Performance Analysis of LCS using Parallel algorithms

Krishna Kabi
kkabi004@ucr.edu

Sheenam Gupta
sgupt068@ucr.edu

Introduction

Longest Common Subsequence is used in a wide variety of applications for similarity comparison between two objects. This algorithm has seen application in bioinformatics, speech and image comparison, file comparison or source control such as git difference between versions. In the field of bioinformatics, common patterns help to discover the relation between two gene sequences. And we would like to use datasets from this domain to find the longest common subsequence between two DNA sequences. Which require use of more processing power.

Until now CPUs were used to carry out such computations, and the algorithms used were mainly serialized algorithm i.e running on one CPU which takes a lot of time. However, these algorithms can be easily parallelized.

With the advent of multi core CPUs it is now possible to run parallel algorithms on parallel processors thus reducing the processing time significantly.

However, we know Graphics Processing Units (GPUs) have come up as major players in parallel computing. Using GPUs, we can use thousands of threads to do parallel calculations and thus making the process much faster. The General-Purpose Computation on Graphic Processing Unit (GPGPU) is a new low-cost technology which take advantage of the GPUs to perform massively parallel calculation, traditionally executed on multi-cores CPU.

In this project we have provided run time comparison of parallel LCS algorithm on multi-core CPU and GPU using different programming application such as openMP, CUDA and Numba and optimized LCS algorithm in Numba.

Motivation

With a naive recursive approach, the time complexity of this algorithm is $O(2^n)$. Using dynamic programming reduces it to O(n*m) where n and m are the lengths of two sequences. With increasing sequence length execution time can be a bottleneck. We aim to implement dynamic programming in a parallel approach to help reduce the execution times significantly.

Goals

- Implementing sequential LCS algorithm using dynamic programming on CPU
- Implementing parallel LCS algorithm using dynamic programing with anti-diagonal approach using CUDA and Numba on GPU and OpenMP on CPU.

- Implementing optimized LCS using Numba on GPU.
- Comparing respective run times of parallel (GPU) with sequential (CPU) algorithm.

Data Collection:

The data used for evaluation is fetched from a DNA sequence generator [3] and from the website of National Center for Biotechnology Information(NCBI) [4].

Method

1. Anti-diagonal approach

To define antidiagonal approach let us consider few assumptions. The first Sequence A varies from 1 to m and second sequence B varies from 1 to n in length of characters. The LCS algorithm using dynamic programming is given by equation (1).

$$S(i,j) = \begin{cases} 0 & if\ \ i = 0\ or\ j = 0 \\ S(i-1, j-1) + 1 & if\ ai = bj \\ max\big(S(i,j-1), S(i-1,j)\big) & otherwise \end{cases} \qquad \text{------ (1)}$$

The antidiagonal approach calculates the same matrix as explained in equation (1) but instead of processing sequentially for each row and column we calculate each cell diagonally from left to right.

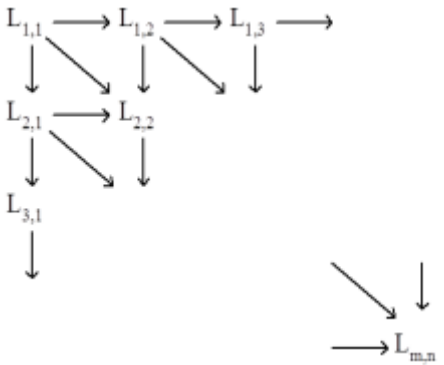


Fig 1: Data dependency in the score (Credit: [1])

Figure 1 explains the data dependency between each cell. So, each current cell gets value from previous row and column. The number of iterations that the kernel runs is equal to the sum of length of both sequences minus one. On each iteration, the kernel is called from CPU and we time this execution for evaluation. Figure 2 shows how the algorithm is implemented.

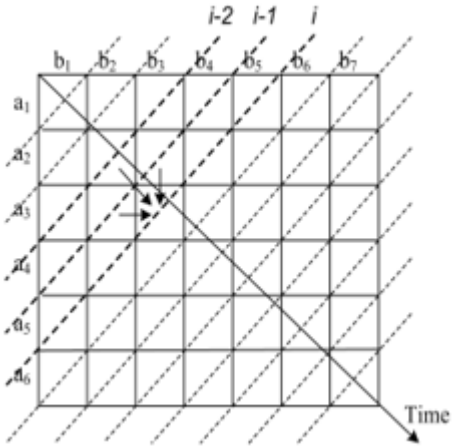


Fig 2: The parallelization approach (Credit: [1])

