FALL SEMESTER 2020 - '21

CSE4001 - Parallel and Distributed Computing
Slot C2

PROJECT REPORT

Parallelization of Local (Gene) Sequence Alignment

Made by -
TEAM 7
19BCE0001 - Roshini Thangavel
18BCE0272 – Nitin Pramod Ranjan
18BCE2017 – Shaurya Singh

Under the guidance of
Prof. Manoov R,
SCOPE,
VIT

Video Link for review 3: [Review 3]
For code briefing, please refer review 2: [pdc.mp4]

Abstract

Smith-Waterman Algorithm is the most sensitive known algorithm for local sequence alignment. It finds extensive use in computational biology, natural language processing and financial sciences. However, owing to its quadratic time complexity, it becomes an algorithm seldom used in its exact form. Several modifications including BLAST have been suggested for improving time complexity, however, that is often at the cost of quality. The authors have tried to study the sequential as well as parallel implementation of the SWA in openMP, MPI and in hybrid form under standard and accelerated hardware conditions in CUDA.

Introduction

Searching databases of DNA and protein sequences is one of the fundamental tasks in bioinformatics. The Smith-Waterman algorithm guarantees the maximal sensitivity for local sequence alignments, but it is slow. It has quadratic complexity in time and a linear complexity in space. It should be further considered that biological databases are growing at a very fast exponential rate, which is greater than the rate of improvement of microprocessors. This trend results in longer time and/or more expensive hardware to manage the problem. Special purpose hardware implementations, as for instance super-computers or field programmable gate arrays (FPGAs) are certainly interesting options, but they tend to be very expensive and not suitable for many users.

For the above reasons, many widespread solutions running on common microprocessors now use some heuristic approaches to reduce the computational cost of sequence alignment. Thus a reduced execution time is reached at the expense of sensitivity. FASTA (Pearson and Lipman, 1988) and BLAST (Altschul et al., 1997) are up to 40 times faster than the best known straight forward CPU implementation of Smith-Waterman. However, both of them reduce time consumption at the cost of accuracy. They are used with the conception that if run several times, the most optimal sequence shall be eventually deduced and this shall take less time than running the CPU implementation of Smith-Waterman to achieve the same level of accuracy.

A number of efforts have also been made to obtain faster implementations of the Smith-Waterman algorithm on commodity hardware. Farrar exploits Intel SSE2, which is the multimedia extension of the CPU. Its implementation is up to 13 times faster than SSEARCH (a quasi-standard implementation of Smith-Waterman).

To our knowledge, the only previous attempt to implement Smith-Waterman on a GPU was done by W. Liu et al. (2006). Their solution relies on OpenGL that has some intrinsic limits as it is based on the graphics pipeline. Thus, a conversion of the problem to the graphical domain is needed, as well as a reverse procedure to convert back the results. Although that approach is up to 5 times faster than SSEARCH, it is considerably slower than BLAST.

There are many existing tools for sequence alignment. Among those, FASTA2 and BLAST3 are two commonly used ones, where the time complexity has been reduced through some heuristic algorithms. These heuristics algorithms obtain efficiency, however, at the expense of sensitivity. As a result, a distantly related sequence may not be found in a search using the above tools.

Researchers have been working on this issue through different approaches. For example, Fa Zhang, XiangZhen Qiao, and Zhi-Yong Liu presented a parallel Smith-Waterman algorithm based on divide and conquer that can reduce running time and memory requirement. However, their method is also at the cost of losing sensitivity.

Other methods that apply standard computer systems such as high performance supercomputers and computer clusters with software solutions for conducting the Smith-Waterman algorithm, although can achieve high sensitivity and reduce running time, are with extremely high cost. With the advance technology in the FPGA, a cost-efficient parallel implementation for the Smith-Waterman algorithm can be obtained.

Related Terms

1. **PAM Matrices:**
   - PAM is 'Point Accepted Mutations'.
   - Are calculated by observing the differences in closely related proteins.
   - One PAM unit (PAM1) specifies one accepted point mutation per 100 amino acid residues or between protiens that have only 1% change.

2. **BLOSUM:**
   - BLOcks SUbstitution Matrix
   - Represent actual percentage identity values i.e. similarity. Blosum 54 means there is 54 % similarity.
3. **Gap score or gap penalty**:
   - Algorithms use gap penalties to maximize the biological meaning i.e. to see mutations or breaks in evolution.
   - Gap penalty is subtracted for each gap that has been introduced.
   - There are different gap penalties such as gap open and gap extension. The gap score defines a penalty given to alignment when we have insertion or deletion. Gap open and gap extension has been introduced when there are continuous gaps (five or more).
   - The open penalty is always applied at the start of the gap, and then the other gaps following it is given with a gap extension penalty which will be less compared to the open penalty.
4. **Assumed scoring schemas:**
   - If the residues are the same in both the sequences being compared, the match score is assumed (Si,j) as +p(say, +5) which is added to the diagonally positioned cell of the current cell (i, j position).
   - If the residues are not the same, the mismatch score is assumed as -q(say, -4).
   - These scores are generally arbitrary.

Smith Waterman Algorithm

A simple way to look at the algorithm is through theses three steps:
1. Initializing the matrix.
2. Filling the matrix with scores using the assumed scoring schema and gap penalties.
3. Trace back the most suitable sequence i.e. one with the highest score.

Related Work

1. A Parallel Pairwise Local Sequence Alignment Algorithm (Sanghamitra Bandyopadhyay, Ramkrishna Mitra, *May 2009, IEEE transactions on nanobioscience*)

In this paper researchers have used a heuristic-based pairwise sequence alignment tools instead of Smith-Waterman (SW) algorithm. Because of this change, it led to a significant loss of sensitivity. Although parallelization was considered as a possible solution, it is restricted to database searching through database fragmentation. The power of a cluster computer was also utilized for developing a parallel algorithm, RPAlign, which involved the detection of regions that are potentially alignable, followed by their actual alignment. RPAlign was also found to reduce the timing requirement by a factor of upto 9 and 99 when used with the basic local alignment search tool (BLAST) and SW. This in turn, however, proved that the Smith-Waterman algorithm was better than the heuristic based pairwise sequence alignment tools.

2. An Overview of Multiple Sequence Alignment Parallel Tools (Fayed F. M. Ghaleb, Naglaa M. Reda, Mohammed Wajid Al-Neama, *June 2013, Conference: WSEAS/NAUN 2nd International Conference of Circuits, Systems, Communications, Computers and Applications (CSCCA '13)At: Dubrovnik, Croatia*)

In this paper, multiple sequence alignment was identified as a key problem to most bioinformatics applications. Various parallel architectures were experimented on for reaching the highest level of accuracy and speed. The most popular tools were looked at to clarify how parallelism accelerates the processing of large biological data sets and improves alignment accuracy. In most of the MSA methods they are based on the two pairwise alignment algorithms: the optimal algorithm proposed by Needleman and Wunsh (NW) for global alignment, and the improvement to the NW algorithm proposed by Smith and Waterman (SW) to obtain the local alignment. Both the methods are compared successfully.

3. Gene Sequencing Parallelization Using Smith-Waterman Algorithm (Deepa B. C1 , Nagaveni. V, ***August 2015, International Journal of Advanced Research in Computer and Communication Engineering*****)**

The objective of modern human genomics is preventive, predictive and individual medicine. Therefore, the authors of this paper try to find an efficient sequence alignment as it is one of the most important and challenging activities in bioinformatics. To perform and accelerate sequence alignment activities several algorithms have been proposed. Smith-Waterman algorithm represents a highly robust and efficient parallel computing system development for biological gene sequence. Therefore, the Smith-Waterman algorithm was successfully implemented and proved to be effective.

4. Parallel Multiple Sequence Alignment: An Overview (G. Scott Lloyd, 25 MARCH 2010)

Multiple Sequence Alignment (MSA) is a fundamental analysis method used in bioinformatics and many comparative genomic applications. The time to compute an optimal MSA grows exponentially with respect to the number of sequences. In the same way, the authors also realised the importance of producing timely results on large problems and that it requires more e-cient algorithms and the use of parallel computing resources. This paper discusses the most successful parallel methods and provides an in-depth review of core contributions in parallel MSA.

5. Fast Exact Sequence Alignment Using Parallel Computing (Ruba A. Al-Hussien, Qanita Bani baker, Mahmoud Al-Ayyoub, ***April 2018, 9th International Conference on Information and Communication Systems (ICICS)*****)**

Bioinformatics is a growing field that attracts many researchers and continues to prove its value and significance. Finding similarities, or even relations between sequences, is a demanding process that requires time and high cost. In this paper the authors realise this and utilize a multi-threading parallelism technique coupled with a block alignment idea in order to improve the sequence alignment performance. The experiments show that the proposed implementation outperforms the sequential implementation by 4.9 times for sequences of lengths ranging between 1024 and 8192. Therefore proving that the parallelism technique with a block alignment idea coupled together produces the best results.

6. Parallel Ant Colony Optimization Algorithm Based Neural Method for Determining Resonant Frequencies of Various Microstrip Antennas (Adem Kalinli, Seref Sagiroglu, Fatih Sarikoc, ***25 September 2009, Taylor & Francis*****)**

Artificial neural networks and heuristic algorithms are popular intelligent techniques in solving complex engineering problems. New approaches based on feed-forward artificial neural networks trained with Levenberg-Marquardt, touring ant colony optimization, and parallel ant colony optimization algorithms to determine the resonant frequencies of the rectangular, circular, and triangular microstrip antennas. The results achieved from heuristic- and gradient-based algorithms were compared to that of the other methods in the literature. The results obtained from the neural models for the various microstrip antennas are in very good agreement with the experimental results in the literature.

7. Multiple sequence alignment modeling: methods and applications (Maria Chatzou, Cedrik Magis, Jia-Ming Chang, Carsten Kemena, Giovanni Bussotti, Ionas Erb, Cedric Notredame, ***27 November 2015, Briefings in Bioinformatics, Volume 17, Issue 6, November 2016, Pages 1009–1023)***

This research paper provides an overview on the development of Multiple sequence alignment (MSA) methods and their main applications. It focuses on the main improvements and achievements made over the past decade. The first three sections review recent algorithmic developments for protein, RNA/DNA and genomic alignment and the fourth section deals with benchmarks and explores the relationship between empirical and simulated data, along with the impact on method developments. Finally, the last section of the review gives an overview on available MSA local reliability estimators and their dependence on various algorithmic properties of available methods.

