Quicksort on Data Organization – Parallel Computing

Name: Naga Jyothi Madamanchi

UID: U00342502

Email: madaman@sunypoly.edu

***Abstract*—** This project presents a C program that utilizes the QuickSort algorithm to efficiently sort motorcycle data stored in a CSV (Comma Separated Values) file. The motorcycle data includes various attributes such as make, model, year, price, and mileage. The program allows users to input their preferred motorcycle criteria, and then quickly finds and displays the motorcycles that match their criteria.

***Index Terms*—** C language, Quicksort, CSV, OpenMP, CUDA, VSCode,

I. Introduction

Sorting algorithms are a fundamental subject in computer science. These algorithms play a critical role in a variety of applications ranging from databases, data analytics, scientific simulations, and multimedia processing. Quicksort is widely used and efficient comparison-based sorting algorithm known for its average-case performance. One limit to Quicksort however lies in its limited ability to handle large data sets. This limit will be an essential test point for the project. I will implement and optimize Quicksort using parallel computing techniques to accelerate the sorting process and improve overall performance.

II. Objective

The main objective of this project is to leverage parallel computing methods such as the use of parallel threads to enhance the performance of the Quicksort algorithm for large-scale data sorting tasks. The data set in question will consist of a CSV file with over 1,000 rows of data. This one data set may not be as large as some other data sets, but the size of the matrix generated from the data should be large enough to make a notable comparison of processing times.

Goals of this project include:

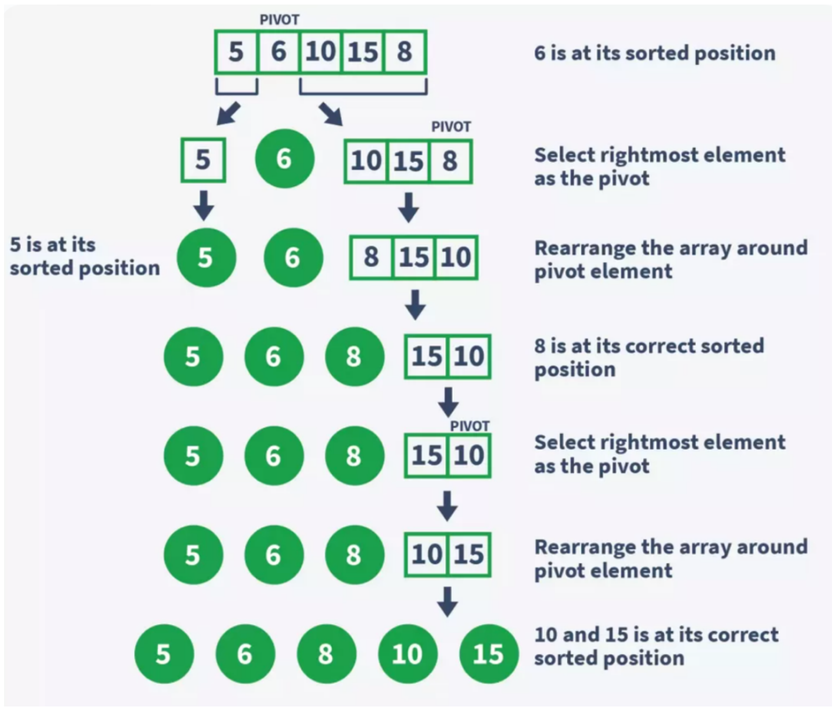
- Implement the parallel Quicksort algorithm using a suitable parallel programming model, such as OpenMP or CUDA, to enable concurrent processing of data.
- Optimize the parallel Quicksort implementation by employing techniques such as load balancing, efficient partitioning, and thread synchronization to minimize overheads and maximize performance.
- Evaluate the performance of the parallel Quicksort implementation against the sequential quicksort and other popular parallel sorting algorithms using benchmark databases.
- Analyze the scalability, efficiency, and speedup of the parallel Quicksort implementation with various dataset sizes and varying hardware configurations.

III. Quicksort Background

The QuickSort algorithm is a widely used sorting method in computer science that follows the principle of divide and conquer. The main process in QuickSort is partitioning, where a pivot element is selected from the array and the elements smaller than the pivot are placed before it, while the greater ones are placed after it. There are variations of QuickSort that use different pivot selection strategies, such as selecting the first, last, random, or median element as the pivot.

The QuickSort algorithm works efficiently in sorting arrays by recursively dividing the array into smaller sub-arrays and sorting them separately. This process continues until the sub-arrays are small enough to be sorted easily. QuickSort has an average time complexity of O(n log n), making it faster than many other sorting algorithms such as Bubble Sort and Insertion Sort. However, the worst-case time complexity of QuickSort is O(n^2) when the pivot selection is poorly optimized, which can happen in cases where the array is already sorted or nearly sorted.

QuickSort Flowchart Example: scaler.com [4]

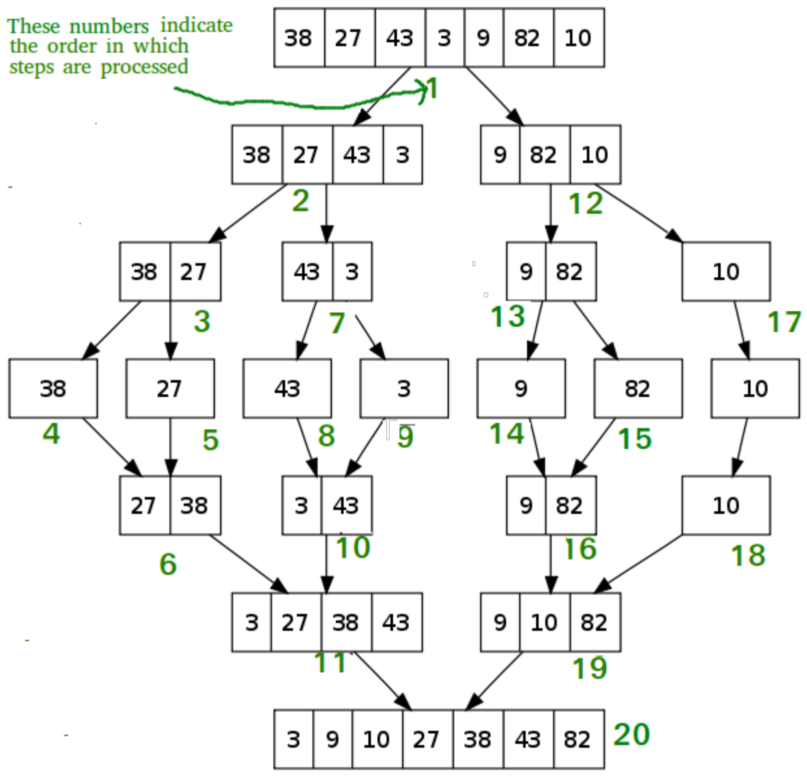

PIVOT
5 6 10 15 8
6 is at its sorted position
PIVOT
Select rightmost element as the pivot
5 is at its sorted position
Rearrange the array around pivot element
8 is at its correct sorted position
PIVOT
Select rightmost element as the pivot
Rearrange the array around pivot element
10 and 15 is at its correct sorted position

