BITS PILANI INTRODUCTION TO PARALLEL AND DISTRIBUTED COMPUTING

IPDP Assignment 1

GROUP 20

10/11/2024

Name	Email ID	BITS ID
VISHAL JEE PANDITA	2024mt03080@wilp.bits-pilani.ac.in	2024MT0308
ZOPE AMIT YASHWANT	2024mt03005@wilp.bits-pilani.ac.in	2024MT0300
BAFNA NEEL PRADEEP	2024mt03021@wilp.bits-pilani.ac.in	2024MT0302
SHAHAPURKAR PURUSHOTTAM VIJAY	2024mt03121@wilp.bits-pilani.ac.in	2024MT0312
BHOLE MRUNMAI VINOD	2024mt03038@wilp.bits-pilani.ac.in	2024MT0303

Dataset Link: Mobile Dataset **(Click to Download)**

GitHub Repo: IPDP-Assignment-1-Group-20

Table of Contents

1 Problem Statement

Design and implement a distributed data preprocessing system to handle a 50 million record dataset with 50 parameters each, utilizing MPI, Pthreads, OpenMP, and CUDA for parallel processing. Preprocess the dataset in chunks of 1000 records, identifying and eliminating noisy or incorrectly entered data using statistical methods and data cleaning algorithms. Employ an event-driven architecture to handle preprocessing progress events, triggering noise elimination and reporting modules upon event receipt. Monitor and report preprocessing status in real-time, evaluating performance using metrics such as processing time, GPU utilization.

Additional Requirement:

Determine and justify the optimal distribution of data processing tasks among MPI, Pthreads, OpenMP, and CUDA. Specifically:

- Decide the proportion of data to be processed by each library

- Explain the rationale behind your distribution decision

- Ensure seamless integration and communication among the libraries

2 Considerations

In this report, we address the preprocessing and optimization efforts applied to a large-scale dataset containing mobile device specifications. Generated through a Python script, this dataset comprises 100,000 records, each with 50 detailed parameters describing various features of mobile devices. These attributes cover a comprehensive range of specifications, including display characteristics (e.g., screen size, resolution, refresh rate), camera capabilities, battery capacity, RAM, and more.

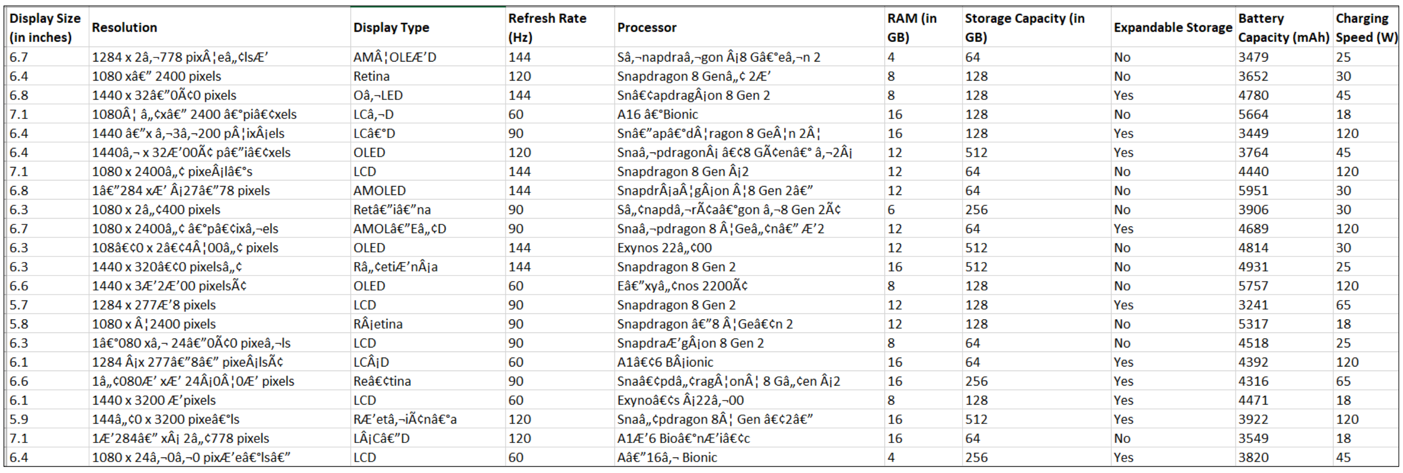
Dataset Overview:

The dataset contains mixed data types—numerical values (e.g., battery capacity, RAM size), categorical variables (e.g., SIM type, operating system), and textual descriptions (e.g., processor details, camera specifications). However, the presence of various Unicode characters, especially within the "Resolution" and "Processor" fields, poses a challenge for effective data processing and analysis. These Unicode characters can lead to inconsistencies and errors in downstream tasks, necessitating a rigorous data-cleaning phase to ensure uniformity and usability.

- Number of Records: 1 Lac
- Number of Parameters: 50

Sample Dataset:

Given the dataset's substantial size, with 100,000 rows and 50 parameters, we provide a sample of the initial few rows and columns for illustration.



Display Size (in inches)	Resolution	Display Type	Refresh Rate (Hz)	Processor	RAM (in GB)	Storage Capacity (in GB)	Expandable Storage	Battery Capacity (mAh)	Charging Speed (W)
6.7	1284 x 2â‚¬778 pixÂ¦eâ„¢lsÆ’	AMÂ¦OLEÆ’D	144	Sâ‚¬napdraâ‚¬gon Â¡8 Gâ€°eâ‚¬n 2	4	64	No	3479	25
6.4	1080 xâ€” 2400 pixels	Retina	120	Snapdragon 8 Genâ„¢ 2Æ’	8	128	No	3652	30
6.8	1440 x 32â€”0Ã¢0 pixels	Oâ‚¬LED	144	Snâ€¢apdragÂ¡on 8 Gen 2	8	128	Yes	4780	45
7.1	1080Â¦ â„¢xâ€” 2400 â€°piâ€¢xels	LCâ‚¬D	60	A16 â€°Bionic	16	128	No	5664	18
6.4	1440 â€”x â‚¬3â‚¬200 pÂ¦ixÂ¡els	LCâ€°D	90	Snâ€”apâ€°dÂ¦ragon 8 GeÂ¦n 2Â¦	16	128	Yes	3449	120
6.4	1440â‚¬ x 32Æ’00Ã¢ pâ€”iâ€¢xels	OLED	120	Snaâ‚¬pdragonÂ¡ â€¢8 GÃ¢enâ€° â‚¬2Â¡	12	512	Yes	3764	45
7.1	1080 x 2400â„¢ pixeÂ¡lâ€°s	LCD	144	Snapdragon 8 Gen Â¡2	12	64	No	4440	120
6.8	1â€”284 xÆ’ Â¡27â€”78 pixels	AMOLED	144	SnapdrÂ¡aÂ¦gÂ¡on Â¦8 Gen 2â€”	12	64	No	5951	30
6.3	1080 x 2â„¢400 pixels	Retâ€”iâ€”na	90	Sâ„¢napdâ‚¬rÃ¢aâ€°gon â‚¬8 Gen 2Ã¢	6	256	No	3906	30
6.7	1080 x 2400â„¢ â€°pâ€¢ixâ‚¬els	AMOLâ€”Eâ„¢D	90	Snaâ‚¬pdragon 8 Â¦Geâ„¢nâ€” Æ’2	12	64	Yes	4689	120
6.3	108â€¢0 x 2â€¢4Â¦00â„¢ pixels	OLED	144	Exynos 22â„¢00	12	512	No	4814	30
6.3	1440 x 320â€¢0 pixelsâ„¢	Râ„¢etiÆ’nÂ¡a	144	Snapdragon 8 Gen 2	16	512	No	4931	25
6.6	1440 x 3Æ’2Æ’00 pixelsÃ¢	OLED	60	Eâ€”xyâ„¢nos 2200Ã¢	8	128	No	5757	120
5.7	1284 x 277Æ’8 pixels	LCD	90	Snapdragon 8 Gen 2	12	128	Yes	3241	65
5.8	1080 x Â¦2400 pixels	RÂ¡etina	90	Snapdragon â€”8 Â¦Geâ€¢n 2	12	128	No	5317	18
6.3	1â€°080 xâ‚¬ 24â€”0Ã¢0 pixeâ‚¬ls	LCD	90	SnapdraÆ’gÂ¡on 8 Gen 2	8	64	No	4518	25
6.1	1284 Â¡x 277â€”8â€” pixeÂ¡lsÃ¢	LCÂ¡D	60	A1â€¢6 BÂ¡ionic	16	64	Yes	4392	120
6.6	1â„¢080Æ’ xÆ’ 24Â¡0Â¦0Æ’ pixels	Reâ€¢tina	90	Snaâ€¢pdâ„¢ragÂ¦onÂ¦ 8 Gâ„¢en Â¡2	16	256	Yes	4316	65
6.1	1440 x 3200 Æ’pixels	LCD	60	Exynoâ€¢s Â¡22â‚¬00	8	128	Yes	4471	18
5.9	144â„¢0 x 3200 pixeâ€°ls	RÆ’etâ‚¬iÃ¢nâ€°a	120	Snaâ„¢pdragon 8Â¦ Gen â€¢2â€”	16	512	Yes	3922	120
7.1	1Æ’284â€” xÂ¡ 2â„¢778 pixels	LÂ¡Câ€”D	120	A1Æ’6 Bioâ€°nÆ’iâ€¢c	16	64	No	3549	18
6.4	1080 x 24â‚¬0â‚¬0 pixÆ’eâ€°lsâ€”	LCD	60	Aâ€”16â‚¬ Bionic	4	256	Yes	3820	45

System Architecture for Preprocessing with OpenMP, Pthreads, MPI, and CUDA

In a high-performance parallel computing environment, preprocessing often involves preparing data or performing calculations before the main computational tasks. Combining **OpenMP**, **Pthreads**, **MPI**, and **CUDA** for preprocessing requires leveraging different levels of parallelism, from **multi-threading** on individual nodes to **distributed computing** across multiple nodes, as well as **GPU acceleration** for intensive computations.

The architecture described here represents how the various parallel programming models can be used together for preprocessing tasks in a distributed and parallelized environment.

1. Overview of Parallel Programming Models:

Pthreads and OpenMP (in C)

- **OpenMP**: OpenMP was utilized for high-level parallelization. Its compiler directives enable easy parallelization of loops and code sections, making it highly effective for multi-core CPU systems. OpenMP automatically manages workload distribution across threads, streamlining parallelization without requiring manual thread management.

- **Pthreads**: Pthreads allowed for low-level threading, offering detailed control over thread creation, synchronization, and management within shared-memory systems. This library is ideal for tasks needing precise thread control, such as custom synchronization or managing shared resources.

CUDA and MPI (in Python)

- **CUDA**: To leverage GPU-based parallelism, we implemented CUDA in Python using the Numba library. CUDA is highly effective for tasks that can be extensively parallelized, as it enables thousands of threads to run simultaneously on the GPU, significantly reducing processing time.

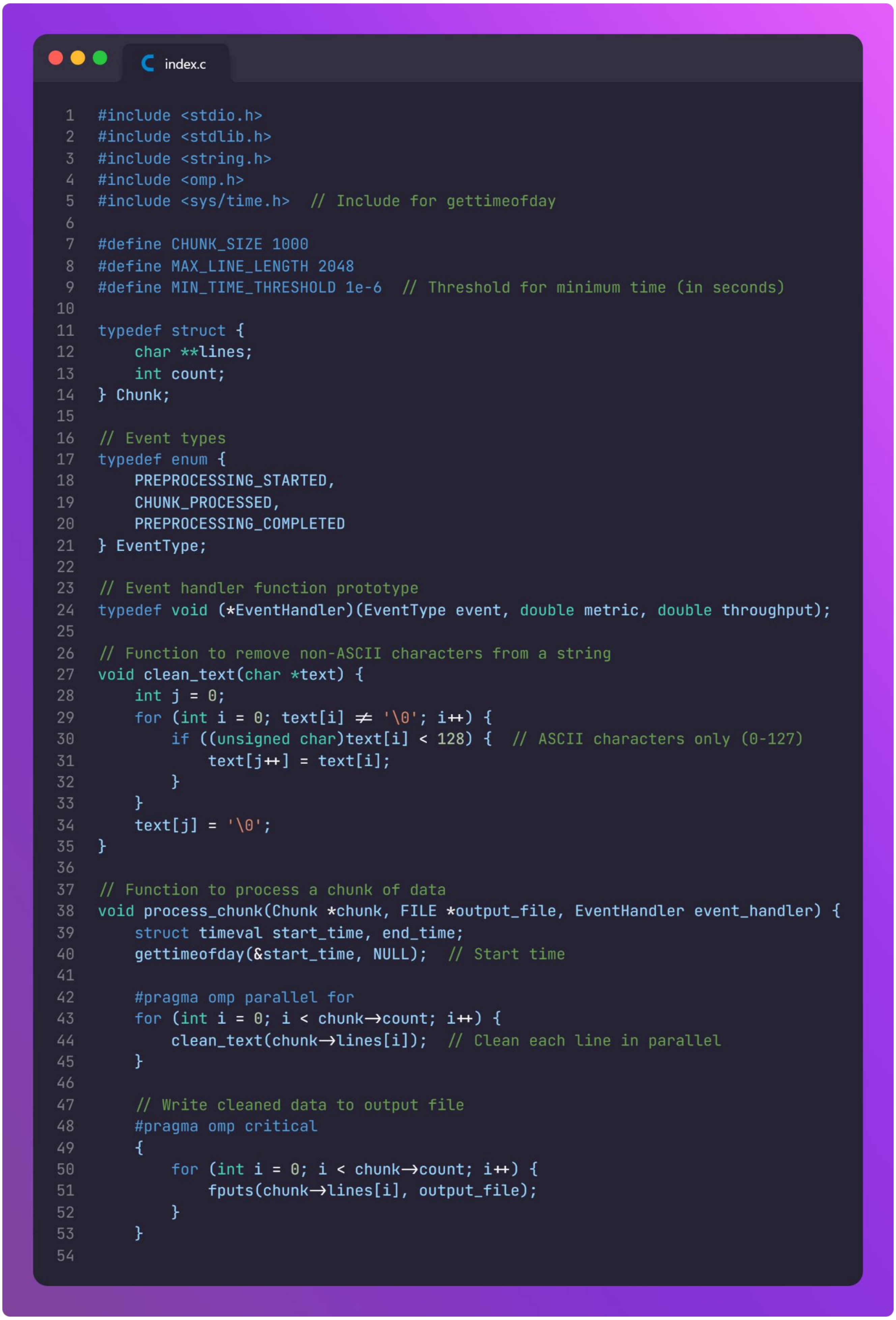
- **MPI**: MPI facilitated distributed computing, implemented in Python using the mpi4py library. This approach is well-suited for spreading tasks across multiple computing nodes.

