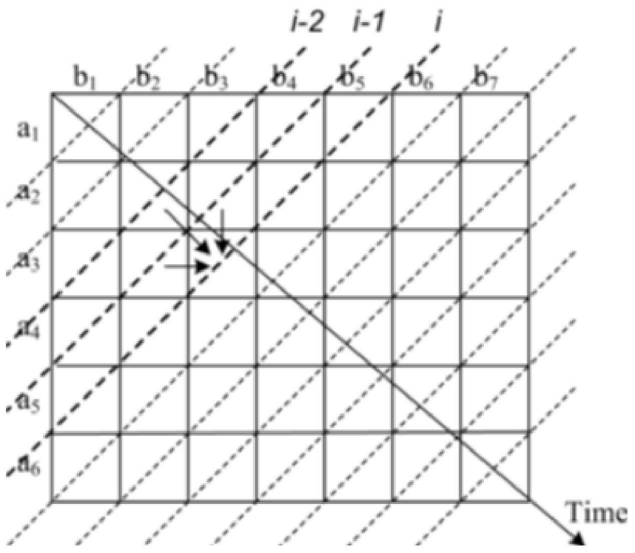


Figure 3: The Parallelisation Approach

For computation of elements in the same diagonal we have applied OpenMP constructs on the inner for loop. We have also restricted the number of threads equals to the number of cores in the machine to avoid the computation overhead. Pseudo-code for our proposed parallel algorithm can be given as follows:

```
ALGORITHM L(A,B)
INPUT: STRING A AND STRING B
OUTPUT: LCS OF A AND B
Begin
(1) Computation of elements of score matrix diagonally in parallel manner and maintaining the
    parent array to store the matched positions
(2) Printing LCS by backtracking score matrix
End
```

4.2 Making Optimal Cache Usage

Though the diagonal approach seems very appropriate for parallelisation, there is a problem with the way we access the data in this parallel implementation. Due to this, when we run the parallel version with one thread, it is slower than serial. This is because of usage of non-cache-friendly data structures. Because of the diagonal dependence, we walk the matrix by diagonals but we still store it the usual way. The data access pattern is then strongly non-linear and hence very cache-unfriendly. There will be a large amount of cache misses when running this code for big problem sizes. To tackle this issue, we store the score matrix as a 1D array such that the **diagonals are stored in linear fashion** in this. To access particular values from this 1D array, we created a mapping function that upon taking as input the indices i,j of the actual 2D matrix, it gives back the corresponding index in this newly formed 1D array. This allows us to use the same parallel algorithm as designed earlier, but with a cache friendly data access pattern.

But again, due to this mapping function called every time we wanted to access any element in the score matrix either to read it or write, there was a significant computational overhead that degraded the overall performance. Observing the pattern, we figured out that to get the length of LCS, it was enough to keep merely a part of the score matrix rather than the whole thing. This part was just the two previous diagonal to calculate the current diagonal. So it would be like (0,0) and (0,1),(1,0) for (0,2),(1,1),(2,0) and so on. Thus, since each diagonal required at most two earlier diagonals, we did not need a mapping function to map whole 2D score matrix into a 1D matrix, but rather use some basic index manipulations to access previous two diagonals. Keeping the values of each diagonal consecutively in memory, the access pattern going up the next diagonal is a linear progression along the previous diagonals which is great for the cache. At any time we **only keep 3 diagonal in memory**: the one we are working on and the two previous ones. We cycle between three such buffers so do not re-allocate memory all the time. Here we made sure to pre-allocate buffers with the maximum diagonal length. Having done this, this method gives improvement over the previous approach with 2D score matrix. However, the only drawback here is that since we are not storing the entire score matrix, there are no scope of backtracking so we cannot cannot find the actual LCS, but we get the length of LCS significantly faster.