2. Efficient Parallelization

The anti-diagonal approach using dynamic programming has a limitation in terms of parallelization. Although it takes advantage of calculating the diagonal cells parallelly, because of the independence of diagonal cells, it however must do that sequentially. So, although the data on same diagonal can be calculated in parallel, the number of cells on each cell is different and varies from 1 to max(m,n). For instance, the first element in the diagonal has only 1 cell to calculate compared to the main diagonal which has maximum threads utilized. Therefore, the thread's distribution is non-uniform which leads to significant warp divergence. A minor change in how we take care of data dependence can solve this problem. [2] explains this optimization by being able to calculate the elements in the same row parallelly. So, each thread block assuming full assignment across the row, would calculate the value at the same time. It suggests changing the data dependence. So, now we initially calculate a P matrix followed by Score(S) matrix.

The P matrix is calculated on the CPU side, which depending on no. of unique characters is calculated by equation (2). The no. of unique characters in a DNA sequence is 4 (i.e ATGC). To achieve more parallelism, this calculation of P is implemented using multiprocessing on CPU by assigning 4 threads for each character.

$$P[I,j] = \begin{cases} 0 & if \;\; j = 0 \\ j & if \;\; b(j-1) = C[i] \\ P[I,j-1] & otherwise \end{cases} \qquad \text{------ (2)}$$

Following calculation of P matrix, the i^{th} row score matrix (S) is calculated by taking values from just $(i\text{-}1)^{th}$ row given by following equation (3).

$$S(i,j) = \begin{cases} 0 & if \; i \vee j = 0 \\ max(S[i-1,j],0) & if \;\; P[c,j] = 0 \\ max(S[i-1,j], S[i-1,P[c,j]-1]+1) & otherwise \end{cases} \qquad \text{-------- (3)}$$

Approach:

We ran parallel LCS algorithms on CPU using openMP and on GPU using CUDA and Numba. We ran the parallel algorithm using input sequence size of 2k, 4k, 8k, 10k, 20k, 30k and 40k

1. **OpenMP**

   OpenMP is an implementation of multithreading, a method of parallelizing whereby a primary thread (a series of instructions executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

   We used OpenMP to run the parallel LCS algorithm with different thread numbers on multi core CPU. The command to run the code in 4 processors is as shown below.

   ```
   export OMP_NUM_THREADS=4

   g++ -fopenmp lcs_openMP.cpp -o lcs_openMP.out

   ./lcs_openMP.out lcs_input1.txt
   ```

2. **Numba**

   Numba is an open-source JIT compiler that translates a subset of Python and NumPy into fast machine code using LLVM, via the llvmlite Python package. It offers a range of options for parallelizing Python code for CPUs and GPUs, often with only minor code changes.

   We used Numba to run parallel LCS algorithm and efficient parallel LCS algorithm on GPU. The command to run the code is as shown below:

Required fields:	Optional fields:
--SeqA/seqB two input sequence	--only_valid: Run LCS only in CPU
--TB Thread block count	--valid: Run for both GPU and CPU

   ```
   python lcs_antidiag.py --seqA ../data/Seq2_200.txt --seqB ../data/Seq1_200.txt --TB 512 --only_valid True

   python optimized_lcs.py --seqA ../data/Seq1_50K.txt --seqB ../data/Seq2_50K.txt --TB 512 --valid True
   ```

3. **CUDA**

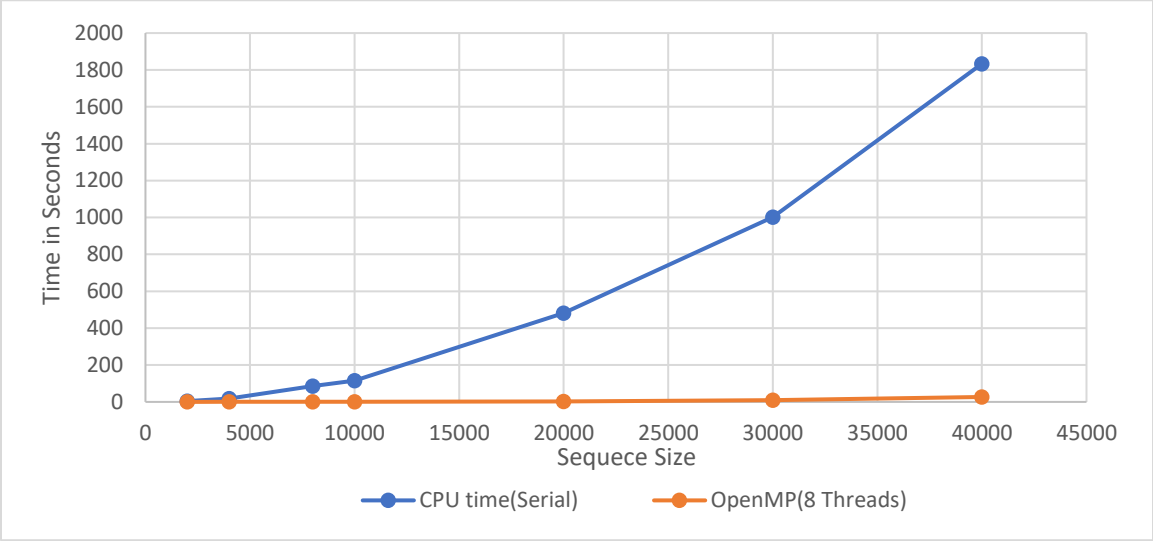
   CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing unit (GPU) for general purpose processing – an approach called general-purpose computing on GPUs (GPGPU).