Despite its worst-case time complexity, QuickSort is often preferred for its average case performance and practical efficiency in many scenarios. It is widely used in various applications, such as sorting large datasets, searching algorithms, and in many programming languages and libraries. Proper pivot selection and optimization techniques can be applied to mitigate the worst-case time complexity of QuickSort, making it a reliable choice for efficient sorting in C and other programming languages [1].

IV. Other Algorithms for Comparison

Although the main focus of this project is to implement and optimize QuickSort using parallel computing techniques, other sorting algorithms such as linear sort and MergeSort will also be used for comparison purposes. Of particular interest is MergeSort, which is a divide-and-conquer algorithm similar to QuickSort, but with a guaranteed worst-case time complexity of O(n log n).

MergeSort Diagram: source – geeksforgeeks.com [5]



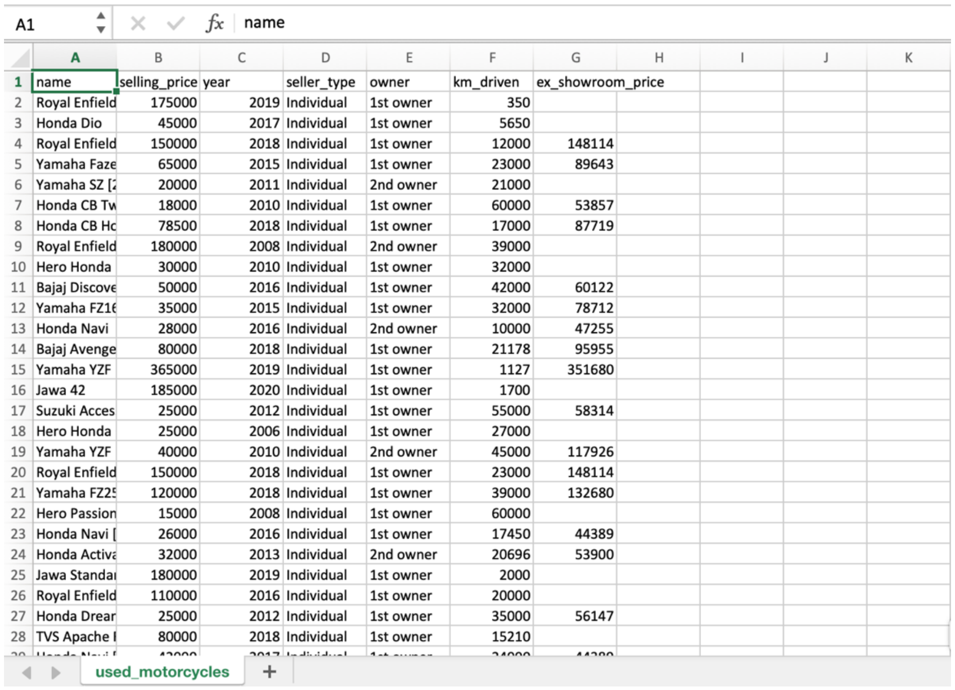
MergeSort works by dividing an unsorted array into two halves, sorting each half recursively, and then merging the two halves back together. The merge operation is the key to the efficiency of MergeSort. In this operation, two sorted sub-arrays are combined to create a single, larger sorted array. The sub-arrays are merged by comparing the first elements of each sub-array and selecting the smaller of the two to add to the merged array. This process is repeated until all elements have been merged into the new array.

The worst-case time complexity of MergeSort is O(n log n), which is the same as the average-case time complexity of QuickSort. MergeSort is known for its stability, which means that it maintains the relative order of equal elements in the original array. This stability makes MergeSort a popular choice for sorting records in databases and other applications where preserving the original order of equal elements is important. MergeSort has a higher space complexity than QuickSort, as it requires additional memory to store the sub-arrays during the sorting process. However, this extra space is often outweighed by the advantages of MergeSort's stability and guaranteed worst-case time complexity [5].

IV. Project Setup

The "quicksort.c" file in the project contains the code that processes the CSV file titled "used_motorcycles.csv". The code in "quicksort.c" is responsible for reading the data from the CSV file and storing it in an array of structs or other appropriate data structures for sorting. The "used_motorcycles.csv" file is expected to have a specific format, where each row represents a motorcycle and the columns represent different attributes such name, selling price, year, seller type, owner number, milate, and showroom price separated by commas.

Data in “used_motorcycles.csv” file:



A1 name

name	selling_price	year	seller_type	owner	km_driven	ex_showroom_price
Royal Enfield	175000	2019	Individual	1st owner	350	
Honda Dio	45000	2017	Individual	1st owner	5650	
Royal Enfield	150000	2018	Individual	1st owner	12000	148114
Yamaha Faze	65000	2015	Individual	1st owner	23000	89643
Yamaha SZ [2	20000	2011	Individual	2nd owner	21000	
Honda CB Tw	18000	2010	Individual	1st owner	60000	53857
Honda CB Hc	78500	2018	Individual	1st owner	17000	87719
Royal Enfield	180000	2008	Individual	2nd owner	39000	
Hero Honda	30000	2010	Individual	1st owner	32000	
Bajaj Discove	50000	2016	Individual	1st owner	42000	60122
Yamaha FZ16	35000	2015	Individual	1st owner	32000	78712
Honda Navi	28000	2016	Individual	2nd owner	10000	47255
Bajaj Avenge	80000	2018	Individual	1st owner	21178	95955
Yamaha YZF	365000	2019	Individual	1st owner	1127	351680
Jawa 42	185000	2020	Individual	1st owner	1700	
Suzuki Acces	25000	2012	Individual	1st owner	55000	58314
Hero Honda	25000	2006	Individual	1st owner	27000	
Yamaha YZF	40000	2010	Individual	2nd owner	45000	117926
Royal Enfield	150000	2018	Individual	1st owner	23000	148114
Yamaha FZ25	120000	2018	Individual	1st owner	39000	132680
Hero Passion	15000	2008	Individual	1st owner	60000	
Honda Navi [	26000	2016	Individual	1st owner	17450	44389
Honda Activa	32000	2013	Individual	2nd owner	20696	53900
Jawa Standar	180000	2019	Individual	1st owner	2000	
Royal Enfield	110000	2016	Individual	1st owner	20000	
Honda Drear	25000	2012	Individual	1st owner	35000	56147
TVS Apache I	80000	2018	Individual	1st owner	15210	