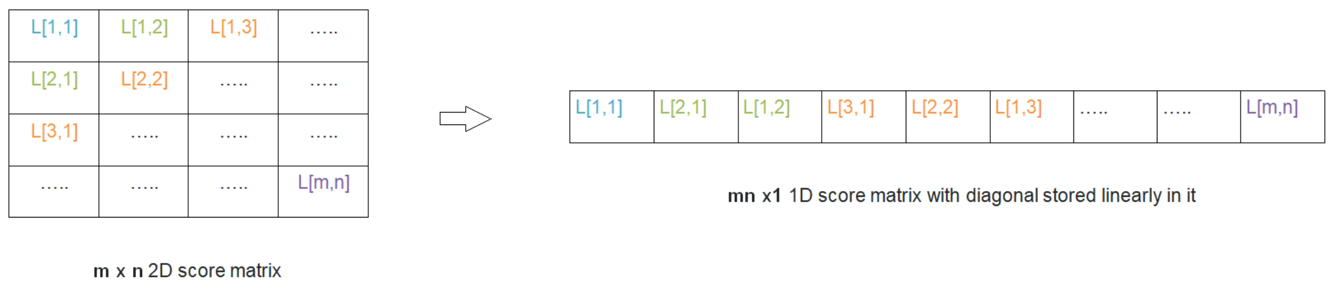


Figure 4: Cache Friendly Score Matrix

4.3 Using different Scheduling Constructs

Due to non-uniformity in the inherent dependence in dynamic programming algorithms, it becomes necessary to schedule the sub problems of dynamic programming effectively to processing cores for optimal utilization of multi-core technology. For the optimization of our parallel algorithm we have used the load balancing. We have divided the score matrix of LCS in three parts: **growing region, stable region and shrinking region** depending on whether the number of sub problems increases, remain stable or decreases uniformly phase by phase respectively.

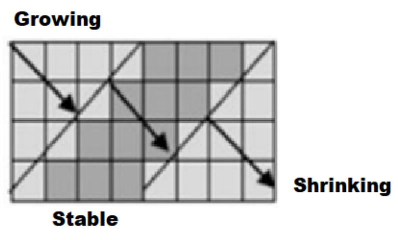


Figure 5: Different regions in score matrix

For each phase, the numbers of sub-problems are assigned to the threads which are handled by the chunk size parameter in OpenMP and finally threads execute those assigned sub-problems over physical cores which are handled by a scheduling policy in OpenMP. We have performed the experiments using three different scheduling policies - **static, dynamic and guided**.

The schedule(static, chunk-size) clause distributes the chunks to threads in a circular order. When no chunk-size is specified, OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread. The static scheduling type is appropriate when all iterations have the same computational cost. For big problem sizes, this property is by and large evident for all the regions where work distribution is fairly balanced. Also, this form of work distribution will help to exploit the **temporal and spatial locality** of the cache. Hence this scheduling construct works best for all regions.

With the schedule(dynamic, chunk-size) clause, each thread executes a chunk 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 dynamic scheduling type is appropriate when the iterations require different computational costs. This means that the iterations are poorly balanced between each other. The dynamic scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime. Since we do not have drastic imbalance in work load, this construct does not offer significant benefit due to its overhead.

Using the schedule (guided, chunk size) clause, the only difference with the dynamic scheduling type is that the size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases. The minimum size of a chunk is set by chunk-size. The guided scheduling type is appropriate when the iterations are poorly balanced between each other. The initial chunks are larger, because they reduce overhead. The smaller chunks fill the schedule towards the end of the computation and improve load balancing. This scheduling type is especially appropriate when poor load balancing occurs toward the end of the computation. With this property being observed in the shrinking region, this construct provides fairly good result if used there.

