PARALLELIZATION USING WATERMAN SMITH ALGORITHM

J-COMPONENT PROJECT

REPORT

SUBMITTED BY

KARTIKAY GUPTA(18BCE2199)
MIHIR HUMAR(18BCE0828)

B. Tech (Computer Science and Engineering)
in
School of Computer Science and Engineering

CSE4001 – Parallel and Distributed Computing

(EPJ)

Project Guide

Dr. DeebakBD Associate Professor (SCOPE), VIT

School of Computer Science and

Engineering FALL SEMESTER 2020-21

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Parallelization Using Smith-Waterman algorithm

1.Domain Introduction

The Smith-Waterman algorithm is a database search algorithm developed by T.F. Smithand M.S. Waterman. The S-W Algorithmis based on dynamic programming, which takes alignments of any length, location, in any sequence, and uses it to determine whether an optimal alignment can be found. Based on this, scores or weights are assigned to each character comparison:

1.)Positive for exact matches/substitutions

2.)Negative for insertions/ deletions.

In weight matrices, scores are added together and the highest scoring alignment is reported.

S-W algorithm compares multi- lengthed segments where it does not compare the whole segment at once but separates it into different parts, looking for whichever segment maximizes the scoring measure. The algorithm itself is recursive in nature:

$$H\ i\ j = \max\{H\ i\text{-}1,\ j\text{-}1 + s(a\ i,\ b\ j);\ H\ i\text{-}k,\ j - W\ k\ ;\ H\ i,\ j\text{-}1 - W1;0\}$$

2.Problem Statement -

Smith–Waterman Algorithm gives us the best local alignment of 2 sequences. This algorithm is mathematically proven to be the best, but it takes large memory space.

***To resolving this problem, we shall implemented Smith-Waterman algorithm using OpenMP to parallelize it.**

3.Techniques & its Related Challenges

Smith–Waterman is the only algorithm that guarantees to find the optimal local alignment but it is also the slowest one as it costs O(mn) for computation & space. Also the volume of biological data is doubling about every six months so the total cost is O(kmn) where k is the size of the database. By using parallel hardware and software architecture accurate results can be achieved in reasonable time.

For resolving the critical requirement of memory space, we present parallel method for Smith-Waterman algorithm using openMP.

4.Research Findings

Smith-Waterman algorithm is a classic dynamic programming algorithm to solve the problem of biological sequence alignment. However, with the rapid increment of the number of DNA and protein sequences, the originally sequential algorithm is very time consuming due to their existing the same computing task computed repeatedly on large-scale data. So we use GPU (graphics processor unit) consists of hundreds of processors, so it has a more powerful computation capacity than the current CPU. And as the programmability of GPU improved continuously, using it to do generous purpose computing is becoming very popular. We design a parallel algorithm which exploits the parallelism of the column of similarity matrix to parallelize the Smith-Waterman algorithm on a heterogeneous system based on CPU and GPU. This parallel algorithm is more efficient , as it takes full advantage of the features of both the CPU and GPU and obtains approximately 37 times speedup compared with the sequential algorithm.

Now that much of the actual process of sequencing is automated (i.e. the genechips in microarrays), a huge amount of quantitative information is being generated. So our project is of quantitative search type.

5.Our Approach i.e. Practical / Theoretical / Review / Survey / Technical Note

Biological sequence comparison is an important tool for researchers in molecular biology. There are several algorithms for sequence comparison. The Smith-Waterman algorithm, is one of the most fundamental algorithms in bioinformatics The Smith-Waterman algorithm is a dynamic programming algorithm that builds a real or implicit array where each cell of the array represents a subproblem in the alignment problem (Smith and Waterman, 1981). For strings a and b and for mismatch scoring function s(a, b) and gap score, W_i, the Smith- Waterman matrix H is -

$H(i,0)=0$, $H(0,j)=0$

$H(i,j)=\max\{0$

$\{H(i-1,j-1)+s(a_i,b_j)$ mismatch

$\{\max_{k>=1} (H(i-k, j)+W_k$ delete

$\{\max_{l>=1} (H(i,j-l)+W_l)$ insert

Where $0<=i<=n$ & $0<=j<=m$, where n and m are the lengths of the string a and b.

The cell H(n, m) contains the optimal score, and the optimal alignment can be found by backtracing starting from the last cell and going in the direction in which matrix was created.

