# Submission Report

* Submission generated at 10/24/2025 at 22:21:32
* Machine info: Linux runnervmwhb2z 6.11.0-1018-azure #18~24.04.1-Ubuntu SMP Sat Jun 28 04:46:03 UTC 2025 x86\_64 x86\_64 x86\_64 GNU/Linux

## Note to Students

Please read this report carefully before submission. Ensure that all sections are complete and accurate. Look for any errors in the build or test outputs. If you find any issues, correct them before submitting. Post any questions on the class discussion board for help.

## README

# Project 3

* Name: Jon Flores
* Email: jonathanflores@u.boisestate.edu
* Class: 452-001

## Known Bugs or Issues

None that I am aware of

## Experience

As I progress through this course, I am slowly but surely becoming more comfortable with the C programming language again. I was very excited to tackle this project after discussing it in class with Dr. Panter, so I had an architecture already in mind when sitting down to code. As such, I started by laying out a skeleton of what I had envisioned, then started to code it up by hand. Finally, I checked all the files through AI to check for validity in terms of syntax, made a few minor adjustments, and then repeated this process for compiler errors and warnings as they began to appear. Eventually I had a working code with minimal input from AI this time, which felt good. I had a little trouble ensuring that I used a lock in this assignment to meet the assignment criteria, as my approach did not necessarily require one, but I did get one in there.

## Analysis

* Were you able to generate soething close to what the example showed?
  + Yes, but on onyx, I had to greatly increase the size of the arrays to be sorted to generate something close to the example plot. The best results I saw were for 10000000 element arrays, but after some discussion with Dr. Panter, he confirmed that this was acceptable.
* Did you see a slowdown at some point? Why or why not?
  + Oddly enough, I did see some slow down on 2 threads, most likely due to some overhead introduced in my methodology, but everything after that followed a pretty smooth pattern of decreased run time, with a very slight upwards trend as the number of threads approached the number of cores available. I have adjusted the generateplot.sh script to go up to my max thread count to further demonstrate this trend on onxy.
* Did your program run faster and faster when you added more threads? Why or Why not?
  + My program did run faster and faster for a good while when adding more threads, but eventually it stabilized, then, as the number of threads approached the number of cores, there was a slight upward trend appearing again.
* What was the optimum number of threads for your machine?
  + On Onyx, it seems like the optimal number of threads was 45.
* What was the slowest number of threads for your machine?
  + On Onyx, the slowest number of threads still seemed to be 2. This is most likely due to some overhead my method introduced.

Here is an example of how to include a plot in your README:

## Onyx Plot: 64 thread max, 10000000 element arrays

## Build Output

This section was generated by running make all in the project root directory.

