# CIS 520 - Project 4 Design Document

Collin Hammond, Jake McKown, Chuhao Chen
Eugene Vasserman
CIS520

## Software Architecture

We designed our implementations for this project as a C program called `scorecard-(pthread/mpi/openmp)`. The pthread and openmp versions take two arguments: first, the filename to read lines of text from, and second, the number of threads to use. They can be called by: `scorecard-(implementation) <filename> <threads>`. The mpi version takes only one command, the filename, and is run with `mpirun -n <threads> scorecard-mpi <filename>`. The first thing the program does after checking for correct arguments is get the size of the passed file and read the file into a buffer array of the same size. After that the program calculates what section of the buffer each thread should read from and starts the number of requested threads. Each thread then reads the highest ASCII value from every complete line in its section of the buffer and stores its results in another buffer. The program waits for each thread to finish its work and, when each one does, it gets the results from that thread and stores them in a new string buffer for all the threads' results, before finally printing all the results. Each version is implemented this way, with slight variations depending on how threads are meant to communiate for the different technologies.

Each thread gets information about the buffer and the work it needs to do through a custom thread_info_t struct. When each thread starts, it is given a start and end index in the struct. First it changes these indices to be at the next newline after each initial given index. Then the thread starts finding the highest ASCII value for each line between the new start and end index in the buffer and stores the value of each line in a results buffer. The results buffer is reallocated during runtime based on the number of lines being read by the thread.

The program was compiled and run on Beocat running the Linux 5.14.0 kernel with the Red Hat GCC 11.4.1 compiler.

## Performance Analysis

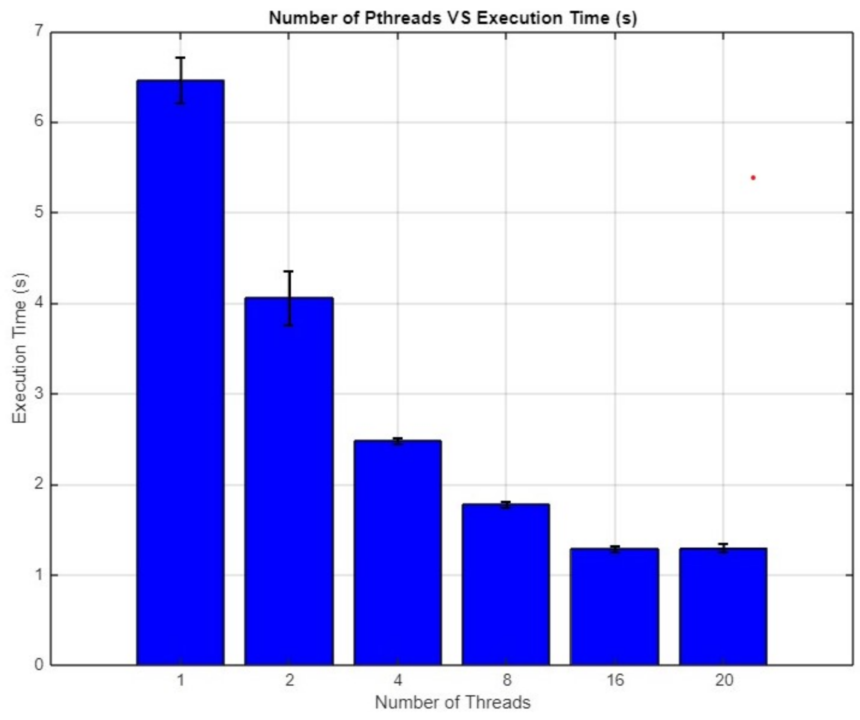
We conducted an evaluation of parallel processing capabilities using a 1.7GB text file of Wikipedia articles, where each line contained a separate article. Our experiments were carried out on Beocat, utilizing three different parallel processing technologies: Pthread, OpenMP, and MPI. Each technology was tested across varying numbers of cores and threads (1, 2, 4, 8, 16, and 20) to measure the impact on performance in terms of execution time and memory usage.

The benchmarking was performed using the command-line tool hyperfine, which enabled us to calculate the average runtime by conducting 40 runs for each core/thread configuration. Additionally, we measured the memory usage by averaging results from five runs at each configuration. This rigorous approach allowed us to gather detailed insights into the scalability and efficiency of each parallel processing method under the same workload.

