HIGH PERFORMANCE COMPUTING

Parallel Implementation of Longest Common Subsequence

Ruchit Shah (201701435)

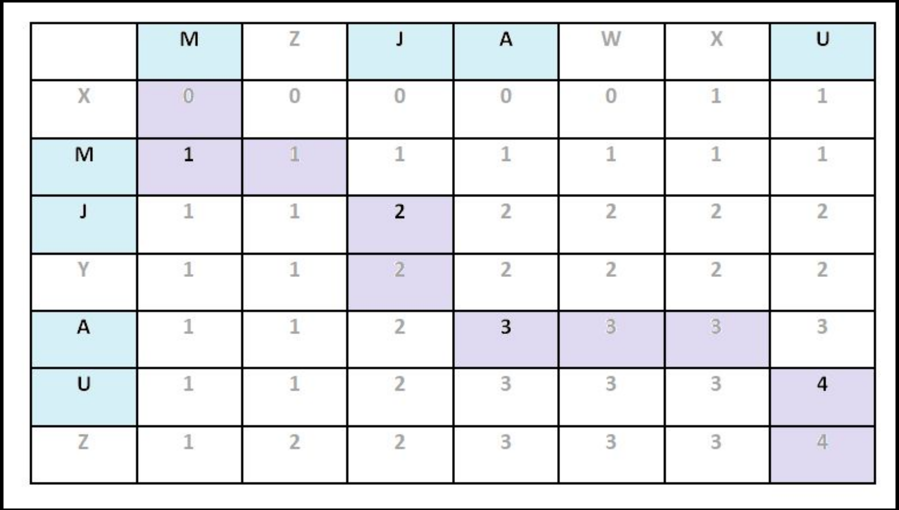
Niharika Dalsania (201701438)

Serial Approach

For input strings X of length m and Y of length n, make a m x n score matrix L and fill it as follows. 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.

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

L[m-1][n-1] is length of required LCS. To find LCS use backtracking on the score matrix starting from L[m-1][n-1]. For every cell L[i][j], if X[i]=Y[j], then include this in 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.



	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

String X: XMJYAUZ

String Y: MZJAWXU

LCS: MJAU with length 4

Highlighted path shows backtracking to find LCS.

Scope of improvement in serial approach

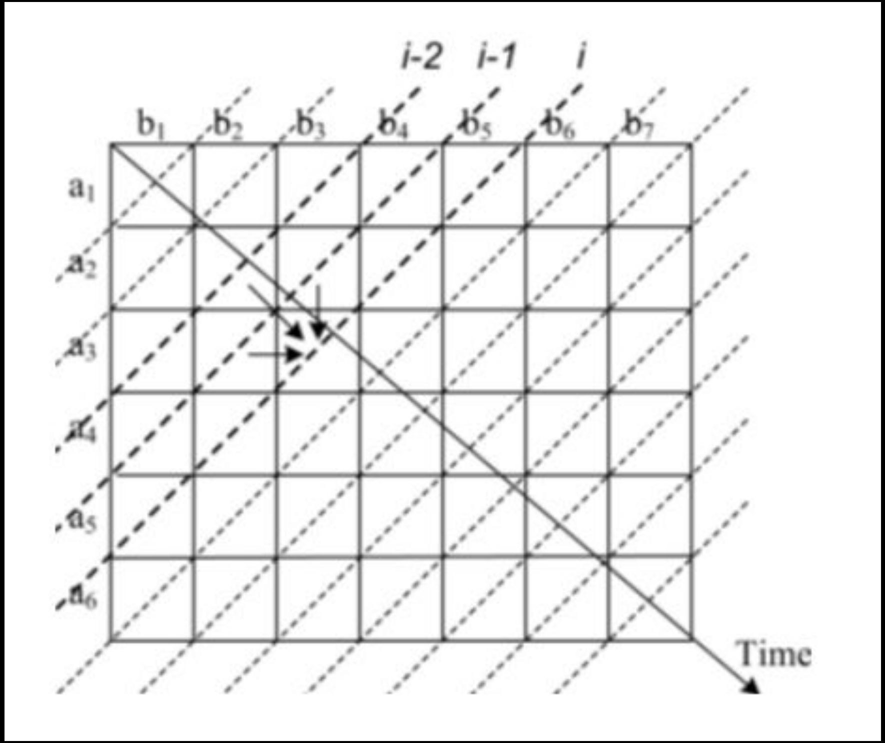
In DP technique, main computational process to get the required LCS is the computation of the score matrix. For the two input sequence of size n, n^2 calculations are required to compute the score matrix . We can parallelise this portion.