We used CUDA to run parallel LCS algorithm on GPU. The command to run the code is as shown below:

nvcc -g -G -arch=sm_52 -o lcs_res lcs.cu

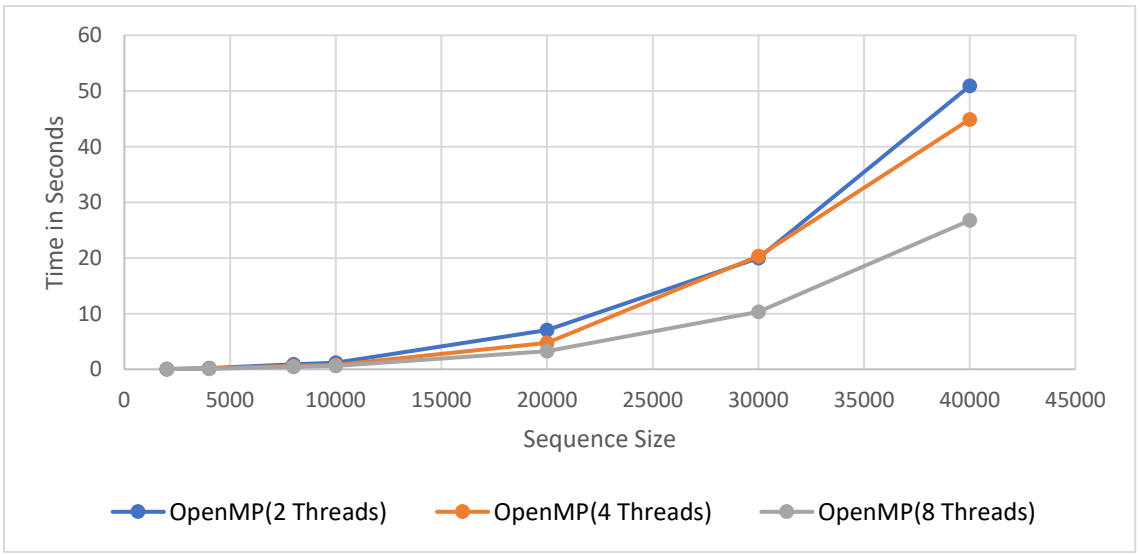
Comparisons:

1. Serial CPU time vs OpenMP (8 threads)



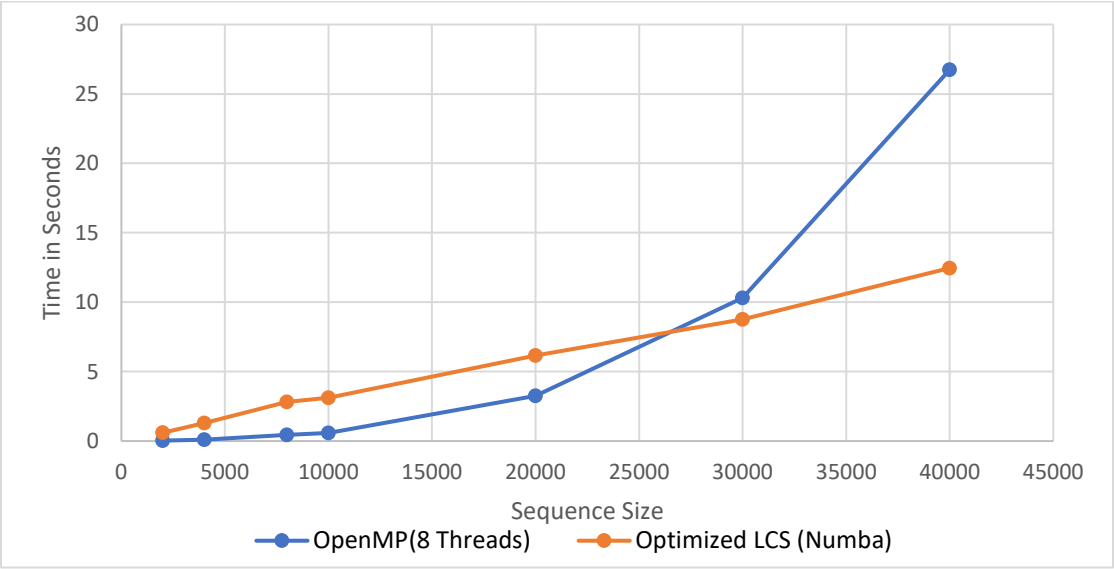
The above comparison shows that with serial LCS algorithm running on single CPU core, the run time is very high as compared to running parallel LCS algorithm using 8 threads. It gives us a speed up of 68.56 when compared with 40K input sequence.