### Pthreads

Scalability: Demonstrated significant reduction in runtime as the number of threads increased. With just one thread, the execution time was 6.466 seconds, which improved to 1.298 seconds at 20 threads.

Memory Usage: Remained constant at 1.683 GB across all thread counts, indicating efficient memory management without additional overhead with increasing threads.

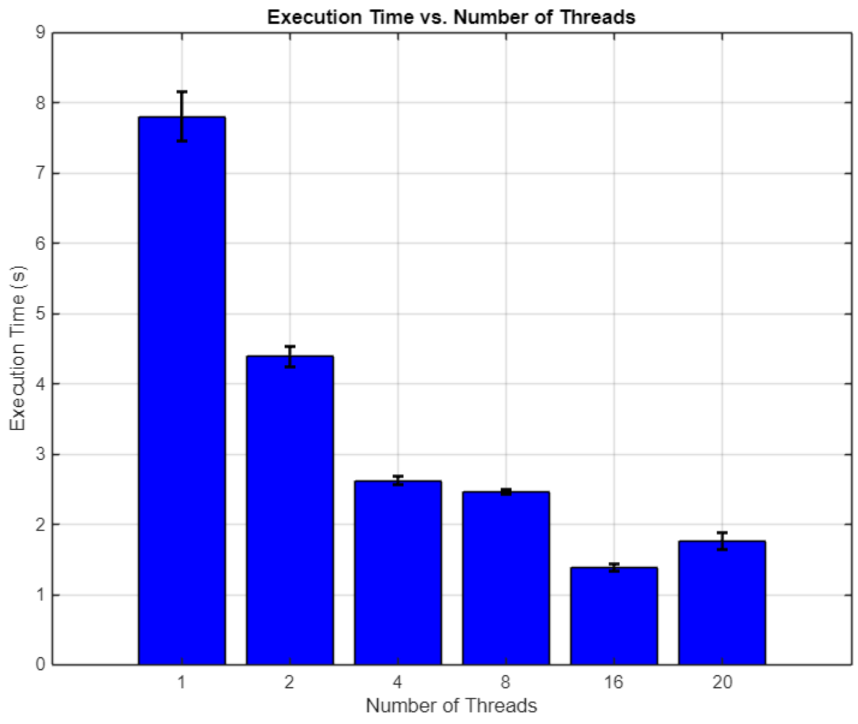


- 1 Thread: Time (mean ± σ) 6.466 s ±  0.253 s Avg. 1.683 GB
- 2 Threads: Time (mean ± σ) 4.058 s ±  0.294 s Avg. 1.683 GB
- 4 Threads: Time (mean ± σ) 2.480 s ±  0.038 s Avg. 1.683 GB
- 8 Threads: Time (mean ± σ):     1.779 s ±  0.030 s Avg. 1.683 GB
- 16 Threads: Time (mean ± σ):     1.288 s ±  0.034 s Avg. 1.683 GB
- 20 Threads: Time (mean ± σ):     1.298 s ±  0.047 s Avg. 1.683 GB

## OpenMP

Scalability: Similar to Pthread, OpenMP showed improved performance with increasing threads. However, OpenMP started slower at 7.806 seconds for one thread and improved to 1.391 seconds at 16 threads but slightly increased at 20 threads to 1.766 seconds.

Memory Usage: Consistently used 1.683 GB across all configurations, supporting the consistent overhead in memory regardless of the thread count.



