MapReduce Systems for Parallel Sorting and Max-Value Aggregation with Constrained Memory

Daisy Aptovska

Computer Science Program, The Pennsylvania State University

CMPSC 472: Operating Systems Concepts

Dr. Janghoon Yang

October 29, 2025

Table of Contents

Project Description……...3

Instructions……...3

Structure of the Code……...4

Description of the Implementation……...8

Performance Evaluation……...9

Conclusion……..13

References……..14

Project Description

In this project, a MapReduce-style system is designed to leverage the strength of parallel computing for sorting and obtaining maximum values from arrays in C. MapReduce is a model that is used across large datasets which distributes clusters across multiple machines for maximizing computing power. In this project, we implement a model inspired by this model without implementing a full-scale MapReduce system. Within this project, the goal is to compare the time/efficiency of sorting/searching across 2 array sizes of 32 and 131,072. In Part 1, the project completes parallel sorting, specifically Merge Sort, across both arrays and times their efficiency in multithreading and multiprocessing, using 1, 2, 4, and 8 worker threads/processes in each implementation. In Part 2, the project works with a shared memory buffer. Each worker thread obtains the maximum of the chunk of the array it is assigned to, and it is written to the shared memory integer only if the local maximum is larger, using 1, 2, 4, and 8 worker threads to execute this.