6.Significance

This local sequence alignment method explores all possible alignments and finds the optimal local alignment. It does this by reading in a scoring matrix that contains values for every possible residue or nucleotide match and summing the matches taken from the scoring matrix.

7.Objective:

•To learn and use openmp and its related packages.

•To learn and use Parallel processing and compare its efficiency with serial execution by implementing the above mentioned algorithm .

•This would be done in parallel and in serial and a comparison between them would be done.

•To work in a team and understand the advantages of teamwork by assigning roles for each member and deadlines for the project, which would help us get suited to a work-based environment and benefit us in the future.

•To improve our report writing and presentation skills through the frequent Reviews conducted to check on our progress.

•To understand the real world applications bioinformatics and to understand the importance of computer science even in the world of biology.

8.Background and Motivation

8.1Motivation

The algorithm that we have chosen for the project is Smith-Waterman, which essentially computes the edit distance between two strings. Out of all the possible choices of algorithms, this one was chosen because there has been substantial prior work on making Smith-Waterman run on platforms with parallel execution capability. As well, we would be able to take advantage of insights and ideas from existing work. We hope to be able to observe similar performance as other solutions and possibly have other advantages as well such as programmability.

8.2Background

The Smith-Waterman algorithm looks for sequence subsets that best match in two large sequences. This algorithm will execute in O(mn) time where m is the length of the main sequence while n is the length of the match sequence. We fix the length of the main and match sequences to be the same, so the algorithm effectively runs in O(n^2) time.

The first step is to generate a similarity matrix. This similarity matrix decides if a pair of codons, a triplet of adjacent DNA nucleotides that code for proteins, are either exact matches, similar matches that are distinct but serve the same function, or dissimilar matches.

This can be generalized as a function sim(a, b)that will return an exact match score, a similar score, or a no match score This algorithm works with a dynamic programming matrix A. This matrix is formed for each element Aij by comparing element i in the main sequence and j in the match sequence using the sim(a, b) function to derive a score that is added to element A(i−1)(j−1) of the score matrix representing the score if the current sequence were matched. A score is derived for a gap in the main sequence by subtracting a gap penalty from element Ai(j−1) in the score matrix.

A score for a gap in the match sequence is also derived by taking element A(i−1)j and subtracting the gap penalty. These scores, for the gap in main, the gap in match, and a similarity value between main and a match, are all compared and the highest score is the score used for element Aij in the matrix to a minimum score of 0.

9. Methodology

In sequence homologous search data-decomposition is used at two levels

1. **Inter-process level:** database is partitioned and distributed to different processes. This level uses MPI.

2. **Intra-process level:** the similarity matrix is partitioned and distributed among a set of threads. This level uses OpenMP.

Here we assume that each node will run only one process and one thread per core.

9.1 Master Node

1. Read query sequence & scoring matrix.
2. Broadcast query sequence & scoring matrix to all workers.
3. Get database size.
4. Statically partitioning database sequences in chunks = number of workers.
5. For(i=1 to number of workers) Send chunk[i] to worker[i].
6. For(i=1 to number of workers) Receive optimal score from any worker.
7. Combine and output the final result.

9.2 Worker Node

1. Receive query sequence & scoring matrix
2. Receive chunk[i]
3. For each target sequence in chunk[i] smith-waterman(query, target) Update optimal score END For
4. Send optimal score to master node.

9.3 Problems in methodology

1. Time and space cost are very high.
2. Finds the alignment with maximal score, but not with maximal percent of matches

10.Tools description

10.1 User Interface

1.) Ubuntu Terminal

10.2 Features / Model

1. It is guaranteed in mathematical sense to provide an optimal alignment for a given set of scoring function.
2. It does not require gap penalty.
3. It simply avoids all data dependences within the alignment matrix.

10.3 Specification

The problem at hand was tackled with a modular approach. Eight functions were constructed, each of which would be explained as follows:

1.) nElement – This function is used to calculate the number of elements that have been found by the Smith Waterman Algorithm. Three conditions are given: One of which is to find out if the number of elements in the diagonal are increasing, decreasing or stable

2.) calcFirstDiagElement – This function is used to calculate the position of the maximum scored value in the matrix. This value needs to be found because the algorithm suggests that the backtracking to find the path should be started from this particular point.

3.) similarityScore – This function is used to find out the optimal order of execution based on three conditions, which are used to calculate the new values of left, upper and the diagonal elements.