8. Protein Multiple Sequence Alignment (Chuong B., DoKazutaka Katoh, ***Part of the Methods In Molecular Biology™ book series (MIMB, volume 484)***

This research paper focuses on protein sequence alignment. It is the task of identifying evolutionarily or structurally related positions in a collection of amino acid sequences. Although the protein alignment problem has been studied for several decades, multiple newer studies have demonstrated considerable progress in improving the accuracy or scalability of multiple and pairwise alignment tools, or in expanding the scope of tasks handled by an alignment program. In this paper the researchers focused on reviewing multiple state-of-the-art protein sequence alignment algorithms including the Smith-Waterman algorithm to compare the pros and cons of them as well as provide practical advice for users of alignment tools.

9. Pairwise statistical significance of local sequence alignment using multiple parameter sets and empirical justification of parameter set change penalty (Ankit Agrawal & Xiaoqiu Huang, ***19 MARCH 2009, Second International Workshop on Data and Text Mining in Bioinformatics (DTMBio) 2008)***

Local sequence alignment plays a major role in the analysis of DNA and protein sequences. It is the basic step of many other applications like detecting homology, finding protein structure and function, deciphering evolutionary relationships, etc. There exist several local sequence alignment programs that use well-known algorithms or their heuristic versions. Database search is a special case of pairwise local sequence alignment where the second sequence is a database in itself consisting of many sequences. The results from the paper proved that the results of pairwise statistical significance using multiple parameter sets are seen to be significantly better than database statistical significance estimates reported by BLAST and PSI-BLAST, and comparable and at times significantly better than SSEARCH.

10. Dataflow acceleration of Smith-Waterman with Traceback for high throughput Next Generation Sequencing by K. Koliogergi, N.Voss, S.Fytraki, S.Xydis, G.Gaydajiev, D.Soudis, ***Microprocessors and Digital Systems Labroratory, ECE, NTUA, "29th International Conference on Field Programmable Logic and Applications, 2019"***

The researchers have used the Maxeler's Dataflow engine and an upscale FPGA from Xilinx VU9P and several matrices were evaluated on the basis of efficiency and accuracy. The effects of the input buffer and accelerators were analysed. Similarly, results were analysed for hardware integrated and nonintegrated accelerations. Using Bowtie2 aligner, an increase of about 35% in performance and 18 times in speed was observed.

11. De(con)struction of the lazy-F loop: improving performance of Smith Waterman alignment by Roman Snyster, Research and AI, Microsoft, ***"IEEE 19th International Conference on Bioinformatics and Bioengineering, 2019"***

The authors have used the Lazy-F loop heuristic to improve performance of Smith-Waterman algorithm. However, the results also suggest that it requires additional memory. So, they improvised the loop heuristic with loop transformations and the asymptomatic performance thus obtained is highly efficient in both space and time complexities.

12. Parallel Smith-Waterman algorithm hardware implementation for ancestors and offspring gene tracer, (Asma G. Seliem, AIA Galal, Wael Abou El-Wafa and Hesham FA Hamed), ***World Symposium on Computer Applications and Research, 2016.***

The authors explore how next generation sequencers(NGS), ABI SOLiD and Illumina/Solexa Genome Analysers work and why Smith Waterman Algorithm is not the most efficient way despite it being the best when it comes to alignment. On similar lines, the authors also discuss BLAST and FASTA heuristics and how they are they are high in speed but poor in performance. The authors have used the HMM (Hidden Markov Model) to create a set of possible future sequences. The software has been accelerated using a Xilinx Zync 7000 based FPGA. Not only was the FPGA usage reduced by 65% to recreate standard FPGA results but reduction in memory and time consumption were also observed.

13. An improved Smith-Waterman Algorithm Based on Spark Parallelisation (Yanfeng Liu, Leixiao Li, Jing Gao), ***International Journal of BioAutomaton, 23(1), 2019.***

The authors propose the utilisation of a spark parallelisation plan for improving Smith-Waterman Algorithm on a cluster environment. They mapped the task on the spark clustering environment, ran a standard Smith-Waterman program under the constraints of reducing the overall number of tasks, setting them up on a distributed environment and finally tabulated the results. The program achieved a 100 percent accuracy with reduced time consumption.

14. A Parallel Implementation of the Smith-Waterman Algorithm for Massive Sequences Searching (Hsien-Yu Liao, Meng-Lia Yin, Yi Cheng), ***Proceedings of the 26th Annual International Conference of the IEEE EMBS, September 2004.***

This research article is not recent. However, this is one of the first research suggestions for using hardware acceleration and FPGAs for improving the Smith-Waterman algorithm in terms of time complexity. The authors have discussed both single sequence comparison and multiple sequence comparison and processing architectures for the same. They have suggested a novel approach for multiple sequence comparison that can reduce the time complexity from O(mn) to O(m+n).

15. Optimizing Smith-Waterman Algorithm on Graphical Processing Unit (Ayman Khalafallah, Omar Mahmoud, Hosain Fath Elbabb and Ahmed Elshamy) ***2nd International Conference on Computer Technology and Development, 2010***

This paper is from 2010, however was one of the first to suggest GPU based acceleration of Smith Waterman Algorithm. The authors have used an OpenCL platform to generate a two times faster execution of the Smith Waterman Algorithm.

16. Optimization for Smith-Waterman Algorithm on FPGA Platform (Xin Chang, A. Escobar, Carlos Valderrama, Vincent Robert) ***2014 International Conference on Computational Science and Computational Intelligence, IEEE***

The authors use FPGA to boost system performance for the smith-Waterman Algorithm, tabulate the cost per genome of sequence alignment from 1999-2013 and realise how the cost has decreased significantly. Further, Programmable Gateway Arrays, SMP machines and MPI improvements have also been discussed. Finally, the authors have suggested a new algorithm improvement in the Smith-Waterman method (for FPGAs only) and concluded that the resource utilisation is way better in FPGAs than in standard SWPEs.

17. DNA Fingerprint Using Smith-Waterman Algorithm by Grid Computing (El-Sayed Orabi, Mohamed A. Asaal, Mustafa Abdel Azim, Yasser Kamal), ***the 9th International Conference on INFOrmatics and Systems, Parallel and Distributed Track, IEEE, 2015***

We have included this research article because it is about grid computing. It involves a user that gives instructions to a coordinator who then distributes the task between several agents. The agents are further systems which have an input/output module, a task executor and a multi-core execution system. The grid setup ensures that resources are utilized to the maximum efficiency. The results thus obtained suggested that increasing the number of cores increases the performance tremendously both in terms of using sequences of differing length and also for across datasets.

18. Modelling and Performance Evaluation of Smith-Waterman Algorithm (Muhammed Shafiq, Jorda Polo, Branimir Dickov, Tassadaq Hussain), ***13th International Bhurban Conference on Applied Sciences and Technology, 2016, IEEE***

The researchers have used a supercomputing framework on an Altix-4700 multiprocessor supercomputing machine to evaluate the performance metrics for the Smith Waterman Algorithm based on OpenMP, MPI and hybrid models of the same. The results obtained showed that hybrid approach is best suited for efficient execution under standard conditions and yields an error of less than 5%.

19. Parallel Processing Cell Score Design of Linear Gap Penalty Smith-Waterman Algorithm (Syed Abdul Mutalib Al Junid, Mohd. Faizul Md, Nooritawati Md. Tahir), ***IEEE 13th International Colloquium on Signal Processing & its Applications, 2017***

The authors used a 2-bit comparator block and a parallelised computational block. The comparator blocks compare the DNA characters while the computational block works towards reducing steps and components involved. The project was built atop a standard Altera Quartus II module to generate results similar to FPGAs. The authors conclude that it is actually a feasible idea to use hardware-optimized element design for Smith Waterman scoring mechanism and by extension for DNA alignment.

20. Improving the performance of Smith-Waterman sequence algorithm on GPU using shared memory for biological protein sequences (Dr Venkata Vara Prasad and Suresh Jaganathan), ***Cluster Computing, 2019, Springer Science+Bussiness, Springer Nature***

The authors have suggested using GPU shared memory instead of the more regularly used GPU global memory to improve the SW algorithm. An architecture that requires a host, a global memory and several shared memory spaces is proposed. This makes it akin to a distributed system with several parallel processing units. Using the uniprotein dataset for their project, they concluded that a speed enhancement of about 2.7% is achieved when using GPU shared memory than the GPU global memory model and sped up by 13.7% as compared to a standard openMP code on a GPU global memory. This performance was even enhanced when the tile size was 32*32 or 32*16.

21. ***"Improving CUDASW++", a Parallelization of Smith-Waterman for CUDA Enabled Devices"*** **by Doug Hains, Zach Cashero et al. ,** ***IEEE International Parallel and Distributed Processing Symposium, 2011.***

The authors have discussed possible improvements in the CUDASW++, a standard tool for exploiting parallelism for bio-informatics. The authors suggest that the real bottle-neck of the program lies in the intra-task kernel and so propose the use of a global storage for local variables and an additional tool that shall allow the exploration of parameter space before and while the algorithm works. The results suggest that this tremendously improves CUDASW++ performance.

22. ***"Implementing Smith-Waterman Algorithm with Two-Dimensional Cache on GPUs"*** **by Xiaowen Feng and others,** ***Second International Conference on Cloud and Green Computing, 2012, IEEE***

The authors note that the Smith-Waterman algorithm needs improvement which can be achieved by accelerating it using GPUs and using a two-dimensional cache for memory. They also suggest using a global memory space for all parameters. The authors note that using the suggested improvisations, the algorithm improves in both speed and accuracy and also becomes easier to be accelerated.

23. "*Accelerating Smith-Waterman on Heterogeneous CPU-GPU Systems*" by Jaideep Singh and Ipseeta Aruni, ***IEEE, 2011.***

The authors have assessed the CUDASW++ version 2.0 and the NCBI BLAST on a standard Xeon X5472 processor and accelerated by GPUs. The performance was evaluated and plotted. The Hybrid CPU-GPU pair selected far outperformed the older Intel Xeon E5420 and Tesla C1060. The results also show that a hybrid CPU+GPU far outperformed a standard CPU only BLAST implementation.

DATA and TOOLS:

Wormbase is an online platform that serves as a repository for genomic data. It is updated regularly. We downloaded about 50 sequences of varying length from the website.
Another repository of genomic data is NCBI or the National Center for Biotechnology Information. The platform offers the option to directly download all the available databases. However, we used about 20 sequences of varying length from this site. Finally, we referred to the CUDASW++ website where the database and the CUDASW++ tool can be downloaded. This tool shall serve as a benchmarking tool for us, so shall the benchmarking data already available for BLAST and FASTA.