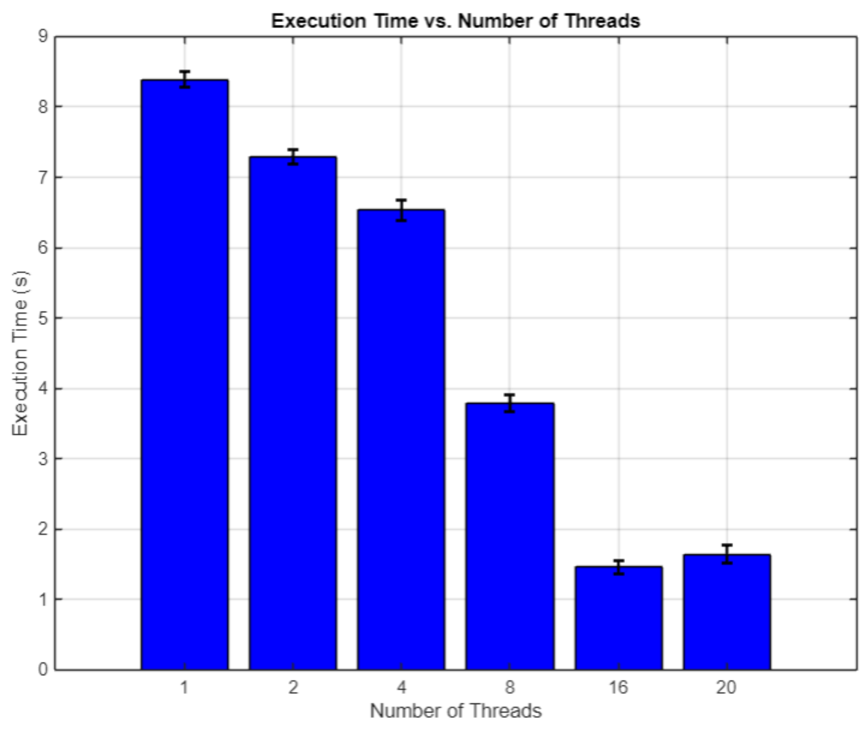
---

- 1 Thread: Time (mean ± σ): 7.806 s ± 0.350 s Avg. 1.683 GB
- 2 Threads: Time (mean ± σ): 4.388 s ± 0.153 s Avg. 1.683 GB
- 4 Threads: Time (mean ± σ): 2.619 s ± 0.061 s Avg. 1.683 GB
- 8 Threads: Time (mean ± σ): 2.465 s ± 0.037 s Avg. 1.683 GB
- 16 Threads: Time (mean ± σ): 1.391 s ± 0.050 s Avg. 1.683 GB
- 20 Threads: Time (mean ± σ): 1.766 s ± 0.121 s Avg. 1.683 GB

## MPI

Scalability: Started with the slowest single-thread performance at 8.386 seconds and achieved a notable reduction to 1.462 seconds at 16 threads. However, performance degradation was observed when moving from 16 to 20 threads.

Memory Usage: Maintained a constant memory usage of 1.683 GB, showing that MPI handled memory similarly to Pthread and OpenMP despite different communication mechanisms between threads.



- 1 Thread: Time (mean ± σ): 8.386 s ± 0.112 s Avg. 1.683 GB
- 2 Threads: Time (mean ± σ): 7.292 s ± 0.101 s Avg. 1.683 GB
- 4 Threads: Time (mean ± σ): 6.538 s ± 0.144 s Avg. 1.683 GB
- 8 Threads: Time (mean ± σ): 3.792 s ± 0.114 s Avg. 1.683 GB
- 16 Threads: Time (mean ± σ): 1.462 s ± 0.098 s Avg. 1.683 GB
- 20 Threads: Time (mean ± σ): 1.647 s ± 0.126 s Avg. 1.683 GB

## Conclusions

Performance Efficiency: Pthread and OpenMP exhibited strong scalability with increasing threads, though OpenMP showed slightly less performance at higher thread counts compared to Pthread. MPI, while starting slower, caught up significantly at higher thread counts but showed signs of inefficiency at the highest count tested.

Memory Management: All three technologies demonstrated stable memory usage across different thread counts, which is indicative of efficient memory management that does not scale with the number of threads. This stability is crucial for applications with fixed memory constraints.

Our comparative analysis highlights the strengths and limitations of Pthread, OpenMP, and MPI in handling large-scale text data. Pthread and OpenMP are more suitable for environments where gradual scaling and resource management are critical. MPI, while effective at scaling, may require careful tuning to optimize performance at very high thread counts. Future work should explore the impact of different data sizes and more complex computational tasks to fully understand the scalability and efficiency of these technologies.

### Recommendations

Further tests with varying data sizes and computational complexity could provide deeper insights into the practical applications of each technology. Investigation into the cause of performance degradation at higher thread counts in MPI could lead to optimizations that enhance its scalability and efficiency.

