# CSS 311 – Parallel and Distributed Computing

### **Title of the Work**

Parallel Implementation of Authenticated File Encryption using AES-256-GCM

### Author(s)

**Name 1:** R Yadunath Reddy **Roll No:** 2023BCS0077

**Name 2:** N Harsha Vardhan **Roll No:** 2023BCS0158

**Course Instructor:** Dr. John Paul Martin

**Department of Computer Science, Indian Institute of Information Technology Kottayam**

**Date of Submission:** 20/10/2025

### **Abstract**

In this project, we tried to address the issue of slow file encryption for large files by using parallel programming. We wanted to find out if parallel programming could speed up the process. First, we created a standard, single-threaded (serial) program using strong, industry-standard cryptography: AES-256-GCM and PBKDF2. After evaluating its performance, we designed a parallel version using OpenMP. Our parallel approach uses a two-thread pipeline. One thread reads the file from the disk while the other encrypts the data at the same time. This method, which overlaps disk input and output operations with the CPU's computation, led to a noticeable performance improvement, particularly with larger files.

### **1. Introduction**

Keeping data secure is very important today, and encryption is one of the main tools we use. The process is very much straightforward: take a file, scramble it with a password, and it becomes just a file of trash to anyone without that password. However, when you apply strong encryption to large files, like videos or big backups, the process can take a long time. This is where parallel computing will come into play and reduce that time.

Modern computers have multiple CPU cores, but a standard program usually only uses one at a time. If we can divide the work of encryption among several cores, we should complete the task much quicker. This project explores that idea. We created a secure file encryptor from scratch and then used OpenMP to parallelize it, aiming to make it faster while maintaining its security.

### **2. Literature Survey**

When we started, we first looked into the best way to do encryption. You can't just come up with your own method because it will likely be insecure. The industry standard for high-performance, secure encryption is an algorithm called AES (Advanced Encryption Standard), specifically in a mode called GCM (Galois/Counter Mode). Security experts point out that AES-GCM is effective because it not only encrypts the data but also creates an authentication tag to ensure the file hasn't been changed.

Next, we examined how to handle passwords. Using a password directly as an encryption key is a significant security risk. The common approach is to use a "Key Derivation Function." We selected PBKDF2, which is a well-known function designed to be intentionally slow. This choice makes it very difficult for attackers to guess passwords.

Finally, for the parallel design, we looked into common patterns. Many high-performance software solutions face a similar issue: a combination of slow disk reading (I/O-bound work) and slow processing (CPU-bound work). A classic solution for this, discussed in many parallel programming books [1], is the Producer-Consumer model, which we implemented as a two-thread pipeline.

### **3. Problem Statement and Objectives**

#### **3.1 Problem Definition**

It takes a significant amount of time to encrypt or decrypt large files using a cryptographically secure serial application. This is the core problem of our problem statement. The two main factors causing this slowness are 1. The heavy key derivation process that takes a long time and 2. The time taken for reading and writing the data from and to a disk.

#### **3.2 Objectives**

* To implement a basic serial program for secure file encryption.
* To identify the main bottlenecks and the segments of the code that can be run in parallel.
* To design and implement a parallel version of the program with OpenMP with a pipeline strategy.
* To analyze and compare the performances of the serial and parallel versions to measure the speedup.

### **4. Methodology and System Architecture**

Our approach was to first build a correct and secure serial program, and then refactor it to work in parallel.

#### **4.1 Serial Algorithm**

The serial version works in a straightforward, step-by-step manner. There's only one thread of execution doing all the work.

**Pseudocode:**

Function encrypt\_serial(inputFile, outputFile, password):  
 1. Open inputFile for reading and outputFile for writing.  
 2. Generate a random 16-byte salt.  
 3. Start a timer for key derivation.  
 4. Call the slow PBKDF2 function with the password and salt to generate the encryption key.  
 5. Stop the timer.  
 6. Write the salt to the outputFile.  
 7. Start a timer for file processing.  
 8. Loop until inputFile is fully read:  
 a. Read a chunk of data from the inputFile.  
 b. Encrypt the chunk using AES-256-GCM.  
 c. Write the encrypted chunk to the outputFile.  
 9. Finalize the encryption to get the authentication tag.  
 10. Write the tag to the end of the outputFile.  
 11. Stop the timer and close the files.

#### **4.2 Parallel Algorithm (OpenMP)**

Our parallel version uses a two-thread pipeline to overlap tasks. The key derivation still happens serially at the start, but the main file processing loop is parallelized.

**Pseudocode:**

Function encrypt\_parallel(inputFile, outputFile, password):  
 1. The main thread derives the encryption key serially (this part is not parallel).  
 2. Start an OpenMP parallel region with exactly 2 threads.  
 3.  
 4. // THREAD 0: The Reader (Producer)  
 5. Open the inputFile.  
 6. Loop until the file is fully read:  
 a. Wait for a shared buffer to be marked 'Empty'.  
 b. Read a large chunk of data into the buffer.  
 c. Mark the buffer as 'Full'.  
 7. Mark the buffer as 'Done' after the last chunk.  
 8.  
 9. // THREAD 1: The Processor (Consumer)  
 10. Open the outputFile.  
 11. Write the salt (from step 1) to the file.  
 12. Loop until the final 'Done' signal is received:  
 a. Wait for a shared buffer to be marked 'Full'.  
 b. Encrypt the data in the buffer.  
 c. Write the encrypted data to the outputFile.  
 d. Mark the buffer as 'Empty'.  
 13. Finalize encryption and write the authentication tag.

