**Name:** Boston Karymshakov

**Course:** CMPSC 472

**Date:** 10/28/2025

**Project Report: MapReduce Systems for Parallel Sorting and Max-Value Aggregation**

**Overview**

This project was about building two MapReduce-style programs that work on a single machine using C. The first part does parallel sorting, and the second part finds the maximum value from a dataset while using shared memory with limited space. Instead of using a distributed system like Hadoop, we simulate MapReduce using threads and processes to understand parallelism, synchronization, and inter-process **communication.**

**Part 1: Parallel Sorting**

In this task, I implemented sorting in two ways using multithreading and multiprocessing.

For the threading version, I divided the array into chunks and gave each thread a part to sort with qsort. After the threads finished, the reducer merged all the sorted chunks together to get one final sorted array. I used the pthread library for thread creation and joining.

For the multiprocessing version, I used fork to create separate worker processes. Each process also sorted a chunk of the array, and I used pipes for inter-process communication. Each worker sent its sorted results to the parent process, which acted as the reducer and merged all the results.

I measured execution times for different numbers of workers like 1, 2, 4, and 8, and compared them to see how parallelization improved performance.

**Part 2: Max-Value Aggregation with Shared Memory**

The second part was about finding the maximum value in a dataset using shared memory but with a constraint that only one integer could be stored there.

In the threading version, each thread found the local maximum of its part, and then all threads tried to update the global maximum. I used a mutex lock to make sure that only one thread updates the shared value at a time to avoid race conditions.

For the multiprocessing version, I used shared memory with mmap to store the global max so that all processes could access it. Each process compared its local maximum with the global one and updated it if necessary.

This part helped me understand how synchronization and communication between processes are needed when multiple threads or processes try to write to the same shared memory space.

**Source Code:**

%%writefile mapreduce\_project.c

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <sys/wait.h>

#include <unistd.h>

#include <sys/mman.h>

#include <string.h>

#include <sys/time.h>

// helper function for timing

double get\_time\_sec() {

struct timeval t;

gettimeofday(&t, NULL);

return t.tv\_sec + t.tv\_usec / 1e6;

}

// comparison function for sorting

int cmpfunc(const void \*a, const void \*b) {

return (\*(int\*)a - \*(int\*)b);

}

// generate random array

int\* generate\_array(int n) {

int \*arr = malloc(n \* sizeof(int));

for (int i = 0; i < n; i++)

arr[i] = rand() % 1000000;

return arr;

}

// check if array is sorted

int is\_sorted(int \*arr, int n) {

for (int i = 0; i < n - 1; i++)

if (arr[i] > arr[i + 1]) return 0;

return 1;

}

// merge two sorted arrays

int\* merge\_two(int \*a, int na, int \*b, int nb) {

int \*merged = malloc((na + nb) \* sizeof(int));

int i = 0, j = 0, k = 0;

while (i < na && j < nb) {

if (a[i] <= b[j]) merged[k++] = a[i++];

else merged[k++] = b[j++];

}

while (i < na) merged[k++] = a[i++];

while (j < nb) merged[k++] = b[j++];

return merged;

}

// struct for thread sort arguments

typedef struct {

int \*arr;

int n;

} ThreadArg;

// thread function for sorting

void\* thread\_sort\_func(void \*arg) {

ThreadArg \*ta = (ThreadArg\*)arg;

qsort(ta->arr, ta->n, sizeof(int), cmpfunc);

return NULL;

}

// sorting using threads

void thread\_sort(int \*arr, int n, int workers) {

int chunk = n / workers;

pthread\_t tids[workers];

ThreadArg args[workers];

double t1 = get\_time\_sec();

// start threads

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

args[i].arr = arr + i \* chunk;

args[i].n = (i == workers - 1) ? n - i \* chunk : chunk;

pthread\_create(&tids[i], NULL, thread\_sort\_func, &args[i]);

}

// wait for all threads

for (int i = 0; i < workers; i++) pthread\_join(tids[i], NULL);

// merge all sorted parts

int \*merged = arr;

int total = chunk;

for (int i = 1; i < workers; i++) {

int size = (i == workers - 1) ? n - i \* chunk : chunk;

int \*newmerged = merge\_two(merged, total, arr + i \* chunk, size);

merged = newmerged;

total += size;

}

double t2 = get\_time\_sec();

printf("[THREAD\_SORT] n=%d workers=%d time=%.6fs sorted=%d\n",

n, workers, t2 - t1, is\_sorted(merged, n));

free(merged);

}

// sorting using processes

void process\_sort(int \*arr, int n, int workers) {

int chunk = n / workers;

int pipes[workers][2];

double t1 = get\_time\_sec();

// make pipes for IPC

for (int i = 0; i < workers; i++) pipe(pipes[i]);

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

pid\_t pid = fork();

if (pid == 0) {

close(pipes[i][0]);

int start = i \* chunk;

int len = (i == workers - 1) ? n - start : chunk;

qsort(arr + start, len, sizeof(int), cmpfunc);

write(pipes[i][1], arr + start, len \* sizeof(int));

close(pipes[i][1]);

exit(0);

}

}

int \*sorted = malloc(n \* sizeof(int));

int offset = 0;

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

close(pipes[i][1]);

int len = (i == workers - 1) ? n - i \* chunk : chunk;

read(pipes[i][0], sorted + offset, len \* sizeof(int));

offset += len;

close(pipes[i][0]);