## Appendix

### Code

scorecard-pthread.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <pthread.h>
#include <stdbool.h>

#define THREAD_RESULTS_START_SIZE 5000
#define LINE_STRING_SIZE 13

// Struct to store thread info
typedef struct {
    int index; // used to keep track of which thread has done work
    int start; // where the thread should start reading from the buffer
    int end; // where thre thread should stop reading from the buffer
    int bufferLength; // the length of the buffer to read from
    int linesCounted; // stores how many lines the thread counted
    int resultsSize; // initial allocation size of the results buffer
    char *buffer; // pointer to the buffer to read from
    char *results; // results buffer stores the highest ASCII value from each line the thread reads from
    bool allocFail; // whether allocation of the results buffer failed for the thread
} thread_info_t;


/// @brief Counts the highest ASCII value in each line of a string between the given indices, starting at
the next newline.
/// To be used with pthreads.
/// @param args Expects a pointer to a thread_info_t struct which stores a buffer to read lines from and
other
/// information needed to know where to start and stop reading from.
/// @return NULL
void *readLines(void *args) {
    thread_info_t *data = (thread_info_t *)args;
    data->results = calloc(data->resultsSize, sizeof(char)); // allocates results
    if (data->results == NULL) {
        data->allocFail = true;
        return NULL;
    }
    data->allocFail = false;

    // Seeks the correct start location
    if (data->index != 0) {
        while (data->buffer[data->start - 1] != '\n') {
            data->start++;
        }
    }

    // Seeks the correct end location
    while (data->buffer[data->end-1] != '\n' && data->end <= data->bufferLength) {
        data->end++;
    }

    char ch = 0; // used to keep track of highest ASCII value per line

    // Loop to look for the highest ASCII value per line
    for (int i = data->start; i < data->end; i++) {

        // Checks if the results buffer needs more memory allocated
        if (data->linesCounted == data->resultsSize) {
            data->resultsSize *= 2;
            data->results = realloc(data->results, data->resultsSize); // reallocated results buffer if
needed
            if (data->results == NULL) {
                data->allocFail = true;
                return NULL;
            }
        }

        // If newline, increment linesCounted and store the highest ASCII value,
        // otherwise check if the current char's value is greater than the highest value encountered so far.
        if (data->buffer[i] == '\n') {
            data->results[data->linesCounted] = ch;
            ch = 0;
            data->linesCounted++;
        }
        else if (data->buffer[i] > ch) {
            ch = data->buffer[i];
        }
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    // Error checking for correct number of arguments
    if (argc < 2) {
        printf("Must give a filename and number of threads to use\n");
        return -1;
    }
    else if (argc < 3) {
        printf("Must give number of threads to use\n");
        return -1;
    }

    int numThreads;
    int divide;
    int lineCount = 0;
    int maxValuesIndex = 0;
    char *results = NULL;
    char *buffer = NULL;

    // Get number of threads requested by user
    sscanf(argv[2], "%d", &numThreads);
    if (numThreads < 1) {
        printf("Cannot use less than 1 thread\n");
        return 1;
    }

    thread_info_t thread_info[numThreads];
    pthread_t threads[numThreads];

    // Get stats of the given file, needed to see how big the file is
    struct stat stats;
    if (stat(argv[1], &stats) == -1) {
        perror("stat");
        return -1;
    }
    printf("%ld bytes read from file\n\n", stats.st_size);

    // Size of chunks that will be divided among the threads
    divide = stats.st_size / numThreads;
    // Open the given file
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        perror(argv[1]);
        return -1;
    }

    // Allocate buffer and read the entire file into it
    buffer = malloc(stats.st_size+1);
    fread(buffer, 1, stats.st_size, file);
    fclose(file);
    if (buffer[stats.st_size-1] != '\n') buffer[stats.st_size] = '\n';

    // Create threads with the correct info and run them
    for (int i = 0; i < numThreads; i++) {
        thread_info[i].bufferLength = stats.st_size;
        thread_info[i].linesCounted = 0;
        thread_info[i].index = i;
        thread_info[i].start = i * divide;
        if (i < numThreads - 1) thread_info[i].end = divide + thread_info[i].start;
        else thread_info[i].end = stats.st_size;
        thread_info[i].buffer = buffer;
        thread_info[i].resultsSize = THREAD_RESULTS_START_SIZE;
        if (pthread_create(&threads[i], NULL, readLines, &(thread_info[i]))) {
            perror("Thread");
            return 1;
        }
    }