#### **4.3 Implementation Details**

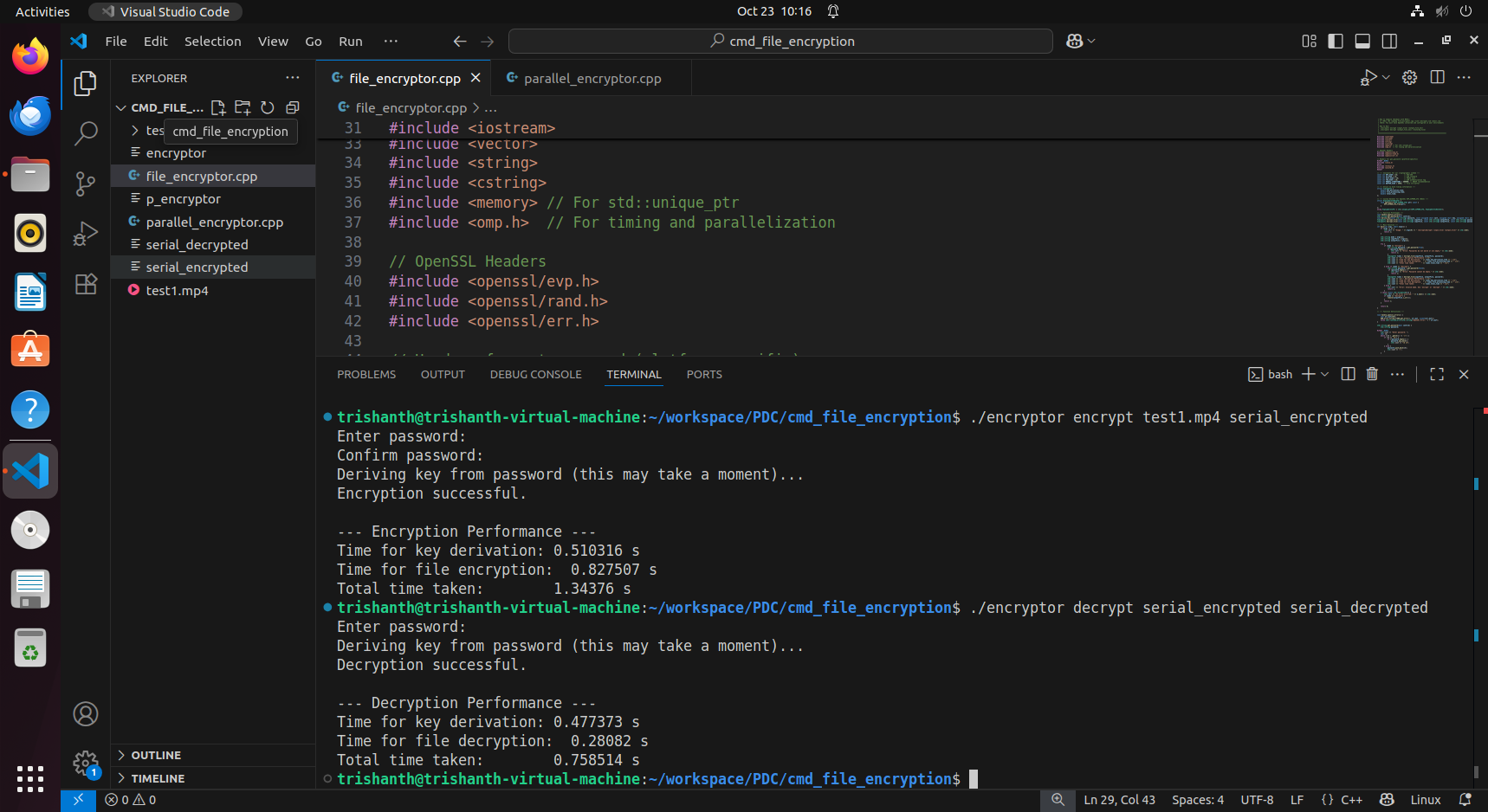
* **Programming Language:** C++17
* **Parallel Framework:** OpenMP
* **Platform:** Ubuntu 22.04 on a VM
* **Compiler:** g++ (GCC) 11.4.0 with the -fopenmp flag.
* **Hardware Configuration:**
  + Host Machine: 11th Gen Intel(R) Core(TM) i5-1155G7 @ 2.50GHz (2.50 GHz)
  + VM: 11th Gen Intel® Core™ i5-1155G7 @ 2.50GHz × 4