PROPOSED WORK:

Serial Code, OpenMP Code, MPI code and hybrid OpenMP and MPI implementations of the problem were studied under standard CPU conditions and with a CUDA GPU acceleration.
Through the same, we tried to study time patterns, effect of size and architecture on SW implementation. Through this experiment, we also tried to hypothesize possible changes or suggestions for a faster SW implementation.

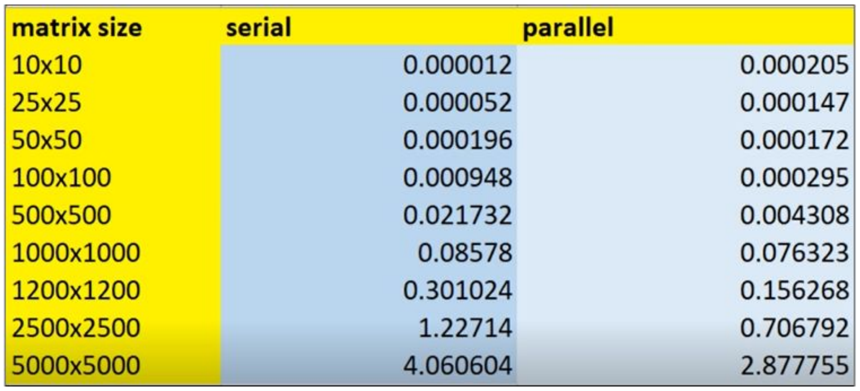
All the experiments and codes are run on two systems:

1. An IDE hosted on AWS cloud 9 architecture.
2. A system with intel i5 1.83GHz processor, 8GB RAM and NVIDIA GTX 1650 graphic card.

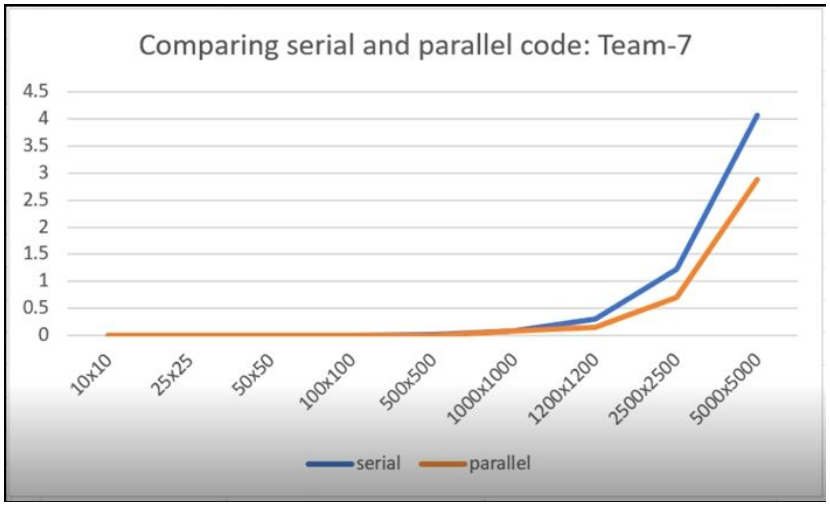
Results:

FOR PARALLEL VS SERIES

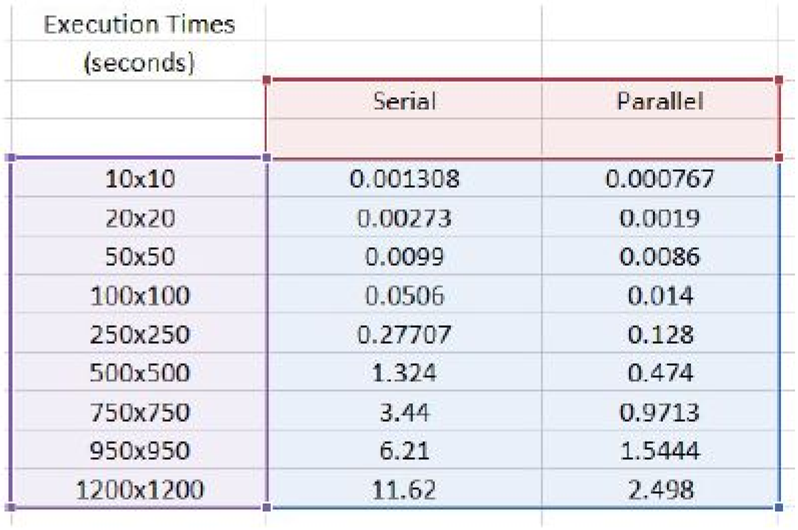
For 4 threads:



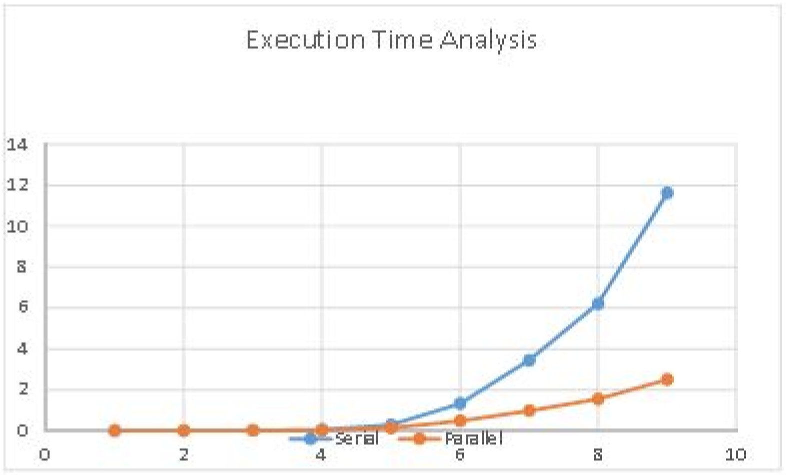
matrix size	serial	parallel
10x10	0.000012	0.000205
25x25	0.000052	0.000147
50x50	0.000196	0.000172
100x100	0.000948	0.000295
500x500	0.021732	0.004308
1000x1000	0.08578	0.076323
1200x1200	0.301024	0.156268
2500x2500	1.22714	0.706792
5000x5000	4.060604	2.877755



For 8 threads:



Execution Times (seconds)	Serial	Parallel
10x10	0.001308	0.000767
20x20	0.00273	0.0019
50x50	0.0099	0.0086
100x100	0.0506	0.014
250x250	0.27707	0.128
500x500	1.324	0.474
750x750	3.44	0.9713
950x950	6.21	1.5444
1200x1200	11.62	2.498



OBSERVATION 1: PARALLEL IMPLEMENTATION IS WAY FASTER THAN SERIAL IMPLEMENTATION IF THE ARRAY SIZE IS SUFFICIENTLY LARGE.

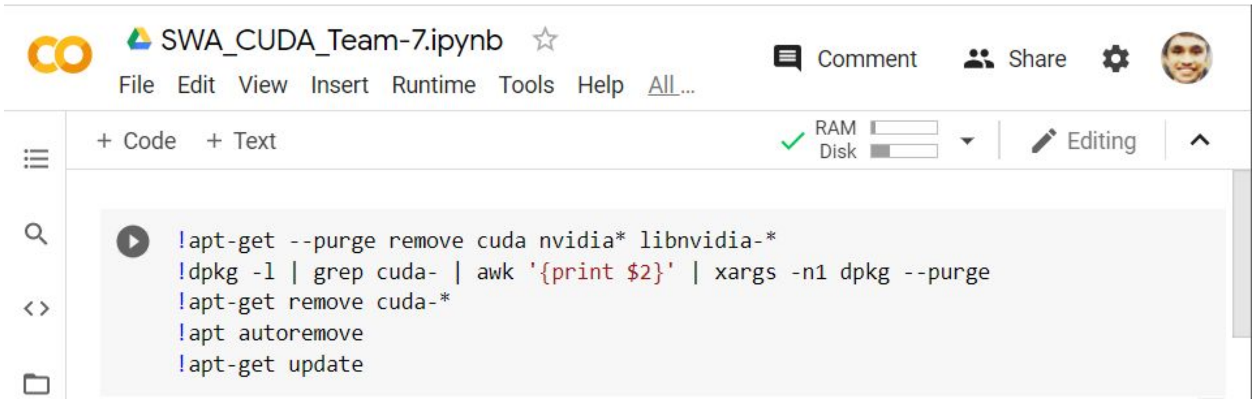
FOR OPENMP VS MPI VS HYBRID ON A CPU:

Our size of experiment was relatively small. So, the performance of the architectures is hard to both assess and compare. However, based on theoretical considerations that MPI has an implicit parallelisation and does not change much upto 32 processors(as per official documentation) , while OpenMP is easy to reach bottleneck, because parallelisation there is not implicit but depends on a multi-thread process and also confirmed by literature survey [18], MPI should perform better than openMP. However, one thing that we conclude is that the Hybrid program outperforms the MPI program as well. It is yet again confirmed by [18] (*in literature survey*).
However, OpenMP is more friendly for a multi-processor CPU.(This was a small survey we participated in and the results explain this conclusion.)

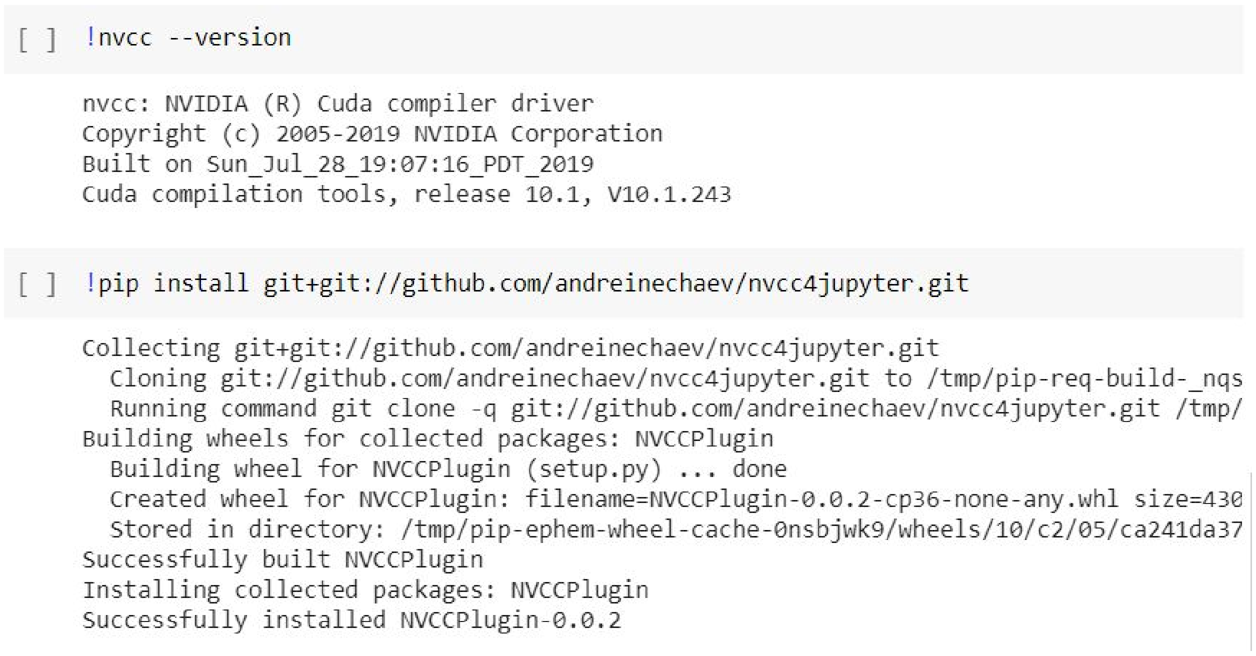
For GPU Driven Parrallelisation on CUDA

CUDA or Compute Unified Device Architecture is a free tool available on Google Collab, can also be downloaded in any GPU-enabled device. We ran our codes in CUDA with minor CUDA-specific changes.