make[1]: Entering directory '/home/runner/work/Fall2025CS452P3/Fall2025CS452P3'  
mkdir -p build/debug  
cc -g -O0 -DDEBUG -fno-omit-frame-pointer -fsanitize=address -c src/lab.c -o build/debug/lab.c.o  
mkdir -p build/debug  
cc -g -O0 -DDEBUG -fno-omit-frame-pointer -fsanitize=address -c src/main.c -o build/debug/main.c.o  
cc -g -O0 -DDEBUG -fno-omit-frame-pointer -fsanitize=address build/debug/lab.c.o build/debug/main.c.o -o build/debug/myapp\_d -fsanitize=address  
make[1]: Leaving directory '/home/runner/work/Fall2025CS452P3/Fall2025CS452P3'  
make[1]: Entering directory '/home/runner/work/Fall2025CS452P3/Fall2025CS452P3'  
mkdir -p build/release  
cc -Wall -Wextra -O2 -fPIE -MMD -MP -Wformat -Wformat=2 -Wconversion -Wsign-conversion -Wimplicit-fallthrough -fstack-protector-strong -Werror=format-security -Werror=implicit -Werror=incompatible-pointer-types -Werror=int-conversion -c src/lab.c -o build/release/lab.c.o  
mkdir -p build/release  
cc -Wall -Wextra -O2 -fPIE -MMD -MP -Wformat -Wformat=2 -Wconversion -Wsign-conversion -Wimplicit-fallthrough -fstack-protector-strong -Werror=format-security -Werror=implicit -Werror=incompatible-pointer-types -Werror=int-conversion -c src/main.c -o build/release/main.c.o  
cc -Wall -Wextra -O2 -fPIE -MMD -MP -Wformat -Wformat=2 -Wconversion -Wsign-conversion -Wimplicit-fallthrough -fstack-protector-strong -Werror=format-security -Werror=implicit -Werror=incompatible-pointer-types -Werror=int-conversion build/release/lab.c.o build/release/main.c.o -o build/release/myapp   
make[1]: Leaving directory '/home/runner/work/Fall2025CS452P3/Fall2025CS452P3'  
make[1]: Entering directory '/home/runner/work/Fall2025CS452P3/Fall2025CS452P3'  
mkdir -p build/tests  
cc -g -O0 -DTEST -fprofile-arcs -ftest-coverage -c src/lab.c -o build/tests/lab.c.o  
mkdir -p build/tests  
cc -g -O0 -DTEST -fprofile-arcs -ftest-coverage -c src/main.c -o build/tests/main.c.o  
mkdir -p build/tests/harness/  
cc -g -O0 -DTEST -fprofile-arcs -ftest-coverage -c tests/harness/unity.c -o build/tests/harness/unity.c.o  
mkdir -p build/tests/  
cc -g -O0 -DTEST -fprofile-arcs -ftest-coverage -c tests/lab-test.c -o build/tests/lab-test.c.o  
cc -g -O0 -DTEST -fprofile-arcs -ftest-coverage build/tests/lab.c.o build/tests/main.c.o build/tests/harness/unity.c.o build/tests/lab-test.c.o -o build/tests/myapp\_t -fprofile-arcs -ftest-coverage  
make[1]: Leaving directory '/home/runner/work/Fall2025CS452P3/Fall2025CS452P3'  
make[1]: Entering directory '/home/runner/work/Fall2025CS452P3/Fall2025CS452P3'  
mkdir -p build/debug-test  
cc -g -O0 -DDEBUG -DTEST -fno-omit-frame-pointer -fsanitize=address -c src/lab.c -o build/debug-test/lab.c.o  
mkdir -p build/debug-test  
cc -g -O0 -DDEBUG -DTEST -fno-omit-frame-pointer -fsanitize=address -c src/main.c -o build/debug-test/main.c.o  
mkdir -p build/debug-test/harness/  
cc -g -O0 -DDEBUG -DTEST -fno-omit-frame-pointer -fsanitize=address -c tests/harness/unity.c -o build/debug-test/harness/unity.c.o  
mkdir -p build/debug-test/  
cc -g -O0 -DDEBUG -DTEST -fno-omit-frame-pointer -fsanitize=address -c tests/lab-test.c -o build/debug-test/lab-test.c.o  
cc -g -O0 -DDEBUG -DTEST -fno-omit-frame-pointer -fsanitize=address build/debug-test/lab.c.o build/debug-test/main.c.o build/debug-test/harness/unity.c.o build/debug-test/lab-test.c.o -o build/debug-test/myapp\_td -fsanitize=address  
make[1]: Leaving directory '/home/runner/work/Fall2025CS452P3/Fall2025CS452P3'  
Builds completed. You can run the application with: ./build/release/myapp  
You can run the debug build with: ./build/debug/myapp\_d  
You can run the test build with: ./build/tests/myapp\_t  
You can run the debug-test build with: ./build/debug-test/myapp\_td

## Coverage Report

This section was generated by running gcovr in the project root directory, with negative branch hits safely ignored.

mkdir -p ./build/report/html  
mkdir -p ./build/report/txt  
  
gcovr -r . --gcov-ignore-parse-errors=negative\_hits.warn --html --html-details --exclude-directories build/tests/harness --exclude '.\*main\.c$' --exclude '.\*test\.c$' -o ./build/report/html/coverage\_report.html 2>&1

## Address Sanitizer Report

This section was generated by running make leak-test in the project root directory.

tests/lab-test.c:161:test\_mergesort\_s:PASS  
tests/lab-test.c:162:test\_split\_sizes\_indices:PASS  
tests/lab-test.c:163:test\_parallel\_sort:PASS  
tests/lab-test.c:164:test\_edge\_cases:PASS  
Stress test: array\_size=10000 threads=1 time=6.752197  
Stress test: array\_size=10000 threads=2 time=3.784180  
Stress test: array\_size=10000 threads=3 time=2.174072  
Stress test: array\_size=10000 threads=4 time=2.855957  
Stress test: array\_size=10000 threads=5 time=2.448975  
Stress test: array\_size=10000 threads=6 time=3.406006  
Stress test: array\_size=10000 threads=7 time=3.619873  
Stress test: array\_size=10000 threads=8 time=3.950928  
Stress test: array\_size=50000 threads=1 time=22.286865  
Stress test: array\_size=50000 threads=2 time=21.600098  
Stress test: array\_size=50000 threads=3 time=10.127930  
Stress test: array\_size=50000 threads=4 time=9.300049  
Stress test: array\_size=50000 threads=5 time=9.194092  
Stress test: array\_size=50000 threads=6 time=9.307129  
Stress test: array\_size=50000 threads=7 time=9.640869  
Stress test: array\_size=50000 threads=8 time=10.314941  
Stress test: array\_size=100000 threads=1 time=40.461914  
Stress test: array\_size=100000 threads=2 time=40.768066  
Stress test: array\_size=100000 threads=3 time=20.662842  
Stress test: array\_size=100000 threads=4 time=18.666992  
Stress test: array\_size=100000 threads=5 time=17.883057  
Stress test: array\_size=100000 threads=6 time=16.906006  
Stress test: array\_size=100000 threads=7 time=18.167969  
Stress test: array\_size=100000 threads=8 time=19.106934  
tests/lab-test.c:165:test\_stress\_sort:PASS  
  