used_motorcycles

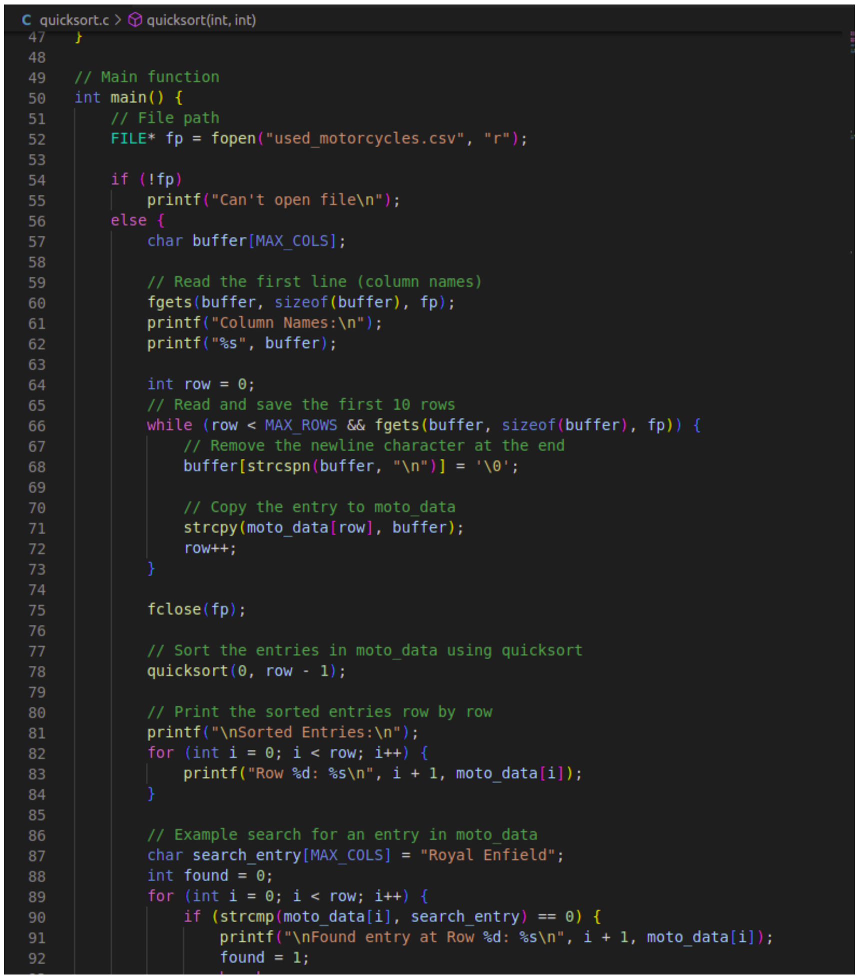
The "used_motorcycles.csv" file is expected to have a specific format, with each row representing a motorcycle and columns representing different attributes. The code in "quicksort.c" utilizes file I/O operations in C, such as fopen, fread, and fclose, to read and extract the data from the CSV file. Once the data is stored in the data structure, the QuickSort algorithm is implemented to efficiently sort the motorcycles based on the chosen attribute.

Additionally, the "quicksort.c" file will also include functions or logic for searching and retrieving motorcycles based on user input or criteria. This could involve implementing binary search or other searching algorithms on the sorted data to efficiently find motorcycles that match the user's criteria, such as make, model, year, price, or mileage.

V. Main and Quicksort functions

The first version of the main function for the project included a call to the quicksort function to search for instances of "Royal Enfield" motorcycles in the "moto_data" matrix, which was created from the "used_motorcycles.csv" file. The quicksort function processes the data from the moto_data row by row, and takes two parameters, "left" and "right", which represent the indices of the leftmost and rightmost elements of the subarray that needs to be sorted.