USING CUDA ON GOOGLE COLLAB



```
!apt-get --purge remove cuda nvidia* libnvidia-*
!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge
!apt-get remove cuda-*
!apt autoremove
!apt-get update
```



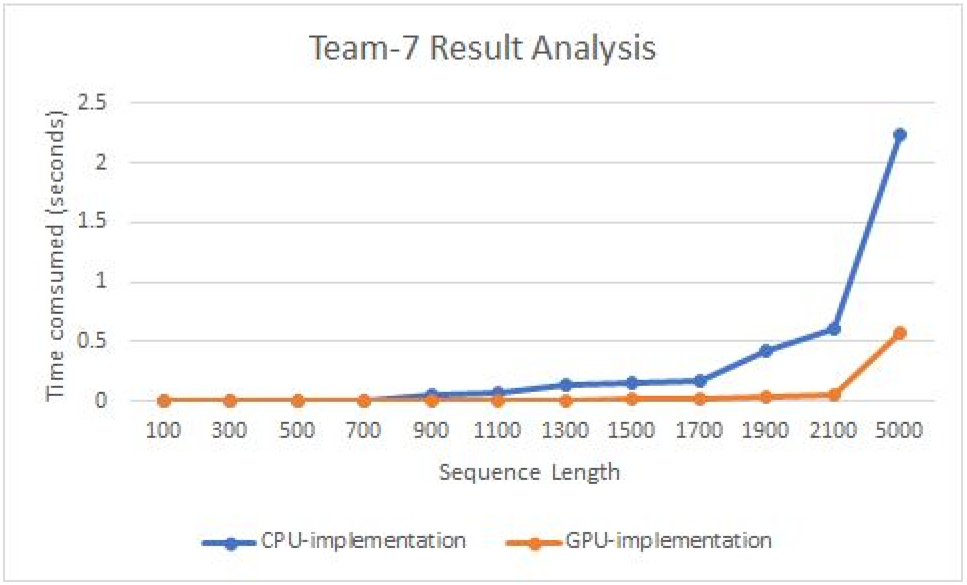
```
[ ] !nvcc --version

    nvcc: NVIDIA (R) Cuda compiler driver
    Copyright (c) 2005-2019 NVIDIA Corporation
    Built on Sun_Jul_28_19:07:16_PDT_2019
    Cuda compilation tools, release 10.1, V10.1.243
```

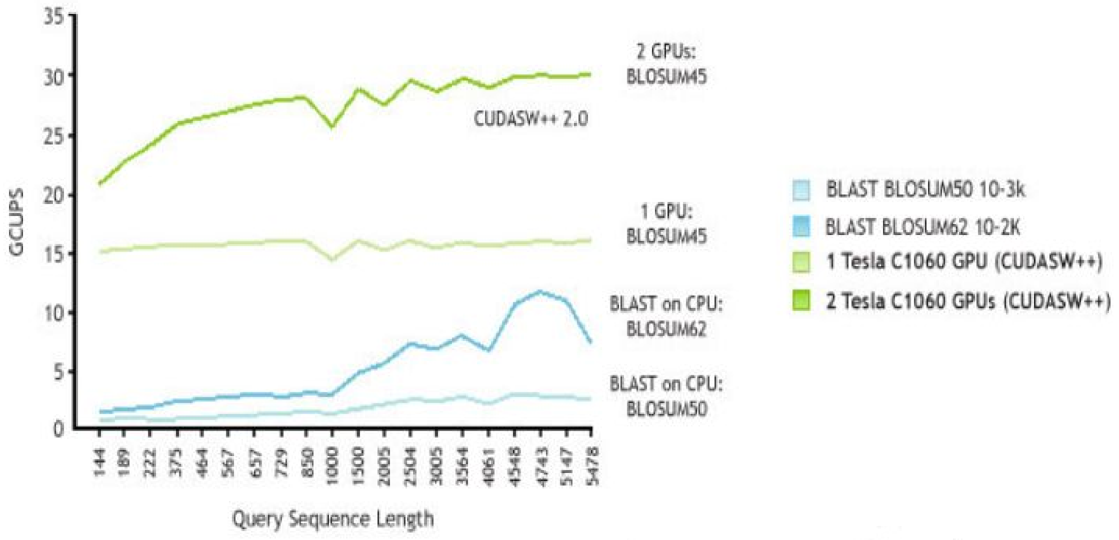
```
[ ] !pip install git+git://github.com/andreinechaev/nvcc4jupyter.git

    Collecting git+git://github.com/andreinechaev/nvcc4jupyter.git
      Cloning git://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-_nqs
      Running command git clone -q git://github.com/andreinechaev/nvcc4jupyter.git /tmp/
    Building wheels for collected packages: NVCCPlugin
      Building wheel for NVCCPlugin (setup.py) ... done
      Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-cp36-none-any.whl size=430
      Stored in directory: /tmp/pip-ephem-wheel-cache-0nsbjwk9/wheels/10/c2/05/ca241da37
    Successfully built NVCCPlugin
    Installing collected packages: NVCCPlugin
    Successfully installed NVCCPlugin-0.0.2
```

Here are the results compiled using GPU acceleration vs normal results and the benchmarked results available online.



Clearly, the GPU implementation is way faster than the CPU implementation.



Benchmarked data : Courtesy: NVIDIA Tesla Bio Workbench, 1 GCUPS is 30 billion updates per second.

CHALLENGES:

1. One of the chief problems we observed was using CUDA for running the codes. CUDA has a very specific dependency system i.e. older codes might simply not work on it. We had to experiment with some versions of CUDA before the optimum version was found.
2. Another problem was to compare the results with the limited architecture we possessed. Google Collab, Kaggle and other cloud services solved the issue.

CONCLUSION:

We studied the Smith Waterman Algorithm, tried to analyse through various researches already done on it, what are the various issues with it. The primary issue is space complexity which is quadratic in nature. We then tried to use parallelisation to resolve that issue. With that, we determined that a hybrid OpenMP + MPI program is most suited to achieving the result. This result is further improved when a GPU acceleration is used. CUDA provides an exclusive GPU parallelisation which improves our results tremendously.
We believe that the chief problem with a CPU implementation of SW algorithm is the fact that a lot of sequences are involved in its matrix, the chief problem with space complexity to be particular. So, GPUs can actually improve its performance to a scale that SW algorithm might actually overcome its basic limitation. However, we are still using a CPU+GPU system and GPU only systems are yet to be realised.

Appendix : Codes Used

Serial V/s Parallel (OpenMP)

I. Serial Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>  //used here to measure time i.e for the omp get wtime() function.

/*--------------------------------------------------------------------
 * Text Tweaks
 */
#define RESET   "\033[0m"
#define BOLDRED "\033[1m\033[31m"      /* Bold Red */
/* End of text tweaks */

/*--------------------------------------------------------------------
 * Constants
 */
#define PATH -1
#define NONE 0
#define UP 1
#define LEFT 2
#define DIAGONAL 3



/* End of constants */


/*--------------------------------------------------------------------
 * Functions Prototypes
 */
void similarityScore(long long int i, long long int j, int* H, int* P, long long int* maxPos);
int matchMissmatchScore(long long int i, long long int j);
void backtrack(int* P, long long int maxPos);
void printMatrix(int* matrix);
void printPredecessorMatrix(int* matrix);
void generate(void);
/* End of prototypes */
```

```
/*--------------------------------------------------------------------
 * Global Variables
 */
//Defines size of strings to be compared
long long int m = 11; //Columns - Size of string a
long long int n = 7;  //Lines - Size of string b

//Defines scores
int matchScore = 5;
int missmatchScore = -3;
int gapScore = -4;

//Strings over the Alphabet Sigma
char *a, *b;

/* End of global variables */

/*--------------------------------------------------------------------
 * Function:    main
 */
int main(int argc, char* argv[]) {
    m = strtoll(argv[1], NULL, 10);
    n = strtoll(argv[2], NULL, 10);

    #ifdef DEBUG
    printf("\nMatrix[%lld][%lld]\n", n, m);
    #endif

    //Allocates a and b
    a = malloc(m * sizeof(char));
    b = malloc(n * sizeof(char));

    //Because now we have zeros
    m++;
    n++;

    //Allocates similarity matrix H
    int *H;
    H = calloc(m * n, sizeof(int));

    //Allocates predecessor matrix P
    int *P;
    P = calloc(m * n, sizeof(int));
    //Gen rand arrays a and b
    generate();
    // a[0] =   'C';
    // a[1] =   'G';
    // a[2] =   'T';
    // a[3] =   'G';
    // a[4] =   'A';
    // a[5] =   'A';
    // a[6] =   'T';
    // a[7] =    'T';
    // a[8] =   'C';
    // a[9] =   'A';
    // a[10] =  'T';

    // b[0] =   'G';
    // b[1] =   'A';
    // b[2] =   'C';
    // b[3] =   'T';
    // b[4] =   'T';
    // b[5] =   'A';
    // b[6] =   'C';


    //Start position for backtrack
    long long int maxPos = 0;

    //Calculates the similarity matrix
    long long int i, j;

    double initialTime = omp_get_wtime();

    for (i = 1; i < n; i++) { //Lines
        for (j = 1; j < m; j++) { //Columns
            similarityScore(i, j, H, P, &maxPos);
        }
    }

    backtrack(P, maxPos);