-----------------------  
5 Tests 0 Failures 0 Ignored   
OK

## Src Files

### lab.c

#include <stdlib.h>  
#include <sys/time.h>  
#include "lab.h"  
#include <pthread.h>  
  
/\*  
 \* AI use: AI Assisted  
 \*/  
typedef struct  
{  
 int \*A;  
 int start;  
 int mid;  
 int end;  
 pthread\_mutex\_t \*merge\_lock; // pointer to global lock  
} merge\_task\_t;  
  
/\*  
 \* @brief Thread function for sorting  
 \*  
 \* @param arg The argument passed to the thread  
 \* @return NULL  
 \* AI use: AI Assisted  
 \*/  
void \*thread\_sort(void \*arg)  
{  
 parallel\_args\_t \*p = (parallel\_args\_t \*)arg;  
 mergesort\_s(p->A, p->start, p->end);  
 return NULL;  
}  
  
  
  
/\*\*  
 \* @brief Worker function for merging two sorted subarrays with locking  
 \*  
 \* @param arg The argument passed to the thread  
 \* @return NULL  
 \* AI use: AI Assisted  
 \*/  
static void \*merge\_worker(void \*arg)  
{  
 merge\_task\_t \*task = (merge\_task\_t \*)arg;  
  
 // Minimal lock: protects merge operation  
 pthread\_mutex\_lock(task->merge\_lock);  
 merge\_s(task->A, task->start, task->mid, task->end);  
 pthread\_mutex\_unlock(task->merge\_lock);  
  
 return NULL;  
}  
  
  
  
/\*\*  
 \* @brief Standard insertion sort that is faster than merge sort for small array's  
 \*  
 \* @param A The array to sort  
 \* @param p The starting index  
 \* @param r The ending index  
 \*/  
static void insertion\_sort(int A[], int p, int r)  
{  
 int j;  
  
 for (j = p + 1; j <= r; j++)  
 {  
 int key = A[j];  
 int i = j - 1;  
 while ((i > p - 1) && (A[i] > key))  
 {  
 A[i + 1] = A[i];  
 i--;  
 }  
 A[i + 1] = key;  
 }  
}  
  
void mergesort\_s(int A[], int p, int r)  
{  
 if (r - p + 1 <= INSERTION\_SORT\_THRESHOLD)  
 {  
 insertion\_sort(A, p, r);  
 }  
 else  
 {  
 int q = (p + r) / 2;  
 mergesort\_s(A, p, q);  
 mergesort\_s(A, q + 1, r);  
 merge\_s(A, p, q, r);  
 }  
}  
  
void merge\_s(int A[], int p, int q, int r)  
{  
 int \*B = (int \*)malloc(sizeof(int) \* (size\_t)(r - p + 1));  
  
 int i = p;  
 int j = q + 1;  
 int k = 0;  
 int l;  
  
  
   
 /\* as long as both lists have unexamined elements \*/  
 /\* this loop keeps executing. \*/  
 while ((i <= q) && (j <= r))  
 {  
 if (A[i] < A[j])  
 {  
 B[k] = A[i];  
 i++;  
 }  
 else  
 {  
 B[k] = A[j];  
 j++;  
 }  
 k++;  
 }  
  
 /\* now only at most one list has unprocessed elements. \*/  
 if (i <= q)  
 {  
 /\* copy remaining elements from the first list \*/  
 for (l = i; l <= q; l++)  
 {  
 B[k] = A[l];  
 k++;  
 }  
 }  
 else  
 {  
 /\* copy remaining elements from the second list \*/  
 for (l = j; l <= r; l++)  
 {  
 B[k] = A[l];  
 k++;  
 }  
 }  
  
 /\* copy merged output from array B back to array A \*/  
 k = 0;  
 for (l = p; l <= r; l++)  
 {  
 A[l] = B[k];  
 k++;  
 }  
  
 free(B);  
}  
  
/\*\*  
 \* @brief Computes the sizes for each split given the total size and number of threads  
 \*  
 \* @param A The array to split  
 \* @param n The size of the array  
 \* @param num\_threads The number of threads  
 \* @return int\* An array of sizes for each split  
 \* AI use: AI Assisted  
 \*/  
int \*split\_sizes(int n, int num\_threads)  
{  
 int working\_threads = (num\_threads <= 1) ? 1 : num\_threads - 1;  
  
 int \*sizes = malloc(sizeof(int) \* (size\_t)working\_threads);  
 if (working\_threads == 1)  
 {  
 sizes[0] = n; // single-thread mode  
 return sizes;  
 }  
  
 int chunk\_remainder = n % working\_threads;  
 int avg\_chunk\_size = n / working\_threads;  
  
 for (int i = 0; i < working\_threads; i++)  
 {  
 sizes[i] = (i == working\_threads - 1) ? avg\_chunk\_size + chunk\_remainder : avg\_chunk\_size;  
 }  
 return sizes;  
}  
  