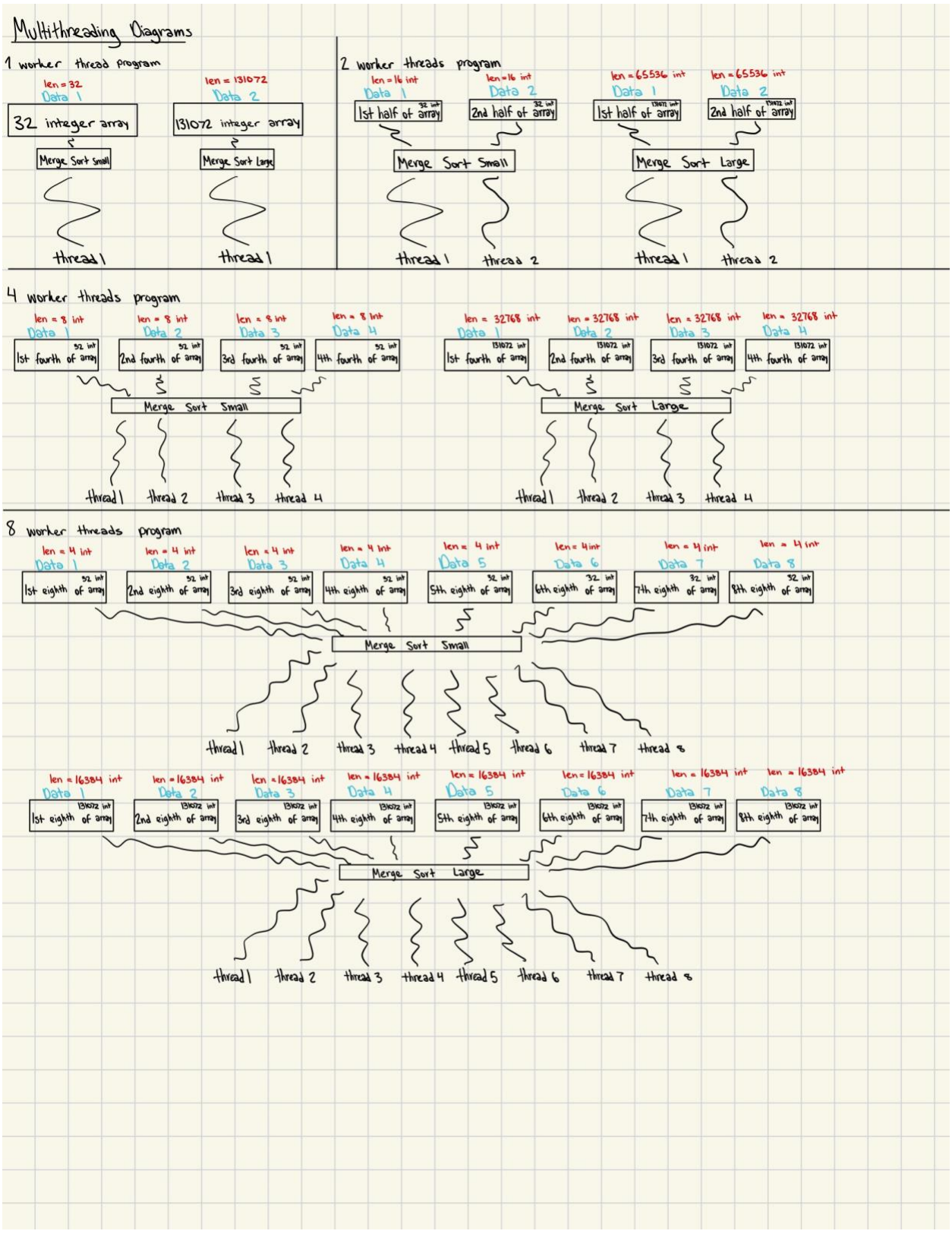
Instructions

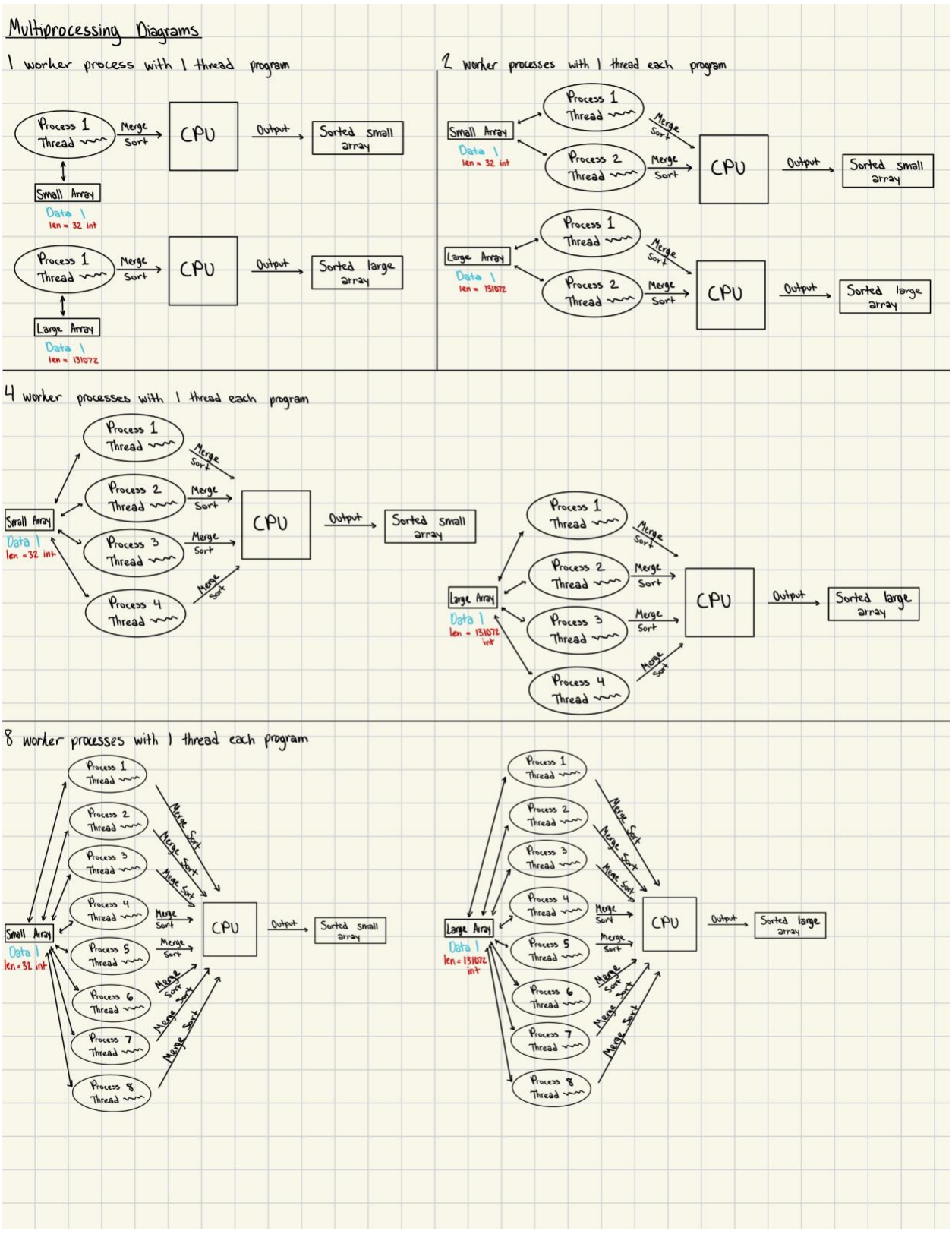
To run this project, it is best to use Google Colab since it is a cloud notebook that provides all the needed libraries. The Google Colab notebooks can be downloaded as Jupyter notebooks and can be edited in other notebook editors (such as Anaconda). The programming language C is used to complete this project. After downloading the three notebooks in this GitHub repository in Google Colab, the code can simply be run one code cell at a time. There are no necessary installations needed to properly execute this project.