    //Gets final time
    double finalTime = omp_get_wtime();  //not necessary for a shell script. While compiling,
compile gcc time <filename> -o <executable>
    printf("\nElapsed time: %f\n\n", finalTime - initialTime);
    #ifdef DEBUG
    printf("\nSimilarity Matrix:\n");
    printMatrix(H);

    printf("\nPredecessor Matrix:\n");
    printPredecessorMatrix(P);
    #endif

    //Frees similarity matrices
    free(H);
    free(P);

    //Frees input arrays
    free(a);
    free(b);

    return 0;
}  /* End of main */

/*--------------------------------------------------------------------
 * Function:    SimilarityScore
 * Purpose:     Calculate  the maximum Similarity-Score H(i,j)
 */
void similarityScore(long long int i, long long int j, int* H, int* P, long long int* maxPos) {

    int up, left, diag;

    //Stores index of element
    long long int index = m * i + j;

    //Get element above
    up = H[index - m] + gapScore;

    //Get element on the left
    left = H[index - 1] + gapScore;

    //Get element on the diagonal
    diag = H[index - m - 1] + matchMissmatchScore(i, j);

    //Calculates the maximum
    int max = NONE;
    int pred = NONE;
    /* === Matrix ===
     *      a[0] ... a[n]
     * b[0]
     * ...
     * b[n]
     *
     * generate 'a' from 'b', if '←' insert e '↑' remove
     * a=GAATTCA
     * b=GACTT-A
     *
     * generate 'b' from 'a', if '←' insert e '↑' remove
     * b=GACTT-A
     * a=GAATTCA
    */

    if (diag > max) { //same letter ↖
        max = diag;
        pred = DIAGONAL;
    }

    if (up > max) { //remove letter ↑
        max = up;
        pred = UP;
    }

    if (left > max) { //insert letter ←
        max = left;
        pred = LEFT;
    }
    //Inserts the value in the similarity and predecessor matrixes
    H[index] = max;
    P[index] = pred;

    //Updates maximum score to be used as seed on backtrack
    if (max > H[*maxPos]) {
        *maxPos = index;
    }

}  /* End of similarityScore */


/*--------------------------------------------------------------------
 * Function:    matchMissmatchScore
 * Purpose:     Similarity function on the alphabet for match/missmatch
 */
```

```
int matchMissmatchScore(long long int i, long long int j) {
   if (a[j-1] == b[i-1])
      return matchScore;
   else
      return missmatchScore;
}  /* End of matchMissmatchScore */

/*--------------------------------------------------------------------
 * Function:    backtrack
 * Purpose:     Modify matrix to print, path change from value to PATH
 */
void backtrack(int* P, long long int maxPos) {
   //hold maxPos value
   long long int predPos;

   //backtrack from maxPos to startPos = 0
   do {
      if(P[maxPos] == DIAGONAL)
         predPos = maxPos - m - 1;
      else if(P[maxPos] == UP)
         predPos = maxPos - m;
      else if(P[maxPos] == LEFT)
         predPos = maxPos - 1;
      P[maxPos]*=PATH;
      maxPos = predPos;
   } while(P[maxPos] != NONE);
}  /* End of backtrack */

/*--------------------------------------------------------------------
 * Function:    printMatrix
 * Purpose:     Print Matrix
 */
void printMatrix(int* matrix) {
   long long int i, j;
   for (i = 0; i < n; i++) { //Lines
      for (j = 0; j < m; j++) {
         printf("%d\t", matrix[m * i + j]);
      }
      printf("\n");
   }

}  /* End of printMatrix */

/*--------------------------------------------------------------------
```

```
 * Function:    printPredecessorMatrix
 * Purpose:     Print predecessor matrix
 */
void printPredecessorMatrix(int* matrix) {
   long long int i, j, index;
   for (i = 0; i < n; i++) { //Lines
      for (j = 0; j < m; j++) {
         index = m * i + j;
         if(matrix[index] < 0) {
            printf(BOLDRED);
            if (matrix[index] == -UP)
               printf("↑ ");
            else if (matrix[index] == -LEFT)
               printf("← ");
            else if (matrix[index] == -DIAGONAL)
               printf("↖ ");
            else
               printf("- ");
            printf(RESET);
         } else {
            if (matrix[index] == UP)
               printf("↑ ");
            else if (matrix[index] == LEFT)
               printf("← ");
            else if (matrix[index] == DIAGONAL)
               printf("↖ ");
            else
               printf("- ");
         }
      }
      printf("\n");
   }

}  /* End of printPredecessorMatrix */

/*------------------------------------------------------------------
 * Function:    generate
 * Purpose:     Generate arrays a and b
 */
 void generate(){
   //Generates the values of a
   long long int i;
   for(i=0;i<m;i++){
      int aux=rand()%4;
```

```
        if(aux==0)
            a[i]='A';
        else if(aux==2)
            a[i]='C';
        else if(aux==3)
            a[i]='G';
        else
            a[i]='T';
    }

    //Generates the values of b
    for(i=0;i<n;i++){
        int aux=rand()%4;
        if(aux==0)
            b[i]='A';
        else if(aux==2)
            b[i]='C';
        else if(aux==3)
            b[i]='G';
        else
            b[i]='T';
    }
} /* End of generate */
```

II. The OpenMP code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include <time.h>

/*--------------------------------------------------------------------
 * Text Tweaks
 */
#define RESET   "\033[0m"
#define BOLDRED "\033[1m\033[31m"      /* Bold Red */
/* End of text tweaks */

/*--------------------------------------------------------------------
 * Constants
 */
#define PATH -1
#define NONE 0
```

```
#define UP 1
#define LEFT 2
#define DIAGONAL 3
/* End of constants */

/*--------------------------------------------------------------------
* Helpers
*/
#define min(x, y) (((x) < (y)) ? (x) : (y))
#define max(a,b) ((a) > (b) ? a : b)

// #define DEBUG
/* End of Helpers */


/*--------------------------------------------------------------------
 * Functions Prototypes
 */
void similarityScore(long long int i, long long int j, int* H, int* P, long long int* maxPos);
int matchMissmatchScore(long long int i, long long int j);
void backtrack(int* P, long long int maxPos);
void printMatrix(int* matrix);
void printPredecessorMatrix(int* matrix);
void generate(void);
long long int nElement(long long int i);
void calcFirstDiagElement(long long int *i, long long int *si, long long int *sj);

/* End of prototypes */


/*--------------------------------------------------------------------
 * Global Variables
 */
//Defines size of strings to be compared
long long int m ; //Columns - Size of string a
long long int n ;  //Lines - Size of string b

//Defines scores
int matchScore = 5;
int missmatchScore = -3;
int gapScore = -4;

//Strings over the Alphabet Sigma
char *a, *b;
```

```
/* End of global variables */

/*--------------------------------------------------------------------
 * Function:    main
 */
int main(int argc, char* argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);
    m = strtoll(argv[2], NULL, 10);
    n = strtoll(argv[3], NULL, 10);

#ifdef DEBUG
    printf("\nMatrix[%lld][%lld]\n", n, m);
#endif

    //Allocates a and b
    a = malloc(m * sizeof(char));
    b = malloc(n * sizeof(char));

    //Because now we have zeros
    m++;
    n++;

    //Allocates similarity matrix H
    int *H;
    H = calloc(m * n, sizeof(int));

    //Allocates predecessor matrix P
    int *P;
    P = calloc(m * n, sizeof(int));


    //Gen rand arrays a and b
    generate();

    //Uncomment this to test the sequence available at
    //http://vlab.amrita.edu/?sub=3&brch=274&sim=1433&cnt=1
    // OBS: m=11 n=7
    // a[0] =   'C';
    // a[1] =   'G';
    // a[2] =   'T';
    // a[3] =   'G';
    // a[4] =   'A';
    // a[5] =   'A';
    // a[6] =   'T';
    // a[7] =   'T';
    // a[8] =   'C';
    // a[9] =   'A';
    // a[10] =  'T';

    // b[0] =   'G';
    // b[1] =   'A';
    // b[2] =   'C';
    // b[3] =   'T';
    // b[4] =   'T';
    // b[5] =   'A';
    // b[6] =   'C';


    //Start position for backtrack
    long long int maxPos = 0;
    //Calculates the similarity matrix
    long long int i, j;

    //Gets Initial time
    double initialTime = omp_get_wtime();

    long long int si, sj, ai, aj;

    //Because now we have zeros ((m-1) + (n-1) - 1)
    long long int nDiag = m + n - 3;
    long long int nEle;

    #pragma omp parallel num_threads(thread_count) \
    default(none) shared(H, P, maxPos, nDiag) private(nEle, i, si, sj, ai, aj)
    {
       for (i = 1; i <= nDiag; ++i)
       {
          nEle = nElement(i);
          calcFirstDiagElement(&i, &si, &sj);
          #pragma omp for
          for (j = 1; j <= nEle; ++j)
          {
             ai = si - j + 1;
             aj = sj + j - 1;
             similarityScore(ai, aj, H, P, &maxPos);
          }
       }
```

```
    }

    backtrack(P, maxPos);

    //Gets final time
    double finalTime = omp_get_wtime();
    printf("\nElapsed time: %f\n\n", finalTime - initialTime);

#ifdef DEBUG
    printf("\nSimilarity Matrix:\n");
    printMatrix(H);

    printf("\nPredecessor Matrix:\n");
    printPredecessorMatrix(P);
#endif

    //Frees similarity matrixes
    free(H);
    free(P);

    //Frees input arrays
    free(a);
    free(b);

    return 0;
}  /* End of main */

/*--------------------------------------------------------------------
 * Function:    nElement
 * Purpose:     Calculate the number of i-diagonal elements
 */
long long int nElement(long long int i) {
    if (i < m && i < n) {
        //Number of elements in the diagonal is increasing
        return i;
    }
    else if (i < max(m, n)) {
        //Number of elements in the diagonal is stable
        long int min = min(m, n);
        return min - 1;
    }
    else {
        //Number of elements in the diagonal is decreasing
        long int min = min(m, n);
        return 2 * min - i + abs(m - n) - 2;
    }
}

/*------------------------------------------------------------------
 * Function:    calcElement
 * Purpose:     Calculate the position of (si, sj)-element
 */
void calcFirstDiagElement(long long int *i, long long int *si, long long int *sj) {
    // Calculate the first element of diagonal
    if (*i < n) {
        *si = *i;
        *sj = 1;
    } else {
        *si = n - 1;
        *sj = *i - n + 2;
    }
}

/*------------------------------------------------------------------
 * Function:    SimilarityScore
 * Purpose:     Calculate  the maximum Similarity-Score H(i,j)
 */
void similarityScore(long long int i, long long int j, int* H, int* P, long long int* maxPos) {

    int up, left, diag;

    //Stores index of element
    long long int index = m * i + j;

    //Get element above
    up = H[index - m] + gapScore;

    //Get element on the left
    left = H[index - 1] + gapScore;

    //Get element on the diagonal
    diag = H[index - m - 1] + matchMissmatchScore(i, j);

    //Calculates the maximum
    int max = NONE;
    int pred = NONE;
    /* === Matrix ===
     *      a[0] ... a[n]
     * b[0]
     * ...
     * b[n]
     *
     * generate 'a' from 'b', if '←' insert e '↑' remove
     * a=GAATTCA
     * b=GACTT-A
     *
     * generate 'b' from 'a', if '←' insert e '↑' remove
     * b=GACTT-A
     * a=GAATTCA
    */

    if (diag > max) { //same letter ↖
        max = diag;
        pred = DIAGONAL;
    }

    if (up > max) { //remove letter ↑
        max = up;
        pred = UP;
    }

    if (left > max) { //insert letter ←
        max = left;
        pred = LEFT;
    }
    //Inserts the value in the similarity and predecessor matrixes
    H[index] = max;
    P[index] = pred;

    //Updates maximum score to be used as seed on backtrack
    if (max > H[*maxPos]) {
        #pragma omp critical
        *maxPos = index;
    }

}  /* End of similarityScore */


/*--------------------------------------------------------------------
 * Function:    matchMissmatchScore
 * Purpose:     Similarity function on the alphabet for match/missmatch
 */
```

```
int matchMissmatchScore(long long int i, long long int j) {
   if (a[j - 1] == b[i - 1])
      return matchScore;
   else
      return missmatchScore;
}  /* End of matchMissmatchScore */

/*--------------------------------------------------------------------
 * Function:    backtrack
 * Purpose:     Modify matrix to print, path change from value to PATH
 */
void backtrack(int* P, long long int maxPos) {
   //hold maxPos value
   long long int predPos;

   //backtrack from maxPos to startPos = 0
   do {
      if (P[maxPos] == DIAGONAL)
         predPos = maxPos - m - 1;
      else if (P[maxPos] == UP)
         predPos = maxPos - m;
      else if (P[maxPos] == LEFT)
         predPos = maxPos - 1;
      P[maxPos] *= PATH;
      maxPos = predPos;
   } while (P[maxPos] != NONE);
}  /* End of backtrack */

/*--------------------------------------------------------------------
 * Function:    printMatrix
 * Purpose:     Print Matrix
 */
void printMatrix(int* matrix) {
   long long int i, j;
   printf("-\t-\t");
   for (j = 0; j < m-1; j++) {
         printf("%c\t", a[j]);
   }
   printf("\n-\t");
   for (i = 0; i < n; i++) { //Lines
      for (j = 0; j < m; j++) {
         if (j==0 && i>0) printf("%c\t", b[i-1]);
         printf("%d\t", matrix[m * i + j]);
      }
```

```
        printf("\n");
    }

}  /* End of printMatrix */

/*-------------------------------------------------------------------
 * Function:    printPredecessorMatrix
 * Purpose:     Print predecessor matrix
 */
void printPredecessorMatrix(int* matrix) {
    long long int i, j, index;
    printf("    ");
    for (j = 0; j < m-1; j++) {
            printf("%c ", a[j]);
    }
    printf("\n  ");
    for (i = 0; i < n; i++) { //Lines
        for (j = 0; j < m; j++) {
            if (j==0 && i>0) printf("%c ", b[i-1]);
            index = m * i + j;
            if (matrix[index] < 0) {
                printf(BOLDRED);
                if (matrix[index] == -UP)
                    printf("↑ ");
                else if (matrix[index] == -LEFT)
                    printf("← ");
                else if (matrix[index] == -DIAGONAL)
                    printf("↖ ");
                else
                    printf("- ");
                printf(RESET);
            } else {
                if (matrix[index] == UP)
                    printf("↑ ");
                else if (matrix[index] == LEFT)
                    printf("← ");
                else if (matrix[index] == DIAGONAL)
                    printf("↖ ");
                else
                    printf("- ");
            }
        }
        printf("\n");
    }
```

```
}  /* End of printPredecessorMatrix */

/*-------------------------------------------------------------------
 * Function:    generate
 * Purpose:     Generate arrays a and b
 */
void generate() {
   //Random seed. Necessary to generate random values so that a more comprehensive
dataset is generated by the program itself.
   //The point is this dataset won't be generated using chaggaf's rule. But, we are
analysing the  openMP program and not the rules themself.
   srand(time(NULL));

   //Generates the values of a
   long long int i;
   for (i = 0; i < m; i++) {
      int aux = rand() % 4;
      if (aux == 0)
         a[i] = 'A';
      else if (aux == 2)
         a[i] = 'C';
      else if (aux == 3)
         a[i] = 'G';
      else
         a[i] = 'T';
   }

   //Generates the values of b
   for (i = 0; i < n; i++) {
      int aux = rand() % 4;
      if (aux == 0)
         b[i] = 'A';
      else if (aux == 2)
         b[i] = 'C';
      else if (aux == 3)
         b[i] = 'G';
      else
         b[i] = 'T';
   }
} /* End of generate */
```