3. Code

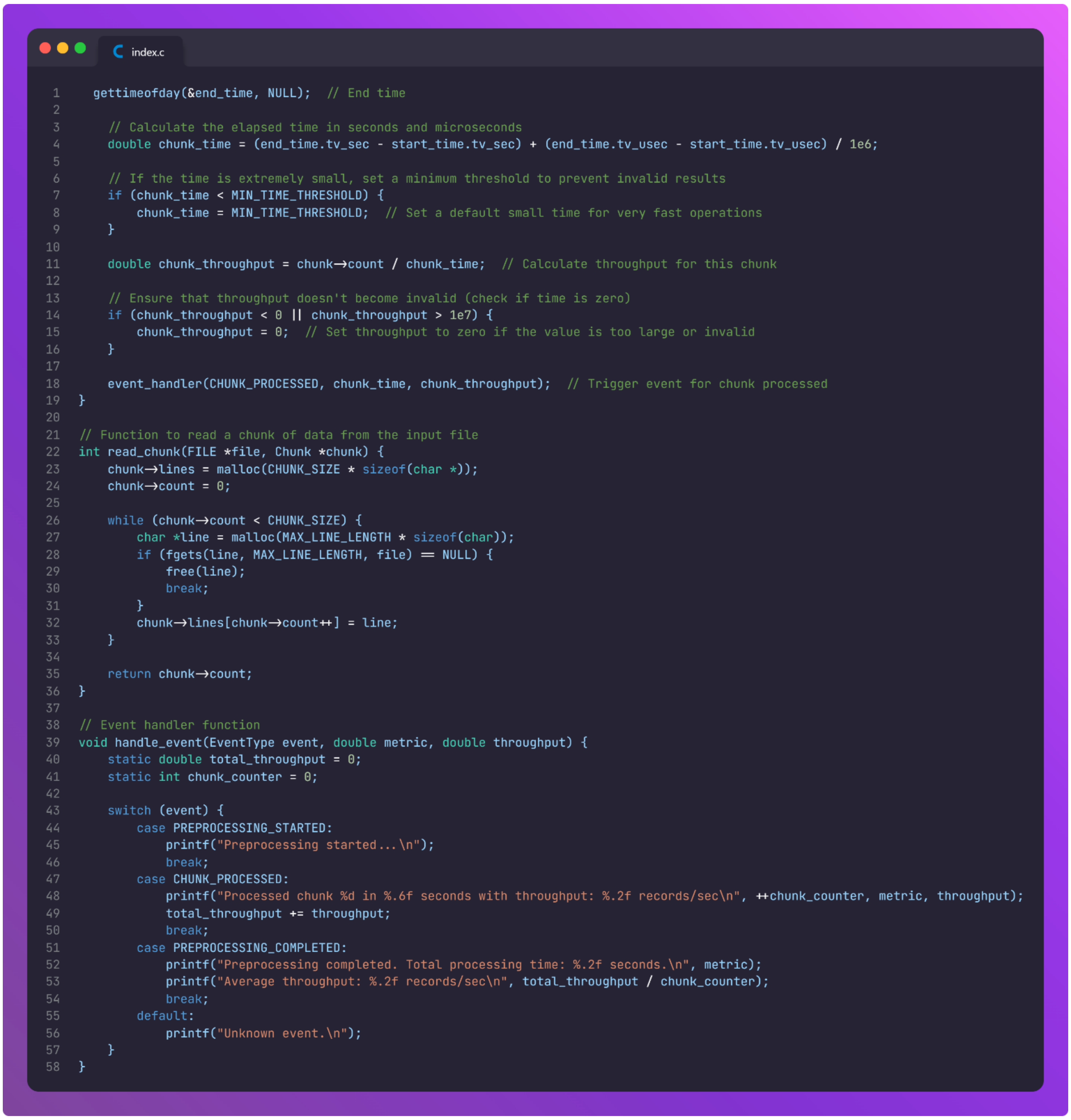
1. OpenMP

This C program is designed to clean a large CSV dataset by removing non-ASCII characters and processing the data efficiently using OpenMP for parallelism.

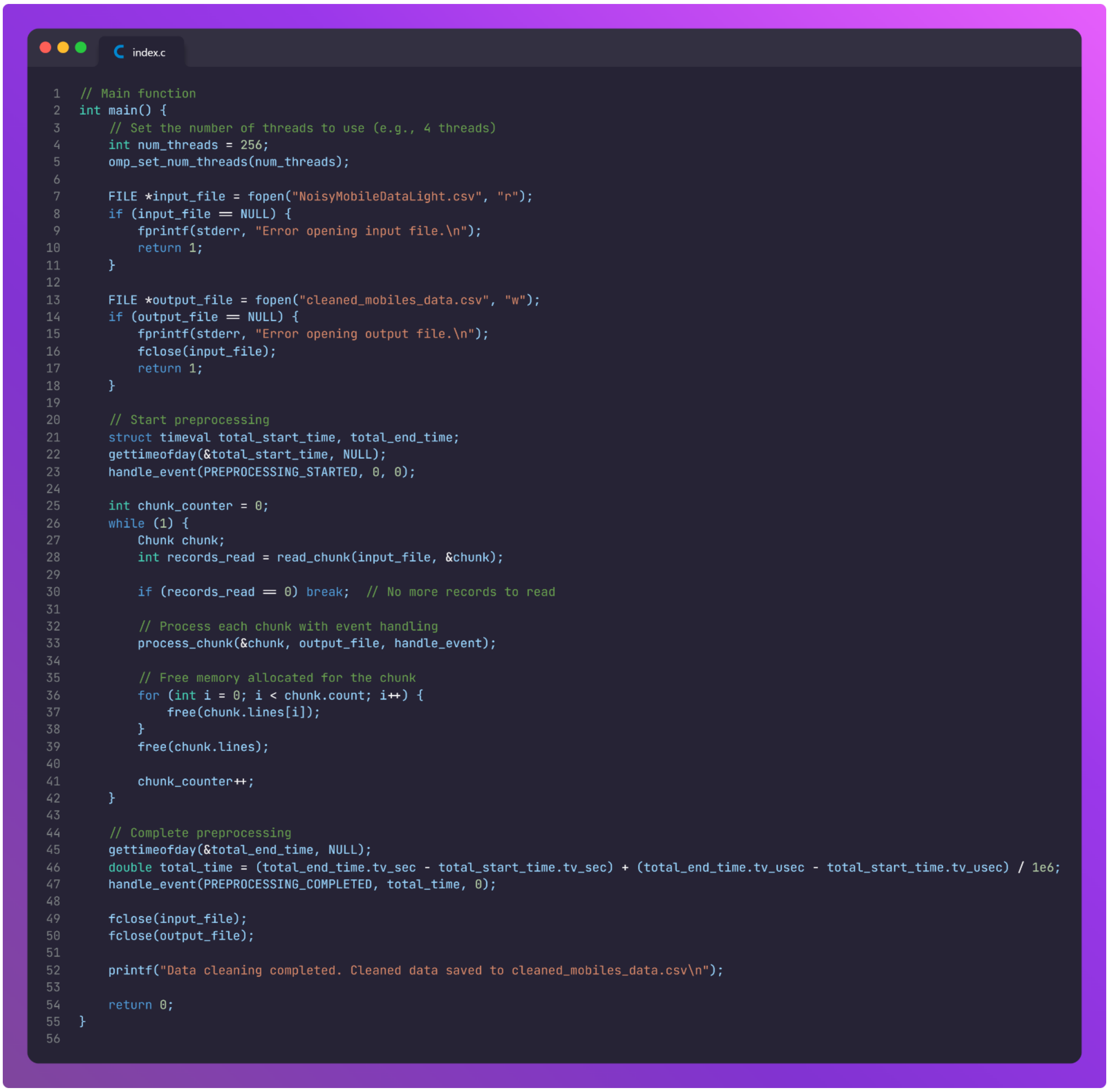
- **Data Chunking**: The input file is read in chunks of 1000 lines, making the program more memory efficient when handling large datasets.
- **Text Cleaning**: The `clean_text()` function iterates over each line in a chunk and removes non-ASCII characters, leaving only characters with ASCII values (0–127).
- **Parallel Processing with OpenMP**: The dataset is processed in parallel using OpenMP directives. Each line within a chunk is cleaned by different threads, speeding up the overall processing of large datasets.
- **Event Handling**: The program logs key events, such as the start and completion of preprocessing, as well as the completion of each chunk. It also calculates throughput (records processed per second) for each chunk processed.
- **File Writing with Critical Section**: The cleaned data is written to the output file (`cleaned_mobiles_data.csv`). To ensure thread safety during file writing, a critical section (`#pragma omp critical`) is used to avoid multiple threads writing simultaneously.
- **Performance Metrics**: The program measures the time taken to process each chunk and calculates throughput (lines processed per second). After processing all chunks, it outputs the total time and average throughput across all chunks.
- **Memory Management**: Memory for each chunk of lines is dynamically allocated and freed after each chunk is processed to handle large datasets without excessive memory usage.
- **Output**: After processing, the cleaned dataset is saved to an output file (`cleaned_mobiles_data.csv`), ready for further analysis.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
#include <sys/time.h>  // Include for gettimeofday

#define CHUNK_SIZE 1000
#define MAX_LINE_LENGTH 2048
#define MIN_TIME_THRESHOLD 1e-6  // Threshold for minimum time (in seconds)

typedef struct {
    char **lines;
    int count;
} Chunk;

// Event types
typedef enum {
    PREPROCESSING_STARTED,
    CHUNK_PROCESSED,
    PREPROCESSING_COMPLETED
} EventType;

// Event handler function prototype
typedef void (*EventHandler)(EventType event, double metric, double throughput);

// Function to remove non-ASCII characters from a string
void clean_text(char *text) {
    int j = 0;
    for (int i = 0; text[i] != '\0'; i++) {
        if ((unsigned char)text[i] < 128) {  // ASCII characters only (0-127)
            text[j++] = text[i];
        }
    }
    text[j] = '\0';
}

// Function to process a chunk of data
void process_chunk(Chunk *chunk, FILE *output_file, EventHandler event_handler) {
    struct timeval start_time, end_time;
    gettimeofday(&start_time, NULL);  // Start time

    #pragma omp parallel for
    for (int i = 0; i < chunk->count; i++) {
        clean_text(chunk->lines[i]);  // Clean each line in parallel
    }

    // Write cleaned data to output file
    #pragma omp critical
    {
        for (int i = 0; i < chunk->count; i++) {
            fputs(chunk->lines[i], output_file);
        }
    }

```

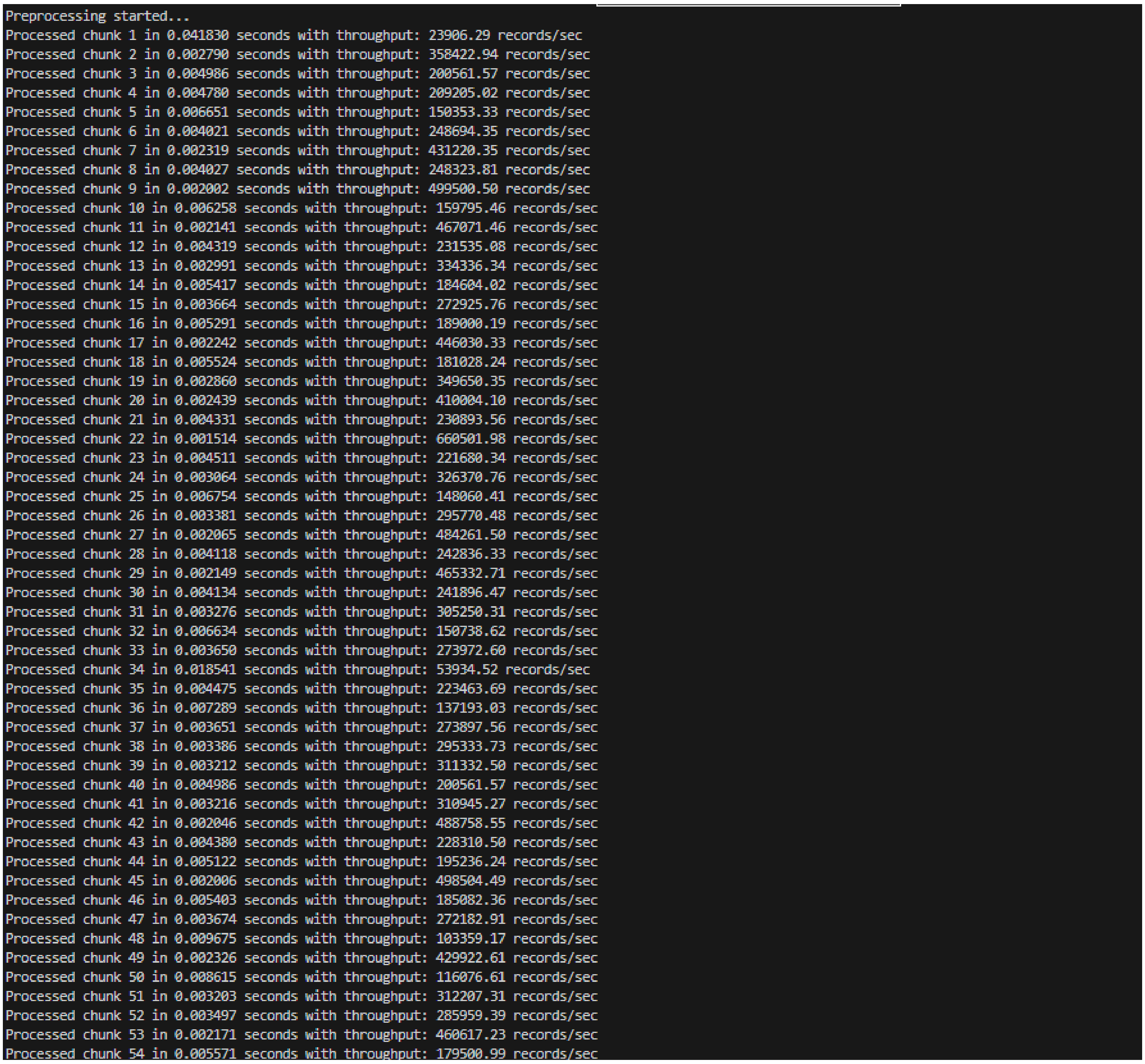


```c
  gettimeofday(&end_time, NULL);  // End time

    // Calculate the elapsed time in seconds and microseconds
    double chunk_time = (end_time.tv_sec - start_time.tv_sec) + (end_time.tv_usec - start_time.tv_usec) / 1e6;

    // If the time is extremely small, set a minimum threshold to prevent invalid results
    if (chunk_time < MIN_TIME_THRESHOLD) {
        chunk_time = MIN_TIME_THRESHOLD;  // Set a default small time for very fast operations
    }

    double chunk_throughput = chunk->count / chunk_time;  // Calculate throughput for this chunk

    // Ensure that throughput doesn't become invalid (check if time is zero)
    if (chunk_throughput < 0 || chunk_throughput > 1e7) {
        chunk_throughput = 0;  // Set throughput to zero if the value is too large or invalid
    }

    event_handler(CHUNK_PROCESSED, chunk_time, chunk_throughput);  // Trigger event for chunk processed
}