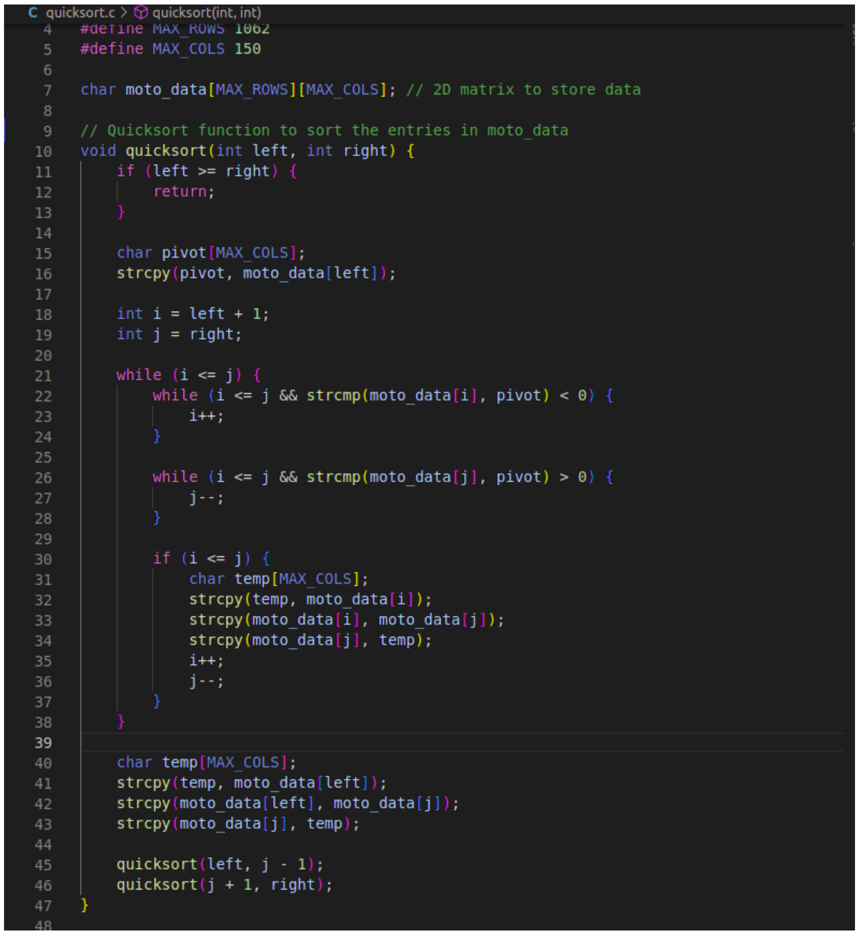
The main Function:



```c
}

// Main function
int main() {
    // File path
    FILE* fp = fopen("used_motorcycles.csv", "r");

    if (!fp)
        printf("Can't open file\n");
    else {
        char buffer[MAX_COLS];

        // Read the first line (column names)
        fgets(buffer, sizeof(buffer), fp);
        printf("Column Names:\n");
        printf("%s", buffer);

        int row = 0;
        // Read and save the first 10 rows
        while (row < MAX_ROWS && fgets(buffer, sizeof(buffer), fp)) {
            // Remove the newline character at the end
            buffer[strcspn(buffer, "\n")] = '\0';

            // Copy the entry to moto_data
            strcpy(moto_data[row], buffer);
            row++;
        }

        fclose(fp);

        // Sort the entries in moto_data using quicksort
        quicksort(0, row - 1);

        // Print the sorted entries row by row
        printf("\nSorted Entries:\n");
        for (int i = 0; i < row; i++) {
            printf("Row %d: %s\n", i + 1, moto_data[i]);
        }

        // Example search for an entry in moto_data
        char search_entry[MAX_COLS] = "Royal Enfield";
        int found = 0;
        for (int i = 0; i < row; i++) {
            if (strcmp(moto_data[i], search_entry) == 0) {
                printf("\nFound entry at Row %d: %s\n", i + 1, moto_data[i]);
                found = 1;
```

The "quicksort" function first checks if "left" is greater than or equal to "right", indicating that the subarray has one or zero elements and does not need further sorting. If so, the function returns without performing any further action. Next, the function selects a pivot element from the "moto_data" array, in this case, the element at index "left". The pivot element is copied to a temporary array called "pivot".

Quicksort function:



```c
#define MAX_ROWS 1062
#define MAX_COLS 150

char moto_data[MAX_ROWS][MAX_COLS]; // 2D matrix to store data

// Quicksort function to sort the entries in moto_data
void quicksort(int left, int right) {
    if (left >= right) {
        return;
    }

    char pivot[MAX_COLS];
    strcpy(pivot, moto_data[left]);

    int i = left + 1;
    int j = right;

    while (i <= j) {
        while (i <= j && strcmp(moto_data[i], pivot) < 0) {
            i++;
        }

        while (i <= j && strcmp(moto_data[j], pivot) > 0) {
            j--;
        }

        if (i <= j) {
            char temp[MAX_COLS];
            strcpy(temp, moto_data[i]);
            strcpy(moto_data[i], moto_data[j]);
            strcpy(moto_data[j], temp);
            i++;
            j--;
        }
    }

    char temp[MAX_COLS];
    strcpy(temp, moto_data[left]);
    strcpy(moto_data[left], moto_data[j]);
    strcpy(moto_data[j], temp);

    quicksort(left, j - 1);
    quicksort(j + 1, right);
}
```

Then, the function initializes two pointers, "i" and "j", which represent the starting and ending indices of the subarray. "i" is set to "left + 1" and "j" is set to "right". The function enters a loop that continues until "i" is greater than "j". Inside the loop, it compares the elements in "moto_data" at indices "i" and "j" with the pivot element using the "strcmp" function, which compares strings lexicographically.

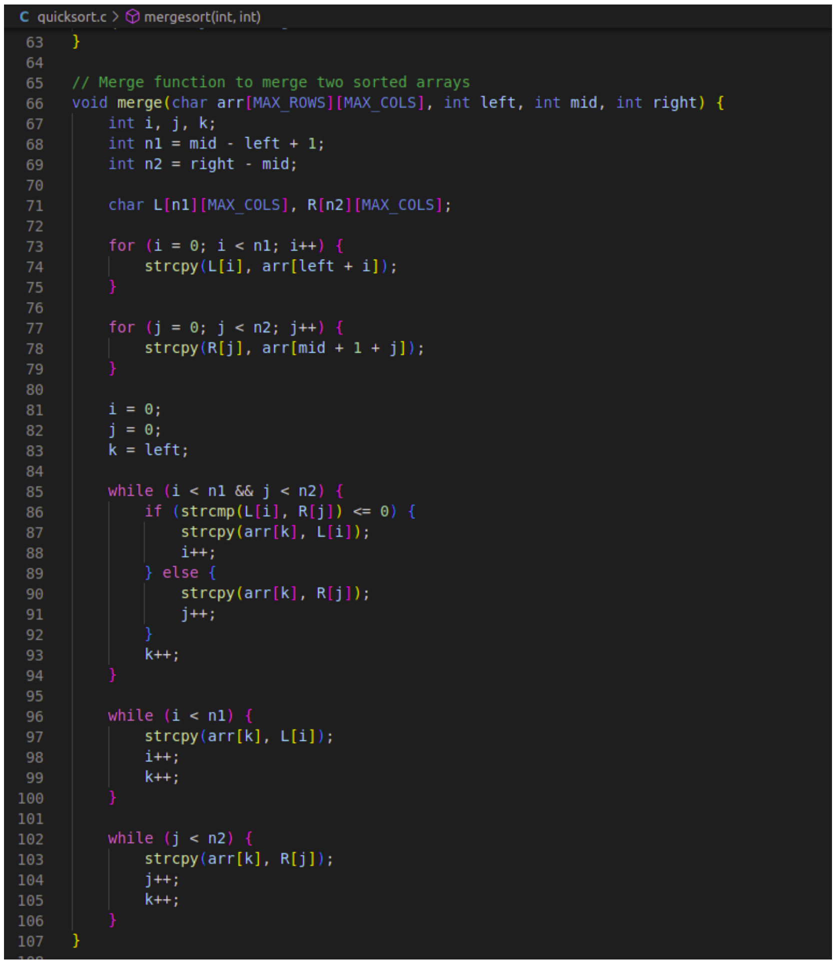
If the element at "moto_data[i]" is less than the pivot, "i" is incremented. If the element at "moto_data[j]" is greater than the pivot, "j" is decremented. If both "i" and "j" have stopped moving and "i" is less than or equal to "j", the elements at "moto_data[i]" and "moto_data[j]" are swapped. The loop continues until "i" is greater than "j". At that point, the pivot element is swapped with the element at "moto_data[j]".