}

// wait for children

for (int i = 0; i < workers; i++) wait(NULL);

// merge all parts

int \*merged = sorted;

int total = chunk;

for (int i = 1; i < workers; i++) {

int size = (i == workers - 1) ? n - i \* chunk : chunk;

int \*newmerged = merge\_two(merged, total, sorted + i \* chunk, size);

merged = newmerged;

total += size;

}

double t2 = get\_time\_sec();

printf("[PROC\_SORT] n=%d workers=%d time=%.6fs sorted=%d\n",

n, workers, t2 - t1, is\_sorted(merged, n));

free(sorted);

free(merged);

}

// struct for max value threads

typedef struct {

int \*arr;

int n;

int \*shared;

pthread\_mutex\_t \*lock;

} MaxArg;

// thread function for finding max

void\* thread\_max\_func(void \*arg) {

MaxArg \*ma = (MaxArg\*)arg;

int local\_max = ma->arr[0];

for (int i = 1; i < ma->n; i++)

if (ma->arr[i] > local\_max) local\_max = ma->arr[i];

pthread\_mutex\_lock(ma->lock);

if (local\_max > \*(ma->shared))

\*(ma->shared) = local\_max;

pthread\_mutex\_unlock(ma->lock);

return NULL;

}

// finding max using threads

void thread\_max(int \*arr, int n, int workers) {

int chunk = n / workers;

pthread\_t tids[workers];

pthread\_mutex\_t lock;

pthread\_mutex\_init(&lock, NULL);

int \*shared = malloc(sizeof(int));

\*shared = -1;

double t1 = get\_time\_sec();

MaxArg args[workers];

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

args[i].arr = arr + i \* chunk;

args[i].n = (i == workers - 1) ? n - i \* chunk : chunk;

args[i].shared = shared;

args[i].lock = &lock;

pthread\_create(&tids[i], NULL, thread\_max\_func, &args[i]);

}

for (int i = 0; i < workers; i++) pthread\_join(tids[i], NULL);

double t2 = get\_time\_sec();

printf("[THREAD\_MAX] n=%d workers=%d time=%.6fs global\_max=%d\n",

n, workers, t2 - t1, \*shared);

free(shared);

}

// finding max using processes

void process\_max(int \*arr, int n, int workers) {

int chunk = n / workers;

int \*shared = mmap(NULL, sizeof(int), PROT\_READ | PROT\_WRITE,

MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

\*shared = -1;

double t1 = get\_time\_sec();

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

pid\_t pid = fork();

if (pid == 0) {

int start = i \* chunk;

int len = (i == workers - 1) ? n - start : chunk;

int local\_max = arr[start];

for (int j = 1; j < len; j++)

if (arr[start + j] > local\_max) local\_max = arr[start + j];

if (local\_max > \*shared) \*shared = local\_max;

exit(0);

}

}

for (int i = 0; i < workers; i++) wait(NULL);

double t2 = get\_time\_sec();

printf("[PROC\_MAX] n=%d workers=%d time=%.6fs global\_max=%d\n",

n, workers, t2 - t1, \*shared);

munmap(shared, sizeof(int));

}

// main function

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

if (argc < 4) {

printf("Usage: %s [thread\_sort|process\_sort|thread\_max|process\_max] <n> <workers>\n", argv[0]);

return 1;

}

char \*mode = argv[1];

int n = atoi(argv[2]);

int workers = atoi(argv[3]);

srand(42);

int \*arr = generate\_array(n);

if (strcmp(mode, "thread\_sort") == 0)

thread\_sort(arr, n, workers);

else if (strcmp(mode, "process\_sort") == 0)

process\_sort(arr, n, workers);

else if (strcmp(mode, "thread\_max") == 0)

thread\_max(arr, n, workers);

else if (strcmp(mode, "process\_max") == 0)

process\_max(arr, n, workers);

else

printf("Invalid mode.\n");

free(arr);

return 0;

}

**Results:**

A computer screen with white text

AI-generated content may be incorrect.

**Code Structure**

* thread\_sort handles sorting using threads. Each thread sorts a portion of the array
* process\_sort handles sorting using processes and pipes for communication
* thread\_max finds the maximum value using threads and a shared integer with a mutex lock
* process\_max finds the maximum using processes and shared memory with synchronization

The reducer is responsible for merging or finalizing results from all the workers.

**Performance Testing**

I tested the code with both small inputs like 32 elements and large ones like 131072 elements. For smaller inputs, the speed difference between 1 and 8 workers wasn’t much, but for larger inputs, more workers significantly reduced execution time.

The multiprocessing version took a bit longer than multithreading because of process creation overhead and communication. Threads share the same memory space, which made communication faster.

When testing the max-value part, I noticed that without synchronization, the shared variable sometimes gave wrong results. After adding the mutex and proper shared memory handling, the program worked correctly every time.

**Conclusion**

This project helped me understand how MapReduce can be simulated with threads and processes. I learned how to manage threads, use synchronization tools like mutexes, and work with communication methods such as pipes and shared memory.

Multithreading was usually faster because it had less overhead, while multiprocessing gave better isolation. The biggest challenge was handling synchronization and merging results efficiently. Overall, this project gave me good hands-on experience with parallelism and process management concepts in operating systems.