/\*\*  
 \* @brief Computes the starting indices for each split given the sizes  
 \*  
 \* @param sizes The sizes of each split  
 \* @param num\_threads The number of threads  
 \* @return int\* An array of starting indices for each split  
 \* AI use: AI Assisted  
 \*/  
int \*split\_indices(int \*sizes, int num\_threads)  
{  
 int num\_splits = (num\_threads <= 1) ? 1 : num\_threads - 1;  
 int \*indices = malloc(sizeof(int) \* (size\_t)num\_splits);  
 indices[0] = 0;  
 for (int i = 1; i < num\_splits; i++)  
 {  
 indices[i] = indices[i - 1] + sizes[i - 1];  
 }  
 return indices;  
}  
  
/\*\*  
 \* @brief Splits the array into num\_threads parts and fills in par\_args  
 \*  
 \* @param A The array to split  
 \* @param indices The starting indices of each split  
 \* @param num\_threads The number of threads  
 \* @return int\*\* An array of pointers to the start of each subarray  
 \* AI use: AI Assisted  
 \*/  
int \*\*split\_arrays\_even(int \*A, int \*indices, int num\_threads)  
{  
 int worker\_threads = num\_threads - 1;  
  
 int \*\*subarrays = malloc(sizeof(int \*) \* (size\_t)worker\_threads);  
 if (!subarrays)  
 return NULL;  
  
 for (int i = 0; i < worker\_threads; i++)  
 {  
 subarrays[i] = &A[indices[i]];  
 }  
  
 return subarrays;  
}  
  
/\*\*  
 \* @brief Splits the array into subarray with the remainder after even splits  
 \*  
 \* @param A The array to split  
 \* @param indices The starting indices of each split  
 \* @param num\_threads The number of threads  
 \* @return int\*\* An array of pointers to the start of each subarray  
 \* AI use: AI Assisted  
 \*/  
int \*\*split\_arrays\_remainder(int \*A, int \*indices, int num\_threads)  
{  
 int worker\_threads = num\_threads - 1;  
  
 int \*\*remainder\_subarray = malloc(sizeof(int \*));  
 if (!remainder\_subarray)  
 return NULL;  
  
 remainder\_subarray[0] = &A[indices[worker\_threads - 1]];  
 return remainder\_subarray;  
}  
  
  
  
/\*\*  
 \* @brief Multi-threaded mergesort  
 \*  
 \* @param A The array to sort  
 \* @param n The size of the array  
 \* @param num\_threads The number of threads to use  
 \* AI use: AI Assisted  
 \*/  
void mergesort\_mt(int \*A, int n, int num\_threads)  
{  
 if (num\_threads < 2)  
 {  
 mergesort\_s(A, 0, n - 1);  
 return;  
 }  
  
 int worker\_threads = num\_threads - 1;  
 parallel\_args\_t \*par\_args = malloc(sizeof(parallel\_args\_t) \* (size\_t)worker\_threads);  
  
 int \*sizes = split\_sizes(n, num\_threads);  
 int \*indices = split\_indices(sizes, num\_threads);  
 int \*\*subarrays = split\_arrays\_even(A, indices, num\_threads);  
  
 // Launch worker threads  
 for (int i = 0; i < worker\_threads; i++)  
 {  
 par\_args[i].A = subarrays[i];  
 par\_args[i].start = 0;  
 par\_args[i].end = sizes[i] - 1;  
 pthread\_create(&par\_args[i].tid, NULL, parallel\_mergesort, &par\_args[i]);  
 }  
  
 // Wait for all workers  
 for (int i = 0; i < worker\_threads; i++)  
 {  
 pthread\_join(par\_args[i].tid, NULL);  
 }  
  
 // Merge all sorted segments in the main thread  
 merge\_all\_tree\_parallel\_locked(A, indices, sizes, num\_threads);  
  
 free(sizes);  
 free(indices);  
 free(subarrays);  
}  
  
  
  