Structure of the Code

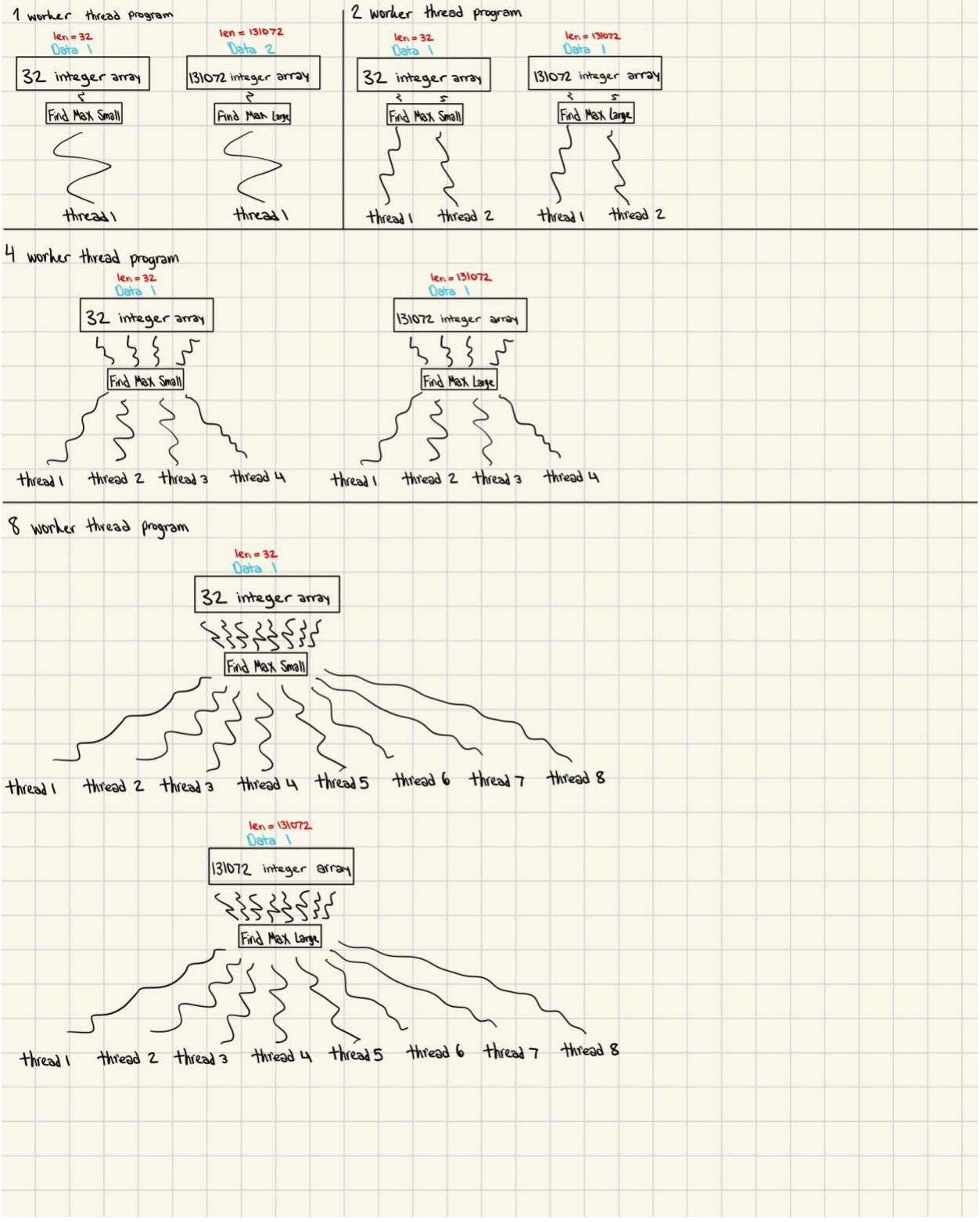
In this project, the code is modularized into three Jupyter notebooks edited in Google Colab. In the multithreading notebook, there are 4 code blocks that execute merge sort on two arrays. In each code block, there are either 1, 2, 4, or 8 worker threads which are assigned an even chunk of the small array and large array. In the multiprocessing notebook, there are 4 code blocks which execute merge sort on the two arrays. In each code block, there are either 1, 2, 4, or 8 worker processes, which are created through the fork() function in C, that are assigned an event chunk of the small array and large array. Lastly, in the synchronization notebook, there are 4 code blocks. In each code block, there are either 1, 2, 4, or 8 worker threads which are assigned an even chunk of the small array and large array. Each thread finds the local maximum of its chunk and compares it to the shared memory maximum. The shared memory maximum is only overwritten if it is smaller than the local maximum of a chunk.

In the following diagrams, the structure of the worker threads/processes is shown for the multithreading, multiprocessing, and synchronization implementations.