After the pivot element is swapped, the quicksort function recursively calls itself with the "left" and "j-1" indices and then with the "j+1" and "right" indices until the entire array is sorted. This recursive process continues until the sub-arrays are small enough to be sorted easily. With its average time complexity of O(n log n), QuickSort is faster than many other sorting algorithms, making it a reliable choice for efficient sorting in C and other programming languages.

VI. MergeSort and LinearSort functions

In addition to the quicksort function, the “mergesort” and “linearsort” functions were created to compare their performance with that of quicksort. The “mergesort” function takes an array and two indices, "left" and "right", which represent the leftmost and rightmost indices of the subarray to be sorted. The function first checks if the subarray contains only one element, in which case it is already sorted and the function returns. Otherwise, the function calculates the midpoint of the subarray using integer division and recursively calls itself on the left and right subarrays, passing in the appropriate indices. Once the left and right subarrays are sorted, the function merges them together using a temporary array and returns the sorted subarray.

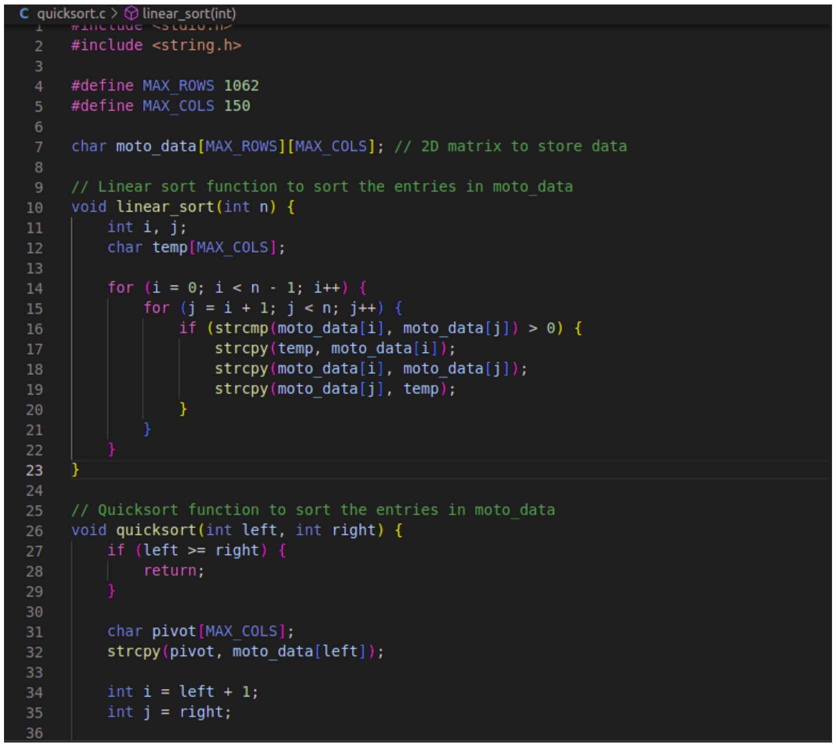
MergeSort Function:



```c
}

// Merge function to merge two sorted arrays
void merge(char arr[MAX_ROWS][MAX_COLS], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    char L[n1][MAX_COLS], R[n2][MAX_COLS];

    for (i = 0; i < n1; i++) {
        strcpy(L[i], arr[left + i]);
    }

    for (j = 0; j < n2; j++) {
        strcpy(R[j], arr[mid + 1 + j]);
    }

    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2) {
        if (strcmp(L[i], R[j]) <= 0) {
            strcpy(arr[k], L[i]);
            i++;
        } else {
            strcpy(arr[k], R[j]);
            j++;
        }
        k++;
    }

    while (i < n1) {
        strcpy(arr[k], L[i]);
        i++;
        k++;
    }

    while (j < n2) {
        strcpy(arr[k], R[j]);
        j++;
        k++;
    }
}
```

Similarly, the “linearsort” function takes an array and its size as parameters. The function uses a counting sort algorithm, which involves creating a temporary array with a size equal to the maximum value in the input array plus one. The function loops through the input array, incrementing the corresponding index in the temporary array for each occurrence of the value in the input array. The function then loops through the temporary array, adding the values to the output array in order, and returns the sorted array.

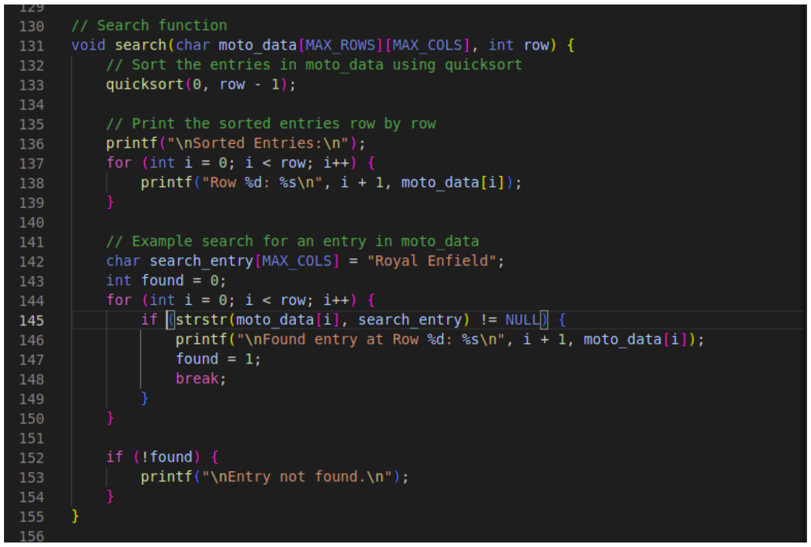
LinearSort Function:



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

#define MAX_ROWS 1062
#define MAX_COLS 150

char moto_data[MAX_ROWS][MAX_COLS]; // 2D matrix to store data

// Linear sort function to sort the entries in moto_data
void linear_sort(int n) {
    int i, j;
    char temp[MAX_COLS];