  
/\*\*  
 \* @brief Merges all sorted segments in a tree-like fashion using multiple threads with locking  
 \*  
 \* @param A The array to merge  
 \* @param indices The starting indices of each sorted segment  
 \* @param sizes The sizes of each sorted segment  
 \* @param num\_threads The number of threads used for sorting  
 \* AI use: AI Assisted  
 \*/  
void merge\_all\_tree\_parallel\_locked(int \*A, int \*indices, int \*sizes, int num\_threads)  
{  
 int worker\_threads = num\_threads - 1;  
 int level\_size = worker\_threads;  
  
 pthread\_mutex\_t merge\_lock = PTHREAD\_MUTEX\_INITIALIZER;  
  
 int \*new\_indices = malloc((size\_t)level\_size \* sizeof(int));  
 int \*new\_sizes = malloc((size\_t)level\_size \* sizeof(int));  
  
 while (level\_size > 1)  
 {  
 int merge\_pairs = level\_size / 2;  
  
 pthread\_t \*threads = malloc(sizeof(pthread\_t) \* (size\_t)merge\_pairs);  
 merge\_task\_t \*tasks = malloc(sizeof(merge\_task\_t) \* (size\_t)merge\_pairs);  
  
 int new\_count = 0;  
 int t = 0;  
  
 for (int i = 0; i < level\_size; i += 2)  
 {  
 if (i + 1 < level\_size)  
 {  
 int start = indices[i];  
 int mid = indices[i] + sizes[i] - 1;  
 int end = indices[i + 1] + sizes[i + 1] - 1;  
  
 tasks[t].A = A;  
 tasks[t].start = start;  
 tasks[t].mid = mid;  
 tasks[t].end = end;  
 tasks[t].merge\_lock = &merge\_lock;  
  
 pthread\_create(&threads[t], NULL, merge\_worker, &tasks[t]);  
 t++;  
  
 new\_indices[new\_count] = start;  
 new\_sizes[new\_count] = sizes[i] + sizes[i + 1];  
 new\_count++;  
 }  
 else  
 {  
 // Odd one out — carry forward  
 new\_indices[new\_count] = indices[i];  
 new\_sizes[new\_count] = sizes[i];  
 new\_count++;  
 }  
 }  
  
 // Wait for all merges at this level  
 for (int j = 0; j < merge\_pairs; j++)  
 {  
 pthread\_join(threads[j], NULL);  
 }  
  
 // Prepare next level  
 for (int j = 0; j < new\_count; j++)  
 {  
 indices[j] = new\_indices[j];  
 sizes[j] = new\_sizes[j];  
 }  
  
 free(threads);  
 free(tasks);  
 level\_size = new\_count;  
 }  
  
 free(new\_indices);  
 free(new\_sizes);  
 pthread\_mutex\_destroy(&merge\_lock);  
}  
  
  
  
  
  
/\*\*  
 \* @brief The function that is called by each thread to sort their chunk  
 \*  
 \* @param args see struct parallel\_args  
 \* @return void\* always NULL  
 \* AI use: AI Assisted  
 \*/  
void \*parallel\_mergesort(void \*args)  
{  
 parallel\_args\_t \*parg = (parallel\_args\_t \*)args;  
 mergesort\_s(parg->A, parg->start, parg->end);  
 return NULL;  
}  
  
  
  
double getMilliSeconds()  
{  
 struct timeval now;  
 gettimeofday(&now, (struct timezone \*)0);  
 return (double)now.tv\_sec \* 1000.0 + (double)now.tv\_usec / 1000.0;  
}

### lab.h

#ifndef LAB\_H  
#define LAB\_H  
#include <pthread.h>  
  
#ifdef \_\_cplusplus  
extern "C"  
{  
#endif  
  
 // The threshold that we will use to switch to insertion sort, make sure that  
 // you use test arrays bigger than 5 so you are testing the merge sort  
#define INSERTION\_SORT\_THRESHOLD 5  
#define MAX\_THREADS 64  
 /\*\*  
 \* @brief Sorts an array of ints into ascending order using the constant  
 \* INSERTION\_SORT\_THRESHOLD internally  
 \*  
 \* @param A A pointer to the start of the array  
 \* @param p The starting index  
 \* @param r The ending index  
 \*/  
 void mergesort\_s(int \*A, int p, int r);  
  
 /\*\*  
 \* @brief Merge two sorted sequences A[p..q] and A[q+1..r] and place merged  
 \* output back in array A. Uses extra space proportional to  
 \* A[p..r].  
 \*  
 \* @param A The array to merge into  
 \* @param p The starting index of the first half  
 \* @param q The middle  
 \* @param r The ending index of the second half  
 \*/  
 void merge\_s(int A[], int p, int q, int r);  
  
 /\*\*  
 \* @brief Sorts an array of ints into ascending order using multiple  
 \* threads  
 \*  
 \* @param A A pointer to the start of the array  
 \* @param n The size of the array  
 \* @param num\_threads The number of threads to use.  
 \*/  
 void mergesort\_mt(int \*A, int n, int num\_thread);  
  