Multithreading Diagrams
1 worker thread program
len = 32
Data 1
32 integer array
Merge Sort Small
thread 1
len = 131072
Data 2
131072 integer array
Merge Sort Large
thread 1
2 worker threads program
len = 16 int
Data 1
1st half of array
32 int
Merge Sort Small
thread 1
thread 2
len = 16 int
Data 2
2nd half of array
len = 65536 int
Data 1
1st half of array
Merge Sort Large
thread 1
thread 2
len = 65536 int
Data 2
2nd half of array
4 worker threads program
len = 8 int
Data 1
1st fourth of array
Merge Sort Small
thread 1
thread 2
thread 3
thread 4
len = 8 int
Data 2
2nd fourth of array
len = 8 int
Data 3
3rd fourth of array
len = 8 int
Data 4
4th fourth of array
len = 32768 int
Data 1
1st fourth of array
Merge Sort Large
thread 1
thread 2
thread 3
thread 4
len = 32768 int
Data 2
2nd fourth of array
len = 32768 int
Data 3
3rd fourth of array
len = 32768 int
Data 4
4th fourth of array
8 worker threads program
len = 4 int
Data 1
1st eighth of array
Merge Sort Small
thread 1
thread 2
thread 3
thread 4
thread 5
thread 6
thread 7
thread 8
len = 4 int
Data 2
2nd eighth of array
len = 4 int
Data 3
3rd eighth of array
len = 4 int
Data 4
4th eighth of array
len = 4 int
Data 5
5th eighth of array
len = 4 int
Data 6
6th eighth of array
len = 4 int
Data 7
7th eighth of array
len = 4 int
Data 8
8th eighth of array
len = 16384 int
Data 1
1st eighth of array
Merge Sort Large
thread 1
thread 2
thread 3
thread 4
thread 5
thread 6
thread 7
thread 8
len = 16384 int
Data 2
2nd eighth of array
len = 16384 int
Data 3
3rd eighth of array
len = 16384 int
Data 4
4th eighth of array
len = 16384 int
Data 5
5th eighth of array
len = 16384 int
Data 6
6th eighth of array
len = 16384 int
Data 7
7th eighth of array
len = 16384 int
Data 8
8th eighth of array


Multiprocessing Diagrams
1 worker process with 1 thread program
Process 1 Thread
Merge Sort
CPU
Output
Sorted small array
Small Array
Data 1
len = 32 int
Process 1 Thread
Merge Sort
CPU
Output
Sorted large array
Large Array
Data 1
len = 131072
2 worker processes with 1 thread each program
Process 1 Thread
Process 2 Thread
Small Array
Data 1
len = 32 int
Merge Sort
CPU
Output
Sorted small array
Process 1 Thread
Process 2 Thread
Large Array
Data 1
len = 131072
Merge Sort
CPU
Output
Sorted large array
4 worker processes with 1 thread each program
Process 1 Thread
Process 2 Thread
Process 3 Thread
Process 4 Thread
Small Array
Data 1
len = 32 int
Merge Sort
CPU
Output
Sorted small array
Large Array
Data 1
len = 131072 int
Sorted large array
8 worker processes with 1 thread each program
Process 1 Thread
Process 2 Thread
Process 3 Thread
Process 4 Thread
Process 5 Thread
Process 6 Thread
Process 7 Thread
Process 8 Thread
Small Array
Data 1
len = 32 int
Merge Sort
CPU
Output
Sorted small array
Large Array
Data 1
len = 131072 int
Sorted large array

Synchronization Diagrams



Based on this project's code, the MapReduce framework is supported because the worker threads/processes model the distributed clusters/machines that a full-scale MapReduce framework would implement. On a smaller scale, this project demonstrates the importance of process synchronization, threading, and concurrency/parallelism.

Description of the Implementation

In this implementation, many built-in includes are used. In the multithreading notebook, the following libraries are included: standard C, C time, POSIX, POSIX pthreads, IPC, shared memory, and string. In the multiprocessing notebook, the following libraries are included: standard C, C time, POSIX, and wait. In the synchronization notebook, the following libraries are included: standard C, POSIX, POSIX pthreads, semaphores, and C time. Through these libraries, the functions can execute properly and efficiently. To assist with function implementation (merge sort codes), a standard recursive implementation of merge sort with 2 functions was implemented along with a thread helper function (Kartik, 2025a-2025e).

