**PDC PROJECT DOCUMENTATION (CS-3006):**

**SEQUENCE ALIGNMENT PROBLEM**



**SUBMITTED TO:**

**DR NADEEM KAFI**

**GROUP MEMBERS:**

**INSHA SAMNANI (20K-0247)**

**ISMAIL AHMED ANSARI (20K-0228)**

**INTRODUCTION:**

The process of analyzing and identifying commonalities between biological sequences is known as **SEQUENCE ALLIGNMENT**. The "**similarities**" that are being found will vary depending on the objectives of the specific alignment process. Numerous bioinformatics applications seem to benefit greatly from sequence alignment. Exact solutions exist for the challenges of calculating **edit distance** and different forms of sequence alignment, such as the methods proposed by **Smith and Waterman (1981)** **and Needleman and Wunsch (1970)**. However, due to their lengthy execution times and memory needs, these methods appear to be unsuitable for DNA alignment in most real-world scenarios.

**Saul B. Needleman** and **Christian D. Wunsch** devised a dynamic programming algorithm to the sequence alignment problem and got it published in **1970**. Since then, numerous improvements have been made to improve the time complexity and space complexity. We got the serial implementation of sequence alignment problem from **Geeks for Geeks**.

**WHAT PROBLEM SEQUENCE ALLIGNMENT SOLVING?**

1. The Sequence Alignment problem is one of the **fundamental problems** of Biological Sciences, aimed at finding the **similarity of two amino-acid sequences**. Comparing amino acids is of prime importance to humans since it gives vital information on evolution and development.
2. Sequence alignment is very widely used in the **biological literature** to demonstrate **conserved regions in a protein alignment**, which we assume to have great functional importance. They may also be used to demonstrate **homology** between a protein family and a distantly related member.
3. Sequence alignments may also be used to investigate **conservation of protein structure** or to predict the **structures of new members** when we know the [tertiary structures](https://www.sciencedirect.com/topics/biochemistry-genetics-and-molecular-biology/tertiary-structure) of one or more members of a sequence data set.

**WHAT ARE THE SIMILAR SOLUTIONS?**

1. We can compute the sequence alignment by making manual combinations. It will use a brute force approach by continuously recomputing the already computed sequence alignments. Due to these repeated computations, the time complexity is much worse which is **O(2^n).**
2. The same problem can be solved by using greedy approach by selecting the sequence alignment with minimum penalty at each stage of computation. Although its time complexity will be **better O (from logn - (n^k) logn)** than brute force approach, but it must be considered that it can ignore the optimum minimum penalty that could be the result. Hence giving you wrong sequence alignment in some cases.
3. The Smith–Waterman algorithm performs local sequence alignment dynamically that is, for determining similar regions between two strings. Instead of looking at the entire sequence. Smith–Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure. Because of its quadratic complexity **O (n\*m)** in time and space, it often cannot be practically applied to large-scale problems.

The alternative solution for the problem of sequence alignment is by using the dynamic approach. This approach will introduce gap into the strings as to equalize the lengths. Since

it can be easily proved that the addition of the extra gaps after equalizing the lengths will only lead to increment of penalty, but the time complexity **(O(n\*m))** will be much lesser than that of brute force approach and maybe possible of greedy approach.