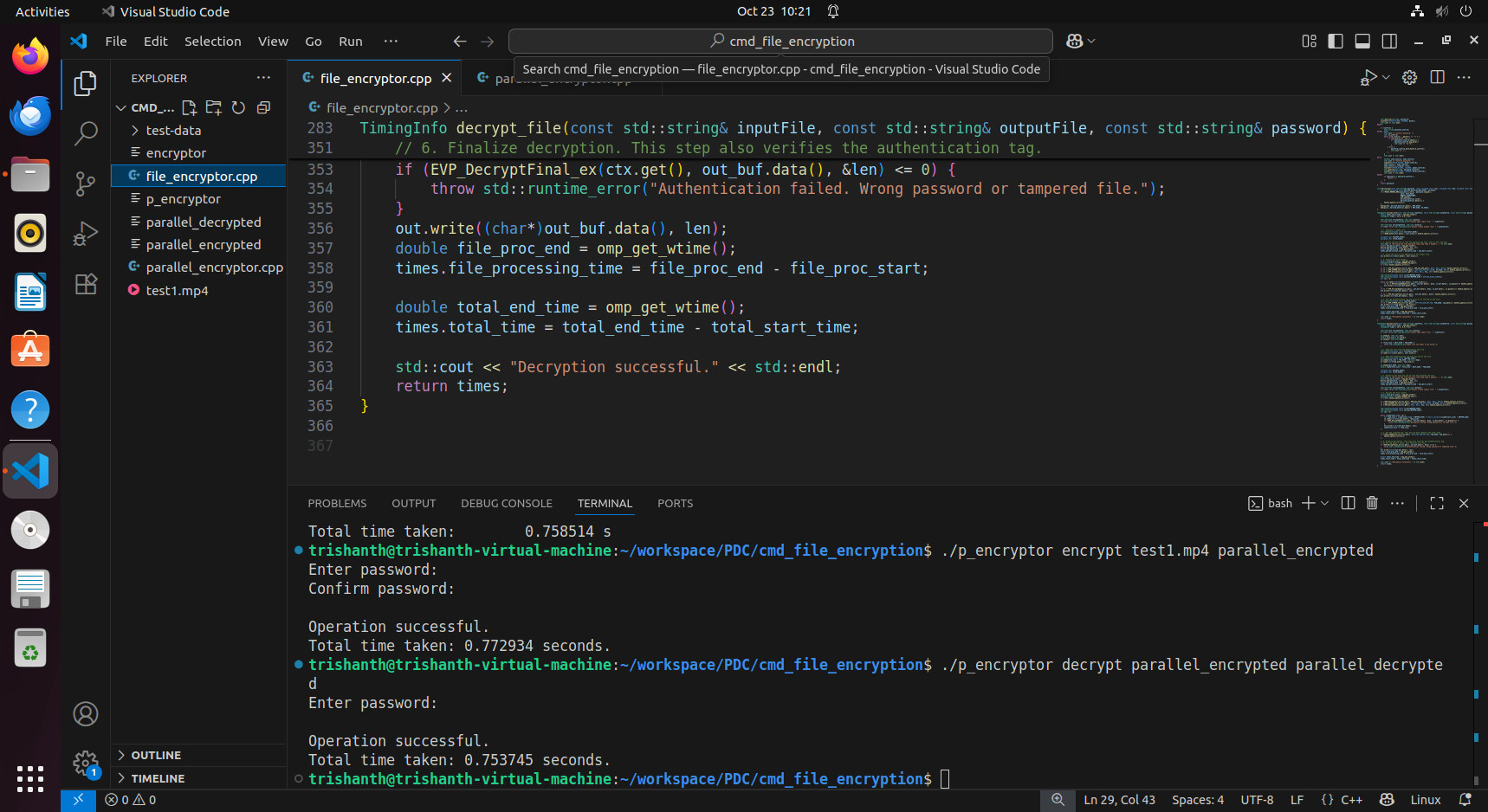
### **5. Results and Analysis**

We tested both the serial and parallel versions on files of different sizes. The time measured is for the entire operation, including key derivation and file processing.