Challenges in parallelisation

1. Removing data dependency
2. Ensuring correctness after loop skewing
3. Making optimal cache usage
4. Use proper scheduling constructs for work load balancing
5. Synchronisation among threads

Parallel Approach - Diagonal

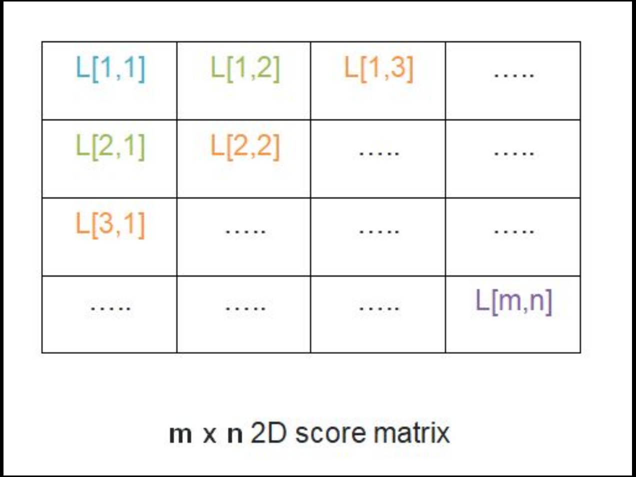
In serial approach, there are data dependencies in same row and column so we cannot compute the cells in the same row or column in parallel. To resolve this issue, we use loop skewing technique. To compute the elements of score matrix in parallel, we start computing diagonally from L[1,1] then L[2,1] and L[1,2] at the same time and so on because all elements on one diagonal are independent of each other and there is dependency only across the diagonals. So diagonals are computed one after another and inner for loop is fully parallelised.



Computation of elements which are in the same diagonal can be done in parallel.

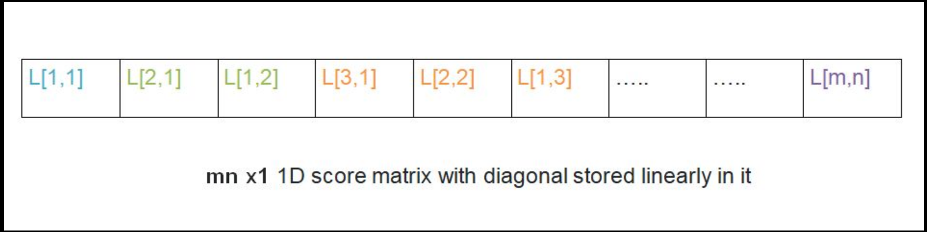
Parallel Approach - Cache Friendly

In diagonal approach, there is a problem with data access pattern due to usage of non-cache-friendly data structures. 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. So we store the score matrix as a 1D array such that the diagonals are stored in linear fashion. To access particular values from this 1D array, we created a mapping function between indices of 2D and 1D array.



L[1,1]	L[1,2]	L[1,3]	.....
L[2,1]	L[2,2]	.....	.....
L[3,1]	.....	.....	.....
.....	.....	.....	L[m,n]

m x n 2D score matrix



L[1,1]	L[2,1]	L[1,2]	L[3,1]	L[2,2]	L[1,3]	.....	.....	L[m,n]

mn x1 1D score matrix with diagonal stored linearly in it

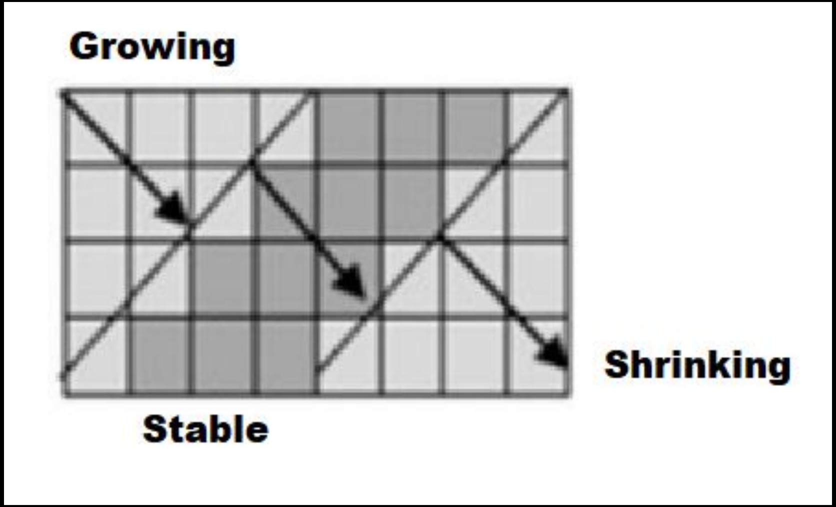
1D score matrix with diagonals stored linearly is cache friendly with efficient data access pattern

Parallel Approach - Cache Friendly and Computationally less intensive

But due to this mapping function called every time we wanted to access any element in the score matrix, there was a significant computational overhead that degraded the overall performance. It was enough to keep just the two previous diagonal to calculate the current diagonal. Then 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. So, 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. This method gives significant improvement over the previous approach with 2D score matrix. The only drawback here is that since we are not storing the entire score matrix, we cannot backtrack to find the actual LCS. However, we get the length significantly faster.