**WORKING OF OUR SELECTED ALGORITHM (Needleman Wunsch Algorithm):**

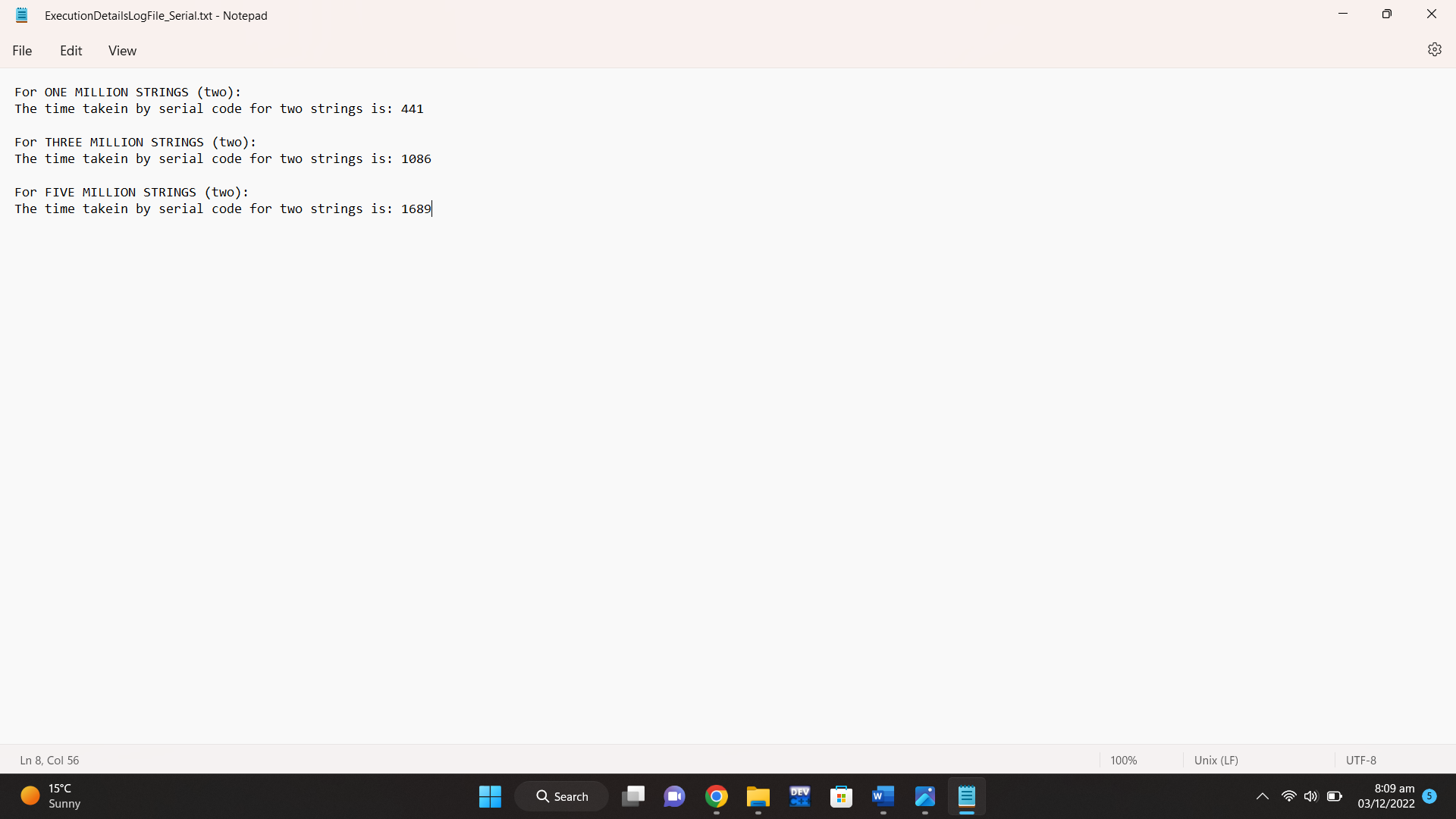
Firstly, it will take **two strings** from user in **two vectors**. It will then **concatenate** those strings up to **1**, **3**, and **5** **million**. Then if will take **gap** and **mis match penalties** from user. This complete data will be sent to a function which will **align** those two strings and save the result in the file of ‘**gene1.txt**’ for string1 and ‘**gene2.txt**’ for string2. The function will first calculate the **lengths of both vectors** (m, n) and will make a **2D vector dp** of m+1 \* n+1. This 2D vector’s **1st row** and **1st column** will be **filled** by gap penalties \* i. Next, the whole 2D vector will be **iterated** and **filled** by gap and mismatch penalties with respect to their neighboring and diagonal elements. After that, it will create **two vectors xans** and **yans** of size n+m. Further, the filled 2D vector will be **backtraced** from **last** element to the **first** element to **fill** xans and yans with the original letter, mismatched letter, or gap with respect to **lowest** **penalty** respectively. Next, the remaining size of xans and yans will be filled with gaps. These gaps will be later removed to **finalize** the two aligned strings to write them back to their respective files.