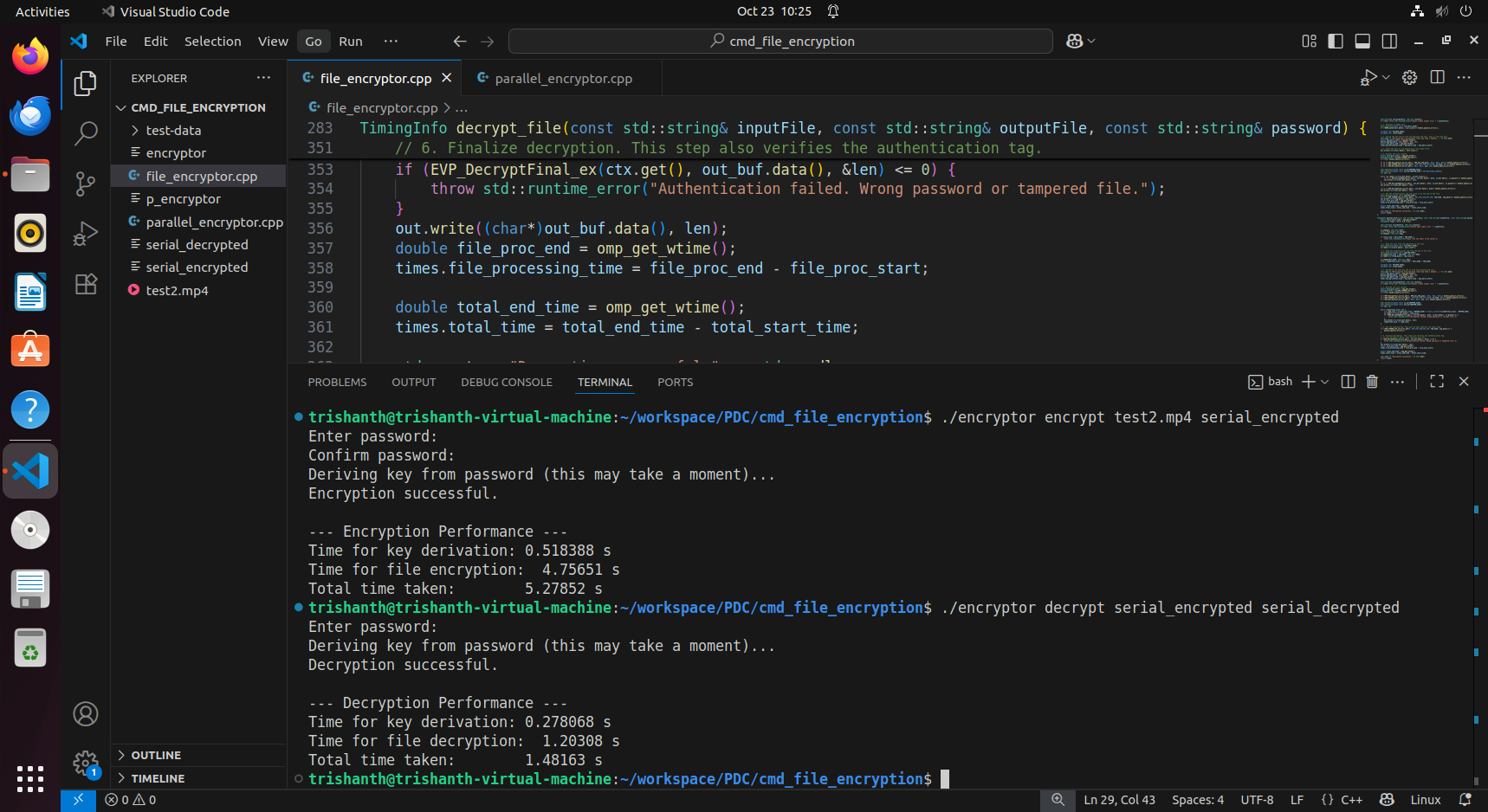
#### **Console Output Screenshot (Example)**