 /\*\*  
 \* @brief retuns the current time as milliseconds  
 \* @return the number of milliseconds  
 \*/  
 double getMilliSeconds();  
  
 /\*\*  
 \* @brief Represents a chunk of the array to be sorted by a thread  
 \* AI use: AI Assisted  
 \*  
 \*/  
typedef struct parallel\_args {  
 int \*A;  
 int start;  
 int end;  
 pthread\_t tid;  
} parallel\_args\_t;  
  
 /\*\*  
 \* @brief The function that is called by each thread to sort their chunk  
 \*  
 \* @param args see struct parallel\_args  
 \* @return void\* always NULL  
 \* AI use: AI Assisted  
 \*/  
 void \*parallel\_mergesort(void \*args);  
  
   
 /\*\*  
 \* @brief Splits the array into num\_threads parts and fills in par\_args  
 \*  
 \* @param n The size of the array  
 \* @param num\_threads The number of threads  
 \* @return int\* An array of sizes for each split  
 \* AI use: AI Assited  
 \*/  
 int \*split\_sizes(int n, int num\_threads);  
  
 /\*\*  
 \* @brief Computes the starting indices for each split given the sizes  
 \*  
 \* @param sizes The sizes of each split  
 \* @param num\_threads The number of threads  
 \* @return int\* An array of starting indices for each split  
 \* AI use: AI Assisted  
 \*/  
 int \*split\_indices(int \*sizes, int num\_threads);  
  
 /\*\*  
 \* @brief Splits the array into num\_threads subarrays  
 \*  
 \* @param A The array to split  
 \* @param indices The starting indices of each split  
 \* @param num\_threads The number of threads  
 \* @return int\*\* An array of pointers to the start of each subarray  
 \* AI use: AI Assisted  
 \*/  
 int \*\*split\_arrays\_even(int \*A, int \*indices, int num\_threads);  
  
 /\*\*  
 \* @brief Splits the array into subarray with the remainder after even splits  
 \*  
 \* @param A The array to split  
 \* @param indices The starting indices of each split  
 \* @param num\_threads The number of threads  
 \* @return int\*\* An array of pointers to the start of each subarray  
 \* AI use: AI Assisted  
 \*/  
 int \*\*split\_arrays\_remainder(int \*A, int \*indices, int num\_threads);  
  
 /\*\*  
 \* @brief Merges all sorted segments in a tree-like fashion  
 \*  
 \* @param A The array to merge  
 \* @param indices The starting indices of each segment  
 \* @param sizes The sizes of each segment  
 \* @param num\_threads The number of threads used (to determine segments)  
 \* AI use: AI Assisted  
 \*/  
 void merge\_all\_tree\_parallel\_locked(int \*A, int \*indices, int \*sizes, int num\_threads);  
  
  
 /\*\*  
 \* @brief Worker thread function to sort a segment  
 \*  
 \* @param arg The parallel\_args\_t pointer  
 \* @return void\* always NULL  
 \* AI use: AI Assisted  
 \*/  
 void \*thread\_sort(void \*arg);  
  
 /\*\*  
 \* @brief Entry point for the main function  
 \*  
 \* @param argc The argument count  
 \* @param argv The argument array  
 \* @return The exit code  
 \*/  
 int myMain(int argc, char \*\*argv);  
  
#ifdef \_\_cplusplus  
} // extern "C"  
#endif  
  
#endif

### main.c

#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include "lab.h"  
  
#ifndef TEST  
  
  
/\*\*  
 \* @brief Main function to run multi-threaded mergesort  
 \*  
 \* @param argc Argument count  
 \* @param argv Argument values  
 \* @return int Exit status  
 \* AI use: AI Assisted  
 \*/  
  
int main(int argc, char \*\*argv) {  
 if (argc < 3) {  
 printf("usage: %s <array\_size> <num\_threads>\n", argv[0]);  
 return 1;  
 }  
  
 int array\_size = atoi(argv[1]);  
 int num\_threads = atoi(argv[2]);  
 if (num\_threads < 1) num\_threads = 1; // enforce minimum  
 if (num\_threads > MAX\_THREADS) num\_threads = MAX\_THREADS; // enforce maximum  
  
 // Allocate and initialize array  
 int \*A = malloc(sizeof(int) \* (size\_t)array\_size);  
 for (int i = 0; i < array\_size; i++)  
 A[i] = rand() % 10000;  
  