**OBJECTIVE OF THIS PROJECT:**

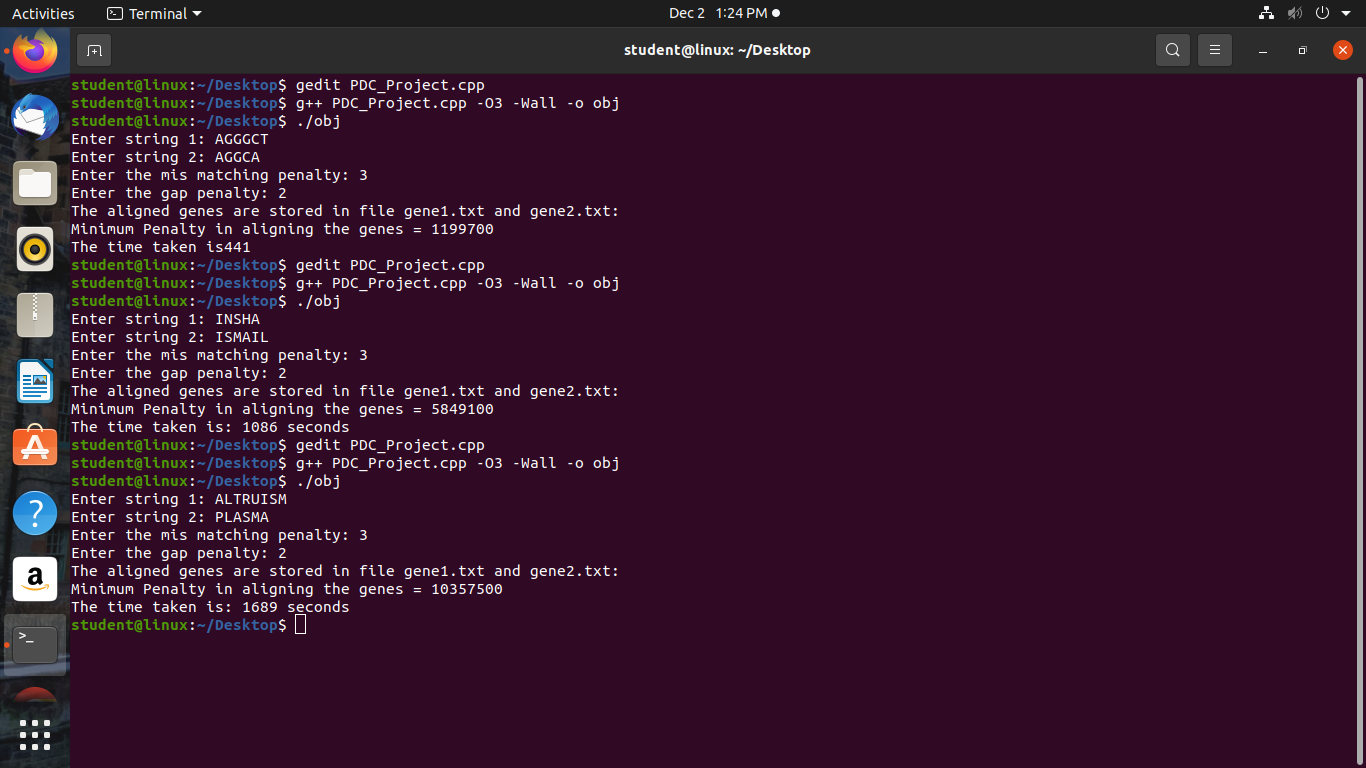
Our objective is to **parallelize** the given algorithm so that it takes less time to execute as compared to its **serial implementation**. As we know that space is very cheap as compared to time. If we execute the serial code on the strings of **one, three** or **five million**, it will take time ranging from **7 minutes – 29 minutes**. What if we can optimize this time from **1 minute – 4 minutes**. Meanwhile, we will be able to align **three to four times** different strings in the parallel implementation than serial. Secondly, our objective is to identify the **specific number of threads** on which our program is executing ideally. For example, it is not necessary that our parallel implementation will execute takings less time than serial on any number of threads. Increasing the number of threads will increase the **overhead of context switching** and **system resources** simultaneously.

**SERIAL IMPLEMENTATION:**

We executed the serial code on the strings of **1 million, 3 million** and **5 million** respectively. On the strings of 1 million, it took **441 seconds (7 minutes 21 seconds)**. On the strings of 3 million, it took **1086 seconds (18 minutes 16 seconds).** On the strings of 5 million, it took **1689 seconds (28 minutes 9 seconds)**. As we can see that the increasing length of strings is increasing the time of execution simultaneously on large scale. On increasing the size of **2 million** respectively, it is taking **10 more minutes** to align those strings. Here is the snapshot of **log file** for the execution of serial code:



The snapshot of **execution of serial code** is as follow:

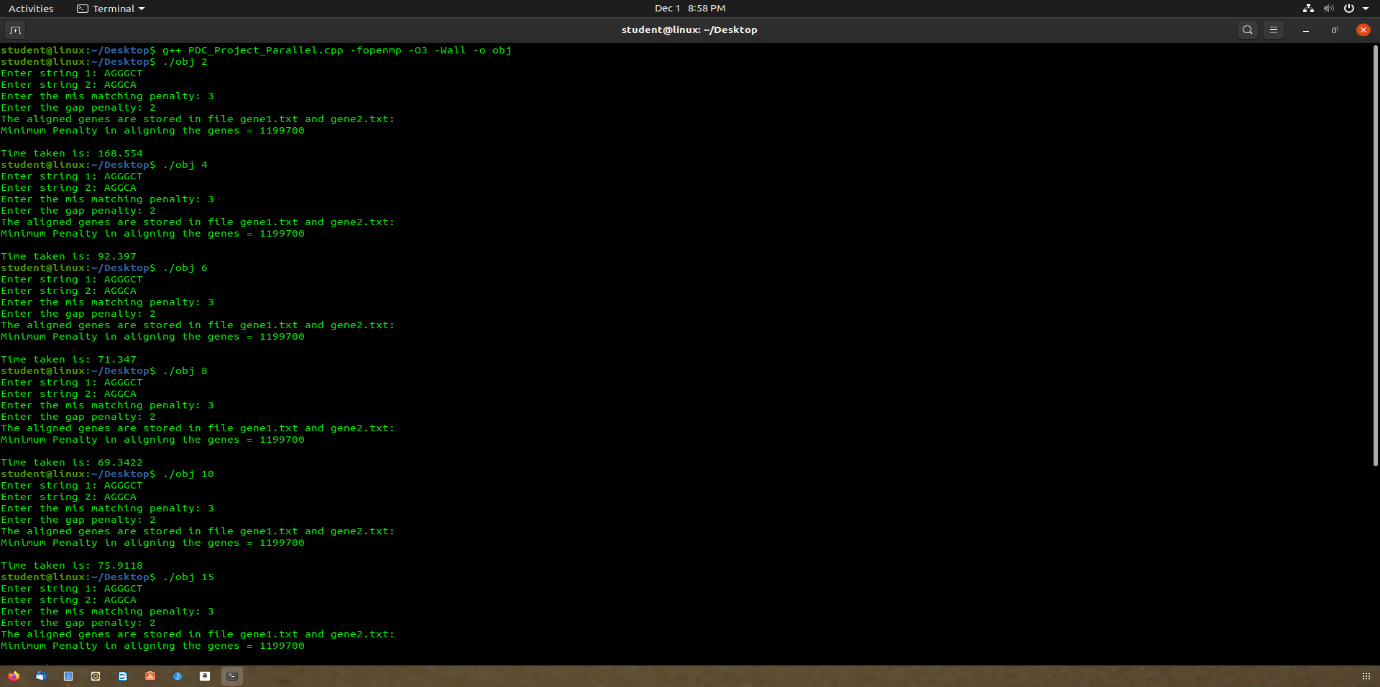


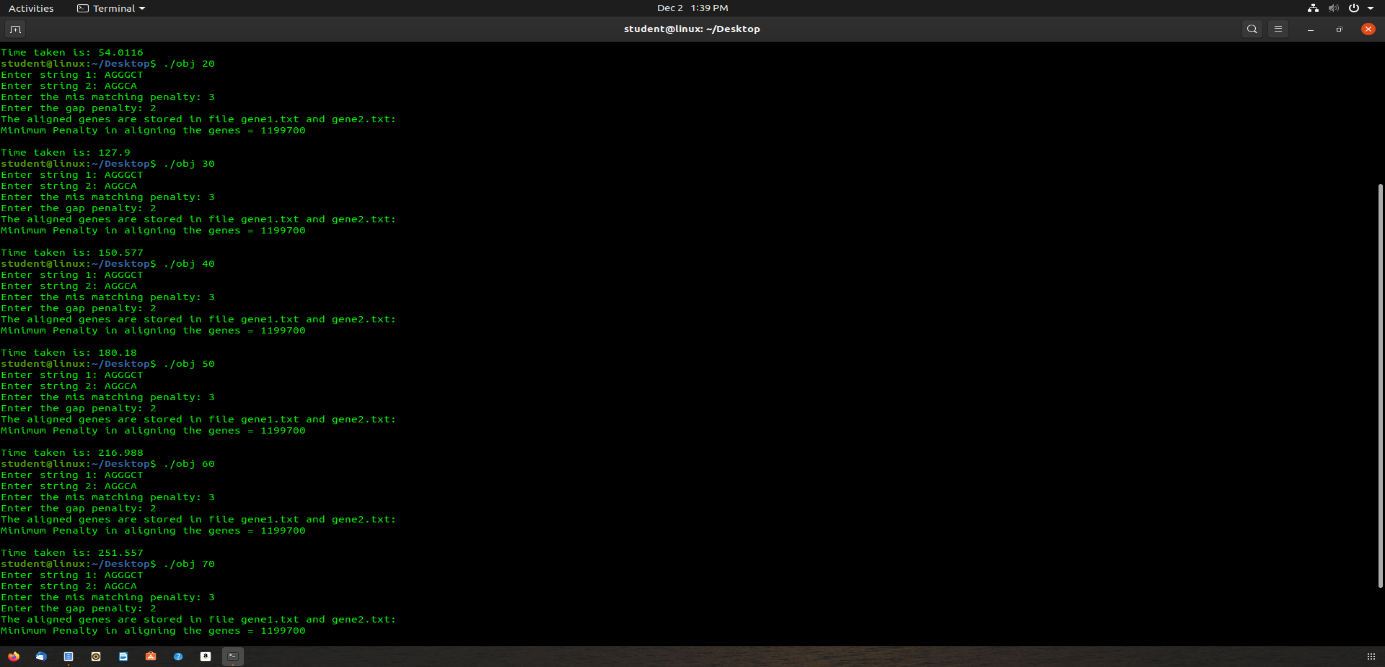
**PROBLEM WITH THIS SERIAL CODE (As to why parallel programming?):**