    // "Joins" all extra threads to main thread, causing main thread to wait on them.
    // After each thread finishes, max ASCII values are updated.
    for (int i = 0; i < numThreads; i++) {
        if (pthread_join(threads[i], NULL)) {
            perror("Thread");
            return 1;
        }

        if (thread_info[i].allocFail) {
            printf("Could not allocate thread %d results", i);
            return 1;
        }

        // Allocates results buffer based on lineCount
        results = realloc(results, (lineCount + thread_info[i].linesCounted) * LINE_STRING_SIZE);
        if (results == NULL) {
            printf("Could not allocate results");
            return 1;
        }

        // Gets the results from each thread and formats them into a string to eventually print
        for (int j = 0; j < thread_info[i].linesCounted; j++) {
            int index = (j + lineCount);
            maxValuesIndex += snprintf(&(results[maxValuesIndex]), LINE_STRING_SIZE, "%d: %d\n", index,
thread_info[i].results[j]);
        }
        lineCount += thread_info[i].linesCounted;
    }

    printf("%s", results); // Print results

    free(buffer);
    free(results);
    for (int i = 0; i < numThreads; i++) {
        free(thread_info[i].results);
    }
}
```

scorecard-mpi.c

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <pthread.h>
#include <stdbool.h>

#define THREAD_RESULTS_START_SIZE 5000
#define LINE_STRING_SIZE 13

// Struct to store thread info
typedef struct {
    int index; // used to keep track of which thread has done work
    int start; // where the thread should start reading from the buffer
    int end; // where thre thread should stop reading from the buffer
    int bufferLength; // the length of the buffer to read from
    int linesCounted; // stores how many lines the thread counted
    int resultsSize; // initial allocation size of the results buffer
    char *buffer; // pointer to the buffer to read from
    char *results; // results buffer stores the highest ASCII value from each line the thread reads from
    int allocFail; // whether allocation of the results buffer failed for the thread
} thread_info_t;

void *readLines(void *args) {
    thread_info_t *data = (thread_info_t *)args;
    data->results = calloc(data->resultsSize, sizeof(char)); // allocates results
    if (data->results == NULL) {
        data->allocFail = 1;
        return NULL;
    }
    data->allocFail = 0;

    // Seeks the correct start location
    if (data->index != 0) {
        while (data->buffer[data->start - 1] != '\n') {
            data->start++;
        }
    }

    // Seeks the correct end location
    while (data->buffer[data->end-1] != '\n' && data->end <= data->bufferLength) {
        data->end++;
    }

    char ch = 0; // used to keep track of highest ASCII value per line

    // Loop to look for the highest ASCII value per line
    for (int i = data->start; i < data->end; i++) {

        // Checks if the results buffer needs more memory allocated
        if (data->linesCounted == data->resultsSize) {
            data->resultsSize *= 2;
            data->results = realloc(data->results, data->resultsSize); // reallocated results buffer if
needed
            if (data->results == NULL) {
                data->allocFail = true;
                return NULL;
            }
        }

        // If newline, increment linesCounted and store the highest ASCII value,
        // otherwise check if the current char's value is greater than the highest value encountered so far.
        if (data->buffer[i] == '\n') {
            data->results[data->linesCounted] = ch;
            ch = 0;
            data->linesCounted++;
        }
        else if (data->buffer[i] > ch) {
            ch = data->buffer[i];
        }
    }
    return NULL;
}


int main(int argc, char *argv[]) {

    int numThreads, divide;
    int lineCount = 0;
    int resultsIndex = 0;
    char *fullResults = NULL;
    thread_info_t thread_info;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numThreads);
    MPI_Comm_rank(MPI_COMM_WORLD, &(thread_info.index));

    // Error checking for correct number of arguments
    if (argc < 2) {
        if (thread_info.index == 0) {
            printf("Must give a filename to use\n");
        }
        return -1;
    }

    // Get file stats
    struct stat stats;
    if (stat(argv[1], &stats) == -1) {
        perror("stat");
        return -1;
    }
    thread_info.buffer = malloc(stats.st_size + 1);

    // Size of chunks that will be divided among the threads
    thread_info.bufferLength = stats.st_size;
    divide = thread_info.bufferLength / numThreads;

    if (thread_info.index == 0) {

        // Open the given file
        FILE *file = fopen(argv[1], "r");
        if (file == NULL) {
            perror(argv[1]);
            return -1;
        }

        // Read the file into a buffer
        if (fread(thread_info.buffer, 1, stats.st_size, file) < 1) {
            printf("Unable to read file: %s\n", argv[1]);
            return 1;
        }
        fclose(file);
        if (thread_info.buffer[stats.st_size-1] != '\n') thread_info.buffer[stats.st_size] = '\n';
    }

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(thread_info.buffer, stats.st_size + 1, MPI_CHAR, 0, MPI_COMM_WORLD);

    // Fill the thread_info struct with info for the readLines method
    thread_info.resultsSize = THREAD_RESULTS_START_SIZE;
    thread_info.linesCounted = 0;
    thread_info.start = thread_info.index * divide;
    if (thread_info.index < numThreads - 1) thread_info.end = divide + thread_info.start;
    else thread_info.end = thread_info.bufferLength;

    readLines(&thread_info);

    // If this is not the first thread, receive the total line count so far from the previous thread
    if (numThreads > 1 && thread_info.index > 0) {
        MPI_Recv(&lineCount, 1, MPI_INT, thread_info.index - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    // Check for allocation failure
    if (thread_info.allocFail) {
        printf("Could not allocate thread %d initial results", thread_info.index);
        return 1;
    }

    // Allocate buffer for results and check for failure
    fullResults = malloc((thread_info.linesCounted) * LINE_STRING_SIZE);
    if (fullResults == NULL) {
        printf("Could not allocate thread %d full results", thread_info.index);
        return 1;
    }

    // Create the full results string for this thread
    int index = lineCount;
    for (int i = 0; i < thread_info.linesCounted; i++) {
        resultsIndex += snprintf(&(fullResults[resultsIndex]), LINE_STRING_SIZE, "%d: %d\n", index,
thread_info.results[i]);
        index++;
    }

    printf("%s", fullResults); // Print results for this thread

    // If this is not the last thread, send the line count so far to the next thread
    if (numThreads > 1 && thread_info.index < numThreads - 1) {
        int totalLineCount = thread_info.linesCounted + lineCount;
        MPI_Send(&totalLineCount, 1, MPI_INT, thread_info.index + 1, 0, MPI_COMM_WORLD);
    }

    free(fullResults);
    free(thread_info.buffer);
    free(thread_info.results);
    MPI_Finalize();
}
```