If the diagonal element > maximum element, Move diagonally upwards

If upper element > maximum element, Move upwards

If left element > maximum element, Move leftwards

Every iteration, the values of maximum element is updated and inserting into the similarity and predecessor matrices.

4.) matchMismatchScore – This function is used to calculate a similarity function or the alphabet for a match ormismatch. If the value of the two elements are equal, it is a match, otherwise it's a mismatch.

5.) Backtrack – The purpose of this function is to modify the matrix that needs to be printed and helps us identify the path that needs to be taken to get the most optimum solution.
6.) printMatrix – It's a looped iteration implementation to display the matrix.
7.) printPredecessorMatrix - It is in this function in which we print the arrows depicting the path of local alignment.
8.) Generate – This function generates the two sequences A and B which would be locally aligned with each other. A random seed is used to ensure the reproducible nature of the output.

10.4 Package Requirement

1.) <stdio.h>The C programming language provides many standard library functions for file input and output. These functions make up the bulk of the C standard library
2.) <stdlib.h> stdlib.h is the header of the general purpose standard library of C programming language which includes functions involving memory allocation, process control, conversions and others. It is compatible with C++ and is known as cstdlib in C++. The name "stdlib" stands for "standard library"
3.) <math.h> The math.h header defines various mathematical functions and one macro. All the functions available in this library take double as an argument and return double as the result.
4.) <omp.h> It is a library that allows memory multiprocessing programming in C.
5.)<time.h> In C programming language time.h (used as ctime in C++) is a header file defined in the C Standard Library that contains time and date function declarations to provide standardized access to time/date manipulation and formatting

11.Code Implementation

```
// IMPORTING ALL THE HEADER FILES NEEDED FOR THE IMPLEMENTATION
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#include <time.h>

// DEFINING THE GLOBAL VARIABLES AND FIXED VARIABLES
#define RESET "\033[0m"
#defineBOLDRED "\033[1m\033[31m"
#define PATH -1
#define NONE0
#define UP 1
#define LEFT 2
#define DIAGONAL 3
#define min(x, y) (((x) < (y)) ? (x) : (y))
#define max(a,b) ((a) > (b) ? a : b)

// FUNCTION DEFINITIONS
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);

// DEFINING THE MAIN VARIABLES
long long int m ; //Columns - Size of string a
long long int n ; //Lines - Size of string b
int matchScore = 5;
int missmatchScore = -3;
int gapScore = -4;
char *a, *b;
```

```
// MAIN FUNCTION FOR EXECUTION
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
a = malloc(m * sizeof(char));
b = malloc(n * sizeof(char));
m++;
n++;
int *H;
H = calloc(m * n, sizeof(int));
int *P;
P = calloc(m * n, sizeof(int));
generate();
long long int maxPos = 0;
long long int i, j;
double initialTime = omp_get_wtime();
long long int si, sj, ai, aj;
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);
#ifdef DEBUG
printf("\nSimilarity Matrix:\n");
printMatrix(H);
printf("\nPredecessor Matrix:\n");
printPredecessorMatrix(P);
#endif
double finalTime = omp_get_wtime();
printf("\nElapsed time: %f\n\n", finalTime -initialTime);
free(H);
free(P);
free(a);
free(b);
return 0;
}
long long int nElement(long long int i) {
if (i < m && i < n) {
return i;
}
else if (i < max(m, n)) {
long int min = min(m, n);
return min - 1;
}
else {
long int min = min(m, n);
return 2 * min - i + abs(m - n) - 2;
}
}

// FUNCTION TO CALCULATE THE ELEMENT FROM WHICH THE ALIGNMENT MUST START
void calcFirstDiagElement(long long int *i, long long int *si, long long int *sj) {
if (*i < n) {
```

```
*si = *i;
*sj = 1;
} else {
*si = n - 1;
*sj = *i - n + 2;
}
}

// FUNCTION TO CALCULATE THE SIMILARITY SCORES OF THETWO SEQUENCES
void similarityScore(long long int i, long long int j, int* H, int* P, long long int* maxPos) {
int up, left, diag;
long long int index = m * i + j;
up = H[index - m] + gapScore;
left = H[index - 1] + gapScore;
diag = H[index - m - 1] + matchMissmatchScore(i, j);
int max = NONE;
int pred = NONE;
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;
}
H[index] = max;
P[index] = pred;
if (max > H[*maxPos]) {
#pragma omp critical
*maxPos = index;
}
}
```

```
// FUNCTION TO CALCULATE THE MISMATCH SCORES FOR THE TWO SEQUENCES
int matchMissmatchScore(long long int i, long long int j) {
if (a[j - 1] == b[i - 1])
return matchScore;
else
return missmatchScore;
}

// THE MAIN BACKTRACKING FUNCTION FOR SEQUENCE ALIGNMENT
void backtrack(int* P, long long int maxPos) {
//hold maxPos value
long long int predPos;
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);
}

// FUNCTION TO PRINT THE 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");
}
}

// FUNCTION TO PRINT THE MAIN ALIGNMENT 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");
}
}
// FUNCTION TO GENERATE THE SEQUENCES RANDOMLY
void generate() {
srand(0);
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';
}
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';
}
}
```