As the time complexity of this algorithm to solve sequence alignment problem is **O (n\*m).** Practically, it can slow down the computation of minimum penalty on large scaled sequences. What if we can optimize its execution by hiding the latency of code by parallelly executing it. It will help us to divide the iterations to calculate the penalty of sequence alignments across multiple threads, simultaneously executing the chunks of few iterations, taking same time as before but this time **parallelly.** This is what **OpenMP** performs. Each single loop will be executed parallelly to initialize the dynamic 2D matrix. Next, to compute each index of the 2D array, outer loop will be executed sequentially, and the result of the former is depended on latter. But the inner loop will be executed parallelly as each index in the same row is independent of the next. The next three single loops to calculate and display the sequence alignments will also be parallelly executed.

**PARALLEL IMPLEMENTATION USING OPENMP:**

We executed the parallel code on the string of **1 million**. The chosen number of threads were **2, 4, 6, 8, 10, 15, 20, 30, 40, 50, 60, 70, 75, 80, 90, 100, 150**. Up to thread **15** the execution time to align two strings was **fluctuating between more and less** but keeping it less from that of serial code. As we saw that the serial code took **441** seconds. The openmp implementation took **168.554** seconds on **2** threads, **92.397** seconds on **4** threads, **71.347** seconds on **6** threads, **69.3422** seconds on **8** threads, **75.9118** seconds on **10** threads, and **54.2566** seconds on **15** threads. This execution helps us to observe that the group of **15 threads** worked **ideal** for the openmp implementation. It took **less than a minute** to align two strings of 1 million. We **optimize** the serial implementation up to **7 times** which is a huge gap. Next up to **100 threads** we observe that the time was **constantly increasing**. It took **127.9** seconds on **20** threads, **150.577** seconds on **30** threads, **180.18** seconds on **40** threads, **216.988** seconds on **50** threads, **251.557** seconds on **60** threads, **284.185** seconds on **70** threads, **305.869** seconds on **75** threads, **323.247** seconds on **80** threads, **357.057** seconds on **90** threads, and **394.752** seconds on **100** threads. Up to **100** threads, the time **being less** than that of the time taken on the execution of serial code. But once we executed it for **150 threads**, it took **573.422 seconds** (around 9 minutes 33.422 seconds). **2 minutes** exceeded than serial implementation. Here we **stopped** the execution of openmp implementation on the strings of 1 million. The snapshot of **execution of parallel code on the strings of 1 million** is as follow:





A screenshot of a computer

Description automatically generated

Next, we executed the parallel code on the string of **3 million**. The chosen number of threads were **2, 4, 6, 8, 10, 15, 20, 30, 40, 50, 60, 70, 75, 80, 90, 100, 150**. Up to thread **15** the execution time to align two strings was **fluctuating between more and less** but keeping it less from that of serial code. As we saw that the serial code took **1086** seconds. The openmp implementation took **526.802** seconds on **2** threads, **283.955** seconds on **4** threads, **226.237** seconds on **6** threads, **205.068** seconds on **8** threads, **237.708** seconds on **10** threads, and **165.639** seconds on **15** threads. This execution helps us to observe that the group of **15 threads** worked **idea**l for the openmp implementation. It took **2 minutes 45.639 seconds** to align two strings of 3 million. We **optimize** the serial implementation up to **7-8 times** which is a huge gap. Next up to **90 threads** we observe that the time was **constantly increasing**. It took **396.683** seconds on **20** threads, **461.083** seconds on **30** threads, **558.163** seconds on **40** threads, **664.821** seconds on **50** threads, **767.931** seconds on **60** threads, **872.878** seconds on **70** threads, **922.725** seconds on **75** threads, **977.2** seconds on **80** threads, and **1087.54** seconds on **90** threads. Up to **90** threads, the time **being less** than that of the time taken on the execution of serial code. But once we executed it for **100 and 150 threads**, it took **1200.84** **seconds** (around 20 minutes 0.84 seconds) on 100 threads and **1739.64 seconds** (around 28 minutes 59.64 seconds) on 150 threads. Here we **stopped** the execution of openmp implementation on the strings of 3 million. The snapshot of **execution of parallel code on the strings of 3 million** is as follow:

Text

Description automatically generated

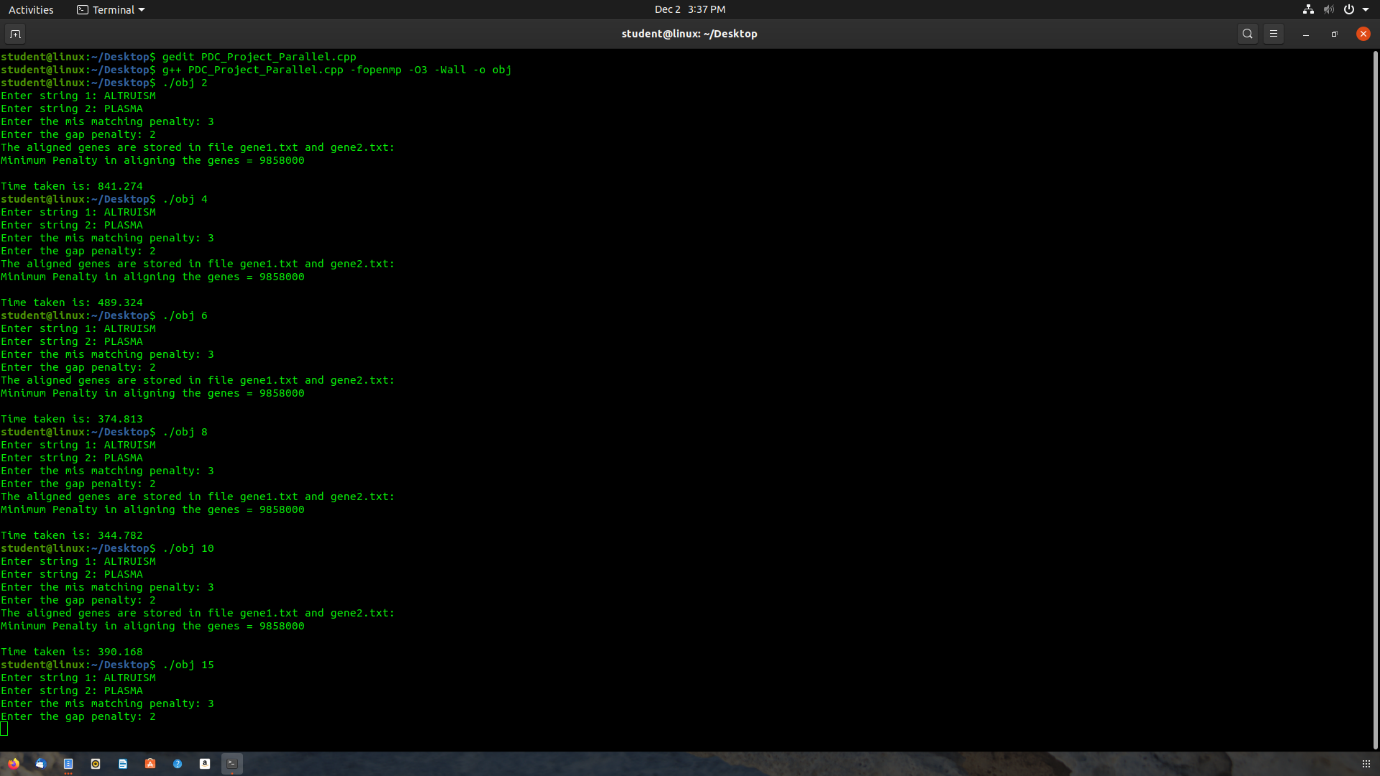
Text

Description automatically generated

Text

Description automatically generated

Next, we executed the parallel code on the string of **5 million**. The chosen number of threads were **2, 4, 6, 8, 10, 15, 20, 30, 40, 50, 60, 70, 75, 80, 90, 100, 150**. Up to thread **15** the execution time to align two strings was **fluctuating between more and less** but keeping it less from that of serial code. As we saw that the serial code took **1689** seconds. The openmp implementation took **841.274** seconds on **2** threads, **489.324** seconds on **4** threads, **374.813** seconds on **6** threads, **344.782** seconds on **8** threads, **390.168** seconds on **10** threads, and **275.796** seconds on **15** threads. This execution helps us to observe that the group of **15 threads** worked **ideal** for the openmp implementation. It took **4 minutes 35.796 seconds** to align two strings of 5 million. We **optimize** the serial implementation up to **7 times** which is a huge gap. Next up to **80 threads** we observe that the time was **constantly increasing**. It took **658.879** seconds on **20 t**hreads, **766.884** seconds on **30** threads, **933.764** seconds on **40** threads, **1108.27** seconds on **50** threads, **1281.54** seconds on **60** threads, **1438.5** seconds on **70** threads, **1541.39** seconds on **75** threads, and **1638.05** seconds on **80** threads. Up to **80** threads, the time **being less** than that of the time taken on the execution of serial code. But once we executed it for **90, 100, and 150 threads**, it took **1806.03 seconds** (around 30 minutes 6.03 seconds) on 90 threads, **1996.71 seconds** (around 33 minutes 16.71 seconds) on 100 threads, and **2947.5 seconds** (around 49 minutes 7.5 seconds) on 150 threads. Here we **stopped** the execution of openmp implementation on the strings of 5 million. The snapshot of **execution of parallel code on the strings of 5 million** is as follow:



A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

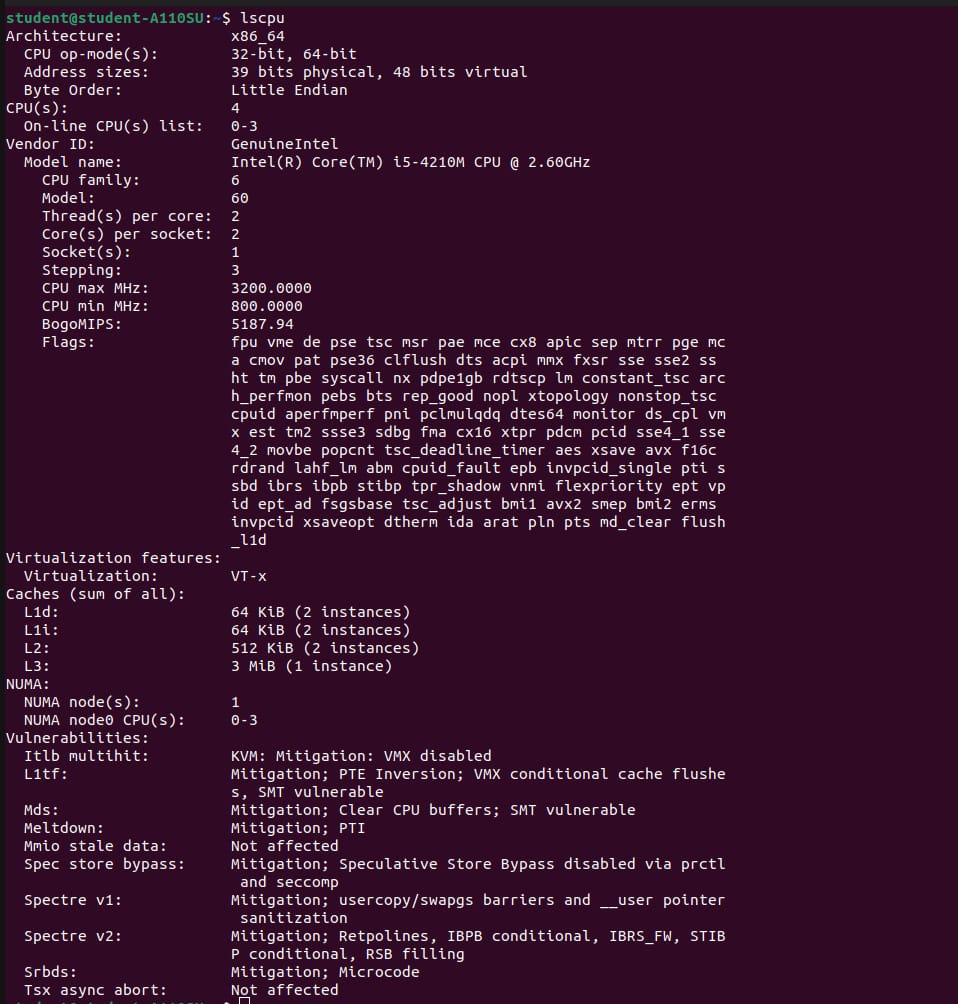
Description automatically generated

Text

Description automatically generated

**ARCHITECTURAL DETAILS:**

We performed the analysis of serial and openmp implementation of sequence alignment on **lab12 PC** having **4 threads per core** (for first computer)and **2 threads per core** (for second computer) and **16GB RAM**. Further architectural details are attached here:



**ANALYSIS OF PARALLEL CONSTRUCTS WITHIN OPENMP IMPLEMENTATION:**

First, the **number of threads** were taken from user through command line.