scorecard-openmp.c

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <stdbool.h>

#define THREAD_RESULTS_START_SIZE 5000
#define LINE_STRING_SIZE 13

typedef struct {
    int index; // used to keep track of which thread has done work
    int start; // where the thread should start reading from the buffer
    int end; // where thre thread should stop reading from the buffer
    int bufferLength; // the length of the buffer to read from
    int linesCounted; // stores how many lines the thread counted
    int resultsSize; // initial allocation size of the results buffer
    char *buffer; // pointer to the buffer to read from
    char *results; // results buffer stores the highest ASCII value from each line the thread reads from
    int allocFail; // whether allocation of the results buffer failed for the thread
} thread_info_t;

void *readLines(void *args) {
    thread_info_t *data = (thread_info_t *)args;
    data->results = calloc(data->resultsSize, sizeof(char)); // allocates results
    if (data->results == NULL) {
        data->allocFail = 1;
        return NULL;
    }
    data->allocFail = 0;

    // Seeks the correct start location
    if (data->index != 0) {
        while (data->buffer[data->start - 1] != '\n') {
            data->start++;
        }
    }

    // Seeks the correct end location
    while (data->buffer[data->end-1] != '\n' && data->end <= data->bufferLength) {
        data->end++;
    }

    char ch = 0; // used to keep track of highest ASCII value per line

    // Loop to look for the highest ASCII value per line
    for (int i = data->start; i < data->end; i++) {

        // Checks if the results buffer needs more memory allocated
        if (data->linesCounted == data->resultsSize) {
            data->resultsSize *= 2;
            data->results = realloc(data->results, data->resultsSize); // reallocated results buffer if
needed
```

```c
            if (data->results == NULL) {
                data->allocFail = true;
                return NULL;
            }
        }