2. OpenMP comparison (2 threads, 4 thread, 8 threads)



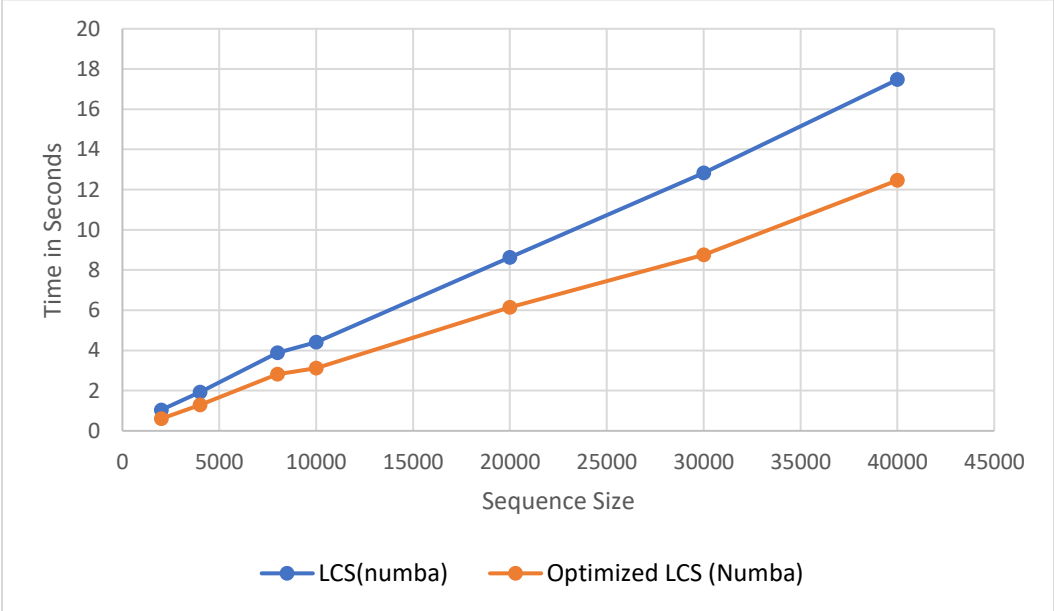
The above comparison shows that as the number of threads increase for parallel LCS code on openMP the run time of algorithm reduces significantly. There is a speedup of almost 2 times when running with 2 threads and 8 threads, when running on 40k sequence.