 // Split array  
 int \*sizes = split\_sizes(array\_size, num\_threads);  
 int \*indices = split\_indices(sizes, num\_threads);  
  
 int worker\_threads = (num\_threads <= 1) ? 0 : num\_threads - 1;  
 parallel\_args\_t \*par\_args = (worker\_threads > 0) ?  
 malloc(sizeof(parallel\_args\_t) \* (size\_t)worker\_threads) : NULL;  
  
 double start = getMilliSeconds();  
  
 if (worker\_threads == 0) {  
 // Single-thread mode  
 mergesort\_s(A, 0, array\_size - 1);  
 } else {  
 // Multi-thread mode  
 for (int i = 0; i < worker\_threads; i++) {  
 par\_args[i].A = A;  
 par\_args[i].start = indices[i];  
 par\_args[i].end = indices[i] + sizes[i] - 1;  
//GCOVR\_EXCL\_START  
 pthread\_create(&par\_args[i].tid, NULL, thread\_sort, &par\_args[i]);  
 }  
  
 for (int i = 0; i < worker\_threads; i++) {  
 pthread\_join(par\_args[i].tid, NULL);  
 }  
  
 merge\_all\_tree\_parallel\_locked(A, indices, sizes, num\_threads);  
 }  
//GCOVR\_EXCL\_STOP  
  
 double end = getMilliSeconds();  
  
 printf("%d %f\n", num\_threads, end - start);  
  
 free(A);  
 free(sizes);  
 free(indices);  
 if (par\_args) free(par\_args);  
  
 return 0;  
}  
#endif

## Tests Files

### lab-test.c

#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
#include "harness/unity.h"  
#include "../src/lab.h"  
#include <time.h>  
  
/\* AI use: AI Assisted \*/  
  
// ----------------------- Test Setup ------------------------//  
  
void setUp(void) { /\* Nothing to setup for now \*/ }  
void tearDown(void) { /\* Nothing to cleanup \*/ }  
  
// Helper: check if array is sorted ascending  
int is\_sorted(int \*A, int n) {  
 for (int i = 1; i < n; i++) {  
 if (A[i - 1] > A[i]) return 0;  
 }  
 return 1;  
}  
  
// --------------------------- Tests ---------------------------  
  
  
/\*AI use: AI Assisted\*/  
// Test single-threaded mergesort  
void test\_mergesort\_s(void) {  
 int A[] = {5, 2, 9, 1, 5, 6};  
 int n = sizeof(A) / sizeof(A[0]);  
 mergesort\_s(A, 0, n - 1);  
 TEST\_ASSERT\_TRUE(is\_sorted(A, n));  
}  
  
//AI use: AI Assisted  
// Test multi-threaded array split  
void test\_split\_sizes\_indices(void) {  
 int n = 10;  
 int num\_threads = 3;  
 int \*A = malloc(sizeof(int) \* n);  
 for (int i = 0; i < n; i++) A[i] = i;  
  
 int \*sizes = split\_sizes(n, num\_threads);  
 int \*indices = split\_indices(sizes, num\_threads);  
  
 int sum = 0;  
 for (int i = 0; i < num\_threads - 1; i++) sum += sizes[i];  
 TEST\_ASSERT\_EQUAL\_INT(n, sum + (n % (num\_threads - 1))); // sum of sizes matches array  
  
 TEST\_ASSERT\_EQUAL\_INT(0, indices[0]);  
 TEST\_ASSERT\_EQUAL\_INT(sizes[0], indices[1]);  
  
 free(A);  
 free(sizes);  
 free(indices);  
}  
  
//AI use: AI Assisted  
// Test multi-threaded sorting  
void test\_parallel\_sort(void) {  
 int n = 20;  
 int num\_threads = 4;  
 int \*A = malloc(sizeof(int) \* n);  
 for (int i = 0; i < n; i++) A[i] = rand() % 100;  
  
 int \*sizes = split\_sizes(n, num\_threads);  
 int \*indices = split\_indices(sizes, num\_threads);  
  
 int worker\_threads = num\_threads - 1;  
 parallel\_args\_t \*par\_args = malloc(sizeof(parallel\_args\_t) \* worker\_threads);  
  
 for (int i = 0; i < worker\_threads; i++) {  
 par\_args[i].A = A;  
 par\_args[i].start = indices[i];  
 par\_args[i].end = indices[i] + sizes[i] - 1;  
 pthread\_create(&par\_args[i].tid, NULL, thread\_sort, &par\_args[i]);  
 }  
  
 for (int i = 0; i < worker\_threads; i++)  
 pthread\_join(par\_args[i].tid, NULL);  
  
 merge\_all\_tree\_parallel\_locked(A, indices, sizes, num\_threads);  
  
 TEST\_ASSERT\_TRUE(is\_sorted(A, n));  
  
 free(A);  
 free(sizes);  
 free(indices);  
 free(par\_args);  
}  
  
//AI use: AI Assisted  
// Test edge cases: empty array, single-element array  
void test\_edge\_cases(void) {  
 int A1[] = {};  
 mergesort\_s(A1, 0, -1); // Should not crash  
 TEST\_ASSERT\_TRUE(1); // Just checking no crash  
  
 int A2[] = {42};  
 mergesort\_s(A2, 0, 0);  
 TEST\_ASSERT\_TRUE(A2[0] == 42);  
}  
  