Based on these experimental, on a whole, **static scheduling is the best policy** in order to get the optimized results

5 RESULTS

5.1 Complexity Based Analysis

1. **Complexity of serial code**
   Time Complexity: O(mn) and Space Complexity: O(mn), where m and n are lengths of two input strings
2. **Complexity of parallel code**
   Ideal Time Complexity: O($\frac{mn}{p}$) and Space Complexity: O(3m) $\approx$ O(m), where p is the number of processors
3. **Cost of parallel algorithm**
   Ideal Cost = O($\frac{mn}{p}$) * p = O(mn)

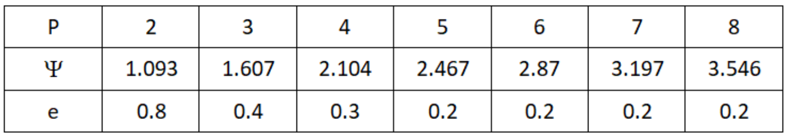
5.2 Performance Modelling

1. **Amdahl's Law**
   This is applied on serial code to analyse the scope of parallelism. Using Amdahl's law,
   Theoretical Speedup $\psi(n,p) \leq \dfrac{1}{s + \frac{1-s}{p}}$, where s is the serial fraction is the code that cannot be parallelised and p is the number of processors. Experimentally, s=$\dfrac{\sigma(n)}{\sigma(n)+\phi(n)}$, with $\sigma(n)$ is the computation time for inherently sequential portion and $\phi(n)$ is the computation time for parallelizable portion for a problem size n. However, in the original serial approach due to dependencies in loops, we cannot calculate $\phi(n)$, so we find the serial fraction from the modified code with removed dependencies that could than be parallelised. Here, for n=4000, with s = 0.020949 and p=8, we get $\psi(n,p) \leq 6.97$. However, this speedup is under very ideal conditions wherein just this one code is running on entire machine. This is impractical in reality, hence the theoretical speedup given is a tight upper bound of the achievable speedup. Also, the speedup

actually calculated considers the original serial code without any loop modification in which case speedup obtained is nearly 4, but after those modification to make the code parallelizable the number of computations obviously increases hence the speedup actually achieved here is close to 2. Also, Amdahl's law assumes that the problem size is fixed and shows how increasing processors can reduce time upto a certain limit. After that, by increasing number of processors further, gain from parallel computation is less than communication overhead, hence overall execution time increases. This is the **Amdahl's effect** and it helps to identify ideal number of processors for a particular problem. However, we can get increased speedup with increasing number of processors if we are allowed to increase the problem size.

2. **Gustafson Barsis's Law**
   Here, unlike Amdahl's law, the problem size can vary, but we fix the execution time. So this metric is used on the parallel code to analyse how efficiently the parallelisation works whereas Amdahl's law just discussed the scope of parallelisation. Given a parallel program of size n using p processors, let s be the fraction of total execution time spent in serial portion of code while parallel execution. So here s=$\dfrac{\sigma(n)}{\sigma(n)+\frac{\phi(n)}{p}}$. The theoretical speedup $\psi(n,p) \leq p + (1-p)s$. Here for n=4000 and p=8, s= 0.012627. This is lower than serial fraction in Amdahl's law beacuse as more computing resources become available, the time spent in the parallelizable part often grows much faster than the inherently serial work. This gives $\psi(n,p) \leq 7.96$ which is a looser bound than Amdahl's law. The difference between Amdahl's Law and Gustafson's Law lies basically in the application's objectives, whether to make the application run faster with the same problem size or run the application in the same time with increasing problem size. We here want to work on ever increasing problem size, hence Gustafson's Law gives a more relevant analysis.

3. **Karp Flatt Metric**
   Both Amdahl's law and Gustafson-Barsis's law ignore the parallelization overhead $\kappa(n,p)$ so they overestimate the achievable speedup. To identify whether or not it would be beneficial to parallellise the code further, we used this metric. Then experimentally determined serial fraction of parallel computation is given by e = $\dfrac{\sigma(n)+\kappa(n,p)}{\sigma(n)+\phi(n)}$. Some further manipulations give e = $\dfrac{\frac{1}{\psi}-\frac{1}{p}}{1-\frac{1}{p}}$. Following table shows how e changes with number of processors and speedup.



P	2	3	4	5	6	7	8
Ψ	1.093	1.607	2.104	2.467	2.87	3.197	3.546
e	0.8	0.4	0.3	0.2	0.2	0.2	0.2

Figure 6: Serial Fraction with Karp Flatt Metric

   Here e does not increase with respect to p meaning that parallelization overhead does not dominates the speedup. Gradually, e saturates to a constant meaning that the parallelization overhead is negligible. This means that further improvement by parallelisation is not possible. The lack in speedup is now due to the inherently sequential part only.