Comparing Various Parallel implementations.

OpenMP Code:

```
#include<stdio.h>
#include<string.h>
#include <stdlib.h>
#include <time.h>
#include "omp.h"

//macros
#define ALPHABET_LENGTH 4
#define max(x,y) ((x)>(y)?(x):(y))


//global variables
char *string_A;
char *string_B;
char *unique_chars_C; //unique alphabets
int c_len;
short **P_Matrix;
short **DP_Results; //to store the DP values

//function prototypes
int get_index_of_character(char *str,char x, int len);
void print_matrix(short **x, int row, int col);
void calc_P_matrix_v1(short **P, char *b, int len_b, char *c, int len_c);
short lcs_yang_v1(short **DP, short **P, char *A, char *B, char *C, int m, int n, int u);
short lcs(short **DP, char *A, char *B, int m, int n);


int get_index_of_character(char *str,char x, int len)
{
   for(int i=0;i<len;i++)
   {
      if(str[i]== x)
      {
         return i;
      }
   }
   return -1;//not found the character x in str
}

void print_matrix(short **x, int row, int col)
```

```
{
   for(int i=0;i<row;i++)
   {
      for(int j=0;j<col;j++)
      {
         printf("%d ",x[i][j]);
      }
      printf("\n");
   }
}

void calc_P_matrix_v1(short **P, char *b, int len_b, char *c, int len_c)
{
   #pragma omp parallel for
   for(int i=0;i<len_c;i++)
   {
      for(int j=2;j<len_b+1;j++)
      {
         if(b[j-2]==c[i]) //j-2 as b we assume here that b has a empty character in the beginning
         {
            P[i][j] = j-1;
         }
         else
         {
            P[i][j] = P[i][j-1];
         }
      }
   }
}


short lcs_yang_v1(short **DP, short **P, char *A, char *B, char *C, int m, int n, int u)
{
   for(int i=1;i<m+1;i++)
   {
      int c_i = get_index_of_character(C,A[i-1],u);
      #pragma omp parallel for schedule(static)
         for(int j=0;j<n+1;j++)
      {
         if(A[i-1]==B[j-1])
         {
            DP[i][j] = DP[i-1][j-1] + 1;
         }
         else if(P[c_i][j]==0)
```

```
            {
               DP[i][j] = max(DP[i-1][j], 0);
            }
            else
            {
               DP[i][j] = max(DP[i-1][j], DP[i-1][P[c_i][j]-1] + 1);
            }
         }
      }
      return DP[m][n];
   }

   short lcs(short **DP, char *A, char *B, int m, int n)
   {
     // printf("%s %d \n%s %d\n",A,m,B,n );

      for(int i=1;i<(m+1);i++)
      {
         for(int j=1;j<(n+1);j++)
         {
            if(A[i-1] == B[j-1])
            {
               DP[i][j] = DP[i-1][j-1] + 1;
            }
            else
            {
               DP[i][j] = max(DP[i-1][j],DP[i][j-1]);
            }
         }
      }

      return DP[m][n];
   }

   int main(int argc, char *argv[])
   {
      if(argc <= 1){
         printf("Error: No input file specified! Please specify the input file, and run again!\n");
         return 0;
      }
      printf("\nYour input file: %s \n",argv[1]);

      FILE *fp;
      int len_a,len_b;
```

```
 double start_time,stop_time,start_time_yang,stop_time_yang;

 fp = fopen(argv[1], "r");
 fscanf(fp, "%d %d %d", &len_a, &len_b, &c_len);
 //printf("1 : %d %d %d\n", len_a, len_b, c_len );

 string_A = (char *)malloc((len_a+1) * sizeof(char *));
 string_B = (char *)malloc((len_b+1) * sizeof(char *));
 unique_chars_C = (char *)malloc((c_len+1) * sizeof(char *));

 fscanf(fp, "%s %s %s", string_A,string_B,unique_chars_C);
// printf("Strings : %s\n %s\n %s\n", string_A, string_B, unique_chars_C );


 //allocate memory for DP Results
 DP_Results = (short **)malloc((len_a+1) * sizeof(short *));
 for(int k=0;k<len_a+1;k++)
 {
    DP_Results[k] = (short *)calloc((len_b+1), sizeof(short));
 }


 //allocate memory for P_Matrix array
 P_Matrix = (short **)malloc(c_len * sizeof(short *));
 for(int k=0;k<c_len;k++)
 {
    P_Matrix[k] = (short *)calloc((len_b+1), sizeof(short));
 }

 start_time = omp_get_wtime();
 printf("lcs is: %d\n",lcs(DP_Results,string_A,string_B,len_a,len_b));
 stop_time = omp_get_wtime();
 printf("time taken by normal algorithm is: %lf",stop_time-start_time);



 //resetting DP to zero values
 for(int k=0;k<len_a+1;k++)
 {
    //memset(DP_Results[k],0,len_b+1);
    for(int l=0;l<len_b+1;l++)
    {
       DP_Results[k][l]=0;
    }
    }
    printf("\n");

    //deallocate pointers
    free(P_Matrix);
    free(DP_Results);
    return 0;
}
```