// Function to read a chunk of data from the input file
int read_chunk(FILE *file, Chunk *chunk) {
    chunk->lines = malloc(CHUNK_SIZE * sizeof(char *));
    chunk->count = 0;

    while (chunk->count < CHUNK_SIZE) {
        char *line = malloc(MAX_LINE_LENGTH * sizeof(char));
        if (fgets(line, MAX_LINE_LENGTH, file) == NULL) {
            free(line);
            break;
        }
        chunk->lines[chunk->count++] = line;
    }

    return chunk->count;
}

// Event handler function
void handle_event(EventType event, double metric, double throughput) {
    static double total_throughput = 0;
    static int chunk_counter = 0;

    switch (event) {
        case PREPROCESSING_STARTED:
            printf("Preprocessing started...\n");
            break;
        case CHUNK_PROCESSED:
            printf("Processed chunk %d in %.6f seconds with throughput: %.2f records/sec\n", ++chunk_counter, metric, throughput);
            total_throughput += throughput;
            break;
        case PREPROCESSING_COMPLETED:
            printf("Preprocessing completed. Total processing time: %.2f seconds.\n", metric);
            printf("Average throughput: %.2f records/sec\n", total_throughput / chunk_counter);
            break;
        default:
            printf("Unknown event.\n");
    }
}
```

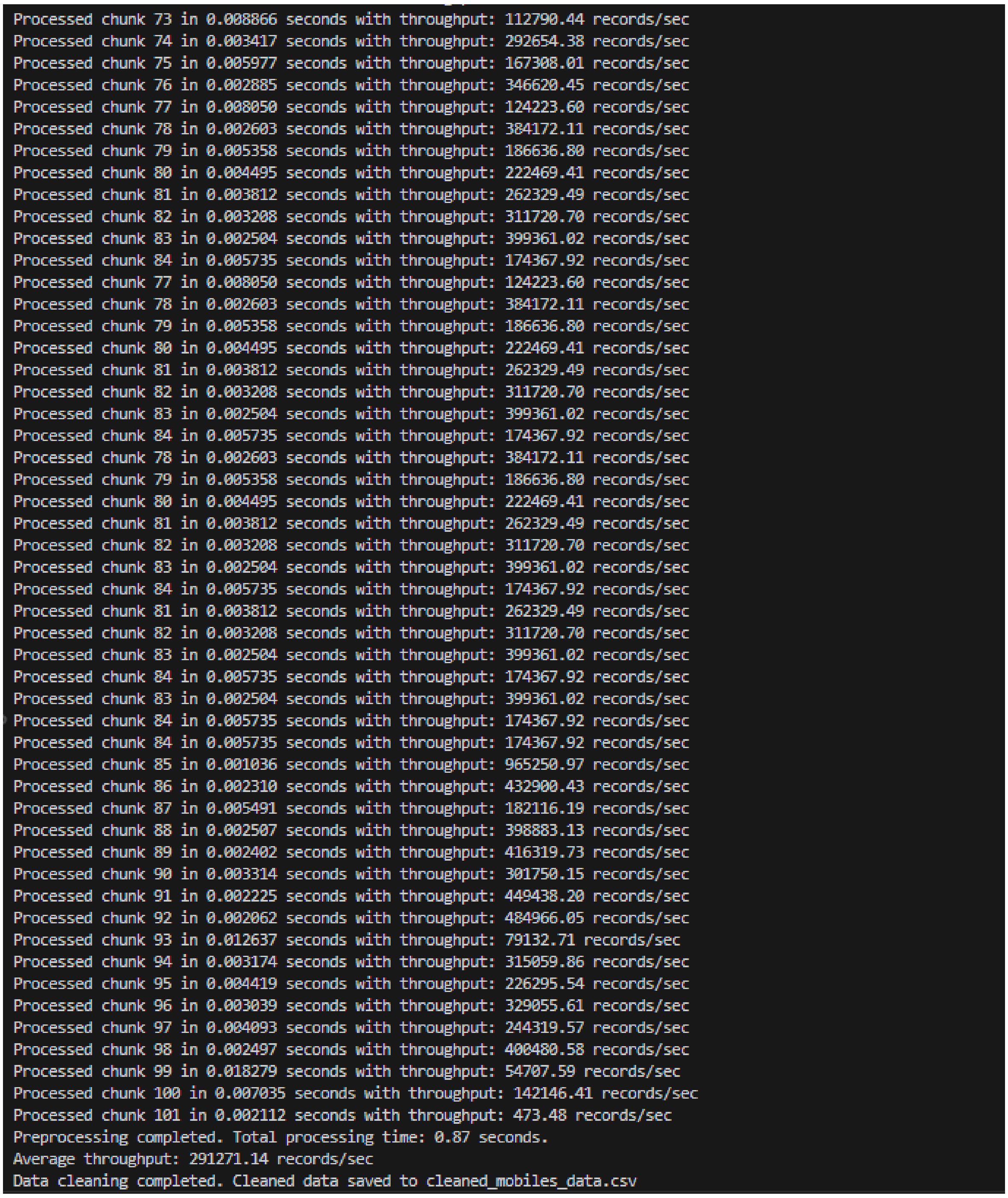

index.c

```c
// Main function
int main() {
    // Set the number of threads to use (e.g., 4 threads)
    int num_threads = 256;
    omp_set_num_threads(num_threads);

    FILE *input_file = fopen("NoisyMobileDataLight.csv", "r");
    if (input_file == NULL) {
        fprintf(stderr, "Error opening input file.\n");
        return 1;
    }

    FILE *output_file = fopen("cleaned_mobiles_data.csv", "w");
    if (output_file == NULL) {
        fprintf(stderr, "Error opening output file.\n");
        fclose(input_file);
        return 1;
    }

    // Start preprocessing
    struct timeval total_start_time, total_end_time;
    gettimeofday(&total_start_time, NULL);
    handle_event(PREPROCESSING_STARTED, 0, 0);

    int chunk_counter = 0;
    while (1) {
        Chunk chunk;
        int records_read = read_chunk(input_file, &chunk);

        if (records_read == 0) break;  // No more records to read

        // Process each chunk with event handling
        process_chunk(&chunk, output_file, handle_event);

        // Free memory allocated for the chunk
        for (int i = 0; i < chunk.count; i++) {
            free(chunk.lines[i]);
        }
        free(chunk.lines);

        chunk_counter++;
    }

    // Complete preprocessing
    gettimeofday(&total_end_time, NULL);
    double total_time = (total_end_time.tv_sec - total_start_time.tv_sec) + (total_end_time.tv_usec - total_start_time.tv_usec) / 1e6;
    handle_event(PREPROCESSING_COMPLETED, total_time, 0);

    fclose(input_file);
    fclose(output_file);

    printf("Data cleaning completed. Cleaned data saved to cleaned_mobiles_data.csv\n");

    return 0;
}
```

Execute the code using: “gcc -fopenmp -o OpenMP_IPDP OpenMP_IPDP.c ”



```
Preprocessing started...
Processed chunk 1 in 0.041830 seconds with throughput: 23906.29 records/sec
Processed chunk 2 in 0.002790 seconds with throughput: 358422.94 records/sec
Processed chunk 3 in 0.004986 seconds with throughput: 200561.57 records/sec
Processed chunk 4 in 0.004780 seconds with throughput: 209205.02 records/sec
Processed chunk 5 in 0.006651 seconds with throughput: 150353.33 records/sec
Processed chunk 6 in 0.004021 seconds with throughput: 248694.35 records/sec
Processed chunk 7 in 0.002319 seconds with throughput: 431220.35 records/sec
Processed chunk 8 in 0.004027 seconds with throughput: 248323.81 records/sec
Processed chunk 9 in 0.002002 seconds with throughput: 499500.50 records/sec
Processed chunk 10 in 0.006258 seconds with throughput: 159795.46 records/sec
Processed chunk 11 in 0.002141 seconds with throughput: 467071.46 records/sec
Processed chunk 12 in 0.004319 seconds with throughput: 231535.08 records/sec
Processed chunk 13 in 0.002991 seconds with throughput: 334336.34 records/sec
Processed chunk 14 in 0.005417 seconds with throughput: 184604.02 records/sec
Processed chunk 15 in 0.003664 seconds with throughput: 272925.76 records/sec
Processed chunk 16 in 0.005291 seconds with throughput: 189000.19 records/sec
Processed chunk 17 in 0.002242 seconds with throughput: 446030.33 records/sec
Processed chunk 18 in 0.005524 seconds with throughput: 181028.24 records/sec
Processed chunk 19 in 0.002860 seconds with throughput: 349650.35 records/sec
Processed chunk 20 in 0.002439 seconds with throughput: 410004.10 records/sec
Processed chunk 21 in 0.004331 seconds with throughput: 230893.56 records/sec
Processed chunk 22 in 0.001514 seconds with throughput: 660501.98 records/sec
Processed chunk 23 in 0.004511 seconds with throughput: 221680.34 records/sec
Processed chunk 24 in 0.003064 seconds with throughput: 326370.76 records/sec
Processed chunk 25 in 0.006754 seconds with throughput: 148060.41 records/sec
Processed chunk 26 in 0.003381 seconds with throughput: 295770.48 records/sec
Processed chunk 27 in 0.002065 seconds with throughput: 484261.50 records/sec
Processed chunk 28 in 0.004118 seconds with throughput: 242836.33 records/sec
Processed chunk 29 in 0.002149 seconds with throughput: 465332.71 records/sec
Processed chunk 30 in 0.004134 seconds with throughput: 241896.47 records/sec
Processed chunk 31 in 0.003276 seconds with throughput: 305250.31 records/sec
Processed chunk 32 in 0.006634 seconds with throughput: 150738.62 records/sec
Processed chunk 33 in 0.003650 seconds with throughput: 273972.60 records/sec
Processed chunk 34 in 0.018541 seconds with throughput: 53934.52 records/sec
Processed chunk 35 in 0.004475 seconds with throughput: 223463.69 records/sec
Processed chunk 36 in 0.007289 seconds with throughput: 137193.03 records/sec
Processed chunk 37 in 0.003651 seconds with throughput: 273897.56 records/sec
Processed chunk 38 in 0.003386 seconds with throughput: 295333.73 records/sec
Processed chunk 39 in 0.003212 seconds with throughput: 311332.50 records/sec
Processed chunk 40 in 0.004986 seconds with throughput: 200561.57 records/sec
Processed chunk 41 in 0.003216 seconds with throughput: 310945.27 records/sec
Processed chunk 42 in 0.002046 seconds with throughput: 488758.55 records/sec
Processed chunk 43 in 0.004380 seconds with throughput: 228310.50 records/sec
Processed chunk 44 in 0.005122 seconds with throughput: 195236.24 records/sec
Processed chunk 45 in 0.002006 seconds with throughput: 498504.49 records/sec
Processed chunk 46 in 0.005403 seconds with throughput: 185082.36 records/sec
Processed chunk 47 in 0.003674 seconds with throughput: 272182.91 records/sec
Processed chunk 48 in 0.009675 seconds with throughput: 103359.17 records/sec
Processed chunk 49 in 0.002326 seconds with throughput: 429922.61 records/sec
Processed chunk 50 in 0.008615 seconds with throughput: 116076.61 records/sec
Processed chunk 51 in 0.003203 seconds with throughput: 312207.31 records/sec
Processed chunk 52 in 0.003497 seconds with throughput: 285959.39 records/sec
Processed chunk 53 in 0.002171 seconds with throughput: 460617.23 records/sec
Processed chunk 54 in 0.005571 seconds with throughput: 179500.99 records/sec
```