5.3 Curve Based Analysis

1. **Speedup with Diagonal Traversal in 2D score matrix**
   As can be seen from the figure below, as number of processors increase, the speedup increases. While parallelising with 1 or 2 threads, the parallelisation and communication overheads are such that it effectively gives no benefit over the serial code. But with 3 or more threads, after a problem size of around 2^{10}, we get speedup greater than 1. Now, around the problem size of 2^{13}, we observe a drop in speedup. This is because of the L1 cache size of 32K. Upto this size, we could fit multiple rows in cache, hence could get more than one element of diagonal at a time. But after this problem size, at max one row fits in cache so we get just one element from each diagonal, leading to cache misses and hence the speedup decreases.

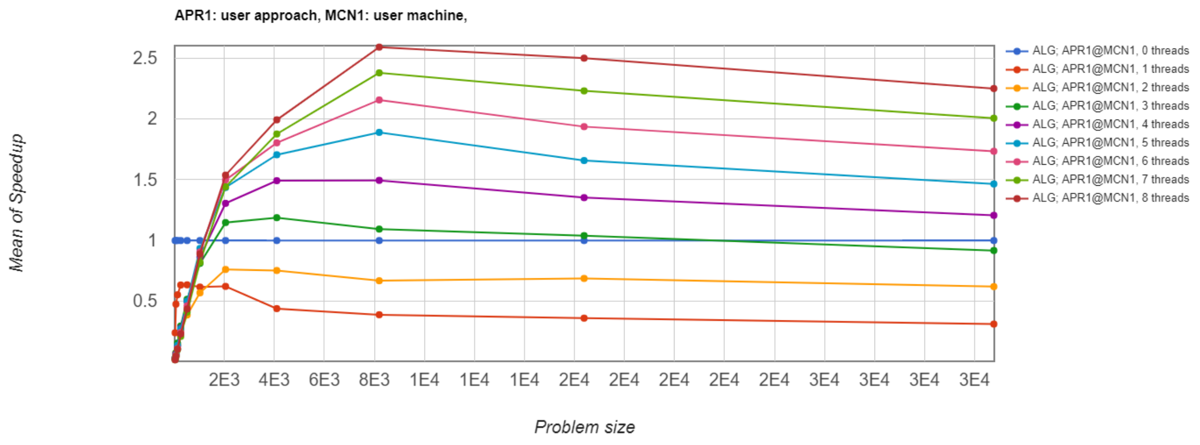


Figure 7

2. **Speedup with Cache Friendly 1D score matrix**
   In the figure below, we see the expected behaviour. For 1 thread, there is no gain in performance over serial and for any more number of threads we get speedup greater than 1. This speedup increases as number of processors and problem size increases, though not linearly due to communication overhead. Now here since we are not storing the entire 2D score matrix, but just the previous 2 diagonals, cache misses do not occur hence there is no drop in the graph.