MPI Code:

```
#include<stdio.h>
#include<string.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"
//macros
#define ALPHABET_LENGTH 4
#define max(x,y) ((x)>(y)?(x):(y))


//global variables
char *string_A;
char *string_B;
char *unique_chars_C; //unique alphabets
int c_len;
short *P_Matrix;
short **DP_Results; //to store the DP values

//function prototypes
int get_index_of_character(char *str,char x, int len);
void print_matrix(short **x, int row, int col);
void calc_P_matrix_v1(short *P, char *b, int len_b, char *c, int len_c, int myrank, int chunk_size);
int lcs_yang_v1(short **DP, short *P, char *A, char *B, char *C, int m, int n, int u, int myrank, int
chunk_size);
int lcs(short **DP, char *A, char *B, int m, int n);


int get_index_of_character(char *str,char x, int len)
{
    for(int i=0;i<len;i++)
    {
        if(str[i]== x)
        {
            return i;
        }
    }
    return -1;//not found the character x in str
}

void print_matrix(short **x, int row, int col)
{
    for(int i=0;i<row;i++)
    {
        for(int j=0;j<col;j++)
        {
            printf("%d ",x[i][j]);
        }
        printf("\n");
    }
}

void print_p_matrix(short *p, int row, int col)
{
            for(int i=0;i<row;i++)
            {
                        for(int j=0;j<col;j++)
                        {
                                    printf("%d ",p[(i*col)+j]);
                        }
                        printf("\n");
            }
}

void calc_P_matrix_v1(short *P, char *b, int len_b, char *c, int len_c, int myrank, int chunk_size)
{
   char receive_array_for_scatter_c[chunk_size];
   short receive_array_for_scatter_p[chunk_size*(len_b+1)];
   if(myrank==0)
   {

   }
    //Scatter the char array chunks by sending each process a particular chunk
    MPI_Scatter(c, chunk_size,
MPI_CHAR,&receive_array_for_scatter_c,chunk_size,MPI_CHAR, 0, MPI_COMM_WORLD);
   //Scatter the char array chunks by sending each process a particular chunk
   MPI_Scatter(P, chunk_size*(len_b+1),
MPI_SHORT,&receive_array_for_scatter_p,chunk_size*(len_b+1),MPI_SHORT, 0,
MPI_COMM_WORLD);
    // Broadcast the whole b  array to everybody
   MPI_Bcast(b, len_b, MPI_CHAR, 0, MPI_COMM_WORLD);

   for(int i=0;i<chunk_size;i++)
   {
      for(int j=2;j<len_b+1;j++)
      {
         if(b[j-2]==receive_array_for_scatter_c[i]) //j-2 as b we assume here that b has a empty
character in the beginning
         {
            receive_array_for_scatter_p[(i*(len_b+1))+j] = j-1;
         }
         else
         {
            receive_array_for_scatter_p[(i*(len_b+1))+j] =
receive_array_for_scatter_p[(i*(len_b+1))+j-1];
         }
      }
   }

   //now gather all the calculated values of P matrix in process 0
   MPI_Gather(receive_array_for_scatter_p, chunk_size*(len_b+1), MPI_SHORT, P,
chunk_size*(len_b+1), MPI_SHORT, 0, MPI_COMM_WORLD);
}


int lcs_yang_v1(short **DP, short *P, char *A, char *B, char *C, int m, int n, int u, int myrank, int
chunk_size)
{

   MPI_Bcast(P, (u*(n+1)), MPI_SHORT, 0, MPI_COMM_WORLD);
   for(int i=1;i<m+1;i++)
   {


         // Broadcast the c_i  array to everybody
         //MPI_Bcast(A_i, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
         // Broadcast the  whole B  array to everybody
         //MPI_Bcast(B, len_b, MPI_CHAR, 0, MPI_COMM_WORLD);

         //Scatter the char array A chunks by sending each process a particular chunk
```

```
        //MPI_Scatter(A, chunk_size, MPI_CHAR,&scatter_receive_a,chunk_size,MPI_CHAR, 0,
MPI_COMM_WORLD);

        int c_i = get_index_of_character(C,A[i-1],u);
        //printf("c_i is %d for %c from %d\n",c_i,A[i-1],myrank);
        short dp_i_receive[chunk_size];
        // Broadcast the  whole B  array to everybody
        MPI_Scatter(DP[i], chunk_size, MPI_SHORT,&dp_i_receive,chunk_size,MPI_SHORT,
0, MPI_COMM_WORLD);
        int start_id = (myrank * chunk_size);
        int end_id = (myrank * chunk_size) + chunk_size;
        for(int j= start_id ;j<end_id;j++)//if myrank=0 then j=start_id+1 else j=start_id
      {
                if(j==start_id && myrank==0)j=j+1;
        if(A[i-1]==B[j-1])
        {
          dp_i_receive[j-start_id] = DP[i-1][j-1] + 1;
        }
        else if(P[(c_i*(n+1))+j]==0)
        {
          dp_i_receive[j-start_id] = max(DP[i-1][j], 0);
        }
        else
        {
          dp_i_receive[j-start_id] = max(DP[i-1][j], DP[i-1][P[(c_i*(n+1))+j]-1] + 1);
        }
      }
        //now gather all the calculated values of P matrix in process 0
        MPI_Allgather(dp_i_receive, chunk_size, MPI_SHORT,DP[i], chunk_size, MPI_SHORT,
MPI_COMM_WORLD);
    }
    return DP[m][n];
}

int lcs(short **DP, char *A, char *B, int m, int n)
{
   // printf("%s %d \n%s %d\n",A,m,B,n );

    for(int i=1;i<(m+1);i++)
    {
      for(int j=1;j<(n+1);j++)
      {
        if(A[i-1] == B[j-1])
        {
            DP[i][j] = DP[i-1][j-1] + 1;
         }
         else
         {
            DP[i][j] = max(DP[i-1][j],DP[i][j-1]);
         }
      }
   }

   return DP[m][n];
}

int main(int argc, char *argv[])
{
   if(argc <= 1){
      printf("Error: No input file specified! Please specify the input file, and run again!\n");
      return 0;
   }


   // Declare process-related vars
   //    // and initialize MPI
   int my_rank;
   int num_procs;
   int chunk_size_p,chunk_size_dp;//chunk_size for P matrix and DP matrix
   int res;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); //grab this process's rank
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs); //grab the total num of processes

   FILE *fp;
   int len_a,len_b;
   double start_time,stop_time,start_time_yang,stop_time_yang;

   if(my_rank == 0)printf("\nYour input file: %s \n",argv[1]);
   fp = fopen(argv[1], "r");
   fscanf(fp, "%d %d %d", &len_a, &len_b, &c_len);
//    printf("1 : %d %d %d\n", len_a, len_b, c_len );

   string_A = (char *)malloc((len_a+1) * sizeof(char *));
   string_B = (char *)malloc((len_b+1) * sizeof(char *));
   unique_chars_C = (char *)malloc((c_len+1) * sizeof(char *));
```

```
    fscanf(fp, "%s %s %s", string_A,string_B,unique_chars_C);
   // printf("Strings : %s\n %s\n %s\n", string_A, string_B, unique_chars_C );
    chunk_size_p = (c_len/num_procs);
    chunk_size_dp = ((len_b+1)/num_procs);

    if(my_rank==0)
{
            printf("chunk_p: %d chunk_dp: %d procs:
%d\n",chunk_size_p,chunk_size_dp,num_procs);
}
    //allocate memory for DP Results
    DP_Results = (short **)malloc((len_a+1) * sizeof(short *));
    for(int k=0;k<len_a+1;k++)
    {
        DP_Results[k] = (short *)calloc((len_b+1), sizeof(short));
    }


    //allocate memory for P_Matrix array
    P_Matrix = (short *)malloc((c_len*(len_b+1)) * sizeof(short));
   // for(int k=0;k<c_len;k++)
   // {
    //    P_Matrix[k] = (int *)calloc((len_b+1), sizeof(int));
   // }
    if(my_rank ==0)
{
    start_time =  MPI_Wtime();
    printf("lcs is: %d\n",lcs(DP_Results,string_A,string_B,len_a,len_b));
    stop_time = MPI_Wtime();
    printf("time taken by normal algorithm is: %lf\n",stop_time-start_time);
   // printf("DP results after normal lcs: \n");
   // print_matrix(DP_Results,len_a+1,len_b+1);
}


    //resetting DP to zero values
    for(int k=0;k<len_a+1;k++)
    {
        //memset(DP_Results[k],0,len_b+1);
        for(int l=0;l<len_b+1;l++)
        {
            DP_Results[k][l]=0;
        }
    }
    printf("\n");

    start_time_yang = MPI_Wtime(); // can use this function to grab a
                              // timestamp (in seconds)
    calc_P_matrix_v1(P_Matrix,string_B,len_b,unique_chars_C,c_len, my_rank, chunk_size_p);
   if(my_rank==0)
{
//         printf("\nP matrix is: \n");
 //        print_p_matrix(P_Matrix,c_len,len_b+1);
}
 //   if(my_rank==0)
//         {
                   res =
lcs_yang_v1(DP_Results,P_Matrix,string_A,string_B,unique_chars_C,len_a,len_b,c_len,my_ra
nk, chunk_size_dp);
//         }
//    int res =
lcs_yang_v1(DP_Results,P_Matrix,string_A,string_B,unique_chars_C,len_a,len_b,c_len);
    stop_time_yang = MPI_Wtime(); // can use this function to grab a
                              // timestamp (in seconds)
   if(my_rank == 0)
         {
                   printf("lcs_yang_v1 is: %d\n",res);
                   printf("time taken for lcs_yang_v1 is: %lf\n",stop_time_yang-start_time_yang);
         //        printf("DP results after yang: \n");
//                 print_matrix(DP_Results,len_a+1,len_b+1);
         }
    // printf("lcs_yang_v1 is: %d\n",res);
   // printf("time taken for lcs_yang_v1 is: %lf\n",stop_time_yang-start_time_yang);


    //deallocate pointers
    free(P_Matrix);
    free(DP_Results);

    // Shutdown MPI (important - don't forget!)
    MPI_Finalize();
    return 0;
}
```