In this project's process management, the worker processes/threads are generated in the main routine of each test to create a fair and accurate measurement of their performance. Different worker threads are used to measure the small and large arrays tests to provide an accurate time measurement. In the multithreading portion of this project,

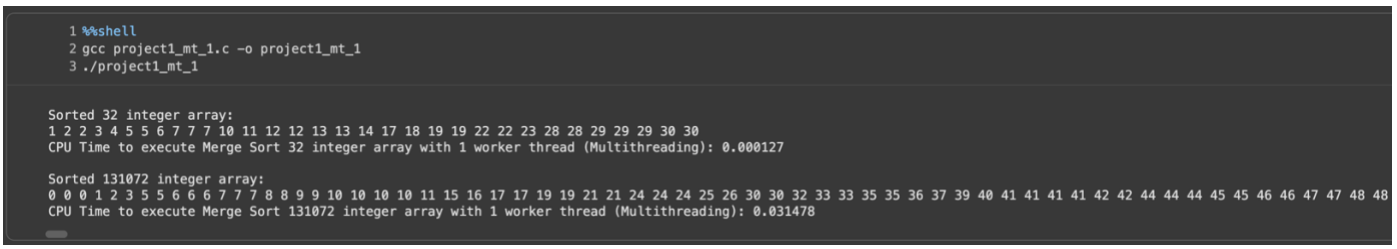
In this project's IPC, the worker threads/processes communicate based on the arrays initialized in the main routine of each section. In the main routine of each test, a random 32 integer array and a random 131,072 integer array are initialized. From these arrays, depending on the number of worker threads/processes, the array is sliced into the corresponding amount of

slices. Then, the slice of the array is passed into a corresponding thread/process for sorting or maximum aggregation.

In this project's threading, the threads are manually created and joined in the main routine one at a time. This mechanism was chosen to ensure the slices of the arrays were being passed properly to reach thread, so there is no overlap in the data (avoiding synchronization issues within arrays). In this project's synchronization strategy and implementation, semaphores were used to find the maximum of the small and large array. The semaphores check for data, allowing for one at a time access of the threads into the function. The semaphores do not implement an empty semaphore since, in this implementation, the buffer (global maximum) need not be empty to overwrite the value. In this project's performance evaluation, the CPU time of each execution was the main performance evaluation. In the multithreading tests and multiprocessing tests, the CPU time of using a different amount of worker threads was compared amongst small and large array merge sorting.

Performance Evaluation

In the following multithreading snapshots, each snapshot shows the execution of each of the 4 code blocks and the CPU time for each array being sorted with 1, 2, 4, and 8 worker threads.



```
%%shell
gcc project1_mt_2.c -o project1_mt_2
./project1_mt_2

Sorted 32 integer array:
1 2 2 3 4 5 5 6 7 7 7 10 11 12 12 13 13 14 17 18 19 19 22 22 23 28 28 29 29 29 30 30
CPU Time to execute Merge Sort 32 integer array with 2 worker threads (Multithreading): 0.000121

Sorted 131072 integer array:
0 0 0 1 2 3 5 5 6 6 6 7 7 7 8 8 9 9 10 10 10 10 11 15 16 17 17 19 19 21 21 24 24 24 25 26 30 30 32 33 33 35 35 36 37 39 40 41 41 41 41 42 42 44 44 44 45 45 46 46 47 47 48 48 49
CPU Time to execute Merge Sort 131072 integer array with 2 worker threads (Multithreading): 0.044298
```

```
%%shell
gcc project1_mt_4.c -o project1_mt_4
./project1_mt_4

Sorted 32 integer array:
1 2 3 4 5 5 6 7 7 7 10 11 12 12 13 13 14 17 18 19 19 22 22 23 28 28 2 29 29 29 30 30
CPU Time to execute Merge Sort 32 integer array with 4 worker threads (Multithreading): 0.000232

Sorted 131072 integer array:
0 0 0 1 2 3 5 5 6 6 6 7 7 7 8 8 9 9 10 10 10 10 11 15 16 17 17 19 19 21 21 24 24 24 25 26 30 30 32 33 33 35 35 36 37 39 40 41 41 41 41 42 42 44 44 44 45 45 46 46 47 47 48 48
CPU Time to execute Merge Sort 131072 integer array with 4 worker threads (Multithreading): 0.040962
```

```
%%shell
gcc project1_mt_8.c -o project1_mt_8
./project1_mt_8

Sorted 32 integer array:
1 2 2 3 4 5 5 6 7 7 7 10 11 12 12 13 13 14 17 18 19 19 22 22 23 28 28 29 29 29 30 30
CPU Time to execute Merge Sort 32 integer array with 8 worker threads (Multithreading): 0.000445

Sorted 131072 integer array:
0 0 0 1 2 3 5 5 6 6 6 7 7 7 8 8 9 9 10 10 10 10 11 15 16 17 17 19 19 21 21 24 24 24 25 26 30 30 32 33 33 35 35 36 37 39 40 41 41 41 41 42 42 44 44 44 45 45 46 46 47 47 48 48 49
CPU Time to execute Merge Sort 131072 integer array with 8 worker threads (Multithreading): 0.035378
```