```
Processed chunk 73 in 0.008866 seconds with throughput: 112790.44 records/sec
Processed chunk 74 in 0.003417 seconds with throughput: 292654.38 records/sec
Processed chunk 75 in 0.005977 seconds with throughput: 167308.01 records/sec
Processed chunk 76 in 0.002885 seconds with throughput: 346620.45 records/sec
Processed chunk 77 in 0.008050 seconds with throughput: 124223.60 records/sec
Processed chunk 78 in 0.002603 seconds with throughput: 384172.11 records/sec
Processed chunk 79 in 0.005358 seconds with throughput: 186636.80 records/sec
Processed chunk 80 in 0.004495 seconds with throughput: 222469.41 records/sec
Processed chunk 81 in 0.003812 seconds with throughput: 262329.49 records/sec
Processed chunk 82 in 0.003208 seconds with throughput: 311720.70 records/sec
Processed chunk 83 in 0.002504 seconds with throughput: 399361.02 records/sec
Processed chunk 84 in 0.005735 seconds with throughput: 174367.92 records/sec
Processed chunk 77 in 0.008050 seconds with throughput: 124223.60 records/sec
Processed chunk 78 in 0.002603 seconds with throughput: 384172.11 records/sec
Processed chunk 79 in 0.005358 seconds with throughput: 186636.80 records/sec
Processed chunk 80 in 0.004495 seconds with throughput: 222469.41 records/sec
Processed chunk 81 in 0.003812 seconds with throughput: 262329.49 records/sec
Processed chunk 82 in 0.003208 seconds with throughput: 311720.70 records/sec
Processed chunk 83 in 0.002504 seconds with throughput: 399361.02 records/sec
Processed chunk 84 in 0.005735 seconds with throughput: 174367.92 records/sec
Processed chunk 78 in 0.002603 seconds with throughput: 384172.11 records/sec
Processed chunk 79 in 0.005358 seconds with throughput: 186636.80 records/sec
Processed chunk 80 in 0.004495 seconds with throughput: 222469.41 records/sec
Processed chunk 81 in 0.003812 seconds with throughput: 262329.49 records/sec
Processed chunk 82 in 0.003208 seconds with throughput: 311720.70 records/sec
Processed chunk 83 in 0.002504 seconds with throughput: 399361.02 records/sec
Processed chunk 84 in 0.005735 seconds with throughput: 174367.92 records/sec
Processed chunk 81 in 0.003812 seconds with throughput: 262329.49 records/sec
Processed chunk 82 in 0.003208 seconds with throughput: 311720.70 records/sec
Processed chunk 83 in 0.002504 seconds with throughput: 399361.02 records/sec
Processed chunk 84 in 0.005735 seconds with throughput: 174367.92 records/sec
Processed chunk 83 in 0.002504 seconds with throughput: 399361.02 records/sec
Processed chunk 84 in 0.005735 seconds with throughput: 174367.92 records/sec
Processed chunk 84 in 0.005735 seconds with throughput: 174367.92 records/sec
Processed chunk 85 in 0.001036 seconds with throughput: 965250.97 records/sec
Processed chunk 86 in 0.002310 seconds with throughput: 432900.43 records/sec
Processed chunk 87 in 0.005491 seconds with throughput: 182116.19 records/sec
Processed chunk 88 in 0.002507 seconds with throughput: 398883.13 records/sec
Processed chunk 89 in 0.002402 seconds with throughput: 416319.73 records/sec
Processed chunk 90 in 0.003314 seconds with throughput: 301750.15 records/sec
Processed chunk 91 in 0.002225 seconds with throughput: 449438.20 records/sec
Processed chunk 92 in 0.002062 seconds with throughput: 484966.05 records/sec
Processed chunk 93 in 0.012637 seconds with throughput: 79132.71 records/sec
Processed chunk 94 in 0.003174 seconds with throughput: 315059.86 records/sec
Processed chunk 95 in 0.004419 seconds with throughput: 226295.54 records/sec
Processed chunk 96 in 0.003039 seconds with throughput: 329055.61 records/sec
Processed chunk 97 in 0.004093 seconds with throughput: 244319.57 records/sec
Processed chunk 98 in 0.002497 seconds with throughput: 400480.58 records/sec
Processed chunk 99 in 0.018279 seconds with throughput: 54707.59 records/sec
Processed chunk 100 in 0.007035 seconds with throughput: 142146.41 records/sec
Processed chunk 101 in 0.002112 seconds with throughput: 473.48 records/sec
Preprocessing completed. Total processing time: 0.87 seconds.
Average throughput: 291271.14 records/sec
Data cleaning completed. Cleaned data saved to cleaned_mobiles_data.csv
```

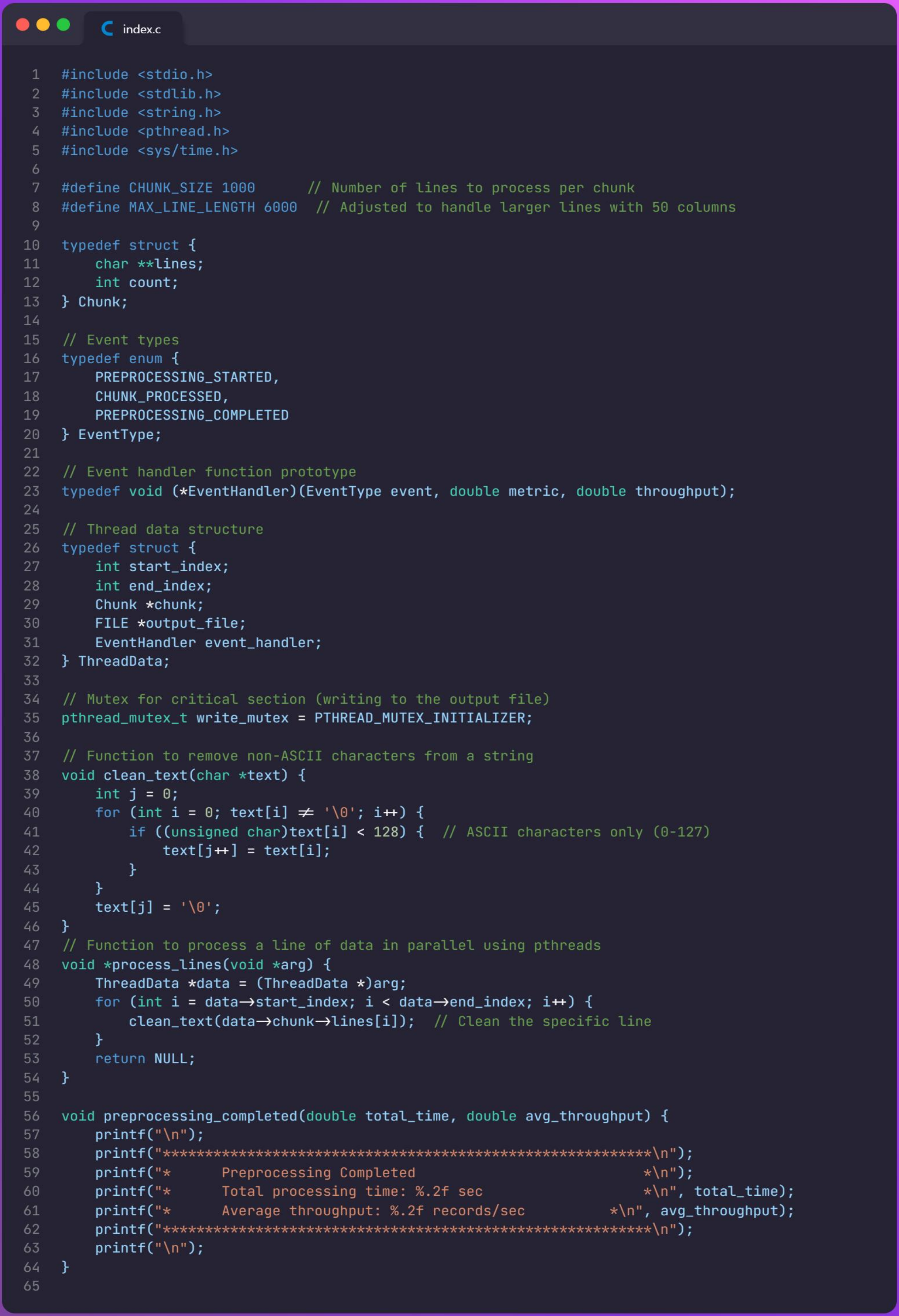
Execution Time for OpenMP – 0.87 seconds

2. Pthreads

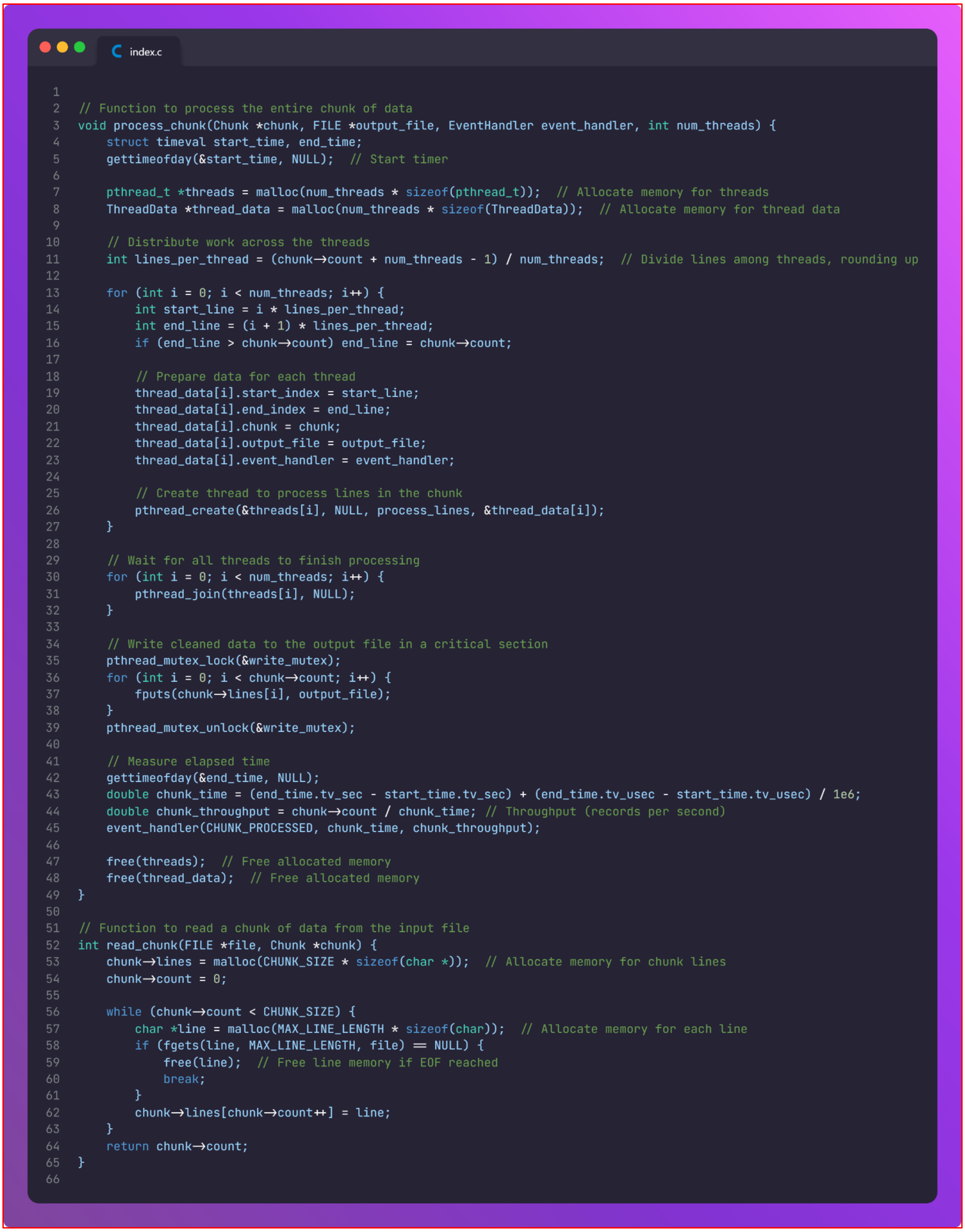
Parallel Data Cleaning Program Overview (C)

This C program is designed to clean a large dataset (in CSV format) by removing non-ASCII characters and processing the data in parallel using Pthreads.

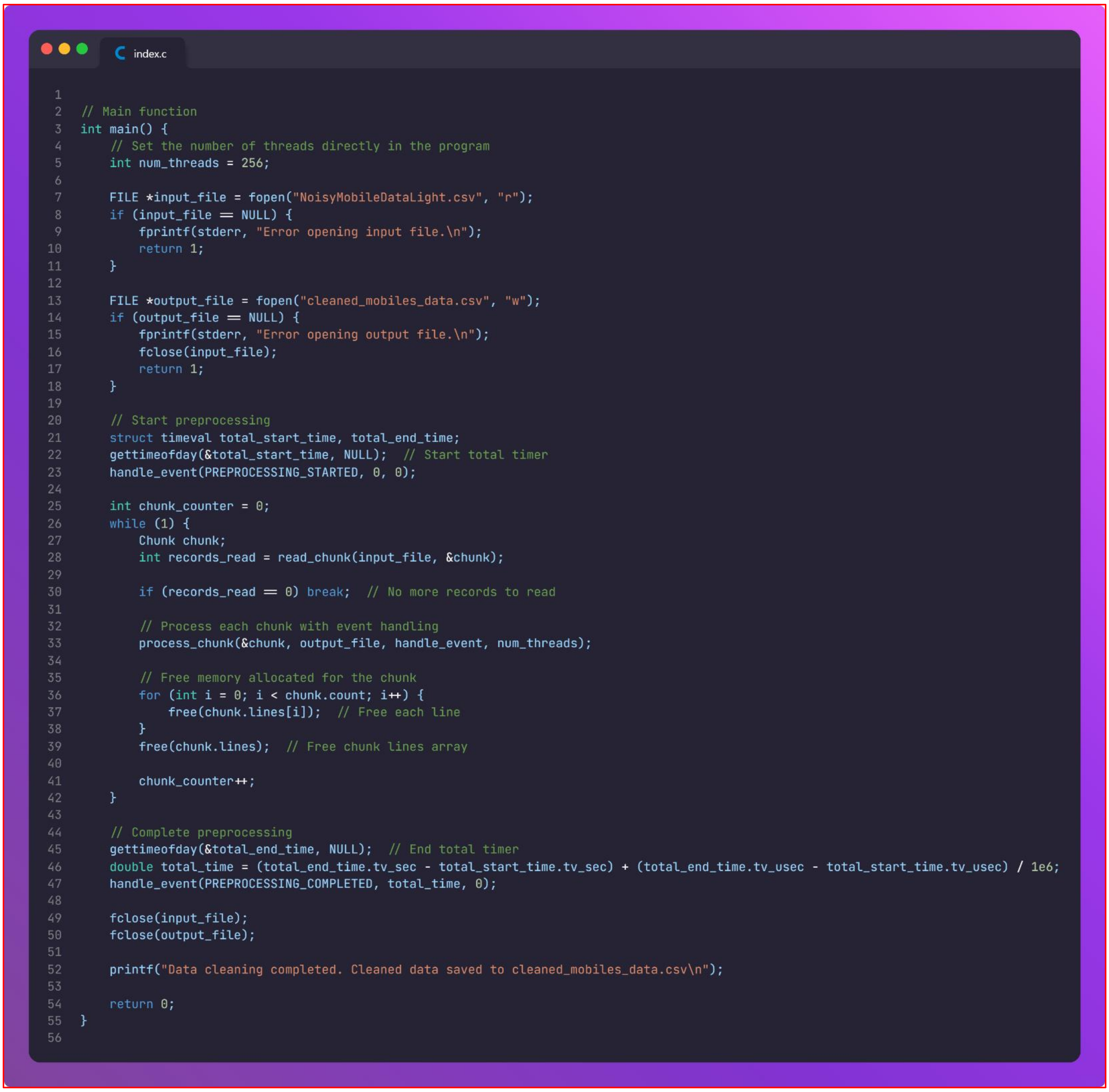
- **Data Chunking**: The input file is read in manageable chunks of 1000 lines, making it more efficient for processing large datasets.
- **Text Cleaning**: A `clean_text()` function iterates over each line, removing non-ASCII characters by retaining only ASCII values (0–127).
- **Multithreading with Pthreads**: The dataset is processed across multiple threads, each cleaning a subset of lines within the current chunk. The number of threads can be configured based on system resources. Threads are created with `pthread_create()` and execute concurrently.
- **Event Handling**: The program logs key processing events, such as the start, completion of each chunk, and overall completion, while calculating throughput (records processed per second) for each chunk.
- **Mutex for Thread Safety**: A mutex (`write_mutex`) ensures that only one thread writes to the output file at a time, maintaining thread safety during file writes.
- **Performance Metrics**: The program monitors and reports the time taken to process each chunk and calculates throughput (records processed per second). After all chunks are processed, the total time and average throughput are displayed.
- **Output**: Cleaned data is written to an output file (`cleaned_mobiles_data.csv`) for further analysis.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sys/time.h>

#define CHUNK_SIZE 1000       // Number of lines to process per chunk
#define MAX_LINE_LENGTH 6000   // Adjusted to handle larger lines with 50 columns

typedef struct {
    char **lines;
    int count;
} Chunk;

// Event types
typedef enum {
    PREPROCESSING_STARTED,
    CHUNK_PROCESSED,
    PREPROCESSING_COMPLETED
} EventType;

// Event handler function prototype
typedef void (*EventHandler)(EventType event, double metric, double throughput);

// Thread data structure
typedef struct {
    int start_index;
    int end_index;
    Chunk *chunk;
    FILE *output_file;
    EventHandler event_handler;
} ThreadData;

// Mutex for critical section (writing to the output file)
pthread_mutex_t write_mutex = PTHREAD_MUTEX_INITIALIZER;

// Function to remove non-ASCII characters from a string
void clean_text(char *text) {
    int j = 0;
    for (int i = 0; text[i] != '\0'; i++) {
        if ((unsigned char)text[i] < 128) {  // ASCII characters only (0-127)
            text[j++] = text[i];
        }
    }
    text[j] = '\0';
}
// Function to process a line of data in parallel using pthreads
void *process_lines(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    for (int i = data->start_index; i < data->end_index; i++) {
        clean_text(data->chunk->lines[i]);  // Clean the specific line
    }
    return NULL;
}

void preprocessing_completed(double total_time, double avg_throughput) {
    printf("\n");
    printf("**********************************************************\n");
    printf("*      Preprocessing Completed                           *\n");
    printf("*      Total processing time: %.2f sec                   *\n", total_time);
    printf("*      Average throughput: %.2f records/sec          *\n", avg_throughput);
    printf("**********************************************************\n");
    printf("\n");
}