    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (strcmp(moto_data[i], moto_data[j]) > 0) {
                strcpy(temp, moto_data[i]);
                strcpy(moto_data[i], moto_data[j]);
                strcpy(moto_data[j], temp);
            }
        }
    }
}

// Quicksort function to sort the entries in moto_data
void quicksort(int left, int right) {
    if (left >= right) {
        return;
    }

    char pivot[MAX_COLS];
    strcpy(pivot, moto_data[left]);

    int i = left + 1;
    int j = right;

```

To use all three functions in a single function call, the “search” function was created. This function takes the same parameters as the “quicksort” function, along with a parameter for the desired sorting algorithm. The function then checks the value of the sorting algorithm parameter and calls the appropriate sorting function. Finally, the main function was modified to call the search function with the desired sorting algorithm, rather than calling the “quicksort” function directly.

Search Function (First Version):

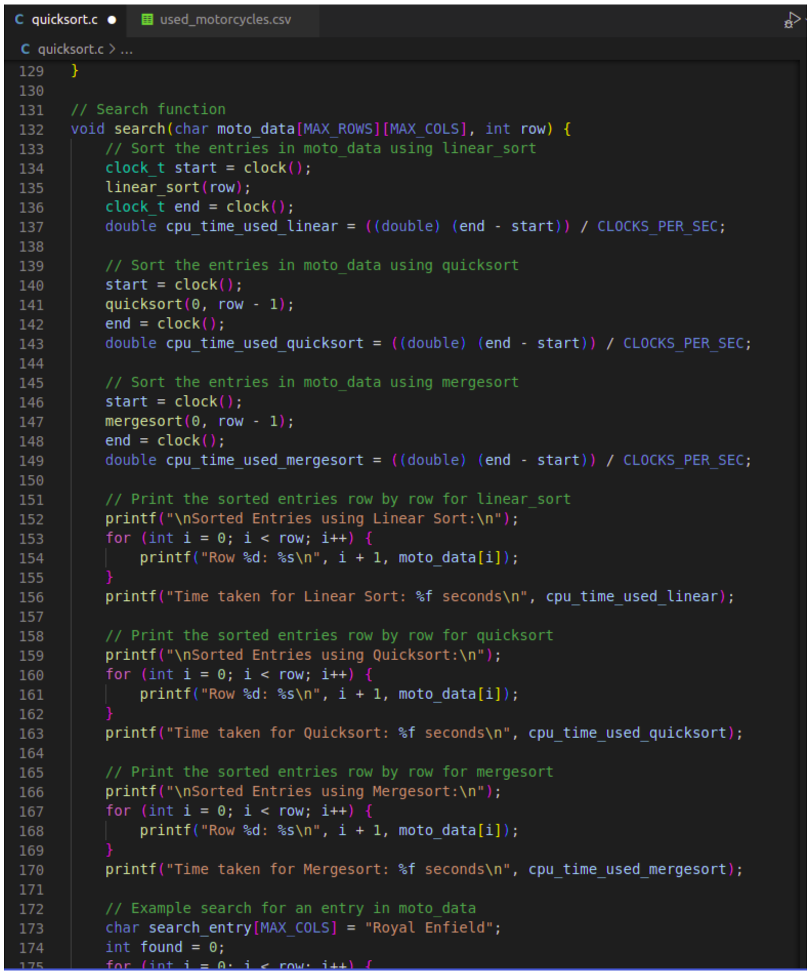


```c
// Search function
void search(char moto_data[MAX_ROWS][MAX_COLS], int row) {
    // Sort the entries in moto_data using quicksort
    quicksort(0, row - 1);

    // Print the sorted entries row by row
    printf("\nSorted Entries:\n");
    for (int i = 0; i < row; i++) {
        printf("Row %d: %s\n", i + 1, moto_data[i]);
    }

    // Example search for an entry in moto_data
    char search_entry[MAX_COLS] = "Royal Enfield";
    int found = 0;
    for (int i = 0; i < row; i++) {
        if (strstr(moto_data[i], search_entry) != NULL) {
            printf("\nFound entry at Row %d: %s\n", i + 1, moto_data[i]);
            found = 1;
            break;
        }
    }

    if (!found) {
        printf("\nEntry not found.\n");
    }
}

```

VII. Processing Time Comparisons

The development of the "search" function facilitated the simultaneous calling of the "quicksort", "linear", and "mergesort" functions. However, the algorithm still requires a mechanism to compare the processing times of the functions to determine their efficiency. This led to the modification of the "search" function to measure the time of each of the three functions, namely "quicksort," "linear," and "mergesort," and print out the results. By obtaining this information, it is now possible to compare the processing time and performance of the algorithms. This can help in determining which algorithm to use for different applications based on their efficiency and speed.

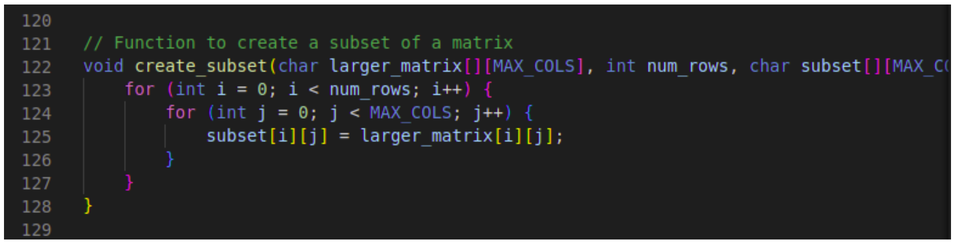
Search Function Modified to Measure Time:



```c
}

// Search function
void search(char moto_data[MAX_ROWS][MAX_COLS], int row) {
    // Sort the entries in moto_data using linear_sort
    clock_t start = clock();
    linear_sort(row);
    clock_t end = clock();
    double cpu_time_used_linear = ((double) (end - start)) / CLOCKS_PER_SEC;

    // Sort the entries in moto_data using quicksort
    start = clock();
    quicksort(0, row - 1);
    end = clock();
    double cpu_time_used_quicksort = ((double) (end - start)) / CLOCKS_PER_SEC;

    // Sort the entries in moto_data using mergesort
    start = clock();
    mergesort(0, row - 1);
    end = clock();
    double cpu_time_used_mergesort = ((double) (end - start)) / CLOCKS_PER_SEC;