In the following multiprocessing snapshots, each snapshot shows the execution of each of the code blocks and the CPU time for each array being sorted with 1, 2, 4, and 8 worker processes. There is 7 code blocks total for multiprocessing since the small and large arrays for 2, 4, and 8 processes are in separate main routines for simplicity, accurate CPU time tracking, and easier readability of the code. Each process creates its own POSIX thread to execute the function.

```
%%shell
gcc project1_mp_1.c -o project1_mp_1
./project1_mp_1

Sorted 32 integer array:
1 2 2 3 4 5 5 6 7 7 7 10 11 12 12 13 13 14 17 18 19 19 22 22 23 28 28 29 29 29 30 30
CPU Time to execute Merge Sort 32 integer array with 1 worker process (Multiprocessing): 0.000007

Sorted 131072 integer array:
0 0 0 1 2 3 5 5 6 6 6 7 7 7 8 8 9 9 10 10 10 10 11 15 16 17 17 19 19 21 21 24 24 24 25 26 30 30 32 33 33 35 35 36 37 39 40 41 41 41 41 42 42 44 44 44 45 45 46 46 47 47 48 48 49 5
CPU Time to execute Merge Sort 131072 integer array with 1 worker process (Multiprocessing): 0.033146
```

```
%%shell
gcc project1_mp_2_small.c -o project1_mp_2_small
./project1_mp_2_small
```

```
project1_mp_2_small.c: In function ‘main’:
project1_mp_2_small.c:109:9: warning: implicit declaration of function ‘pthread_create’; did you mean ‘timer_create’? [-Wimplicit-function-declaration]
  109 |         pthread_create(&process2, NULL, mergeSortSmall, &small_args);
      |         ^~~~~~~~~~~~~~
      |         timer_create

Sorted 32 integer array:
0 1 2 2 3 4 5 5 6 7 7 7 10 11 12 12 13 13 14 17 18 19 19 22 22 23 28 28 29 29 29 30
CPU Time to execute Merge Sort 32 integer array with 2 worker processes (Multiprocessing): 0.000105
```

```
%%shell
gcc project1_mp_2_large.c -o project1_mp_2_large
./project1_mp_2_large
```

```
project1_mp_2_large.c: In function ‘main’:
project1_mp_2_large.c:109:9: warning: implicit declaration of function ‘pthread_create’; did you mean ‘timer_create’? [-Wimplicit-function-declaration]
  109 |         pthread_create(&process2, NULL, mergeSortLarge, &large_args);
      |         ^~~~~~~~~~~~~~
      |         timer_create

Sorted 131072 integer array:
0 40 81 106 209 230 255 257 273 292 331 334 352 428 477 479 486 487 488 497 509 519 529 565 580 607 610 615 645 647 667 668 683 706 719 749 752 774 775 776 785 806 809 816 824 82
CPU Time to execute Merge Sort 131072 integer array with 2 worker processes (Multiprocessing): 0.000355
```

```
%%shell
gcc project1_mp_4_small.c -o project1_mp_4_small
./project1_mp_4_small
```

```
project1_mp_4_small.c: In function ‘main’:
project1_mp_4_small.c:111:9: warning: implicit declaration of function ‘pthread_create’; did you mean ‘timer_create’? [-Wimplicit-function-declaration]
  111 |         pthread_create(&process2, NULL, mergeSortSmall, &small_args);
      |         ^~~~~~~~~~~~~~
      |         timer_create

Sorted 32 integer array:
0 1 2 2 3 4 5 5 6 7 7 7 10 11 12 12 13 13 14 17 18 19 19 22 22 23 28 28 29 29 29 30
CPU Time to execute Merge Sort 32 integer array with 4 worker processes (Multiprocessing): 0.000082
```

```
%%shell
gcc project1_mp_4_large.c -o project1_mp_4_large
./project1_mp_4_large
```

```
project1_mp_4_large.c: In function ‘main’:
project1_mp_4_large.c:111:9: warning: implicit declaration of function ‘pthread_create’; did you mean ‘timer_create’? [-Wimplicit-function-declaration]
  111 |         pthread_create(&process2, NULL, mergeSortLarge, &large_args);
      |         ^~~~~~~~~~~~~~
      |         timer_create

Sorted 131072 integer array:
0 0 10 40 46 73 81 106 116 123 175 181 209 230 248 257 268 273 286 292 318 331 334 343 352 357 405 428 463 465 477 479 486 487 488 497 509 519 532 539 546 552 565 576 580 585 603
CPU Time to execute Merge Sort 131072 integer array with 4 worker processes (Multiprocessing): 0.000246
```

```
%%shell
gcc project1_mp_8_small.c -o project1_mp_8_small
./project1_mp_8_small
```

```
project1_mp_8_small.c: In function ‘main’:
project1_mp_8_small.c:115:9: warning: implicit declaration of function ‘pthread_create’; did you mean ‘timer_create’? [-Wimplicit-function-declaration]
  115 |         pthread_create(&process2, NULL, mergeSortSmall, &small_args);
      |         ^~~~~~~~~~~~~~
      |         timer_create

Sorted 32 integer array:
0 1 2 2 3 4 5 5 6 7 7 7 10 11 12 12 13 13 14 17 18 19 19 22 22 23 28 28 29 29 29 30
CPU Time to execute Merge Sort 32 integer array with 8 worker processes (Multiprocessing): 0.000070
```