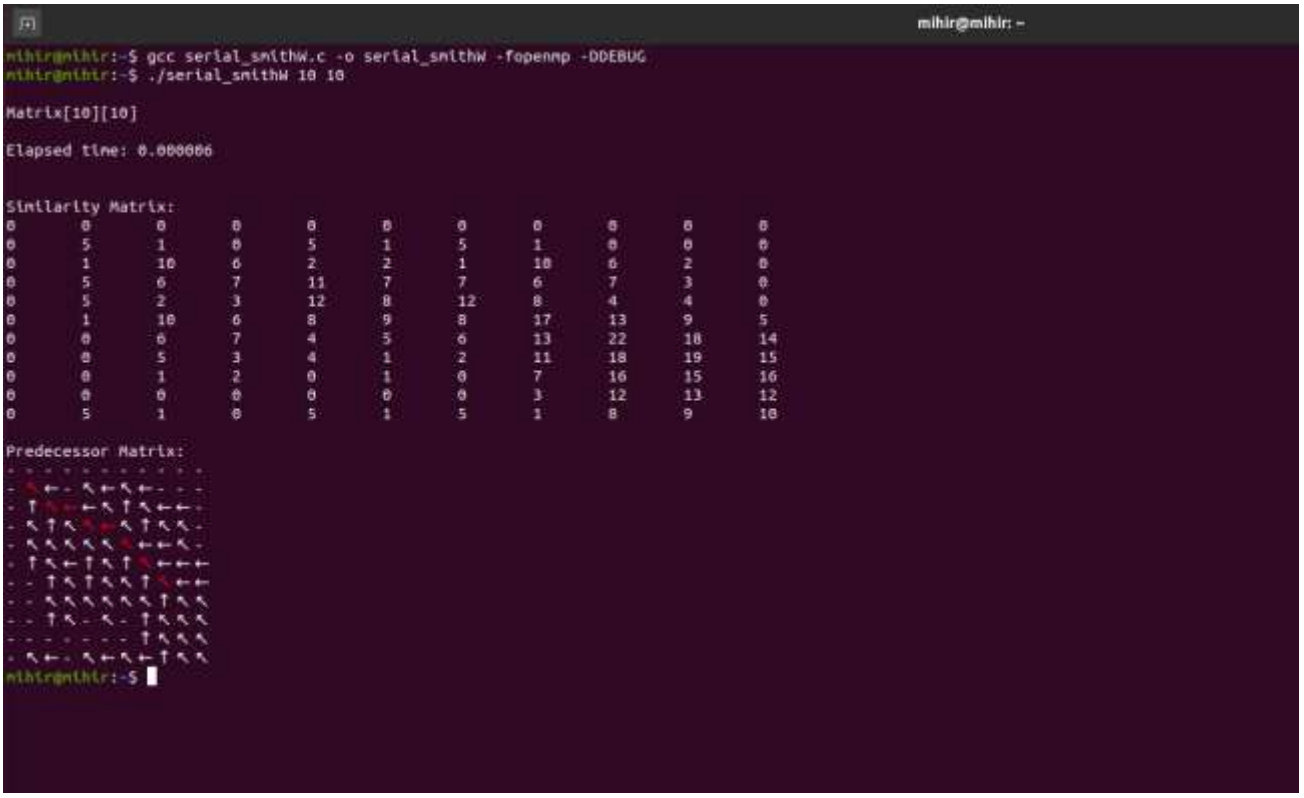
12.Contribution

Mihir Humar	**Problem statement, Domain Introduction, Techniques and related challenges, objective, Tools description, parallel code implementation**
Kartikay Gupta	**Research Findings Individual Contribution, Approach, Significance, Background Motivation , Methedology, serial code implementation**