        // If newline, increment linesCounted and store the highest ASCII value,
        // otherwise check if the current char's value is greater than the highest value encountered so far.
        if (data->buffer[i] == '\n') {
            data->results[data->linesCounted] = ch;
            ch = 0;
            data->linesCounted++;
        }
        else if (data->buffer[i] > ch) {
            ch = data->buffer[i];
        }
    }
    return NULL;
}


int main(int argc, char *argv[]) {
    // Error checking for correct number of arguments
    if (argc < 2) {
        printf("Must give a filename and number of threads to use\n");
        return -1;
    }
    else if (argc < 3) {
        printf("Must give number of threads to use\n");
        return -1;
    }

    int numThreads;
    int divide;
    int lineCount = 0;
    int maxValuesIndex = 0;
    char *results = NULL;
    char *buffer = NULL;


    // Get number of threads requested by user
    sscanf(argv[2], "%d", &numThreads);
    if (numThreads < 1) {
        printf("Cannot use less than 1 thread\n");
        return 1;
    }
    omp_set_num_threads(numThreads); // Set the number of threads
    thread_info_t thread_info[numThreads];
    struct stat stats;
    if (stat(argv[1], &stats) == -1) {
        perror("stat");
        return -1;
    }
    printf("%ld bytes read from file\n\n", stats.st_size);

    // Size of chunks that will be divided among the threads
    divide = stats.st_size / numThreads;

    // Open the given file
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        perror(argv[1]);
        return -1;
    }

    // Allocate buffer and read the entire file into it
    buffer = malloc(stats.st_size+1);
    fread(buffer, 1, stats.st_size, file);
    fclose(file);
    if (buffer[stats.st_size-1] != '\n') buffer[stats.st_size] = '\n';

    // Create threads with the correct info and run them
    for (int i = 0; i < numThreads; i++) {
        thread_info[i].bufferLength = stats.st_size;
        thread_info[i].linesCounted = 0;
        thread_info[i].index = i;
        thread_info[i].start = i * divide;
        if (i < numThreads - 1) thread_info[i].end = divide + thread_info[i].start;
        else thread_info[i].end = stats.st_size;
        thread_info[i].buffer = buffer;
        thread_info[i].resultsSize = THREAD_RESULTS_START_SIZE;
    }

#pragma omp parallel
{
    int threadNum = omp_get_thread_num();
    readLines(&(thread_info[threadNum]));
    //printf("printing from thread: %d\n", threadNum);

}

for (int i = 0; i < numThreads; i++) {

        if (thread_info[i].allocFail) {
            printf("Could not allocate thread %d results", i);
            return 1;
        }

        // Allocates results buffer based on lineCount
        results = realloc(results, (lineCount + thread_info[i].linesCounted) * LINE_STRING_SIZE);
        if (results == NULL) {
            printf("Could not allocate results");
            return 1;
        }

        // Gets the results from each thread and formats them into a string to eventually print
        for (int j = 0; j < thread_info[i].linesCounted; j++) {
            int index = (j + lineCount);
            maxValuesIndex += snprintf(&(results[maxValuesIndex]), LINE_STRING_SIZE, "%d: %d\n", index,
thread_info[i].results[j]);
        }
        lineCount += thread_info[i].linesCounted;
    }

    printf("%s", results); // Print results

    free(buffer);
    free(results);
    for (int i = 0; i < numThreads; i++) {
        free(thread_info[i].results);
    }
}
```

## Scripts

There are versions of these scripts for 1, 2, 4, 8, 16, 20, and 40 threads for each implementation. Shown is the 4thread script for each one.

pthread 4thread.sh

```
#!/bin/bash
#SBATCH --job-name=4thread-pthread
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --mem-per-core=1G
#SBATCH --time=00:15:00
#SBATCH --constraint=moles
#SBATCH --output=results/4thread.out

../../hyperfine '../build/scorecard-pthread ~dan/625/wiki_dump.txt 4' --warmup 2 --runs 50 --export-json
results/4thread.json

/usr/bin/time -f 'Run 1: %M Bytes Used' ../build/scorecard-pthread ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
/usr/bin/time -f 'Run 2: %M Bytes Used' ../build/scorecard-pthread ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
/usr/bin/time -f 'Run 3: %M Bytes Used' ../build/scorecard-pthread ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
/usr/bin/time -f 'Run 4: %M Bytes Used' ../build/scorecard-pthread ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
/usr/bin/time -f 'Run 5: %M Bytes Used' ../build/scorecard-pthread ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
```

mpi 4thread.sh