3. OpenMP (8 threads) vs Numba optimized parallel algorithm



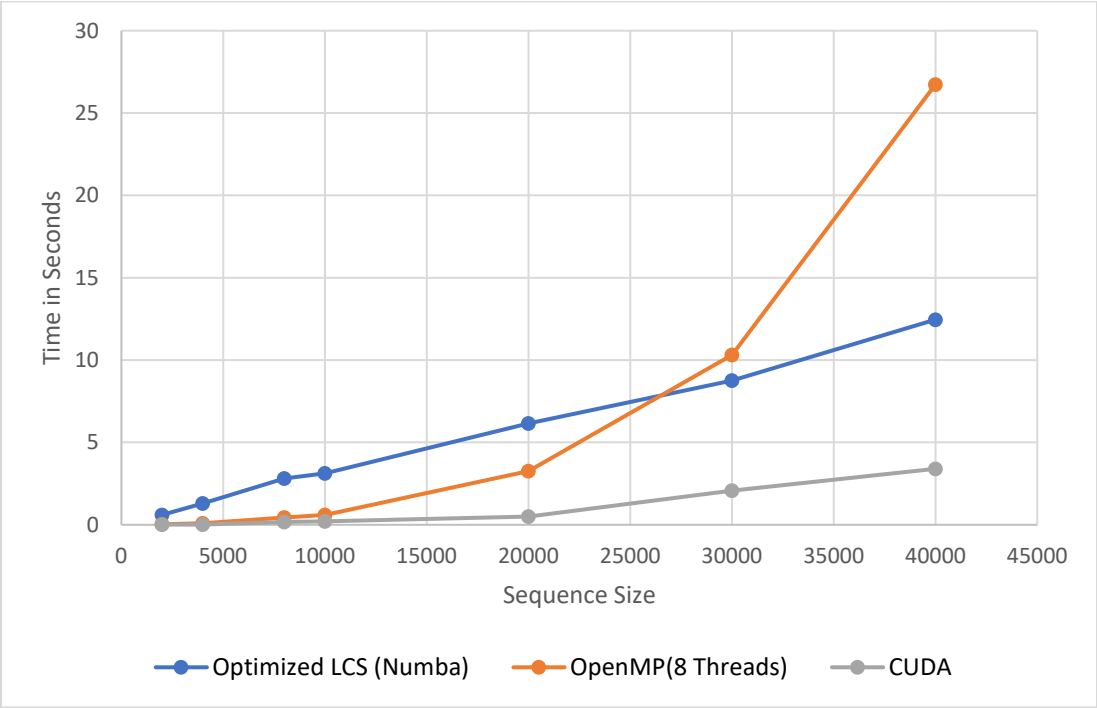
The above comparison shows that as the number of sequences increases Numba performs better than OpenMP with 8 threads, with speed up of more than 2 times when running on 40k sequence.

4. Numba optimized parallel algorithm vs Numba parallel algorithm



The above comparison shows that the optimized parallel LCS algorithm runs faster than anti-diagonal parallel LCS algorithm giving a speedup of almost 1.5 times when running on 40K sequence.

5. CUDA, Numba optimized parallel algorithm and OpenMP (8 threads)



In the above comparison it shows that as the sequence size increases Numba and CUDA perform better that OpenMP. Also, CUDA performs a lot better than Numba when running on GPU and provides a speed up of 4 times as compared to Numba when and CUDA is 8 times faster than openMP running on 40k sequence.

Results:

We measure the speedup that is achieved using parallel algorithm is OpenMP on CPU and CUDA and Numba on GPU.

Speed up can be defined as:

$$Speedup = \frac{Sequential \;\; executiontime}{execution \;\; time}$$

In case of GPU, we initially see the CPU is faster for lower range sequence.

This is due to scheduling of threads which delays the execution time overall. But beyond sequence of 800 input sequence size, we see GPU outperforms CPU. There is also significant change in execution time of CUDA. In C compared to Numba, CUDA performs 4 times better than Numba when running on 40K input sequence.

Status of your project:

The project implementation is complete. We have taken into consideration both square and rectangular matrices i.e sequences of different length can be processed successfully.

Task Breakdown:

Task	Breakdown
Open MP	Sheenam Gupta: 100%, Krishna Kabi: 0%
CUDA C LCS	Krishna Kabi: 70%, Sheenam Gupta: 30%
Numba LCS	Krishna Kabi: 100%, Sheenam Gupta: 0%
Numba optimized LCS	Krishna Kabi: 100%, Sheenam Gupta: 0%
Report	Sheenam Gupta: 80% Krishna Kabi: 20%

Conclusion:

We have focused in this project on the parallelization of a dynamic programming algorithm for solving the LCS problem.

For this purpose, we have studied some languages for parallel development on GPU (CUDA and Numba). Then, we have provided a parallelization approach and an optimized parallelization approach for solving the LCS problem on GPU. We have evaluated our proposed algorithm on an NVIDIA GPU using CUDA, Numba and on CPU using the OpenMP API.

The results have shown that the algorithm enables higher degree of parallelism and achieves a good speedup on GPU.

In addition, during this experiment, we have seen that CUDA is more suitable for NVIDIA devices than Numba.

References

[1] A. Dhraief, R. Issaoui, and A. Belghith, "Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability," in Proceedings of The First International Conference onAdvanced Communications and Computation (INFOCOMP), 2011.

[2] Jiaoyun Yang, Yun Xu, Yi Shang, "An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs", World Congress Engineering, 2010.

[3] http://www.faculty.ucr.edu/~mmaduro/random.htm

[4] https://ftp.ncbi.nlm.nih.gov/genomes/