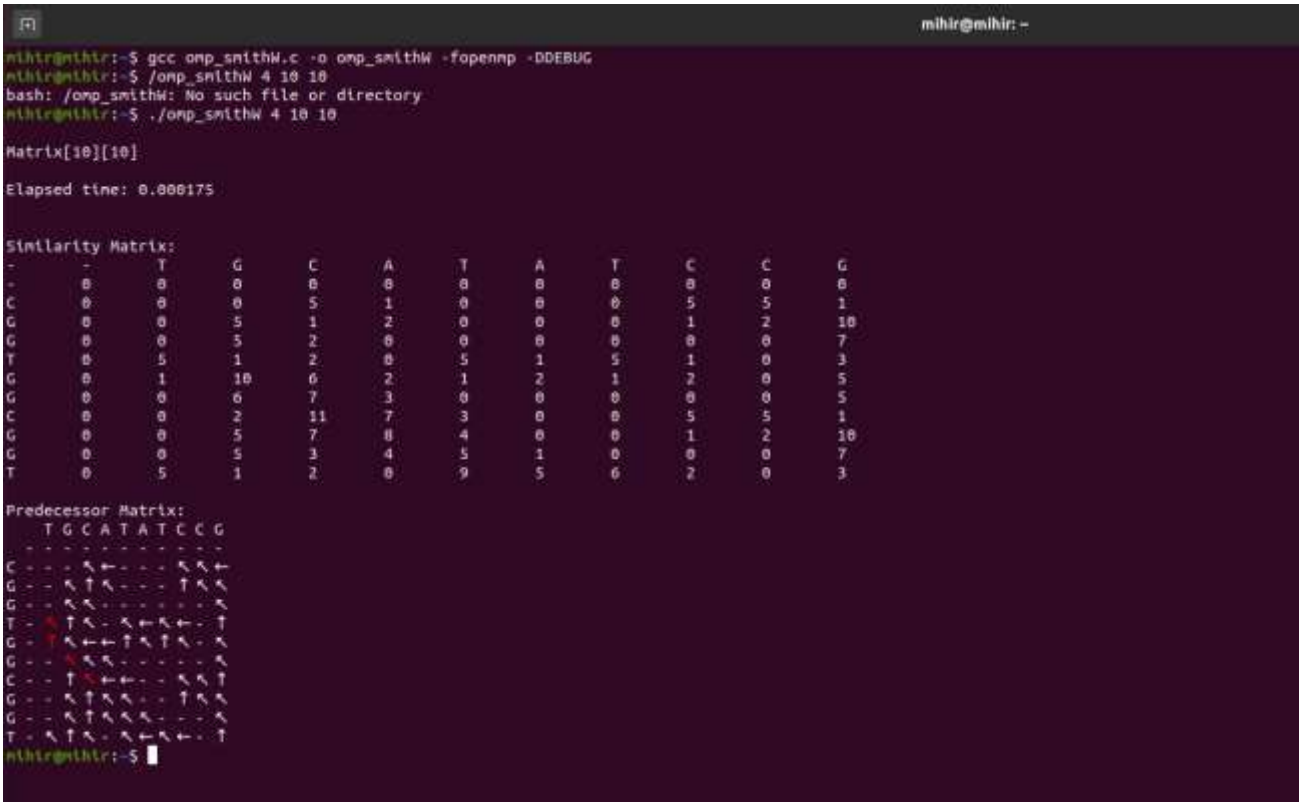
13.Result and Discussion

Output-

Serial implementation –



Parallel Implementation –



14.Result Analysis

The code was run for various lengths of sequences, and the elapsed time was recorded in each case (as shown in the aforementioned outputs). After tabulating all the execution times, we get:

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

15. Conclusion & Further Study

As we can see in the above graph, for small lengths of the sequence, serial and parallel programs tend to give the output in almost the same amount of time. However, as the length increases, the execution time for serial implementation also increases exponentially, hence forming a steep graph. However, the parallel implementation remains stable with not a high rise in the execution time because of the parallel execution of the task with two threads, making the process faster than its serial counterparts.

16.References:

[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 26

[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] H. Li, R. Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler transform", Bioinformatic., vol. 26, pp. 589-595, 2010.27

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

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

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

[18] 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