Hybrid OpenMP + MPI Code:

```
#include<stdio.h>
#include<string.h>
#include <stdlib.h>
```

```
#include<mpi.h>
#include<omp.h>
#include<time.h>
//macros
#define ALPHABET_LENGTH 4
#define max(x,y) ((x)>(y)?(x):(y))


//global variables
char *string_A;
char *string_B;
char *unique_chars_C; //unique alphabets
int c_len;
short *P_Matrix;
short **DP_Results; //to store the DP values

//function prototypes
int get_index_of_character(char *str,char x, int len);
void print_matrix(short **x, int row, int col);
void calc_P_matrix_v2(short *P, char *b, int len_b, char *c, int len_c, int myrank, int chunk_size);
short lcs_yang_v2(short **DP, short *P, char *A, char *B, char *C, int m, int n, int u, int myrank,
int chunk_size);
short lcs(short **DP, char *A, char *B, int m, int n);


int get_index_of_character(char *str,char x, int len)
{
   for(int i=0;i<len;i++)
   {
      if(str[i]== x)
      {
         return i;
      }
   }
   return -1;//not found the character x in str
}

void print_matrix(short **x, int row, int col)
{
   for(int i=0;i<row;i++)
   {
      for(int j=0;j<col;j++)
      {
         printf("%d ",x[i][j]);
```

```
        }
        printf("\n");
    }
}



void calc_P_matrix_v2(short *P, char *b, int len_b, char *c, int len_c, int myrank, int chunk_size)
{
            char receive_array_for_scatter_c[chunk_size];
   short receive_array_for_scatter_p[chunk_size*(len_b+1)];
//Scatter the char array chunks by sending each process a particular chunk
MPI_Scatter(c, chunk_size, MPI_CHAR,&receive_array_for_scatter_c,chunk_size,MPI_CHAR,
0, MPI_COMM_WORLD);
//Scatter the char array chunks by sending each process a particular chunk
MPI_Scatter(P, chunk_size*(len_b+1),
MPI_SHORT,&receive_array_for_scatter_p,chunk_size*(len_b+1),MPI_SHORT, 0,
MPI_COMM_WORLD);
// Broadcast the whole b  array to everybody
MPI_Bcast(b, len_b, MPI_CHAR, 0, MPI_COMM_WORLD);


    #pragma omp parallel for
for(int i=0;i<chunk_size;i++)
    {
        for(int j=1;j<len_b+1;j++)
        {
            if(b[j-1]==receive_array_for_scatter_c[i])
            {
                    receive_array_for_scatter_p[(i*(len_b+1))+j] = j;
            }
            else
            {
                    receive_array_for_scatter_p[(i*(len_b+1))+j] =
receive_array_for_scatter_p[(i*(len_b+1))+j-1];
            }
        }
    }
//now gather all the calculated values of P matrix in process 0
MPI_Gather(receive_array_for_scatter_p, chunk_size*(len_b+1), MPI_SHORT, P,
chunk_size*(len_b+1), MPI_SHORT, 0, MPI_COMM_WORLD);
}
```

```
short lcs_yang_v2(short **DP, short *P, char *A, char *B, char *C, int m, int n, int u, int myrank,
int chunk_size)
{
   MPI_Bcast(P, (u*(n+1)), MPI_SHORT, 0, MPI_COMM_WORLD);
   for(int i=1;i<m+1;i++)
   {
      int c_i = get_index_of_character(C,A[i-1],u);
         short dp_i_receive[chunk_size];
         MPI_Scatter(DP[i], chunk_size, MPI_SHORT,&dp_i_receive,chunk_size,MPI_SHORT,
0, MPI_COMM_WORLD);
         int start_id = (myrank * chunk_size);
      int end_id = (myrank * chunk_size) + chunk_size;

      int t,s;
         #pragma omp parallel for private(t,s) schedule(static)
      for(int j=start_id;j<end_id;j++)
      {
            if(j==start_id && myrank==0)j=j+1;
         t= (0-P[(c_i*(n+1))+j])<0;
         s= (0 - (DP[i-1][j] - (t*DP[i-1][P[(c_i*(n+1))+j]-1]) ));
         dp_i_receive[j-start_id] = ((t^1)||(s^0))*(DP[i-1][j]) +
(!((t^1)||(s^0)))*(DP[i-1][P[(c_i*(n+1))+j]-1] + 1);
      }
         //now gather all the calculated values of P matrix in process 0
         MPI_Allgather(dp_i_receive, chunk_size, MPI_SHORT,DP[i], chunk_size, MPI_SHORT,
MPI_COMM_WORLD);
   }
   return DP[m][n];
}


short lcs(short **DP, char *A, char *B, int m, int n)
{
  // printf("%s %d \n%s %d\n",A,m,B,n );

   //print_matrix(DP,m+1,n+1);

   for(int i=1;i<(m+1);i++)
   {
      for(int j=1;j<(n+1);j++)
      {
         if(A[i-1] == B[j-1])
         {
            DP[i][j] = DP[i-1][j-1] + 1;
            }
            else
            {
                DP[i][j] = max(DP[i-1][j],DP[i][j-1]);
            }
        }
    }

    return DP[m][n];
}

int main(int argc, char *argv[])
{
    if(argc <= 1){
        printf("Error: No input file specified! Please specify the input file, and run again!\n");
        return 0;
    }

            int my_rank;
    int num_procs;
    int chunk_size_p,chunk_size_dp;//chunk_size for P matrix and DP matrix
    int res;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); //grab this process's rank
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs); //grab the total num of processes

    FILE *fp;
    int len_a,len_b;
    double start_time,stop_time,start_time_yang,stop_time_yang;

    if(my_rank == 0)printf("\nYour input file: %s \n",argv[1]);
    fp = fopen(argv[1], "r");
    fscanf(fp, "%d %d %d", &len_a, &len_b, &c_len);
    string_A = (char *)malloc((len_a+1) * sizeof(char *));
    string_B = (char *)malloc((len_b+1) * sizeof(char *));
    unique_chars_C = (char *)malloc((c_len+1) * sizeof(char *));

    fscanf(fp, "%s %s %s", string_A,string_B,unique_chars_C);

    chunk_size_p = (c_len/num_procs);
    chunk_size_dp = ((len_b+1)/num_procs);

    if(my_rank==0)
```

```
{
        printf("chunk_p: %d chunk_dp: %d procs:
%d\n",chunk_size_p,chunk_size_dp,num_procs);
}

     DP_Results = (short **)malloc((len_a+1) * sizeof(short *));
    for(int k=0;k<len_a+1;k++)
    {
        DP_Results[k] = (short *)calloc((len_b+1), sizeof(short));
    }

    P_Matrix = (short *)malloc((c_len*(len_b+1)) * sizeof(short));


    for(int k=0;k<len_a+1;k++)
    {
        for(int l=0;l<len_b+1;l++)
        {
            DP_Results[k][l]=0;
        }
    }
    start_time_yang = MPI_Wtime();

    calc_P_matrix_v2(P_Matrix,string_B,len_b,unique_chars_C,c_len, my_rank, chunk_size_p);

        res =
lcs_yang_v2(DP_Results,P_Matrix,string_A,string_B,unique_chars_C,len_a,len_b,c_len,my_ra
nk, chunk_size_dp);

    stop_time_yang = MPI_Wtime();

    if(my_rank == 0)
        {
                printf("lcs_yang_v2 is: %d\n",res);
                printf("time taken for lcs_yang_v2 is: %lf\n",stop_time_yang-start_time_yang);

        }
    //deallocate pointers
    free(P_Matrix);
    free(DP_Results);

    // Shutdown MPI (important - don't forget!)
    MPI_Finalize();
    return 0;      }
```

Reference Works:

[1] Cuong Cao Dang, Vincent Lefort, Vinh Sy Le, Quang Si Le, and Olivier Gascuel ,“Maximum likelihood estimation of amino acid replacement rate matrix”, Bioinformatics. 2011, 27(19):2758-60.

[2] Frank Keul, Martin Hess , Michael Goesele and Kay Hamacher , “PFASUM: a substitution matrix from Pfam structural alignments” , June 5 2017

[3] S Henikoff and J G Henikoff , “Amino acid substitution matrices from protein blocks.”, Proc Natl Acad Sci U S A. 1992 Nov 15; 89(22): 10915–10919.

[4] Gary Benson ,Yozen Hernandez and Joshua Loving ,” A Bit-Parallel, General Integer-Scoring Sequence Alignment Algorithm” , 2004

[5] Vincent Ranwez and Yang Zhang ,” Two Simple and Efficient Algorithms to Compute the SP-Score Objective Function of a Multiple Sequence Alignment”, PLoS One, 2016

[6] Robert C. Edgar1 and Kimmen Sjölander ,” A comparison of scoring functions for protein sequence profile alignment “

[7] Cheng Ling, Khaled Benkrid, Ahmet T. Erdogan, "High performance Intra-task parallelization of Multiple Sequence Alignments on CUDA-compatible GPUs", Adaptive Hardware and Systems (AHS) 2011 NASA/ESA Conference on, pp. 360-366, 2011.

[8]Chao-Chin Wu, Jenn-Yang Ke, Heshan Lin, Wu-chun Feng, "Optimizing Dynamic Programming on Graphics Processing Units via Adaptive Thread-Level Parallelism", Parallel and Distributed Systems (ICPADS) 2011 IEEE 17th International Conference on, pp. 96-103, 2011.

[9]Chao-Chin Wu, Kai-Cheng Wei, Ting-Hong Lin, "Optimizing Dynamic Programming on Graphics Processing Units Via Data Reuse and Data Prefetch with Inter-Block Barrier Synchronization", Parallel and Distributed Systems (ICPADS) 2012 IEEE 18th International Conference on, pp. 45-52, 2012.

[10]Dzmitry Razmyslovich, Guillermo Marcus, Markus Gipp, Marc Zapatka, Andreas Szillus, "Implementation of Smith-Waterman Algorithm in OpenCL for GPUs", Parallel and Distributed Methods in Verification 2010 Ninth International Workshop on and High Performance Computational Systems Biology Second International Workshop on, pp. 48-56, 2010.

[11] T. F. Smith, M. S. Waterman, "Identification of common molecular subsequences", Journal of molecular biolog., vol. 147, pp. 195-197, 1981.

[12] E. Rucci, C. García, G. Botella, A. De Giusti, M. Naiouf, M. Prieto-Matías, "State-of-the-Art in Smith-Waterman Protein Database Search on HPC Platforms" in Big Data Analytics in Genomics, Springer, pp. 197-223, 2016.

[13] H. Li, R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform", Bioinformatic., vol. 25, pp. 1754-1760, 2009.

[14] Y. Liu, B. Schmidt, "Long read alignment based on maximal exact match seeds", Bioinformatic., vol. 28, pp. i318-i324, 2012.

[15] B. Langmead, S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2", Nature method., vol. 9, pp. 357-359, 2012.

[16] B. Buchfink, C. Xie, D. H. Huson, "Fast and sensitive protein alignment using DIAMOND", Nature method., vol. 12, pp. 59-60, 2015.

[17] T. Rognes, E. Seeberg, "Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors", Bioinformatic., vol. 16, pp. 699-706, 2000.