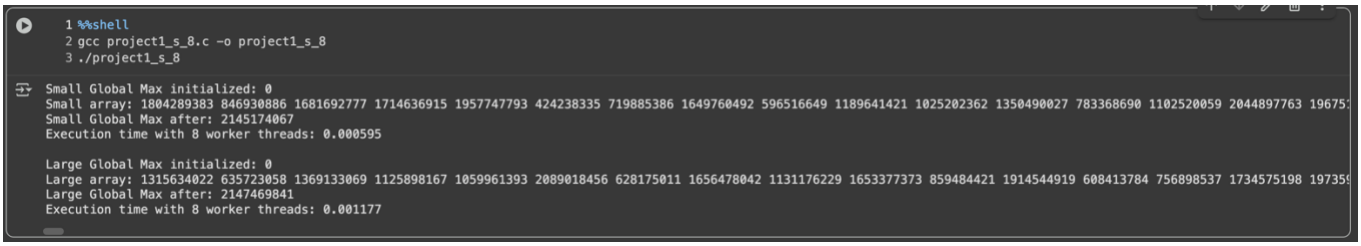
```
%%shell
gcc project1_mp_8_large.c -o project1_mp_8_large
./project1_mp_8_large

project1_mp_8_large.c: In function 'main':
project1_mp_8_large.c:115:9: warning: implicit declaration of function 'pthread_create'; did you mean 'timer_create'? [-Wimplicit-function-declaration]
  115 |         pthread_create(&process2, NULL, mergeSortLarge, &large_args);
      |         ^~~~~~~~~~~~~
      |         timer_create

Sorted 131072 integer array:
0 40 81 106 106 106 106 106 123 123 123 123 175 175 175 175 181 181 181 181 209 230 255 257 268 268 268 268 273 292 318 318 318 318 331 334 352 428 465 465 477 479 486 487 488 49
CPU Time to execute Merge Sort 131072 integer array with 8 worker processes (Multiprocessing): 0.000187
```

In the following synchronization snapshots, each snapshot shows the execution of each of the 4 code blocks and the CPU time for each array being sorted with 1, 2, 4, and 8 worker processes. Each thread is passed the array and the find maximum function. Concurrently, they compare the values of the array to the current global maximum, and the threads only change the global maximum if the value found in the array is larger. Through semaphores, the code ensures that only one thread at a time can compare.

```
%%shell
gcc project1_s_1.c -o project1_s_1
./project1_s_1

Small Global Max initialized: 0
Small array: 1804289383 846930886 1681692777 1714636915 1957747793 424238335 719885386 1649760492 596516649 1189641421 1025202362 1350490027 783368690 1102520059 2044897763 19675
Small Global Max after: 2145174067
Execution time with 1 worker thread: 0.000058

Large Global Max initialized: 0
Large array: 1315634022 635723058 1369133069 1125898167 1059961393 2089018456 628175011 1656478042 1131176229 1653377373 859484421 1914544919 608413784 756898537 1734575198 19735
Large Global Max after: 2147469841
Execution time with 1 worker thread: 0.000081
```

```
%%shell
gcc project1_s_2.c -o project1_s_2
./project1_s_2

Small Global Max initialized: 0
Small array: 1804289383 846930886 1681692777 1714636915 1957747793 424238335 719885386 1649760492 596516649 1189641421 1025202362 1350490027 783368690 1102520059 2044897763 19675
Small Global Max after: 2145174067
Execution time with 2 worker threads: 0.000110

Large Global Max initialized: 0
Large array: 1315634022 635723058 1369133069 1125898167 1059961393 2089018456 628175011 1656478042 1131176229 1653377373 859484421 1914544919 608413784 756898537 1734575198 19735
Large Global Max after: 2147469841
Execution time with 2 worker threads: 0.000551
```

```
%%shell
gcc project1_s_4.c -o project1_s_4
./project1_s_4

Small Global Max initialized: 0
Small array: 1804289383 846930886 1681692777 1714636915 1957747793 424238335 719885386 1649760492 596516649 1189641421 1025202362 1350490027 783368690 1102520059 2044897763 19675
Small Global Max after: 2145174067
Execution time with 4 worker threads: 0.000159

Large Global Max initialized: 0
Large array: 1315634022 635723058 1369133069 1125898167 1059961393 2089018456 628175011 1656478042 1131176229 1653377373 859484421 1914544919 608413784 756898537 1734575198 19735
Large Global Max after: 2147469841
Execution time with 4 worker threads: 0.000459
```



```
%%shell
gcc project1_s_8.c -o project1_s_8
./project1_s_8

Small Global Max initialized: 0
Small array: 1804289383 846930886 1681692777 1714636915 1957747793 424238335 719885386 1649760492 596516649 1189641421 1025202362 1350490027 783368690 1102520059 2044897763 19675
Small Global Max after: 2145174067
Execution time with 8 worker threads: 0.000595

Large Global Max initialized: 0
Large array: 1315634022 635723058 1369133069 1125898167 1059961393 2089018456 628175011 1656478042 1131176229 1653377373 859484421 1914544919 608413784 756898537 1734575198 19735
Large Global Max after: 2147469841
Execution time with 8 worker threads: 0.001177
```