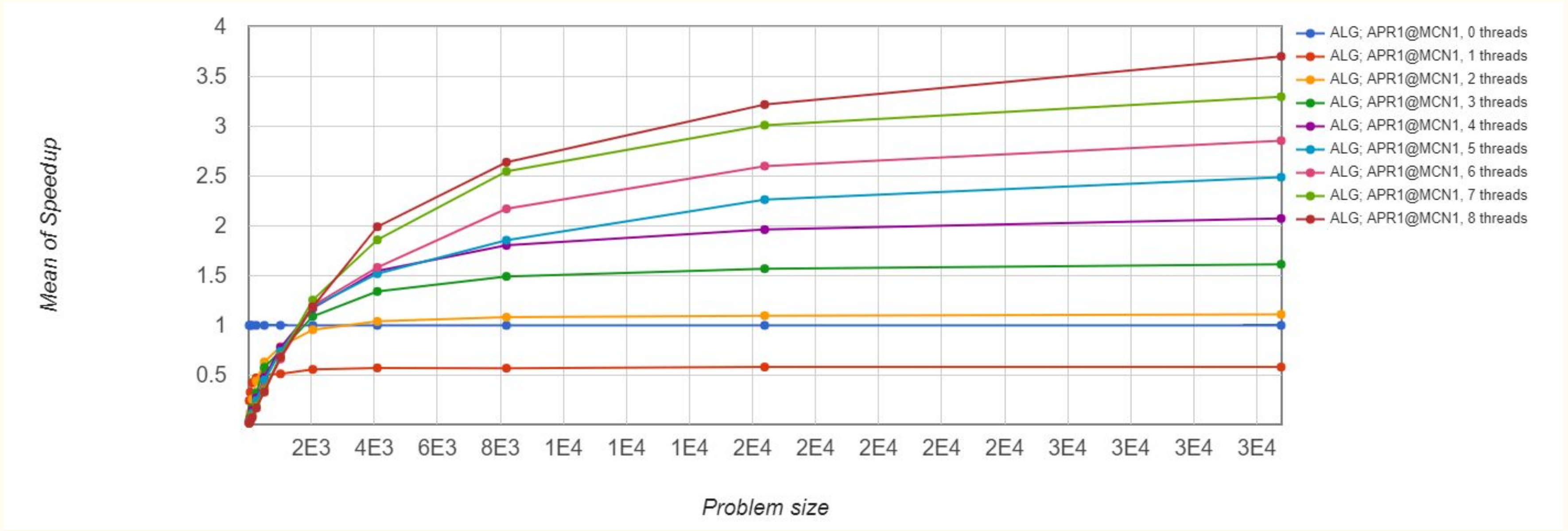
Different scheduling constructs

We divided the score matrix in three parts: growing region, stable region and shrinking region. We used three different scheduling policies - static, dynamic and guided - on each region. Static scheduling worked best for all regions because all subproblems have the same computational cost and work distribution is fairly balanced. Also, this form of work distribution will help to exploit the temporal and spatial locality of the cache for each thread better than random dynamic scheduling.



Score matrix divided into 3 different regions to observe effects of different types of scheduling on each region

Speedup Curve



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.

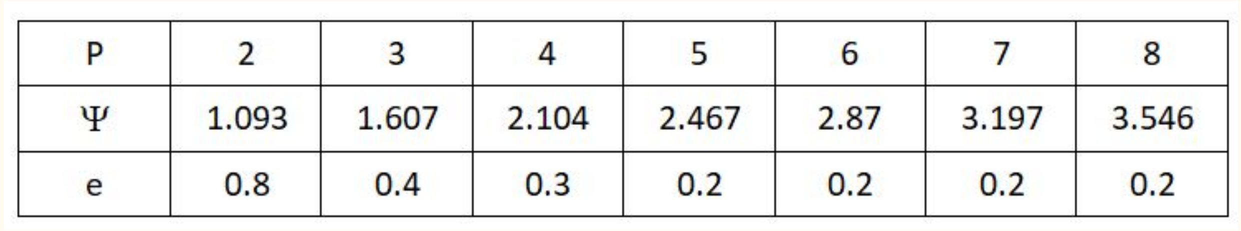
Performance analysis and Important conclusions

Amdahl’s Law - This is applied on serial code. Using the formula for n=4000 with serial fraction s = 0.020949 and p=8, we get theoretical speedup 6.97. But this is under very ideal conditions which is impractical in reality. 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.