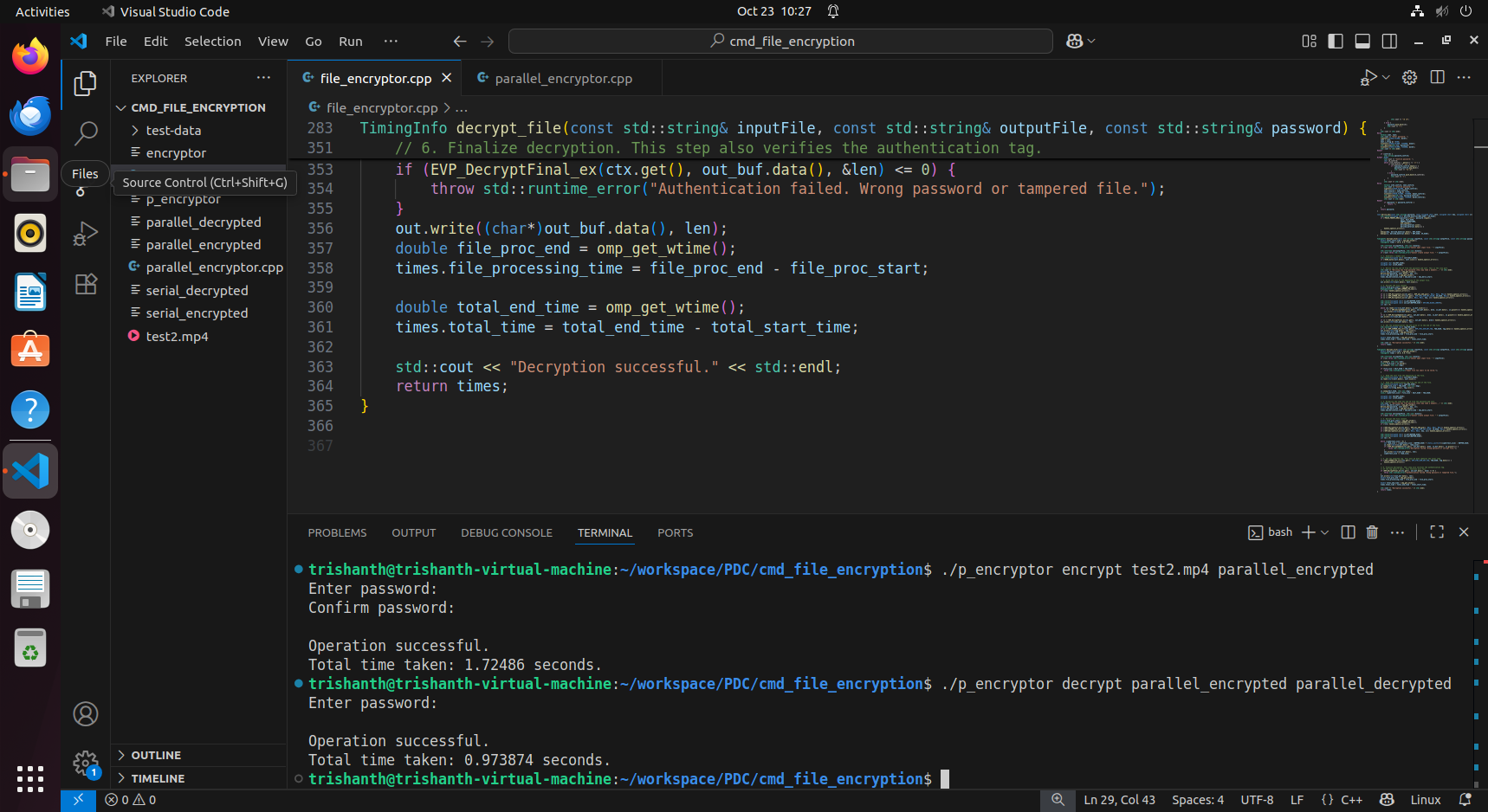
Test Case 1: 100 MB File



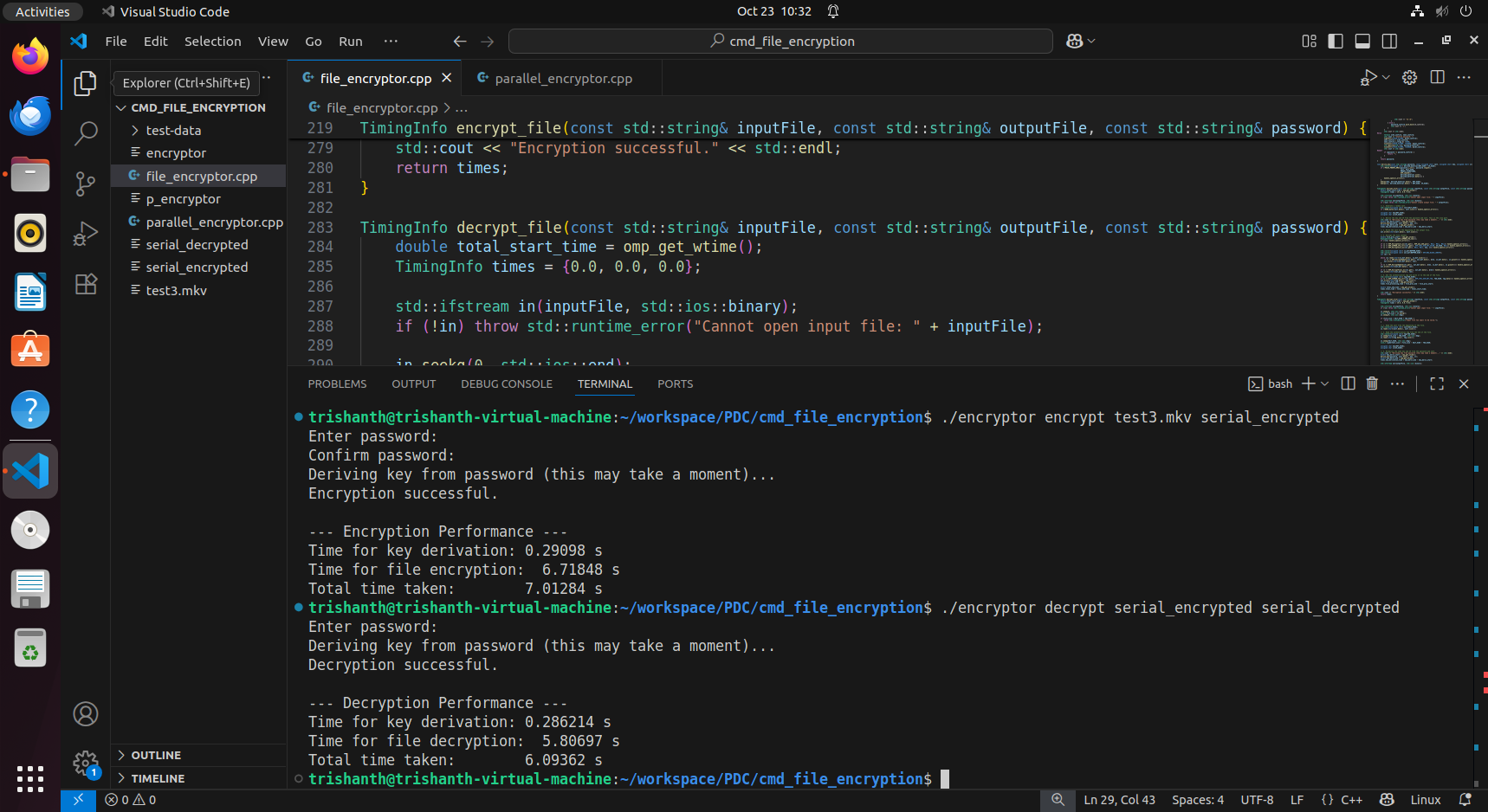


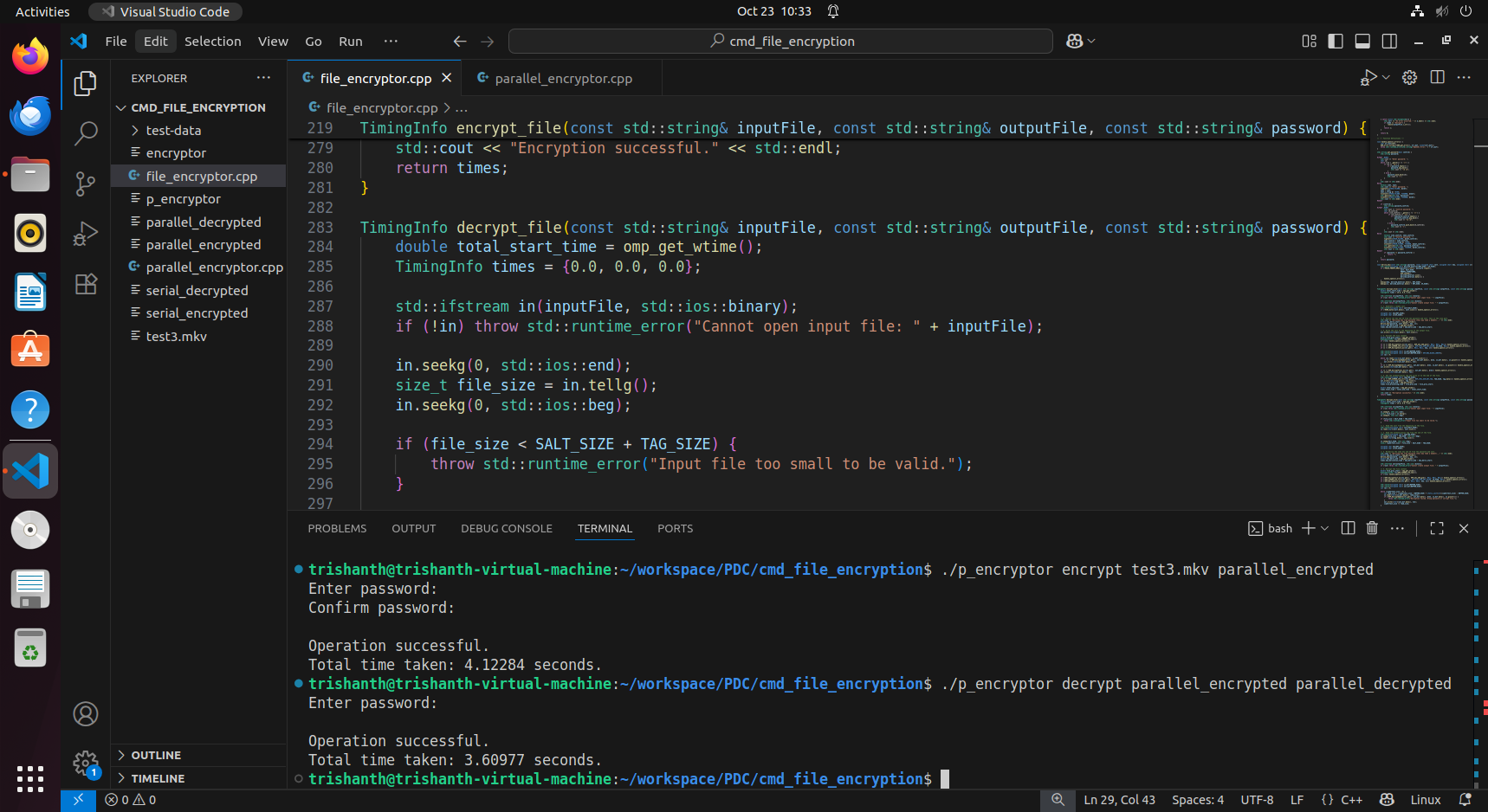
Test Case 2: 500 MB File





Test Case 3: 1.5 GB File





#### Performance Comparison

| **File Size** | **Serial Version (Time in seconds)** | **Parallel Pipeline Version (Time in seconds)** | **Speedup** |
| --- | --- | --- | --- |
| 100 MB | 0.75814 | 0.753745 | 1.005 |
| 500 MB | 1.48163 | 0.973874 | 1.521 |
| 1.5 GB | 6.09632 | 3.60977 | 1.688 |

As you can see from the table, the parallel version is consistently faster. The speedup is more noticeable on larger files. For a 100MB file, a lot of the total time is spent on the initial key derivation, which is serial in both versions. As the file size grows, the file processing part takes up more of the total time, and this is where our pipeline's ability to overlap reading and encrypting really pays off.