```


index.c

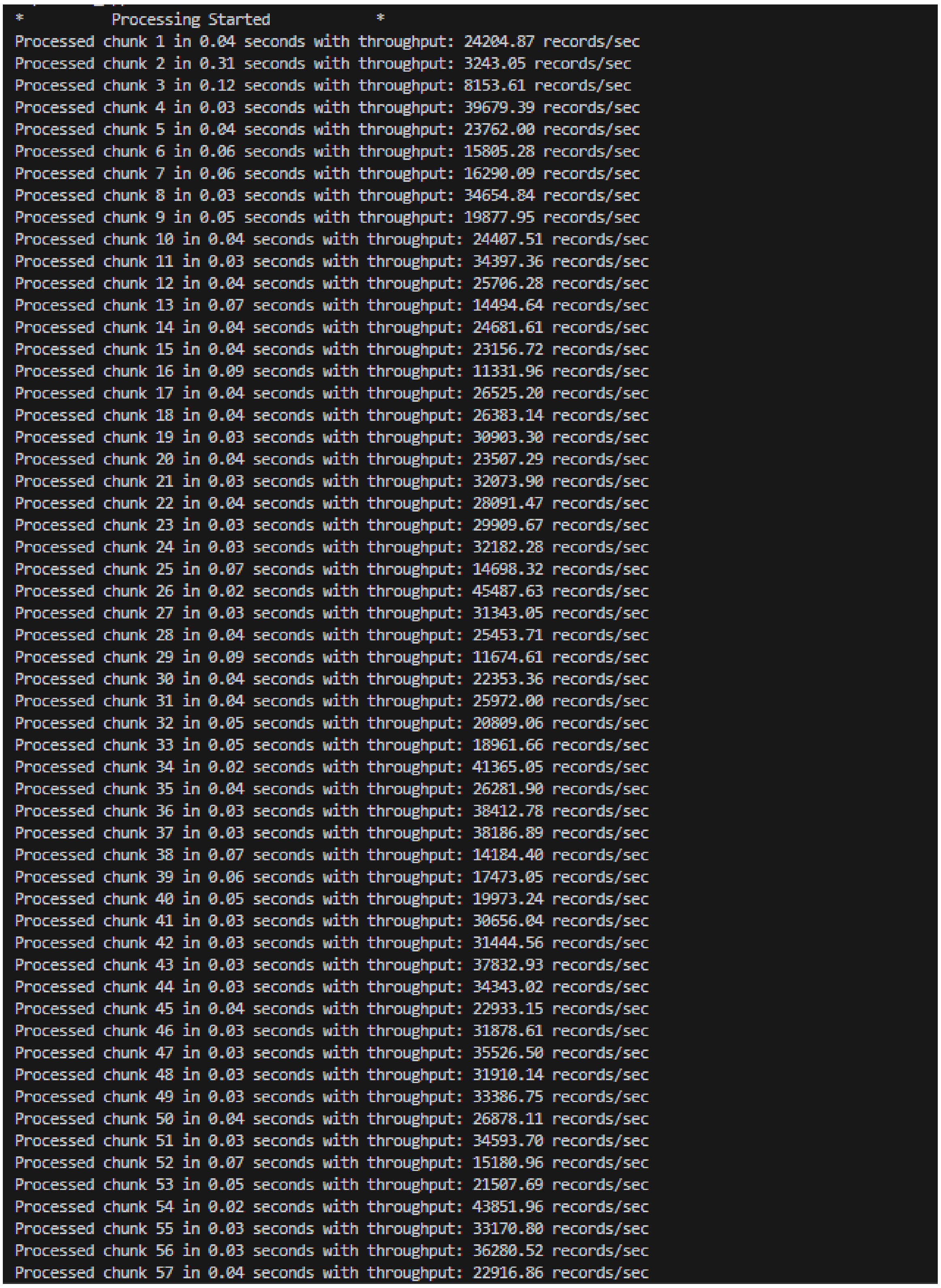
```c

// Function to process the entire chunk of data
void process_chunk(Chunk *chunk, FILE *output_file, EventHandler event_handler, int num_threads) {
    struct timeval start_time, end_time;
    gettimeofday(&start_time, NULL);  // Start timer

    pthread_t *threads = malloc(num_threads * sizeof(pthread_t));  // Allocate memory for threads
    ThreadData *thread_data = malloc(num_threads * sizeof(ThreadData));  // Allocate memory for thread data

    // Distribute work across the threads
    int lines_per_thread = (chunk->count + num_threads - 1) / num_threads;  // Divide lines among threads, rounding up

    for (int i = 0; i < num_threads; i++) {
        int start_line = i * lines_per_thread;
        int end_line = (i + 1) * lines_per_thread;
        if (end_line > chunk->count) end_line = chunk->count;

        // Prepare data for each thread
        thread_data[i].start_index = start_line;
        thread_data[i].end_index = end_line;
        thread_data[i].chunk = chunk;
        thread_data[i].output_file = output_file;
        thread_data[i].event_handler = event_handler;

        // Create thread to process lines in the chunk
        pthread_create(&threads[i], NULL, process_lines, &thread_data[i]);
    }

    // Wait for all threads to finish processing
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    // Write cleaned data to the output file in a critical section
    pthread_mutex_lock(&write_mutex);
    for (int i = 0; i < chunk->count; i++) {
        fputs(chunk->lines[i], output_file);
    }
    pthread_mutex_unlock(&write_mutex);

    // Measure elapsed time
    gettimeofday(&end_time, NULL);
    double chunk_time = (end_time.tv_sec - start_time.tv_sec) + (end_time.tv_usec - start_time.tv_usec) / 1e6;
    double chunk_throughput = chunk->count / chunk_time; // Throughput (records per second)
    event_handler(CHUNK_PROCESSED, chunk_time, chunk_throughput);

    free(threads);  // Free allocated memory
    free(thread_data);  // Free allocated memory
}

// Function to read a chunk of data from the input file
int read_chunk(FILE *file, Chunk *chunk) {
    chunk->lines = malloc(CHUNK_SIZE * sizeof(char *));  // Allocate memory for chunk lines
    chunk->count = 0;

    while (chunk->count < CHUNK_SIZE) {
        char *line = malloc(MAX_LINE_LENGTH * sizeof(char));  // Allocate memory for each line
        if (fgets(line, MAX_LINE_LENGTH, file) == NULL) {
            free(line);  // Free line memory if EOF reached
            break;
        }
        chunk->lines[chunk->count++] = line;
    }
    return chunk->count;
}

```



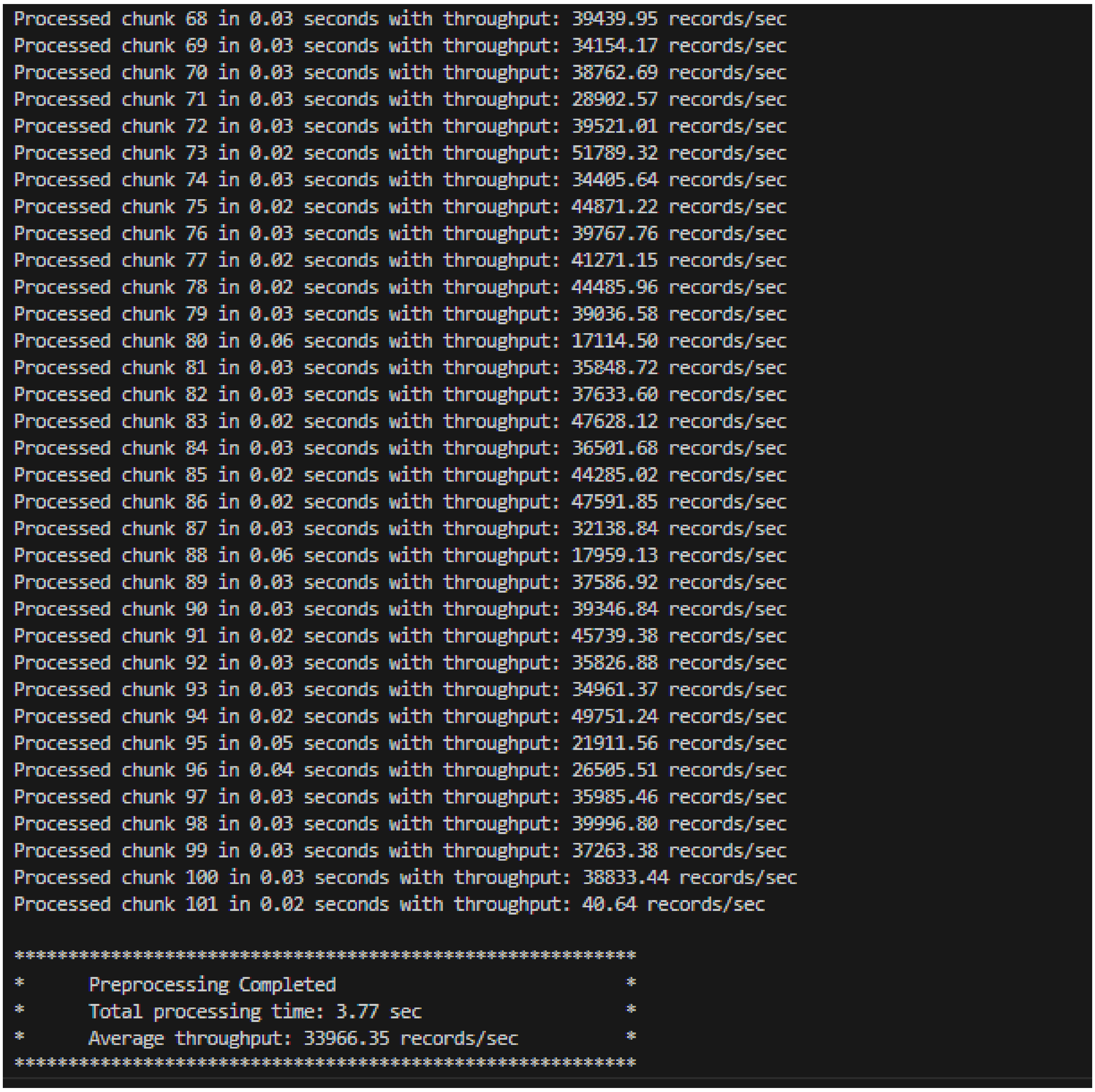
```c

// Main function
int main() {
    // Set the number of threads directly in the program
    int num_threads = 256;

    FILE *input_file = fopen("NoisyMobileDataLight.csv", "r");
    if (input_file == NULL) {
        fprintf(stderr, "Error opening input file.\n");
        return 1;
    }

    FILE *output_file = fopen("cleaned_mobiles_data.csv", "w");
    if (output_file == NULL) {
        fprintf(stderr, "Error opening output file.\n");
        fclose(input_file);
        return 1;
    }

    // Start preprocessing
    struct timeval total_start_time, total_end_time;
    gettimeofday(&total_start_time, NULL);  // Start total timer
    handle_event(PREPROCESSING_STARTED, 0, 0);

    int chunk_counter = 0;
    while (1) {
        Chunk chunk;
        int records_read = read_chunk(input_file, &chunk);

        if (records_read == 0) break;  // No more records to read

        // Process each chunk with event handling
        process_chunk(&chunk, output_file, handle_event, num_threads);

        // Free memory allocated for the chunk
        for (int i = 0; i < chunk.count; i++) {
            free(chunk.lines[i]);  // Free each line
        }
        free(chunk.lines);  // Free chunk lines array

        chunk_counter++;
    }

    // Complete preprocessing
    gettimeofday(&total_end_time, NULL);  // End total timer
    double total_time = (total_end_time.tv_sec - total_start_time.tv_sec) + (total_end_time.tv_usec - total_start_time.tv_usec) / 1e6;
    handle_event(PREPROCESSING_COMPLETED, total_time, 0);

    fclose(input_file);
    fclose(output_file);

    printf("Data cleaning completed. Cleaned data saved to cleaned_mobiles_data.csv\n");

    return 0;
}

```

Output:

Execute Code using: “gcc pthread_unicode.c -o pthread_unicode -lpthread”



```
*          Processing Started           *
Processed chunk 1 in 0.04 seconds with throughput: 24204.87 records/sec
Processed chunk 2 in 0.31 seconds with throughput: 3243.05 records/sec
Processed chunk 3 in 0.12 seconds with throughput: 8153.61 records/sec
Processed chunk 4 in 0.03 seconds with throughput: 39679.39 records/sec
Processed chunk 5 in 0.04 seconds with throughput: 23762.00 records/sec
Processed chunk 6 in 0.06 seconds with throughput: 15805.28 records/sec
Processed chunk 7 in 0.06 seconds with throughput: 16290.09 records/sec
Processed chunk 8 in 0.03 seconds with throughput: 34654.84 records/sec
Processed chunk 9 in 0.05 seconds with throughput: 19877.95 records/sec
Processed chunk 10 in 0.04 seconds with throughput: 24407.51 records/sec
Processed chunk 11 in 0.03 seconds with throughput: 34397.36 records/sec
Processed chunk 12 in 0.04 seconds with throughput: 25706.28 records/sec
Processed chunk 13 in 0.07 seconds with throughput: 14494.64 records/sec
Processed chunk 14 in 0.04 seconds with throughput: 24681.61 records/sec
Processed chunk 15 in 0.04 seconds with throughput: 23156.72 records/sec
Processed chunk 16 in 0.09 seconds with throughput: 11331.96 records/sec
Processed chunk 17 in 0.04 seconds with throughput: 26525.20 records/sec
Processed chunk 18 in 0.04 seconds with throughput: 26383.14 records/sec
Processed chunk 19 in 0.03 seconds with throughput: 30903.30 records/sec
Processed chunk 20 in 0.04 seconds with throughput: 23507.29 records/sec
Processed chunk 21 in 0.03 seconds with throughput: 32073.90 records/sec
Processed chunk 22 in 0.04 seconds with throughput: 28091.47 records/sec
Processed chunk 23 in 0.03 seconds with throughput: 29909.67 records/sec
Processed chunk 24 in 0.03 seconds with throughput: 32182.28 records/sec
Processed chunk 25 in 0.07 seconds with throughput: 14698.32 records/sec
Processed chunk 26 in 0.02 seconds with throughput: 45487.63 records/sec
Processed chunk 27 in 0.03 seconds with throughput: 31343.05 records/sec
Processed chunk 28 in 0.04 seconds with throughput: 25453.71 records/sec
Processed chunk 29 in 0.09 seconds with throughput: 11674.61 records/sec
Processed chunk 30 in 0.04 seconds with throughput: 22353.36 records/sec
Processed chunk 31 in 0.04 seconds with throughput: 25972.00 records/sec
Processed chunk 32 in 0.05 seconds with throughput: 20809.06 records/sec
Processed chunk 33 in 0.05 seconds with throughput: 18961.66 records/sec
Processed chunk 34 in 0.02 seconds with throughput: 41365.05 records/sec
Processed chunk 35 in 0.04 seconds with throughput: 26281.90 records/sec
Processed chunk 36 in 0.03 seconds with throughput: 38412.78 records/sec
Processed chunk 37 in 0.03 seconds with throughput: 38186.89 records/sec
Processed chunk 38 in 0.07 seconds with throughput: 14184.40 records/sec
Processed chunk 39 in 0.06 seconds with throughput: 17473.05 records/sec
Processed chunk 40 in 0.05 seconds with throughput: 19973.24 records/sec
Processed chunk 41 in 0.03 seconds with throughput: 30656.04 records/sec
Processed chunk 42 in 0.03 seconds with throughput: 31444.56 records/sec
Processed chunk 43 in 0.03 seconds with throughput: 37832.93 records/sec
Processed chunk 44 in 0.03 seconds with throughput: 34343.02 records/sec
Processed chunk 45 in 0.04 seconds with throughput: 22933.15 records/sec
Processed chunk 46 in 0.03 seconds with throughput: 31878.61 records/sec
Processed chunk 47 in 0.03 seconds with throughput: 35526.50 records/sec
Processed chunk 48 in 0.03 seconds with throughput: 31910.14 records/sec
Processed chunk 49 in 0.03 seconds with throughput: 33386.75 records/sec
Processed chunk 50 in 0.04 seconds with throughput: 26878.11 records/sec
Processed chunk 51 in 0.03 seconds with throughput: 34593.70 records/sec
Processed chunk 52 in 0.07 seconds with throughput: 15180.96 records/sec
Processed chunk 53 in 0.05 seconds with throughput: 21507.69 records/sec
Processed chunk 54 in 0.02 seconds with throughput: 43851.96 records/sec
Processed chunk 55 in 0.03 seconds with throughput: 33170.80 records/sec
Processed chunk 56 in 0.03 seconds with throughput: 36280.52 records/sec
Processed chunk 57 in 0.04 seconds with throughput: 22916.86 records/sec
```


```
Processed chunk 68 in 0.03 seconds with throughput: 39439.95 records/sec
Processed chunk 69 in 0.03 seconds with throughput: 34154.17 records/sec
Processed chunk 70 in 0.03 seconds with throughput: 38762.69 records/sec
Processed chunk 71 in 0.03 seconds with throughput: 28902.57 records/sec
Processed chunk 72 in 0.03 seconds with throughput: 39521.01 records/sec
Processed chunk 73 in 0.02 seconds with throughput: 51789.32 records/sec
Processed chunk 74 in 0.03 seconds with throughput: 34405.64 records/sec
Processed chunk 75 in 0.02 seconds with throughput: 44871.22 records/sec
Processed chunk 76 in 0.03 seconds with throughput: 39767.76 records/sec
Processed chunk 77 in 0.02 seconds with throughput: 41271.15 records/sec
Processed chunk 78 in 0.02 seconds with throughput: 44485.96 records/sec
Processed chunk 79 in 0.03 seconds with throughput: 39036.58 records/sec
Processed chunk 80 in 0.06 seconds with throughput: 17114.50 records/sec
Processed chunk 81 in 0.03 seconds with throughput: 35848.72 records/sec
Processed chunk 82 in 0.03 seconds with throughput: 37633.60 records/sec
Processed chunk 83 in 0.02 seconds with throughput: 47628.12 records/sec
Processed chunk 84 in 0.03 seconds with throughput: 36501.68 records/sec
Processed chunk 85 in 0.02 seconds with throughput: 44285.02 records/sec
Processed chunk 86 in 0.02 seconds with throughput: 47591.85 records/sec
Processed chunk 87 in 0.03 seconds with throughput: 32138.84 records/sec
Processed chunk 88 in 0.06 seconds with throughput: 17959.13 records/sec
Processed chunk 89 in 0.03 seconds with throughput: 37586.92 records/sec
Processed chunk 90 in 0.03 seconds with throughput: 39346.84 records/sec
Processed chunk 91 in 0.02 seconds with throughput: 45739.38 records/sec
Processed chunk 92 in 0.03 seconds with throughput: 35826.88 records/sec
Processed chunk 93 in 0.03 seconds with throughput: 34961.37 records/sec
Processed chunk 94 in 0.02 seconds with throughput: 49751.24 records/sec
Processed chunk 95 in 0.05 seconds with throughput: 21911.56 records/sec
Processed chunk 96 in 0.04 seconds with throughput: 26505.51 records/sec
Processed chunk 97 in 0.03 seconds with throughput: 35985.46 records/sec
Processed chunk 98 in 0.03 seconds with throughput: 39996.80 records/sec
Processed chunk 99 in 0.03 seconds with throughput: 37263.38 records/sec
Processed chunk 100 in 0.03 seconds with throughput: 38833.44 records/sec
Processed chunk 101 in 0.02 seconds with throughput: 40.64 records/sec

**********************************************************
*      Preprocessing Completed                           *
*      Total processing time: 3.77 sec                   *
*      Average throughput: 33966.35 records/sec          *
**********************************************************
```

Execution Time for Pthreads – 3.77 seconds

3. CUDA

CUDA Data-Cleaning Program Summary

Purpose:
This Python program leverages CUDA parallel processing to clean large datasets (100,000 records) by removing non-ASCII characters from specific columns in a CSV file. It uses GPU processing for faster data handling than traditional CPU-based methods.