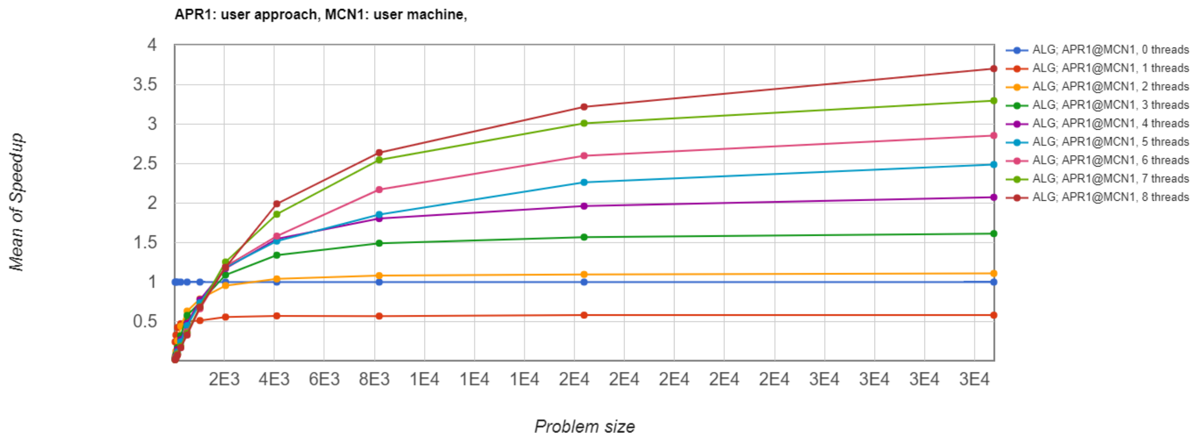


Figure 8

5.4 Additional Analysis

1. **Major serial and parallel overheads**
   The only overhead with the serial code is the amount of computation required to traverse through the entire score matrix to generate LCS because there is no track of matched positions. With the parallel code, several others emerge like the amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead includes thread creation and termination time, thread synchronizations, data communications, software overhead imposed by parallel languages, libraries, operating system, etc. However the rate of decrease in computation is higher than the rate of increase in communication with increasing number of processors upto a certain limit. Hence, we observe better performance with the parallel code.

2. **Compute to Memory Access Ratio (CMA)**
   Accessing the memory is expensive. So we want to perform more number of computations with the data fetched at once from the memory. This increases the CMA ratio. Assuming cache line size of 64 bytes and two input strings of size n and m:
   For serial code, number of computation and memory access is almost the same for very element in the score matrix, hence we get the following:
   Number of computations = $2nm$
   Number of Memory Accesses = $\frac{n}{64} + \frac{m}{64} + \frac{nm}{64}$
   Taking m=n, CMA = $\dfrac{2n^2}{\frac{n}{32} + \frac{n^2}{64}} \approx 128$ for large problem sizes.
   For parallel code, the exact number of computation is input dependent as per what condition block it enters. Assuming a fairly random string, taking average number of computation and memory access for each element in score matrix, we get the following:
   Number of computations = $6nm$
   Number of Memory Accesses = $\frac{n}{64} + m + \frac{1}{64}[\frac{(m)(m+1)}{2} + (n-m-1)m + \frac{(m)(m+1)}{2}]$
   Taking m=n, CMA = $\dfrac{6n^2}{\frac{n}{64} + n + \frac{n^2}{64}} \approx 384$ for large problem sizes. As we see, the **CMA increases for the parallel code**, which is desirable.

3. **Peak Performance**
   Peak performance (in GFlops) = Frequency (in GHz) * Instruction per cycle * Number of cores = (2.6)*(4)*(16) = 166.4 GFlops/s

4. **Code Balance**
   Code Balance = $\dfrac{\text{Data traffic}}{\text{Floating point operations}}$. This is the inverse of CMA ratio i.e. the computational intensity.
   Here, for parallel code Code Balance = 0.04167 Bytes/Flops

5. **Machine Balance**
   Machine Balance = $\dfrac{\text{Memory Bandwidth}}{\text{Peak performance}}$. Due to lack of admin rights, we could not find the exact memory bandwidth, but typical value lies around 0.06 words/Flops (with 8 bytes/word). In this case, with machine balance greater than code balance, the code is not bandwidth bound.

6. **Memory Wall analysis**
   Ideally we want machine balance close to 1, but it is very less than 1. Also historically this has been decreasing due to processor - memory performance gap i.e. the DRAM gap. Memory is often a bottleneck to achieving high performance. This limitation is the memory wall. If machine balance is less than code balance, then the machine cannot satisfy the data traffic requirements and performance is limited by memory bandwidth. However, machine balance is fixed with the architecture, but we can change the code balance with the algorithm. Hence we try reducing the code balance as much as possible i.e. increase the CMA as much as possible for better performance. This has been exploited with the cache friendly parallel approach that we took. Fetching data from the constant memory is not a high-latency operation hence generous caching improved performance.