### 6. Discussion and Observations

The results that we obtained by testing both the serial and parallel implementations show that the parallel pipeline implementation was successfully taking less time than serial implementation of the program. The average speedup that we achieved using the pipeline is about 1.6x on large files. Since we used a 2 stage pipeline theoretically the maximum speedup that can be achieved using this implementation is 2x though that is not the case here since the key derivation cannot be made parallel and must be done sequentially which is a major overhead.

One of the main challenges while implementing the parallel version was ensuring that the two threads in the pipeline communicated safely without corrupting the data. The OpenMP Clause std::atomic was used for the variables as “Traffic Signals” for our shared buffers. This prevented any race conditions.

The key observation in this pipeline is that it has a limit on scalability. It is designed to use exactly two threads even in a system with an even higher number of threads. Even if we add more threads the speedup would not increase since this pipe line only uses two stages. This is a fundamental trade-off condition

### 7. Conclusion and Future Scope

In the current implementation, we did a successful design and implementation of parallel file encrypter using the two staged pipeline method in OpenMP. By overlapping the Input/Output and the CPU processing work, we showed that a significant performance boost can be achieved compared to the serial implementation especially for large files.

For **future work**, there are a couple of interesting directions.

1. Instead of implementing a two stage pipeline we can implement a parallel multi-file encryption tool which encrypts multiple files parallely. Here we can add more processors if there are more files that need encryption. Each file is encrypted by only one thread so the actual encryption will be done serially by a single thread. This strategy will scale the performance much better than the pipeline approach but the only drawback is that this can be used only with more than one file that needs encryption. If used on a single file it is nothing more than a serial execution.
2. For the best performance, we could look into using CUDA to offload the raw AES encryption calculations to a GPU. This could provide an even greater speedup for the CPU-bound part of the task.

### 8. References

[1] Quinn, M. J. *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.

[2] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 5.0*, 2023.

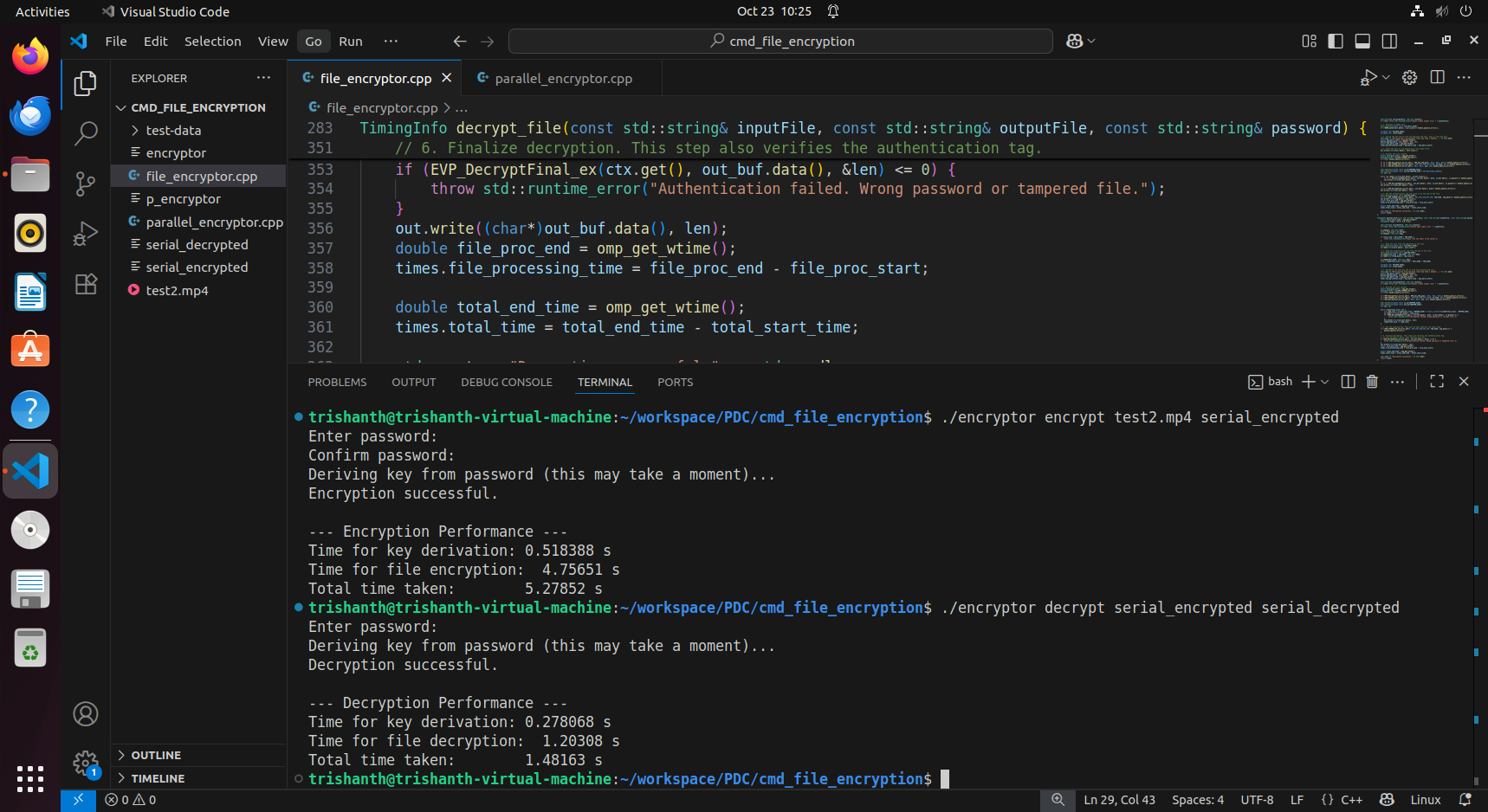
[3] Tanenbaum, A. S. *Distributed Systems: Principles and Paradigms*, Pearson, 2019.