Process Overview:

1. **Data Loading & Preprocessing:**
   a. The program loads the CSV dataset (`NoisyMobileDataLight.csv`) into a Pandas DataFrame.
   b. A specified list of columns (`columns_to_clean`) is identified for ASCII cleaning.
   c. Non-empty values are converted to strings and missing values are filled with an empty string.

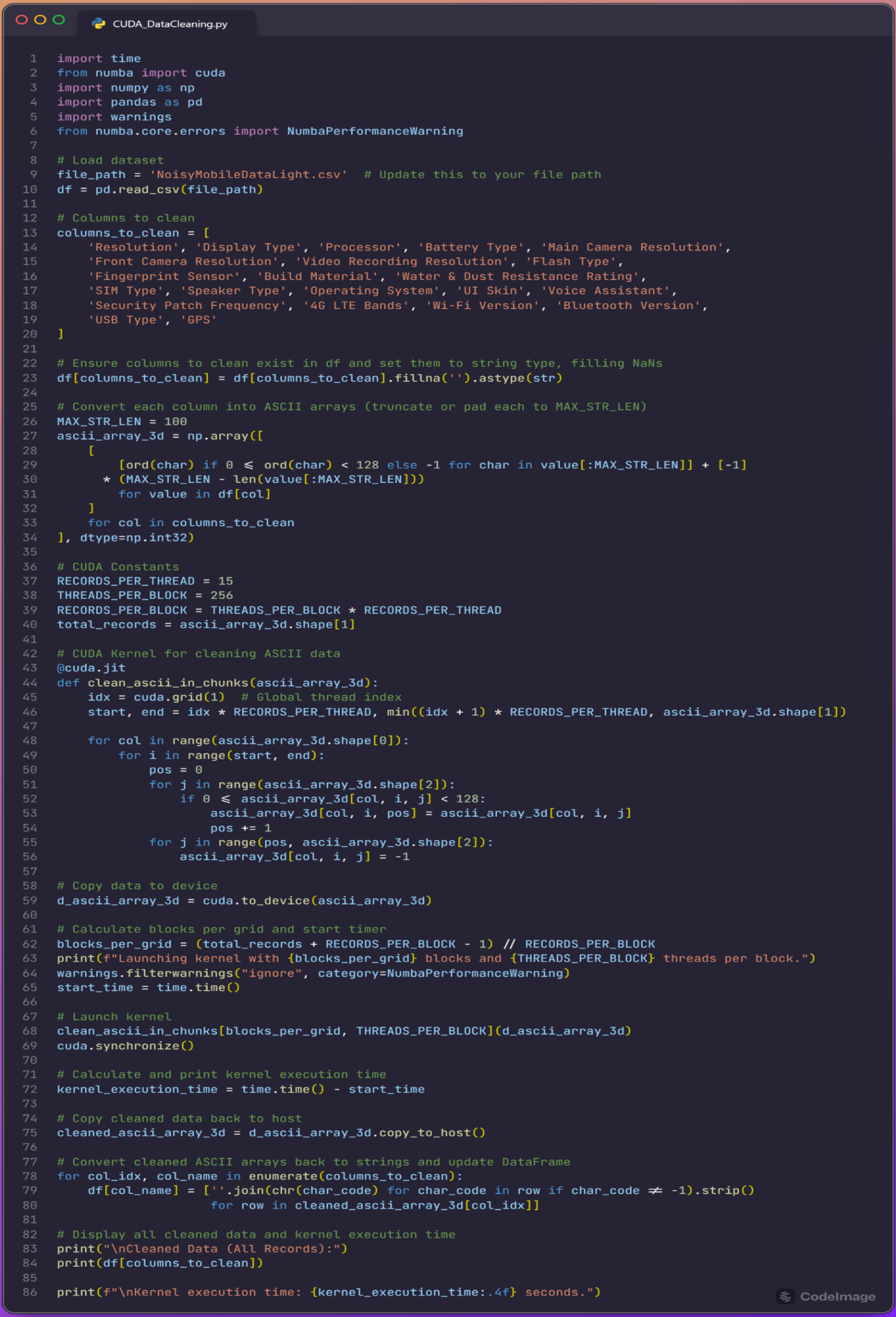
2. **Data Conversion to ASCII Arrays:**
   a. For each value in the target columns, characters are converted into ASCII codes (values between 0 and 127).
   b. The resulting ASCII codes are stored in a 3D NumPy array (`ascii_array_3d`), with each value padded to a fixed length (`MAX_STR_LEN`) for uniformity.

3. **CUDA Kernel Setup:**
   a. **Kernel Function (`@cuda.jit`):** A CUDA function `clean_ascii_in_chunks` is defined to filter non-ASCII characters, with each thread handling multiple records.
   b. **Global Thread Index (`cuda.grid(1)`):** Determines the unique index for each thread, allowing for distributed processing of records.
   c. **Device Data Transfer (`cuda.to_device`):** Data is transferred to GPU memory for fast access.
   d. **Kernel Execution:** The kernel is launched with a configured number of blocks and threads per block, and `cuda.synchronize()` ensures all threads complete before data retrieval.

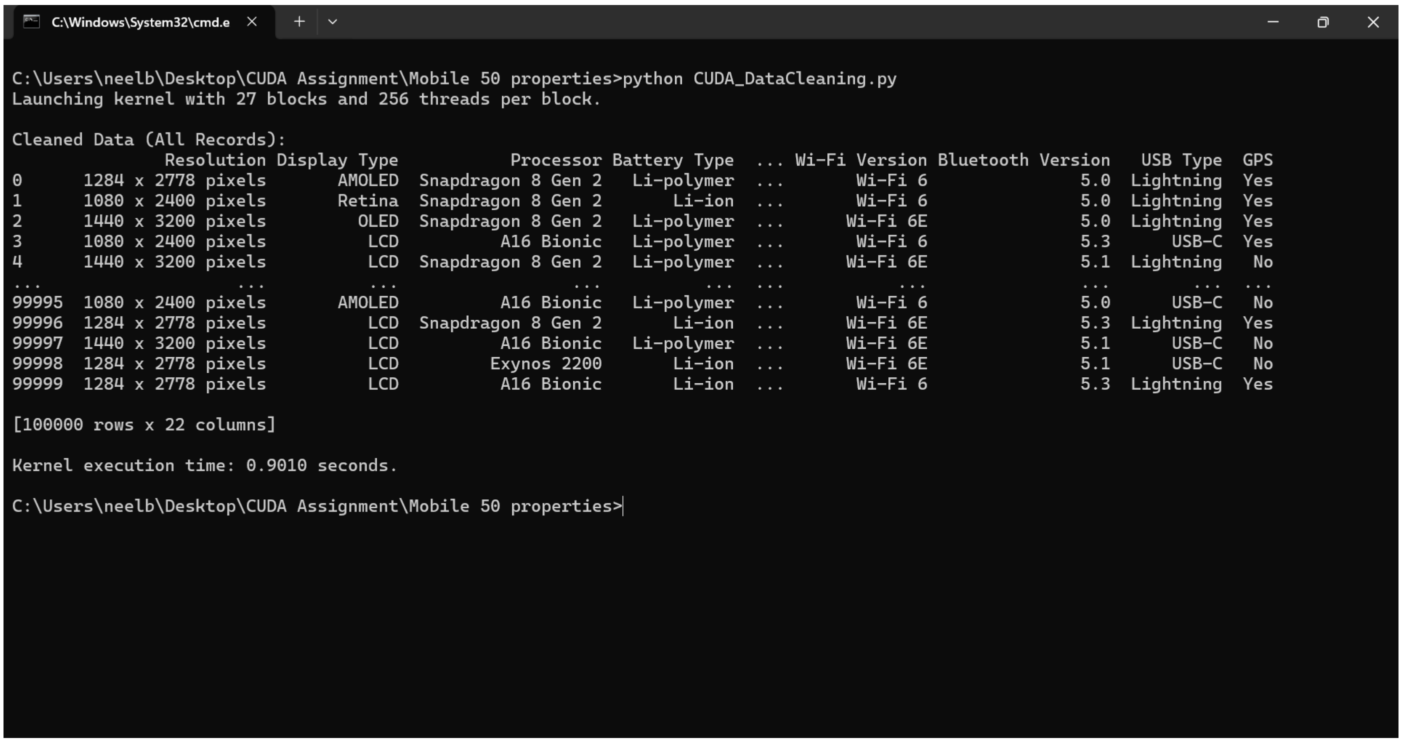
4. **Reconstruction & Final Output:**
   a. Cleaned ASCII codes are converted back to strings, updating the DataFrame with cleaned data.
   b. The cleaned data and kernel execution time are displayed as output.

Performance Measurement:

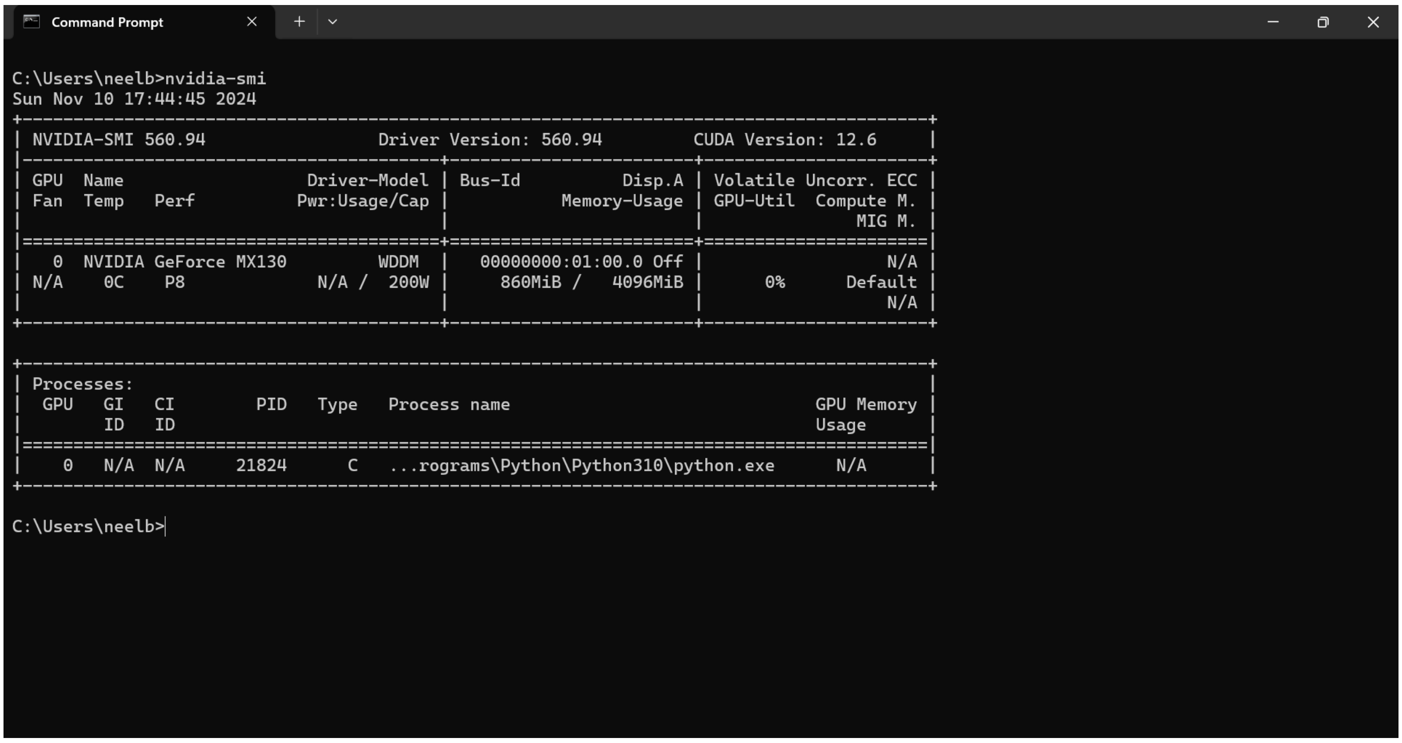
The program tracks and outputs the kernel execution time to highlight the efficiency gains achieved through parallel processing on the GPU.