Gustafson Barsis Law - This is applied on parallel code. Using the formula for n=4000 with serial fraction s= 0.012627 and p=8, the theoretical speedup is 7.96 which is a looser bound than Amdahl's law.

Performance analysis and Important conclusions

Karp Flatt Metric - The experimentally determined serial fraction gradually saturates which means that no further parallelisation is beneficial. Lack in speedup is due to the inherently sequential portion of the code and not due to the parallelisation overhead.



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

CMA - For large problem size, on an average the Compute to Memory Access ratio for serial code is 128 and for parallel code is 384. We ideally want the CMA higher because that means more number of computations per memory access which is desirable since accessing main memory is expensive. So the parallel code performs better as expected.

Performance analysis and Important conclusions

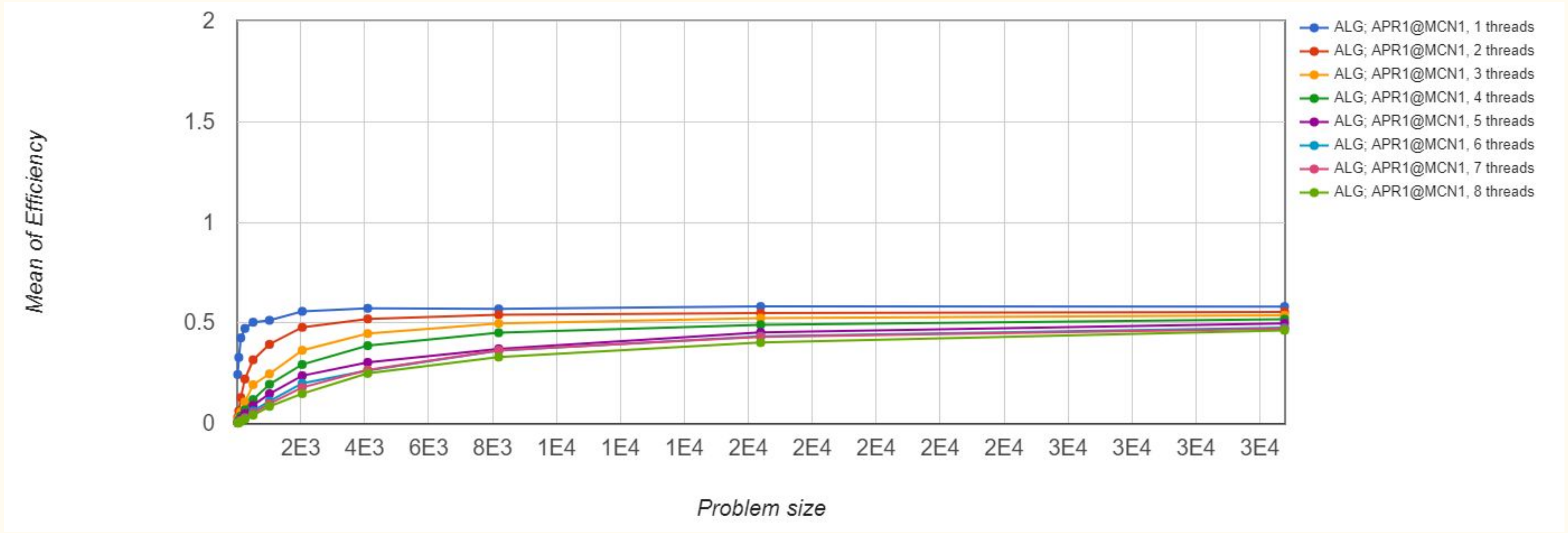
Parallel Overheads - Thread creation and termination time, thread synchronizations, data communications, software overhead imposed by parallel languages, libraries, operating system, etc.

Synchronisation - The implicit barrier associated with pragma omp for ensures synchronisation among threads across diagonals. There are no other critical regions.

Granularity - Loop level parallelism is used here which is medium grained. Optimal performance is obtained between fine grained and coarse grained because of both reduced communication overhead and load imbalance

Performance analysis and Important conclusions

Scalability - For given problem size, as we increase number of processors, overall efficiency of parallel system goes down. 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.


Mean of Efficiency
2E3
4E3
6E3
8E3
1E4
1E4
1E4
2E4
2E4
2E4
2E4
2E4
3E4
3E4
3E4
3E4
Problem size
ALG; APR1@MCN1, 1 threads
ALG; APR1@MCN1, 2 threads
ALG; APR1@MCN1, 3 threads
ALG; APR1@MCN1, 4 threads
ALG; APR1@MCN1, 5 threads
ALG; APR1@MCN1, 6 threads
ALG; APR1@MCN1, 7 threads
ALG; APR1@MCN1, 8 threads