1. We haven’t placed any pragma for concatenating the strings up to 1,3 or 5 million as we know that scheduling of threads occurs arbitrarily. It was possible that it concatenate the strings in an improper way. This section of concatenation of strings was executed serially.
2. The 2D vector dp was allocated memory parallelly as there was no dependence between the iterations of for loop in memory allocation. Hence, we used #pragma omp parallel for num\_threads(numthreads). Simply parallelism applied on for loop without any concerns as the shared variable is used only as read-only variable. Rest of the calculations are independent.
3. The 1st row and 1st column of dp vector was initialized parallelly as there was no dependence between the iterations of for loop in initializing it. Hence, we used #pragma omp parallel for num\_threads(numthreads). Simply parallelism applied on for loop without any concerns as the shared variable is used only as read-only variable. Rest of the calculations are independent.
4. Calculating minimum penalty section was executed serially as there were no for loops, only if else blocks were executing.
5. The code (of filling out dp matrix) is modified (from serial) to achieve parallelism for the very important part of code. The original block of code was not supporting the parallelism because each iteration of the outer and inner for loop was dependent on each other. And the parallelism for the program except that part of code was not worth. After the modification of code, to calculate the matrix dp as per the rules of dynamic programming, it was observed that at each iteration of nested for loop, the previous elements are required. Hence it could be dangerous to use parallel for clause for outer loop as well as inner loop. As it was possible that the thread executing the inner loop iteration, demanding for previous elements of dp vector that are not yet calculated by the other threads, reading the garbage values. Therefore, we decided to go only for inner loop parallelism. Hence, we used #pragma omp parallel num\_threads(numthreads) before outer loop and #pragma omp for for inner loop.
6. The same parallelism explained in point 5 was applied for next 3 nested for loops sections.
7. Now there is a while loop to extract the aligned strings from 2D vector. This while loop can easily be converted in for loop, we tried to do so to apply parallel for clause. Since each variable is being updated at each iteration, it would require blocking whole section of code (if else if clauses) OR apply multiple critical sections to synchronize the updating of each variable. Hence being time consuming and increasing parallel overhead, it is better to sequentially execute this while loop.
8. To fill the remaining portions of new aligned strings with gaps, we were trying to nest for loop within sections. To complete the initialization of arrays xans and yans, being independent, we created two sections for them. Later we come to know that we can't nest for loop within sections as for loop require all threads assigned to the code to participate in its iterations whereas the threads are divided between sections. Hence, we thought to make 2 functions, each called by respective section. Then we used parallel for clause within those functions. Hence, to create sections, we used #pragma omp parallel sections num\_threads(2) and #pragma omp section.
9. Now each function have only concern while applying parallel for clause is of updating shared variable i or j and using it as index to fill up the xans or yans vector. Hence it was required to either lock the variable or the code snippet or apply reduction. Locking the code snippet is not required here so no use of critical as it is less efficient also. Also, we can apply atomic, but what if hardware instructions are not available at the time of code execution. Therefore, we thought to use reduction. Hence, we used #pragma omp parallel for reduction(-:i) num\_threads(numthreads) and #pragma omp parallel for reduction(-:j) num\_threads(numthreads).
10. Next, we modified the working of for loop to store the index from xand and yans so that we can store the aligned strings without false gaps in the file. The original code had for loop executing in reverse. We tried to execute it in forward direction with parallel for clause so that we can store the last index in id variable where the condition met. But still the iteration of k that we require to get stored in id is possible to be executed before. CORRECT OUTPUT IS BETTER THAN FALSE PARALLELISM.
11. To store the updated aligned strings, we made sections. As we know that storing two different strings in two different files are completely independent tasks. Less benefit is better than 0. Hence, we can apply sections for them. Therefore, we used #pragma omp parallel sections num\_threads(2) and #pragma omp section.

**ANOTHER PARALLEL SOLUTION AND PROBLEMS IN THAT APPROACH?**

Other than **OpenMP**, **MPI** is a parallel solution for sequence alignment problem. Sequence Alignment using MPI will create unnecessary overheads and it will not be fast enough than OpenMP, because of the overhead in communicating with the output generated in the other processor and then linking with one input three times for generating dynamic 2D array. For every output generation on the matrix unnecessary communication on different processors will halt which increase the overhead of communication.

**CONCLUSION:**

In conclusion, we have **tons** of approaches to save time in execution of serial implementations. It depends on us that **how** we are utilizing these resources to parallelize our codes. What if we start parallelizing **all** the codes in serial? We will be able to do **three times** the same work in a particular amount of time that the serial code is doing only **one time**. Also, it is not necessary that we should parallelize whole code. It will depend on the working of different portions of code that do they require parallelism or **not**? And will they execute **faster** being in parallel portion or serial? Also, it is not hard and fast to use **one type of pragma** for all portions. It depends that what the code requires. For example, one portion of code can be solved with **reduction**, **critical**, and **atomic**. So, you think you can apply **any** one of those. **No**, it depends that what your code **demands**. Although if **all three works** and parallelize your code but it is **guaranteed** that only one of this will come up with **efficient parallelism**. Hence, parallelizing the code **is not that easy but that much beneficial**.

**REFERENCES:**

* [*https://www.geeksforgeeks.org/sequence-alignment-problem/*](https://www.geeksforgeeks.org/sequence-alignment-problem/)
* *https://stackoverflow.com/questions/7916411/openmp-for-loop-inside-section*