```
#!/bin/bash
#SBATCH --job-name=4thread-mpi
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --mem-per-core=2G
#SBATCH --time=00:30:00
#SBATCH --constraint=moles
#SBATCH --output=results/4thread.out

../../hyperfine 'mpirun -n 4 ../build/scorecard-mpi ~dan/625/wiki_dump.txt' --warmup 2 --runs 20 --export-
json results/4thread.json

/usr/bin/time -f 'Run 1: %M Bytes Used' mpirun -n 4 ../build/scorecard-mpi ~dan/625/wiki_dump.txt | grep
'Bytes Used'
/usr/bin/time -f 'Run 2: %M Bytes Used' mpirun -n 4 ../build/scorecard-mpi ~dan/625/wiki_dump.txt | grep
'Bytes Used'
/usr/bin/time -f 'Run 3: %M Bytes Used' mpirun -n 4 ../build/scorecard-mpi ~dan/625/wiki_dump.txt | grep
'Bytes Used'
/usr/bin/time -f 'Run 4: %M Bytes Used' mpirun -n 4 ../build/scorecard-mpi ~dan/625/wiki_dump.txt | grep
'Bytes Used'
/usr/bin/time -f 'Run 5: %M Bytes Used' mpirun -n 4 ../build/scorecard-mpi ~dan/625/wiki_dump.txt | grep
'Bytes Used'
```

openmp 4thread.sh

```
#!/bin/bash
#SBATCH --job-name=4thread-openmp
#SBATCH --nodes=1
```

```
#SBATCH --ntasks=4
#SBATCH --mem-per-core=1G
#SBATCH --time=00:15:00
#SBATCH --constraint=moles
#SBATCH --output=results/4thread.out

../../hyperfine '../build/scorecard-openmp ~dan/625/wiki_dump.txt 4' --warmup 2 --runs 50 --export-json
results/4thread.json

/usr/bin/time -f 'Run 1: %M Bytes Used' ../build/scorecard-openmp ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
/usr/bin/time -f 'Run 2: %M Bytes Used' ../build/scorecard-openmp ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
/usr/bin/time -f 'Run 3: %M Bytes Used' ../build/scorecard-openmp ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
/usr/bin/time -f 'Run 4: %M Bytes Used' ../build/scorecard-openmp ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
/usr/bin/time -f 'Run 5: %M Bytes Used' ../build/scorecard-openmp ~dan/625/wiki_dump.txt 4 | grep 'Bytes
Used'
```

Sample Output

```
999899: 125
999900: 126
999901: 125
999902: 125
999903: 125
999904: 125
999905: 125
999906: 125
999907: 125
999908: 125
999909: 125
999910: 125
999911: 125
999912: 125
999913: 125
999914: 125
999915: 125
999916: 125
999917: 125
999918: 124
999919: 125
999920: 125
999921: 125
999922: 125
999923: 125
999924: 125
999925: 125
999926: 124
999927: 125
999928: 125
999929: 125
999930: 125
999931: 125
999932: 125
```

```
999933: 125
999934: 125
999935: 125
999936: 125
999937: 125
999938: 125
999939: 125
999940: 125
999941: 125
999942: 125
999943: 125
999944: 125
999945: 125
999946: 125
999947: 124
999948: 125
999949: 125
999950: 125
999951: 125
999952: 125
999953: 125
999954: 125
999955: 125
999956: 125
999957: 125
999958: 125
999959: 125
999960: 125
999961: 125
999962: 125
999963: 125
999964: 125
999965: 125
999966: 125
999967: 125
999968: 125
999969: 125
999970: 125
999971: 125
999972: 125
999973: 125
999974: 125
999975: 125
999976: 124
999977: 125
999978: 124
999979: 125
999980: 125
999981: 125
999982: 125
999983: 125
999984: 125
999985: 125
999986: 125
```

```
999987: 125
999988: 125
999989: 125
999990: 125
999991: 125
999992: 125
999993: 125
999994: 124
999995: 125
999996: 125
999997: 125
999998: 125
999999: 125
```