CUDA_DataCleaning.py

```python
import time
from numba import cuda
import numpy as np
import pandas as pd
import warnings
from numba.core.errors import NumbaPerformanceWarning

# Load dataset
file_path = 'NoisyMobileDataLight.csv'  # Update this to your file path
df = pd.read_csv(file_path)

# Columns to clean
columns_to_clean = [
    'Resolution', 'Display Type', 'Processor', 'Battery Type', 'Main Camera Resolution',
    'Front Camera Resolution', 'Video Recording Resolution', 'Flash Type',
    'Fingerprint Sensor', 'Build Material', 'Water & Dust Resistance Rating',
    'SIM Type', 'Speaker Type', 'Operating System', 'UI Skin', 'Voice Assistant',
    'Security Patch Frequency', '4G LTE Bands', 'Wi-Fi Version', 'Bluetooth Version',
    'USB Type', 'GPS'
]

# Ensure columns to clean exist in df and set them to string type, filling NaNs
df[columns_to_clean] = df[columns_to_clean].fillna('').astype(str)

# Convert each column into ASCII arrays (truncate or pad each to MAX_STR_LEN)
MAX_STR_LEN = 100
ascii_array_3d = np.array([
    [
        [ord(char) if 0 ≤ ord(char) < 128 else -1 for char in value[:MAX_STR_LEN]] + [-1]
      * (MAX_STR_LEN - len(value[:MAX_STR_LEN]))
        for value in df[col]
    ]
    for col in columns_to_clean
], dtype=np.int32)

# CUDA Constants
RECORDS_PER_THREAD = 15
THREADS_PER_BLOCK = 256
RECORDS_PER_BLOCK = THREADS_PER_BLOCK * RECORDS_PER_THREAD
total_records = ascii_array_3d.shape[1]

# CUDA Kernel for cleaning ASCII data
@cuda.jit
def clean_ascii_in_chunks(ascii_array_3d):
    idx = cuda.grid(1)  # Global thread index
    start, end = idx * RECORDS_PER_THREAD, min((idx + 1) * RECORDS_PER_THREAD, ascii_array_3d.shape[1])

    for col in range(ascii_array_3d.shape[0]):
        for i in range(start, end):
            pos = 0
            for j in range(ascii_array_3d.shape[2]):
                if 0 ≤ ascii_array_3d[col, i, j] < 128:
                    ascii_array_3d[col, i, pos] = ascii_array_3d[col, i, j]
                    pos += 1
            for j in range(pos, ascii_array_3d.shape[2]):
                ascii_array_3d[col, i, j] = -1

# Copy data to device
d_ascii_array_3d = cuda.to_device(ascii_array_3d)

# Calculate blocks per grid and start timer
blocks_per_grid = (total_records + RECORDS_PER_BLOCK - 1) // RECORDS_PER_BLOCK
print(f"Launching kernel with {blocks_per_grid} blocks and {THREADS_PER_BLOCK} threads per block.")
warnings.filterwarnings("ignore", category=NumbaPerformanceWarning)
start_time = time.time()

# Launch kernel
clean_ascii_in_chunks[blocks_per_grid, THREADS_PER_BLOCK](d_ascii_array_3d)
cuda.synchronize()

# Calculate and print kernel execution time
kernel_execution_time = time.time() - start_time

# Copy cleaned data back to host
cleaned_ascii_array_3d = d_ascii_array_3d.copy_to_host()

# Convert cleaned ASCII arrays back to strings and update DataFrame
for col_idx, col_name in enumerate(columns_to_clean):
    df[col_name] = [''.join(chr(char_code) for char_code in row if char_code ≠ -1).strip()
                    for row in cleaned_ascii_array_3d[col_idx]]

# Display all cleaned data and kernel execution time
print("\nCleaned Data (All Records):")
print(df[columns_to_clean])

print(f"\nKernel execution time: {kernel_execution_time:.4f} seconds.")
```



```
C:\Users\neelb\Desktop\CUDA Assignment\Mobile 50 properties>python CUDA_DataCleaning.py
Launching kernel with 27 blocks and 256 threads per block.

Cleaned Data (All Records):
               Resolution Display Type           Processor Battery Type  ... Wi-Fi Version Bluetooth Version   USB Type  GPS
0      1284 x 2778 pixels       AMOLED  Snapdragon 8 Gen 2   Li-polymer  ...        Wi-Fi 6               5.0  Lightning  Yes
1      1080 x 2400 pixels       Retina  Snapdragon 8 Gen 2       Li-ion  ...        Wi-Fi 6               5.0  Lightning  Yes
2      1440 x 3200 pixels         OLED  Snapdragon 8 Gen 2   Li-polymer  ...       Wi-Fi 6E               5.0  Lightning  Yes
3      1080 x 2400 pixels          LCD          A16 Bionic   Li-polymer  ...        Wi-Fi 6               5.3      USB-C  Yes
4      1440 x 3200 pixels          LCD  Snapdragon 8 Gen 2   Li-polymer  ...       Wi-Fi 6E               5.1  Lightning   No
...                   ...          ...                 ...          ...  ...            ...               ...        ...  ...
99995  1080 x 2400 pixels       AMOLED          A16 Bionic   Li-polymer  ...        Wi-Fi 6               5.0      USB-C   No
99996  1284 x 2778 pixels          LCD  Snapdragon 8 Gen 2       Li-ion  ...       Wi-Fi 6E               5.3  Lightning  Yes
99997  1440 x 3200 pixels          LCD          A16 Bionic   Li-polymer  ...       Wi-Fi 6E               5.1      USB-C   No
99998  1284 x 2778 pixels          LCD         Exynos 2200       Li-ion  ...       Wi-Fi 6E               5.1      USB-C   No
99999  1284 x 2778 pixels          LCD          A16 Bionic       Li-ion  ...        Wi-Fi 6               5.3  Lightning  Yes

[100000 rows x 22 columns]

Kernel execution time: 0.9010 seconds.

C:\Users\neelb\Desktop\CUDA Assignment\Mobile 50 properties>
```



```
C:\Users\neelb>nvidia-smi
Sun Nov 10 17:44:45 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 560.94                 Driver Version: 560.94         CUDA Version: 12.6     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Driver-Model | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce MX130         WDDM  |   00000000:01:00.0 Off |                  N/A |
| N/A    0C    P8              N/A /  200W |     860MiB /   4096MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A     21824      C   ...rograms\Python\Python310\python.exe      N/A      |
+-----------------------------------------------------------------------------------------+

C:\Users\neelb>
```

Execution Time for CUDA – 0.9010 seconds

4. MPI

MPI Program Overview:

The provided MPI code is designed to clean a large CSV file (NoisyMobileDataLight.csv) in parallel, efficiently removing unwanted characters from the data using multiple processes.

Code Explanation

1. **Initialization and Setup:**
    - The code initializes MPI, determining the rank (process ID) and size (total number of processes).
    - Only the root process (rank 0) reads the CSV file and splits it into chunks of 1000 rows for parallel processing.

2. **Broadcasting and Distributing Chunks:**
    - The root process broadcasts the number of chunks to all other processes.
    - It distributes the chunks among the available processes using comm.scatter(), allowing each process to work on a subset of the data.

3. **Data Cleaning Process:**
    - Each process checks for received chunks. If chunks are present, it defines a cleaning function (clean_cell) that uses a regular expression to remove unwanted characters (Unicode noise) from each cell.
    - Processes iterate over their assigned chunks, applying the cleaning function, counting the cleaned cells, and printing progress messages.

4. **Error Handling and Monitoring:**
    - If an error occurs during cleaning, it is caught and logged.
    - Each process records its processing time and memory usage using the psutil library.

5. **Gathering Cleaned Data:**
    - Each process sends its cleaned chunks back to the root process using comm.gather().
    - The root process concatenates all cleaned chunks into a single DataFrame and saves it as cleaned_NoisyMobileDataLight.csv.

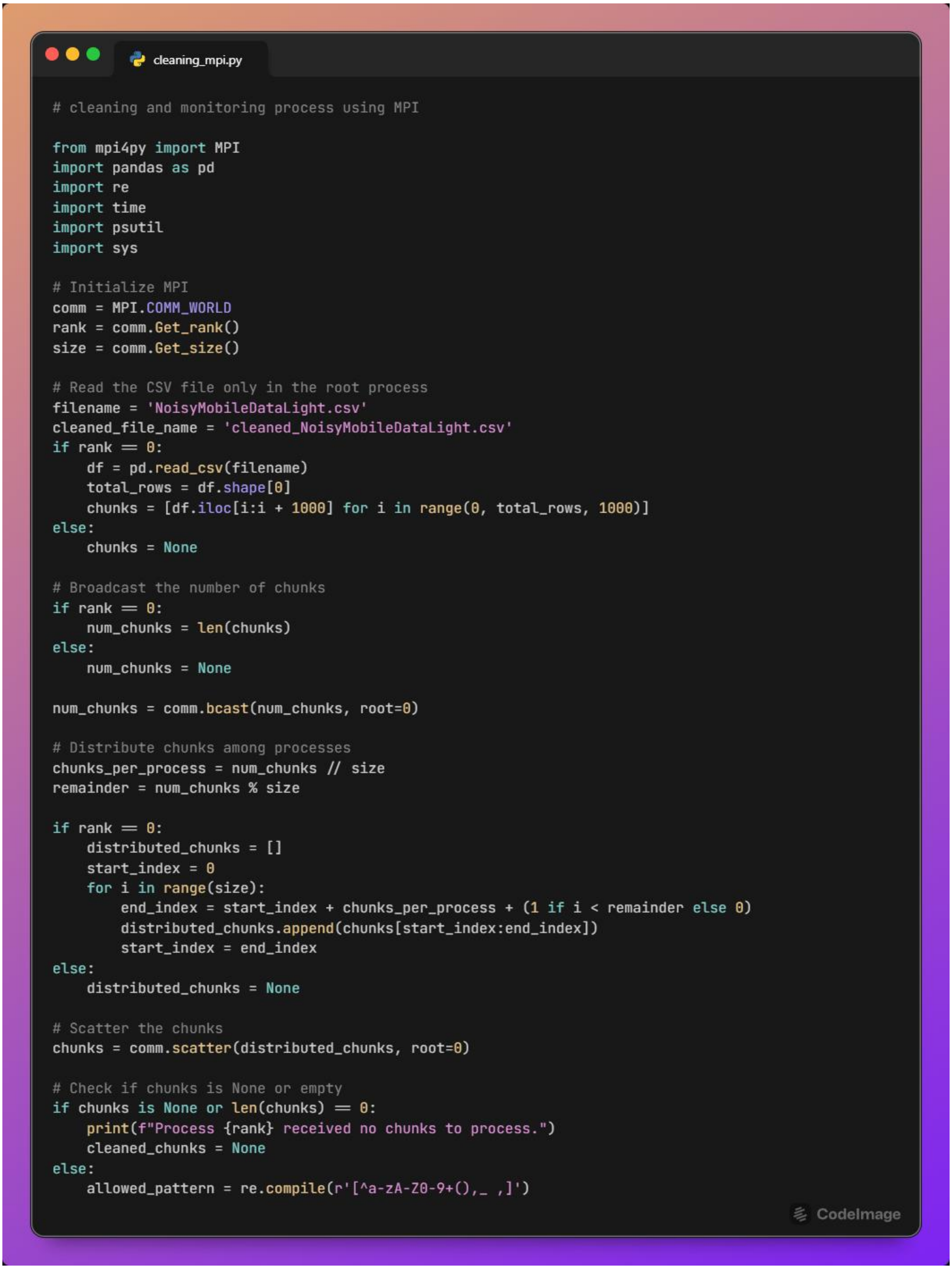
6. **Final Output:**
    - The root process prints the total number of cells processed and the overall cleaning time. saves cleaned file as cleaned_NoisyMobileDataLight.csv.

Command Explanation

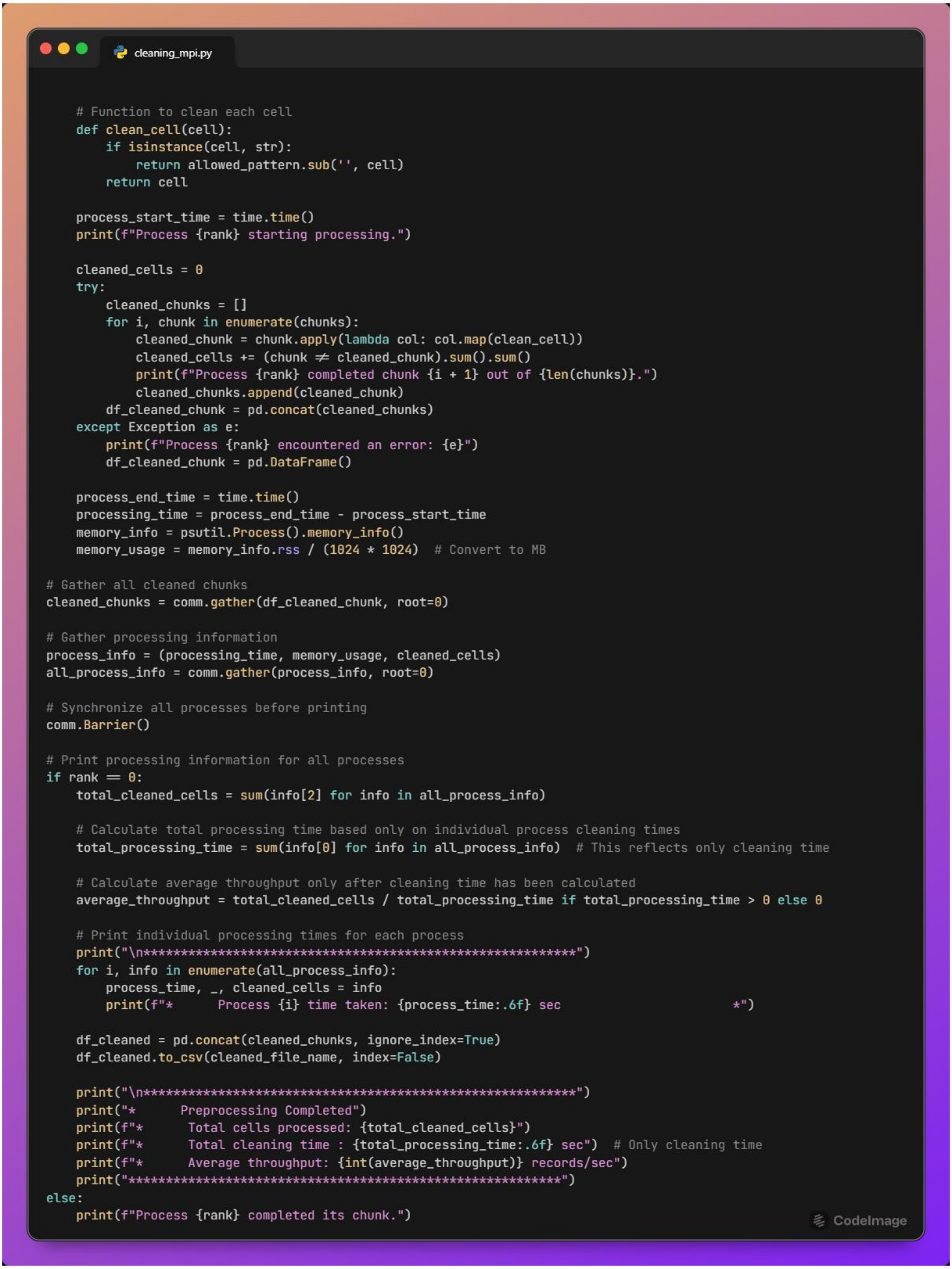
- The command mpiexec -n 5 python cleaning_mpi.py runs the script across 5 processes:
    - mpiexec: Initializes the MPI environment.
    - -n 5: Specifies the number of processes.
    - python cleaning_mpi.py: Executes the Python script containing the MPI code.

Summary for MPI

This code efficiently cleans a large CSV file in parallel, specifically targeting the removal of unwanted characters (Unicode noise) from the data. By leveraging MPI, it distributes the workload across multiple processes, significantly speeding up the data cleaning process and enhancing resource utilization