Based on these execution results, multithreading, multiprocessing, and synchronization all performed very well. However, based on the large array's sorting results, it appears that multiprocessing outperforms multithreading for merge sorting a large integer array with low CPU time execution. The CPU time was lower the more worker processes/threads were given to the multiprocessing test. Surprisingly, with more worker threads, the synchronization maximum test performed worse than with only one worker thread. In the multithreading tests, the more worker threads provided did not improve the performance.

Conclusion

Based on the findings of this project, my results suggest that multiprocessing outperforms multithreading for a large dataset. Additionally, the results suggest that further optimization is needed to improve multithreading and synchronization for the increase of worker threads/processes. Furthermore, it is crucial to provide threads and processes with separate portions of data to ensure a proper mapping phase and reducing phase. In this project, the mapping phase accounts for the sorting portion in multithreading and multiprocessing and the iteration portion in synchronization. The reduce phase accounts for the merging portion in multithreading and multiprocessing and global maximum buffer writing in synchronization.

This project faced a couple of challenges. Initially, it was difficult to implement the sliced arrays to the threads function. However, after slicing the array, the function seemed to function

normally. Another challenge was implementing the thread function. Based on the debugging suggestion from a Google search AI Overview, it seemed best to implement 2 merge sort functions, one as a thread function which organizes the arguments of the thread and another regular recursive merge sort function (which receives its arguments within the thread function). This change helped to make sure the arguments were properly passed to the recursive merge sort calls.

To further improve and optimize this project in the future, I would be interested to see the performance of this code applied to a specific dataset. Additionally, to further optimize this code, the use of more threads or processes could potentially create any even faster CPU time. Overall, this project yielded good results.

References

BARTWALL, U. (2024, October 11). *Generating random numbers in a range in C*. GeeksforGeeks. https://www.geeksforgeeks.org/c/generating-random-number-range-c/

CMPSC 472 – Lectures and Google Colabs 3, 4, 6

Google search AI Overview – used for syntax questions, comparison of function arguments, debugging errors and explanations, and suggestions for solving bugs

Kartik. (2025a, July 23). *C program for Merge Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/c/c-program-for-merge-sort/

Kartik. (2025b, September 8). *Thread in operating system*. GeeksforGeeks. https://www.geeksforgeeks.org/operating-systems/thread-in-operating-system/

Kartik. (2025c, September 22). *Memcpy() in C*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp/memcpy-in-cc/

Kartik. (2025d, October 3). *Merge sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/merge-sort/

Kartik. (2025e, October 18). *Merge sorted arrays*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/merge-k-sorted-arrays/

Linux manual page. Man(1) - linux manual page. (2025, May 2). https://man7.org/linux/man-pages/man1/man.1.html

Michael. (2009, August 30). *Multiple arguments to function called by pthread_create()?*. Stack Overflow. https://stackoverflow.com/questions/1352749/multiple-arguments-to-function-called-by-pthread-create

Rana, S. (2024, February 16). *Merge sort using multi-threading*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/merge-sort-using-multi-threading/

temptemp324. (2024, April 15). *Concurrent merge sort in shared memory*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/concurrent-merge-sort-in-shared-memory/

W3schools.com. W3Schools Online Web Tutorials. (n.d.-a). https://www.w3schools.com/c/ref_math_fmax.php

W3schools.com. W3Schools Online Web Tutorials. (n.d.-a). https://www.w3schools.com/c/c_structs_pointers.php

W3schools.com. W3Schools Online Web Tutorials. (n.d.-a). https://www.w3schools.com/dsa/dsa_algo_mergesort.php