7. **Synchronisation analysis**
   In the parallel approach, we are parallelising the inner for loop. There is an **implicit barrier associated with *pragma omp for***, which ensures that we move to the computation of next diagonal only when the current diagonal is completed. This guarantees thread synchronisation across diagonals.

8. **Granularity analysis**
   Granularity is the ratio of computation time to communication time. Depending on the amount of work which is performed by a parallel task, parallelism can be fine-grained, medium-grained or coarse-grained. In our approach, we are using **loop level parallelism** which is medium grained. This is optimal because although using fine grains or small tasks results in more parallelism and hence increases the speedup, but

synchronization overhead, scheduling strategies etc. can negatively impact the performance of fine-grained tasks. So increasing parallelism alone cannot give the best performance.To reduce the communication overhead, granularity can be increased. Coarse grained tasks have less communication overhead but they often cause load imbalance. Hence optimal performance is achieved between the two extremes of fine-grained and coarse-grained parallelism.

9. **Scalability analysis**
For a given problem size, as we increase the number of processors, the overall efficiency of the parallel system goes down. A scalable parallel system is one in which the efficiency can be kept constant as the number of processing elements is increased, provided that the problem size is also increased. The rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed determines the degree of scalability of the parallel system. Isoefficiency metric helps to quantitatively determining this degree. A lower rate is more desirable than a higher growth rate in problem size. The parallel approach used here shows that on increasing the number of threads, the **efficiency remains almost constant for the same problem size**. Hence it is a **strongly scalable code**.

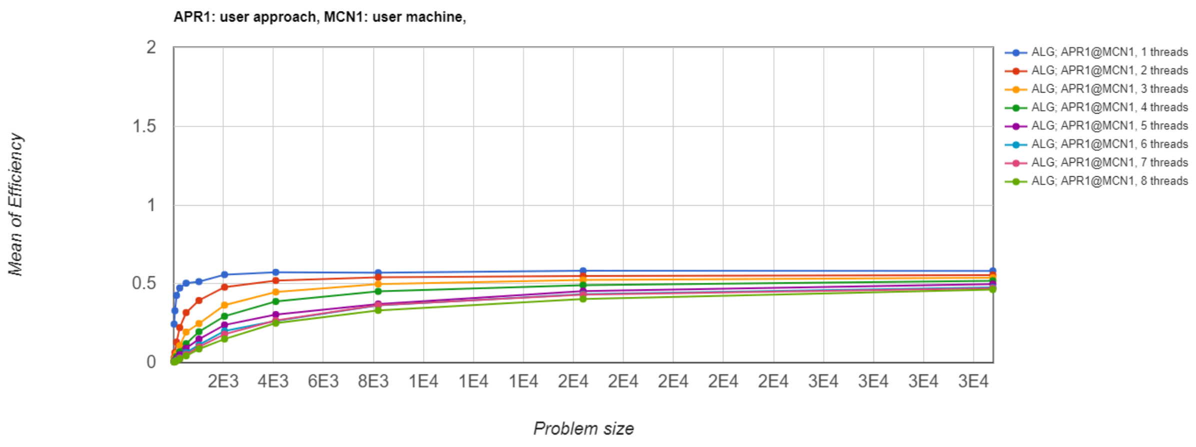


Figure 9

6 CONCLUSION

Problem of LCS has found variety of applications in fields of bioinformatics, pattern recognition and data mining. For the problems having large size input, parallel algorithms using OpenMP are one of the best ways to solve these problems. In this report we have presented an Cache Optimized Parallel algorithm using OpenMP for Multi-Core processors which was considerably faster than the sequential LCS algorithm. We also concluded that static scheduling works best in this case. With analysis of several metrics we showed that this is very close to the maximum achievable performance improvement. In future we can extend this algorithm to support Multiple Longest Common Subsequence also.

7 REFERENCES

1. https://www.irjet.net/archives/V3/i6/IRJET-V3I6183.pdf

2. https://stackoverflow.com/questions/13371122/why-is-this-parallel-function-for-computing-the-longe

3. https://moodle.daiict.ac.in/pluginfile.php/13425/mod_resource/content/1/module-2-final.pdf