//AI use: AI Assisted  
// Stress test with larger arrays and varying thread counts  
void test\_stress\_sort(void) {  
 const int sizes[] = {10000, 50000, 100000};  
 const int max\_threads = 8; // adjust as needed  
 for (int s = 0; s < sizeof(sizes)/sizeof(sizes[0]); s++) {  
 int n = sizes[s];  
 int \*A = malloc(sizeof(int) \* n);  
 for (int i = 0; i < n; i++) A[i] = rand() % 100000;  
  
 for (int num\_threads = 1; num\_threads <= max\_threads; num\_threads++) {  
 int \*copy = malloc(sizeof(int) \* n);  
 memcpy(copy, A, sizeof(int) \* n);  
  
 int \*sizes\_split = split\_sizes(n, num\_threads);  
 int \*indices\_split = split\_indices(sizes\_split, num\_threads);  
  
 int worker\_threads = (num\_threads <= 1) ? 0 : num\_threads - 1;  
 parallel\_args\_t \*par\_args = (worker\_threads > 0) ?  
 malloc(sizeof(parallel\_args\_t) \* worker\_threads) : NULL;  
  
 double start = getMilliSeconds();  
  
 if (worker\_threads == 0) {  
 mergesort\_s(copy, 0, n - 1);  
 } else {  
 for (int i = 0; i < worker\_threads; i++) {  
 par\_args[i].A = copy;  
 par\_args[i].start = indices\_split[i];  
 par\_args[i].end = indices\_split[i] + sizes\_split[i] - 1;  
 pthread\_create(&par\_args[i].tid, NULL, (void\*(\*)(void\*))thread\_sort, &par\_args[i]);  
 }  
  
 for (int i = 0; i < worker\_threads; i++)  
 pthread\_join(par\_args[i].tid, NULL);  
  
 merge\_all\_tree\_parallel\_locked(copy, indices\_split, sizes\_split, num\_threads);  
 }  
  
 double end = getMilliSeconds();  
 printf("Stress test: array\_size=%d threads=%d time=%f\n", n, num\_threads, end - start);  
  
 TEST\_ASSERT\_TRUE(is\_sorted(copy, n));  
  
 free(copy);  
 if (par\_args) free(par\_args);  
 free(sizes\_split);  
 free(indices\_split);  
 }  
 free(A);  
 }  
}  
  
/\* AI use: AI Assisted \*/  
// ----------------------- Test Runner ------------------------//  
int main(void) {  
 UNITY\_BEGIN();  
 RUN\_TEST(test\_mergesort\_s);  
 RUN\_TEST(test\_split\_sizes\_indices);  
 RUN\_TEST(test\_parallel\_sort);  
 RUN\_TEST(test\_edge\_cases);  
 RUN\_TEST(test\_stress\_sort);  
 return UNITY\_END();  
}

## Scripts Files

### createplot.sh

#!/usr/bin/env bash  
function usage() {  
 echo "$0 usage:" && grep " .)\ #" $0  
 exit 0  
}  
[ $# -eq 0 ] && usage  
while getopts "hs:f:" arg; do  
 case $arg in  
 s) # The size of the array to sort.  
 size=${OPTARG}  
 ;;  
 f) # The plot file name  
 name=${OPTARG}  
 ;;  
 h | \*) # Display help.  
 usage  
 exit 0  
 ;;  
 esac  
done  
MYAPP="../build/release/myapp"  
  
if [ "$name" == "" ] || [ "$size" == "" ]  
then  
 usage  
 exit 0  
fi  
if [ -e $MYAPP ]; then  
 if [ -e "data.dat" ]; then  
 rm -f data.dat  
 fi  
 echo "Running myprogram to generate data"  
 echo "#Time Threads" >> data.dat  
 for n in {1..64}; do  
 echo -ne "running $n thread \r"  
 $MYAPP "$size" "$n" >> data.dat  
 done  
 gnuplot -e "filename='$name.png'" graph.plt  
 echo "Created plot $name.png from data.dat file"  
else  
 echo "myapp is not present in the build directory. Did you compile your code?"  
fi

Report generated on 10/24/2025 at 22:21:34

## End of Report

SHA-256 Hash of the report: 0058c04f95d33743f90cb1af61fe6dbd1bba4574adfbf31118d4154790b49da5

Do not edit the generated report. Any changes will be reported as academic dishonesty

## GitHub Info

* GitHub repo name: JonFlores3475/Fall2025CS452P3
* The repository visibility is public.
* The workflow was triggered by JonFlores3475