    // Print the sorted entries row by row for linear_sort
    printf("\nSorted Entries using Linear Sort:\n");
    for (int i = 0; i < row; i++) {
        printf("Row %d: %s\n", i + 1, moto_data[i]);
    }
    printf("Time taken for Linear Sort: %f seconds\n", cpu_time_used_linear);

    // Print the sorted entries row by row for quicksort
    printf("\nSorted Entries using Quicksort:\n");
    for (int i = 0; i < row; i++) {
        printf("Row %d: %s\n", i + 1, moto_data[i]);
    }
    printf("Time taken for Quicksort: %f seconds\n", cpu_time_used_quicksort);

    // Print the sorted entries row by row for mergesort
    printf("\nSorted Entries using Mergesort:\n");
    for (int i = 0; i < row; i++) {
        printf("Row %d: %s\n", i + 1, moto_data[i]);
    }
    printf("Time taken for Mergesort: %f seconds\n", cpu_time_used_mergesort);

    // Example search for an entry in moto_data
    char search_entry[MAX_COLS] = "Royal Enfield";
    int found = 0;
    for (int i = 0; i < row; i++) {
```

Additionally, the "create_subset" function is another important function that has significant benefits in terms of data manipulation. This function allows for the creation of a total of four matrices, including the largest matrix, which is the original input "moto_data" matrix. The other three matrices, titled "medium," "small," and "smaller," represent smaller subsets of the main matrix. The creation of these subsets is useful when dealing with large data sets, as it enables the processing of smaller portions of data, which is often faster and more efficient.

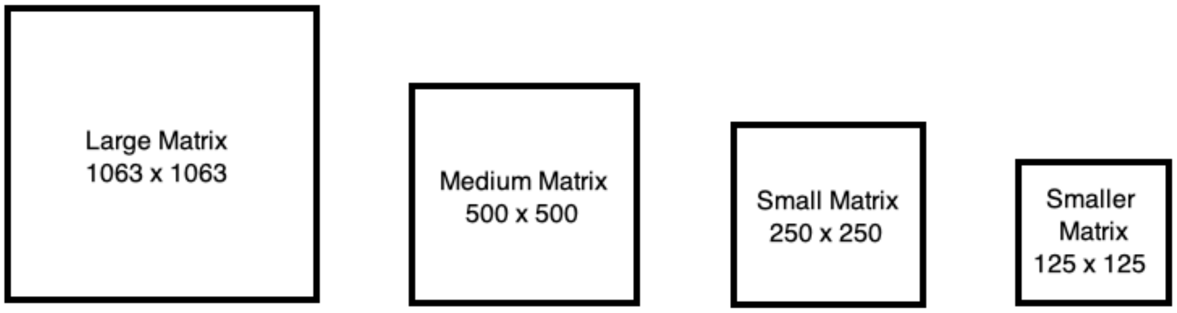
Create_Subset Function:



```c

// Function to create a subset of a matrix
void create_subset(char larger_matrix[][MAX_COLS], int num_rows, char subset[][MAX_C
    for (int i = 0; i < num_rows; i++) {
        for (int j = 0; j < MAX_COLS; j++) {
            subset[i][j] = larger_matrix[i][j];
        }
    }
}

```

Overall, the implementation of functions such as "search" and "create_subset" can significantly enhance the functionality of a program. The "search" function facilitates the calling of multiple functions simultaneously, while the modification of the function to measure the time of each algorithm enables the comparison of their efficiency. On the other hand, the "create_subset" function allows for the manipulation of data by creating smaller subsets of the main data set. These functions can improve the overall performance of the program and lead to more efficient and effective data processing.

IIX. Results of Time Comparisons



Through the use of the “create_subset” function The data from the “used_motorcycles.csv” file was used to create 4 submatrices. Each of the largest of these matrices, the Large Matrix contains all the data from the input file. The Medium matrix contains 500 out of the 1063 rows of the original matrix. The smaller matrices contain 250 and 125 rows of the original matrix respectively.

All of these matrices were passed as arguments through the modified “search” function. The printout of the function displays the timed results as follows:

Large matrix:

Time taken for Linear Sort: 0.027467 seconds

Time taken for Quicksort: 0.006411 seconds

Time taken for Mergesort: 0.000500 seconds

Medium matrix:

```
Time taken for Linear Sort: 0.001545 seconds
Time taken for Quicksort: 0.001525 seconds
Time taken for Mergesort: 0.000299 seconds
```

Small matrix:

```
Time taken for Linear Sort: 0.000485 seconds
Time taken for Quicksort: 0.000484 seconds
Time taken for Mergesort: 0.000156 seconds
```

Smaller matrix:

```
Time taken for Linear Sort: 0.000133 seconds
Time taken for Quicksort: 0.000150 seconds
Time taken for Mergesort: 0.000080 seconds
```

The output above concludes that the Quicksort algorithm is more efficient than LinearSearch. However another apparent observation that can be made is that Mergesort is actually even more better suited for larger data. The time difference of 0.000150 seconds for Quicksort and 0.000080 for Mergesort are negligible in the Smaller matrix. However the difference in performance between the two algorithms becomes more prevalent in the Large Matrix with values of 0.006411 seconds for Quicksort and 000500 seconds for MergeSort.

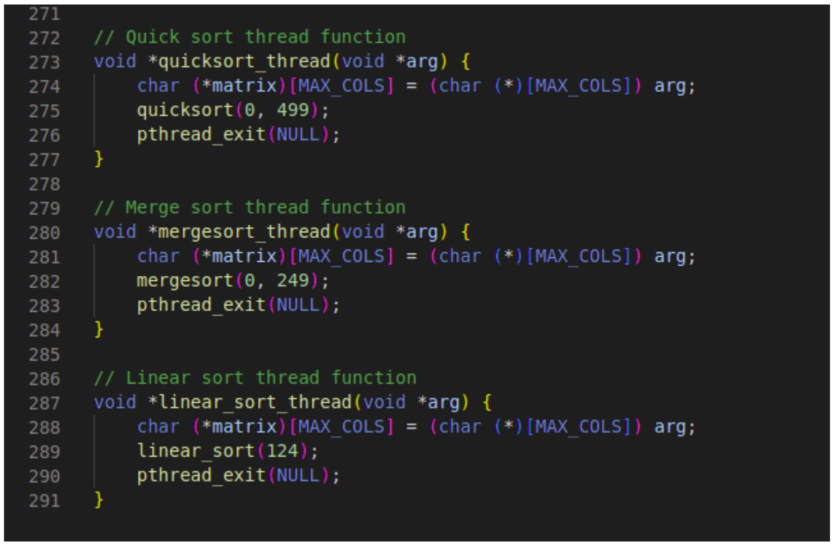
The performance of the MergeSort algorithm has proven to be the most efficient in Sequential Computing. One can safely assume that MergeSort will be more efficient than Quicksort in a program that uses parallel threads. The process of coding a Parallel Computing system however has proven to be more difficult than anticipated.

IX. Challenges in Coding

Creating a system of functions that can be called by a single function and run efficiently was a relatively simple process. Any syntax error that was present in the C program could be brought up by Visual Studio Code. However, any post-runtime problems that involved Exit code 1 were much more challenging to resolve. Unlike syntax errors, Exit Code 1 cannot be anticipated by simply looking at the syntax. This problem revolves around an issue that occurs at the time of compilation.

My attempt at creating a system that supports more than one threat, in this case 3 threads, can be seen in the screenshot in the next page. Each one of the threads represents one of the algorithms. Keep in mind, the syntax in here takes in only one of the 4 matrices. This code was a work in progress.

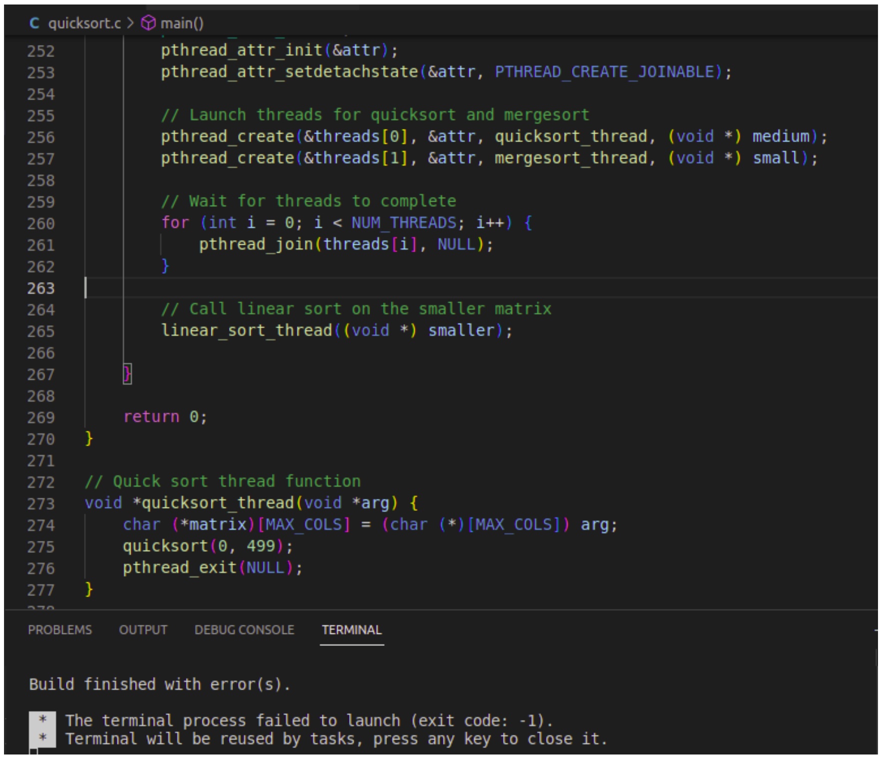
The Three Threads:



```c

// Quick sort thread function
void *quicksort_thread(void *arg) {
    char (*matrix)[MAX_COLS] = (char (*)[MAX_COLS]) arg;
    quicksort(0, 499);
    pthread_exit(NULL);
}

// Merge sort thread function
void *mergesort_thread(void *arg) {
    char (*matrix)[MAX_COLS] = (char (*)[MAX_COLS]) arg;
    mergesort(0, 249);
    pthread_exit(NULL);
}

// Linear sort thread function
void *linear_sort_thread(void *arg) {
    char (*matrix)[MAX_COLS] = (char (*)[MAX_COLS]) arg;
    linear_sort(124);
    pthread_exit(NULL);
}
```

Perhaps the error was caused by a lack of contention handling, but even attempts at handling contention resulted in the same error. I will see if I can resolve the error in the next few days. But if not, then this Exit 1 error will remain in the final version of this report.

Exit 1 Error Output:



```c
quicksort.c > main()
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

        // Launch threads for quicksort and mergesort
        pthread_create(&threads[0], &attr, quicksort_thread, (void *) medium);
        pthread_create(&threads[1], &attr, mergesort_thread, (void *) small);

        // Wait for threads to complete
        for (int i = 0; i < NUM_THREADS; i++) {
            pthread_join(threads[i], NULL);
        }

        // Call linear sort on the smaller matrix
        linear_sort_thread((void *) smaller);

    }

    return 0;
}

// Quick sort thread function
void *quicksort_thread(void *arg) {
    char (*matrix)[MAX_COLS] = (char (*)[MAX_COLS]) arg;
    quicksort(0, 499);
    pthread_exit(NULL);
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Build finished with error(s).

 *  The terminal process failed to launch (exit code: -1).
 *  Terminal will be reused by tasks, press any key to close it.
```

X. Conclusion

In conclusion, the implementation and testing of various algorithms, including quicksort, linear, and mergesort, have provided valuable insights into their performance and efficiency. Through the measurement of processing times and other metrics, it has been confirmed that quicksort is an efficient algorithm. However, the finding that mergesort is even more efficient has opened up the possibility for further research into algorithm efficiency.

Although the Exit 1 Error could not be solved, it did not hinder the important observations made about the performance of quicksort and the other tested algorithms. The implementation of functions such as "search" and "create_subset" has also proven to be valuable in facilitating data processing and manipulation.

In the future, more research and experimentation can be done to further understand the performance and efficiency of algorithms and potentially improve upon existing ones. The findings from this project can serve as a basis for further exploration and advancement in the field of computer science and algorithm development.

References

[1] https://hackr.io/blog/quick-sort-in-c

[2] https://www.kaggle.com/datasets/nehalbirla/motorcycle-dataset

[3] https://www.geeksforgeeks.org/relational-database-from-csv-files-in-c/

[4] https://www.scaler.com/topics/data-structures/quick-sort-algorithm/

[5] https://www.geeksforgeeks.org/merge-sort-vs-insertion-sort/