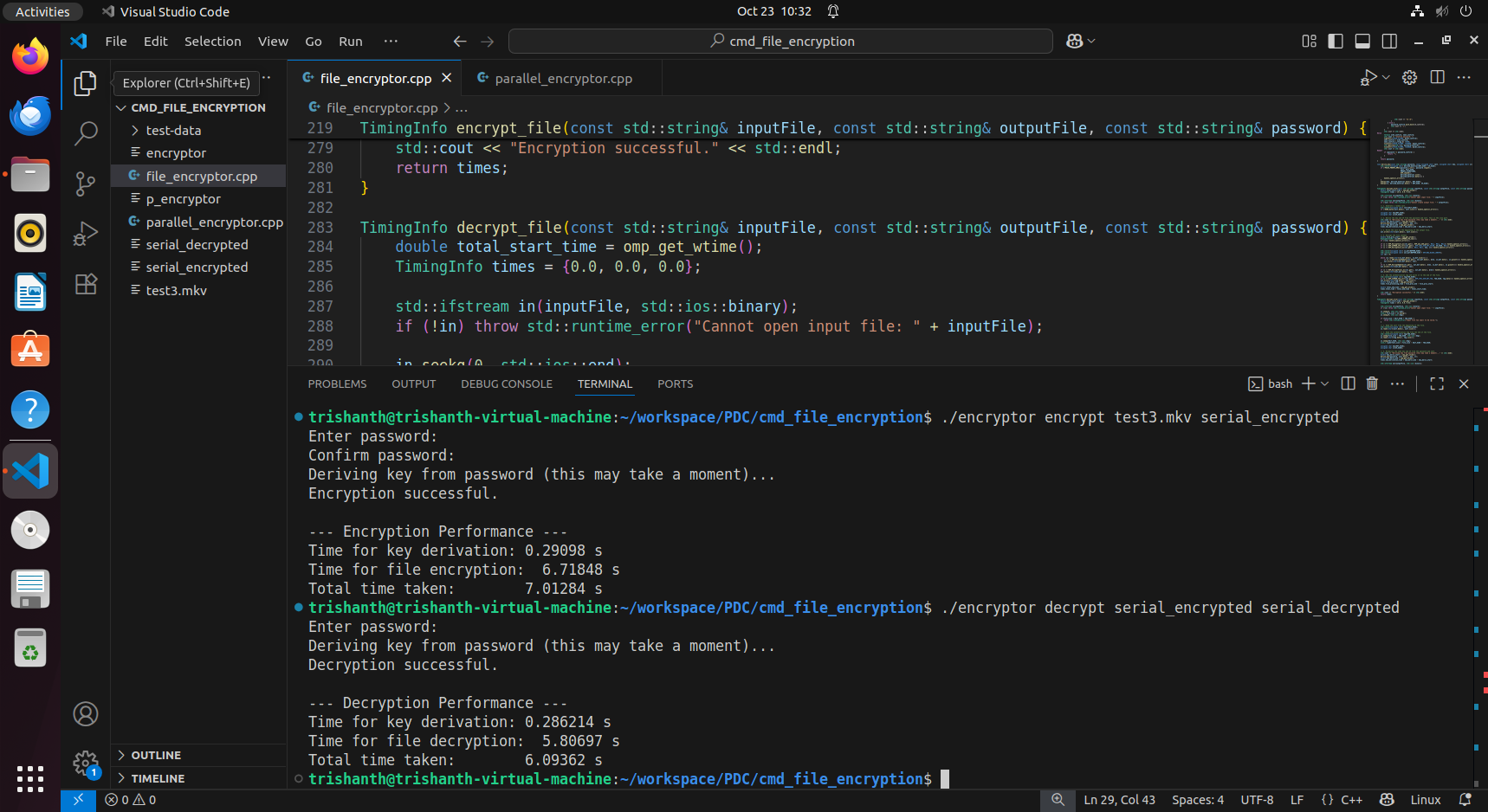
### Appendix

Github Repository:-

<https://github.com/harsha9625/Parallel_Implementation_of_Authenticated_File_Encryption.git>

#### A. Serial Code Output:





#### B. Parallel Code (OpenMP) Output