cleaning_mpi.py

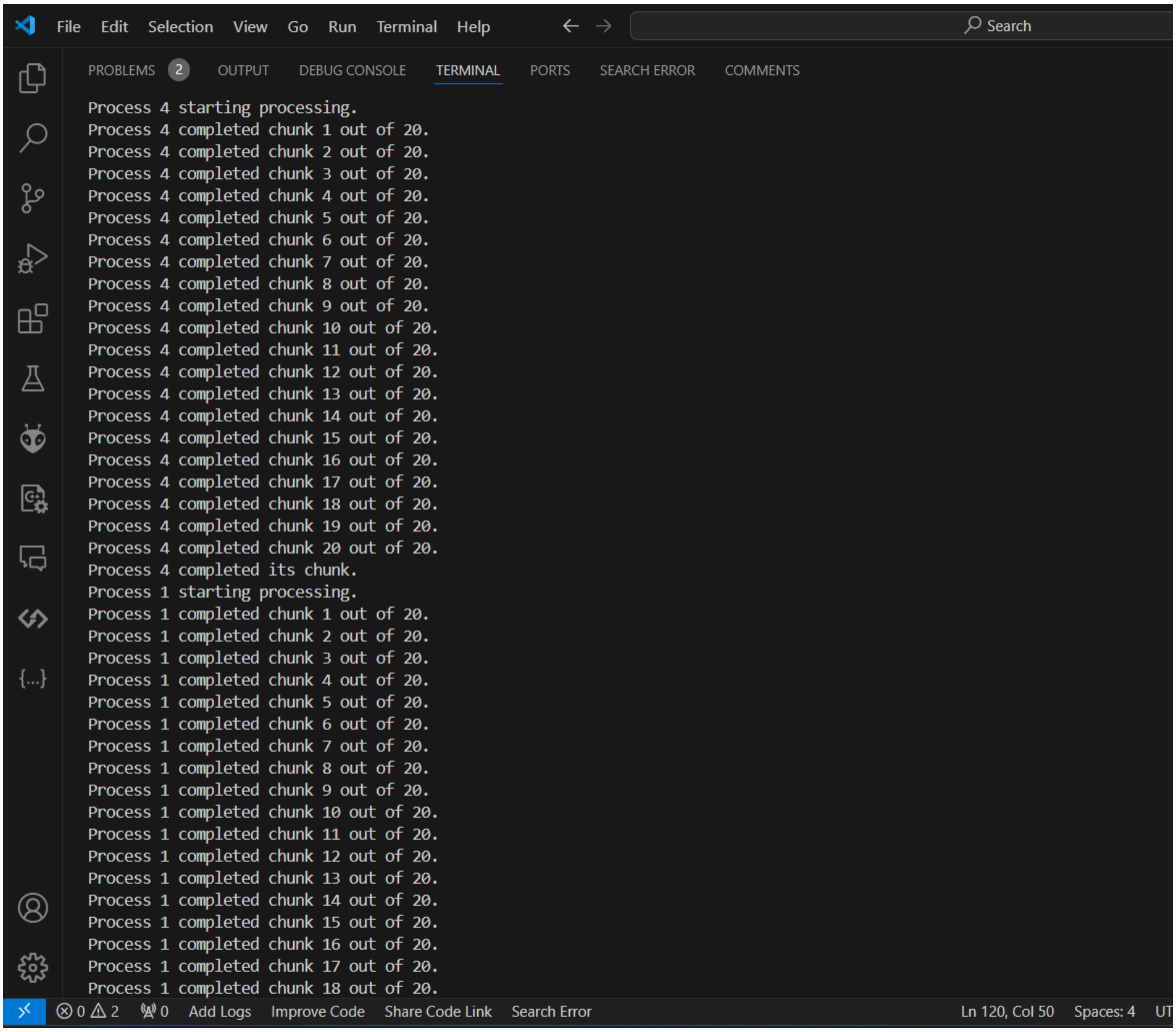
```python
# cleaning and monitoring process using MPI

from mpi4py import MPI
import pandas as pd
import re
import time
import psutil
import sys

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Read the CSV file only in the root process
filename = 'NoisyMobileDataLight.csv'
cleaned_file_name = 'cleaned_NoisyMobileDataLight.csv'
if rank == 0:
    df = pd.read_csv(filename)
    total_rows = df.shape[0]
    chunks = [df.iloc[i:i + 1000] for i in range(0, total_rows, 1000)]
else:
    chunks = None

# Broadcast the number of chunks
if rank == 0:
    num_chunks = len(chunks)
else:
    num_chunks = None

num_chunks = comm.bcast(num_chunks, root=0)

# Distribute chunks among processes
chunks_per_process = num_chunks // size
remainder = num_chunks % size

if rank == 0:
    distributed_chunks = []
    start_index = 0
    for i in range(size):
        end_index = start_index + chunks_per_process + (1 if i < remainder else 0)
        distributed_chunks.append(chunks[start_index:end_index])
        start_index = end_index
else:
    distributed_chunks = None

# Scatter the chunks
chunks = comm.scatter(distributed_chunks, root=0)

# Check if chunks is None or empty
if chunks is None or len(chunks) == 0:
    print(f"Process {rank} received no chunks to process.")
    cleaned_chunks = None
else:
    allowed_pattern = re.compile(r'[^a-zA-Z0-9+(),_ ,]')
```



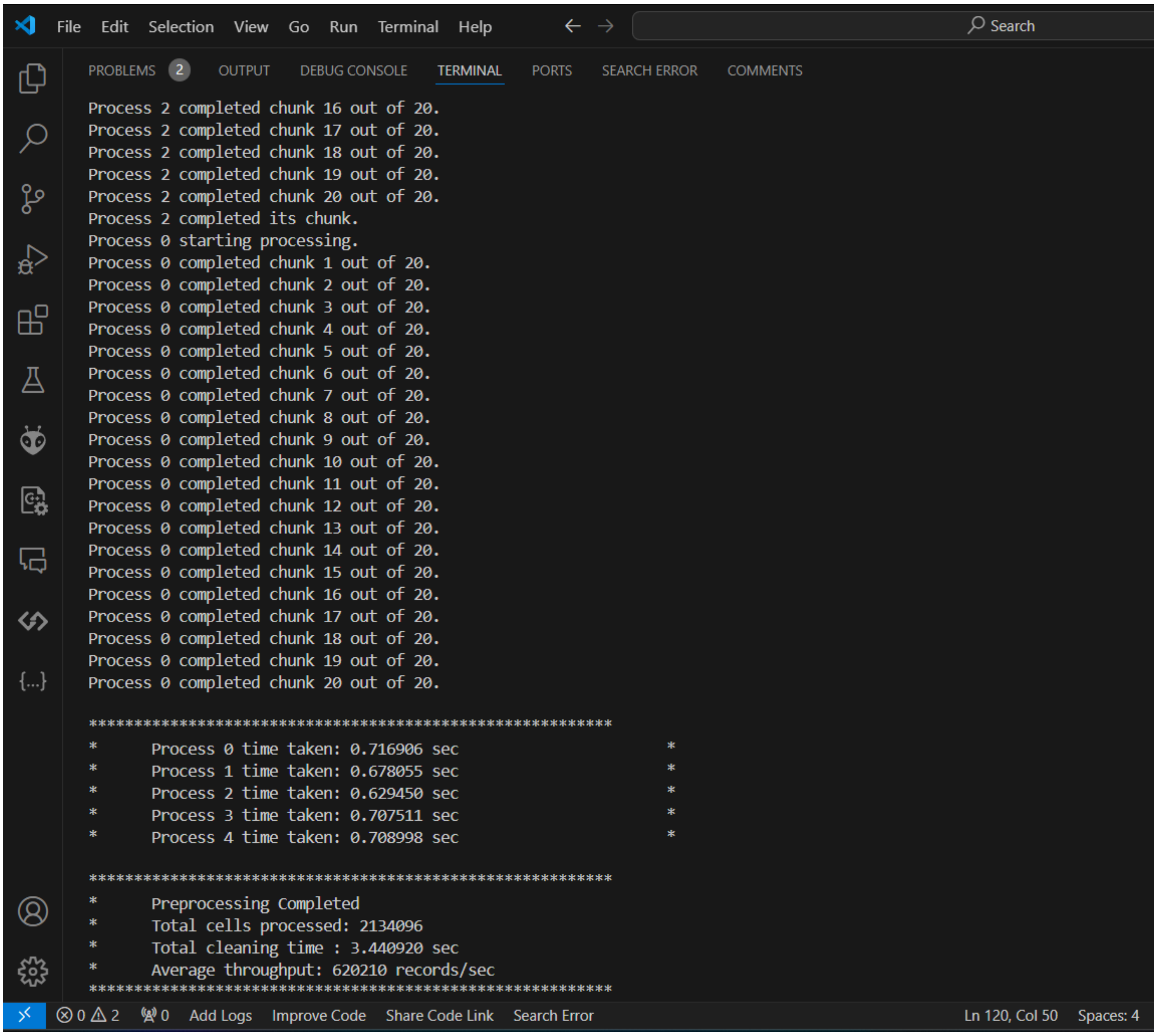
```python
    # Function to clean each cell
    def clean_cell(cell):
        if isinstance(cell, str):
            return allowed_pattern.sub('', cell)
        return cell

    process_start_time = time.time()
    print(f"Process {rank} starting processing.")

    cleaned_cells = 0
    try:
        cleaned_chunks = []
        for i, chunk in enumerate(chunks):
            cleaned_chunk = chunk.apply(lambda col: col.map(clean_cell))
            cleaned_cells += (chunk != cleaned_chunk).sum().sum()
            print(f"Process {rank} completed chunk {i + 1} out of {len(chunks)}.")
            cleaned_chunks.append(cleaned_chunk)
        df_cleaned_chunk = pd.concat(cleaned_chunks)
    except Exception as e:
        print(f"Process {rank} encountered an error: {e}")
        df_cleaned_chunk = pd.DataFrame()

    process_end_time = time.time()
    processing_time = process_end_time - process_start_time
    memory_info = psutil.Process().memory_info()
    memory_usage = memory_info.rss / (1024 * 1024)  # Convert to MB

# Gather all cleaned chunks
cleaned_chunks = comm.gather(df_cleaned_chunk, root=0)

# Gather processing information
process_info = (processing_time, memory_usage, cleaned_cells)
all_process_info = comm.gather(process_info, root=0)

# Synchronize all processes before printing
comm.Barrier()

# Print processing information for all processes
if rank == 0:
    total_cleaned_cells = sum(info[2] for info in all_process_info)

    # Calculate total processing time based only on individual process cleaning times
    total_processing_time = sum(info[0] for info in all_process_info)  # This reflects only cleaning time

    # Calculate average throughput only after cleaning time has been calculated
    average_throughput = total_cleaned_cells / total_processing_time if total_processing_time > 0 else 0

    # Print individual processing times for each process
    print("\n**********************************************************")
    for i, info in enumerate(all_process_info):
        process_time, _, cleaned_cells = info
        print(f"*      Process {i} time taken: {process_time:.6f} sec                       *")

    df_cleaned = pd.concat(cleaned_chunks, ignore_index=True)
    df_cleaned.to_csv(cleaned_file_name, index=False)

    print("\n**********************************************************")
    print("*      Preprocessing Completed")
    print(f"*      Total cells processed: {total_cleaned_cells}")
    print(f"*      Total cleaning time : {total_processing_time:.6f} sec")  # Only cleaning time
    print(f"*      Average throughput: {int(average_throughput)} records/sec")
    print("**********************************************************")
else:
    print(f"Process {rank} completed its chunk.")
```

MPI **Output:**

mpiexec -n 5 python cleaning_mpi.py


File Edit Selection View Go Run Terminal Help
Search
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS SEARCH ERROR COMMENTS
0 2 0 Add Logs Improve Code Share Code Link Search Error Ln 120, Col 50 Spaces: 4

```
Process 4 starting processing.
Process 4 completed chunk 1 out of 20.
Process 4 completed chunk 2 out of 20.
Process 4 completed chunk 3 out of 20.
Process 4 completed chunk 4 out of 20.
Process 4 completed chunk 5 out of 20.
Process 4 completed chunk 6 out of 20.
Process 4 completed chunk 7 out of 20.
Process 4 completed chunk 8 out of 20.
Process 4 completed chunk 9 out of 20.
Process 4 completed chunk 10 out of 20.
Process 4 completed chunk 11 out of 20.
Process 4 completed chunk 12 out of 20.
Process 4 completed chunk 13 out of 20.
Process 4 completed chunk 14 out of 20.
Process 4 completed chunk 15 out of 20.
Process 4 completed chunk 16 out of 20.
Process 4 completed chunk 17 out of 20.
Process 4 completed chunk 18 out of 20.
Process 4 completed chunk 19 out of 20.
Process 4 completed chunk 20 out of 20.
Process 4 completed its chunk.
Process 1 starting processing.
Process 1 completed chunk 1 out of 20.
Process 1 completed chunk 2 out of 20.
Process 1 completed chunk 3 out of 20.
Process 1 completed chunk 4 out of 20.
Process 1 completed chunk 5 out of 20.
Process 1 completed chunk 6 out of 20.
Process 1 completed chunk 7 out of 20.
Process 1 completed chunk 8 out of 20.
Process 1 completed chunk 9 out of 20.
Process 1 completed chunk 10 out of 20.
Process 1 completed chunk 11 out of 20.
Process 1 completed chunk 12 out of 20.
Process 1 completed chunk 13 out of 20.
Process 1 completed chunk 14 out of 20.
Process 1 completed chunk 15 out of 20.
Process 1 completed chunk 16 out of 20.
Process 1 completed chunk 17 out of 20.
Process 1 completed chunk 18 out of 20.
```



```
Process 2 completed chunk 16 out of 20.
Process 2 completed chunk 17 out of 20.
Process 2 completed chunk 18 out of 20.
Process 2 completed chunk 19 out of 20.
Process 2 completed chunk 20 out of 20.
Process 2 completed its chunk.
Process 0 starting processing.
Process 0 completed chunk 1 out of 20.
Process 0 completed chunk 2 out of 20.
Process 0 completed chunk 3 out of 20.
Process 0 completed chunk 4 out of 20.
Process 0 completed chunk 5 out of 20.
Process 0 completed chunk 6 out of 20.
Process 0 completed chunk 7 out of 20.
Process 0 completed chunk 8 out of 20.
Process 0 completed chunk 9 out of 20.
Process 0 completed chunk 10 out of 20.
Process 0 completed chunk 11 out of 20.
Process 0 completed chunk 12 out of 20.
Process 0 completed chunk 13 out of 20.
Process 0 completed chunk 14 out of 20.
Process 0 completed chunk 15 out of 20.
Process 0 completed chunk 16 out of 20.
Process 0 completed chunk 17 out of 20.
Process 0 completed chunk 18 out of 20.
Process 0 completed chunk 19 out of 20.
Process 0 completed chunk 20 out of 20.

***********************************************************
*      Process 0 time taken: 0.716906 sec                *
*      Process 1 time taken: 0.678055 sec                *
*      Process 2 time taken: 0.629450 sec                *
*      Process 3 time taken: 0.707511 sec                *
*      Process 4 time taken: 0.708998 sec                *

***********************************************************
*      Preprocessing Completed
*      Total cells processed: 2134096
*      Total cleaning time : 3.440920 sec
*      Average throughput: 620210 records/sec
***********************************************************
```

Execution Time for MPI – 3.440920 seconds

4. Performance Comparison Summary

Library	Time taken to pre-process the data
Open MP	0.8700 seconds
Pthreads	3.7700 seconds
MPI	3.4409 seconds
CUDA	0.9010 seconds

5. Observations

1. **OpenMP (0.87 seconds)**:
    a. **OpenMP** is typically a very efficient method for parallelizing CPU-bound tasks, especially for tasks that can be easily split into independent chunks. It works well when the task has a clear parallel structure (like your text cleaning loop) and the machine has multiple cores.
    b. OpenMP can provide good performance on multi-core CPUs without much additional complexity in the code. Given that it outperforms both **Pthreads** and **MPI**, it seems to be the most efficient choice for your task.

2. **Pthreads (3.77 seconds)**:
    a. **Pthreads** is a low-level threading model that requires manual management of threads and synchronization, which can introduce significant overhead and complexity. It is often slower than OpenMP for tasks that are well-suited to the higher-level parallel constructs in OpenMP.
    b. Managing threads manually in **Pthreads** can lead to issues with load balancing, synchronization overhead, and context-switching, especially if the workload is not optimally partitioned.

3. **MPI (3.44 seconds)**:
    a. **MPI** is designed for distributed systems and parallel programming across multiple machines, so for tasks that don't need distributed computing, it can be overkill.
    b. Even though **MPI** can be very efficient on clusters, it introduces additional complexity and overhead from communication between processes, which might be why it's slower in your case compared to **OpenMP**.

4. **CUDA (0.90 seconds)**:
    a. **CUDA** is used for GPU-based parallel computing, which is highly optimized for massive parallelism with thousands of threads on the GPU. **CUDA** is highly effective when you can break down a problem into many simple, parallel operations, such as matrix multiplications or image processing.
    b. Your result for **CUDA** (0.90 seconds) is close to **OpenMP**'s 0.87 seconds. This is expected, as the task you're performing (cleaning text lines) may not fully leverage the capabilities of a GPU, which excels in handling large datasets with parallel operations. In contrast, **OpenMP** and **CUDA** both perform similarly because the GPU likely isn't fully utilized for a relatively small-scale problem like text processing.

6. Performance Insights

- **OpenMP**'s speed is likely because it can effectively utilize the CPU's multiple cores without the overhead of managing threads manually, as seen in **Pthreads**. It abstracts away much of the complexity, making it easy to achieve parallelism.

- **Pthreads** is more low-level and generally requires more care in managing resources (e.g., thread synchronization, load balancing), which can introduce inefficiencies and slower performance.

- **MPI**, though very efficient in distributed computing environments (e.g., on clusters of machines), incurs communication overheads that seem unnecessary for your task, as it's designed for inter-process communication across nodes in a cluster.

- **CUDA** is optimized for parallel computing on the GPU, and while it performs well in your case (close to **OpenMP**), GPUs excel when you have a large number of parallelizable tasks (e.g., matrix operations, image processing). Text processing tasks may not fully leverage the GPU's potential.