Project 6

Due Apr 14 by 11:59pm **Points** 19 **Submitting** a file upload
Available after Mar 30 at 12am

CS537 Spring 2023, Project 6

In this project, you will write a C application to perform parallel sorting.

Administrivia

- **Due Date** by April 14th, 2023 at 11:59 PM
- Questions: We will be using Piazza for all questions.
- Collaboration: This is a group assignment. It can be solved individually, or in a group of 2 students. We highly recommend doing this in a group of 2. **[IMPORTANT] Complete the canvas project 6 quiz (https://canvas.wisc.edu/courses/330415/quizzes/438272) to tell us about your group**.
- Copying code (from others) is considered cheating. [Read this](http://pages.cs.wisc.edu/~remzi/Classes/537/Spring2018/dontcheat.html (http://pages.cs.wisc.edu/~remzi/Classes/537/Spring2018/dontcheat.html)) for more info on what is OK and what is not. Please help us all have a good semester by not doing this.
- Some tests *will be* provided at `~cs537-1/tests/P6`. Read more about the tests, including how to run them, by executing the command `cat ~cs537-1/tests/P6/README` on any lab machine. Note these test cases are not complete, and you are encouraged to create more on your own.
- Handing it in: Copy all your files to `~cs537-1/handin/login/P6` where login is your CS login.

Parallel sorting

Sorting, or alphabetizing as you called it as a child, is still a critical task for data-intensive applications, including databases, spreadsheets, and many other data-oriented applications. In this project, you'll be build a high-performance parallel sort and evaluate its performance!

There are three specific objectives to this assignment:

- To familiarize yourself with the Linux pthreads.
- To learn how to parallelize a program.
- To learn how to program for high performance.

Background

To understand how to make progress on any project that involves concurrency, you should understand the basics of thread creation, mutual exclusion (with

locks), and signaling/waiting (with condition variables). These are described in the following book chapters:

- [Intro to Threads](https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf (https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf))
- [Threads API](http://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf))
- [Locks](http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf))
- [Using Locks](http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks-usage.pdf))
- [Condition Variables](http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf))

Read these chapters in order to prepare yourself for this project better.

A **multi-processor machine** is one that can run more than one thread simultaneously. To get the number of processors your machine has, run `nproc` on Linux, or `sysctl -n hw.ncpu` on Mac. You will need a multi-processor machine to see the difference in performance of your parallel sort. (Most of all lab machines are!)

Parallel algorithms exploit this fact by dividing their instruction execution into different threads, and then combine all the individual outputs to produce the final result.

Project Specification

Your parallel sort (`psort`) will take three command-line arguments.

- **input** The input file to read records for sort
- **output** The output file where records will be written after sort
- **numThreads** Number of threads that shall perform the sort operation.

`prompt> ./psort input output 4`

The input file will consist of records; within each record is a key. The key is the first four bytes of the record. The records are fixed-size, and are each 100 bytes (which includes the key). A successful sort will read all the records into memory from the input file, **sort them by key**, and then write out the “key, value records” to output file. The exact implementation of the parallel sort operation is left to you, the **only constraint** we impose is that your code should be a parallel algorithm.

An example input and output file are given below.

input

```
aaaavvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
ccccyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
```

```
bbbbxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

output

```
aaaavvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
bbbbxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ccccyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
```

In the above example input, the following are the key, value records,
1. key = aaaa, value = vvv..v (96 times)
2. key = cccc, value = yyy..y(96 times)
3. key = bbbb, value = xxx..x(96 times)

Obtain key for each record using the following code snipped, `int key = *(int*)b.` Where b is a pointer to a 100-byte record. **[Important] Ordering of keys is to be performed using signed integer comparison.**

Use the following command to generate a 10^6 B file with random data. Read about the command line utility [dd (https://linux.die.net/man/1/dd)](https://linux.die.net/man/1/dd) to understand more about the arguments.

```
dd if=/dev/urandom iflag=fullblock of=test.img bs=1000000 count=1000
```

You also have to force writes to disk by calling `fsync()` on the output file before finishing.

You can assume that this is a one-pass sort, i.e., the data can fit into memory. You do not have to implement a multi-pass sort.

You can also assume that the input file will always be multiples of 100 in size.

Grading & Submission Details

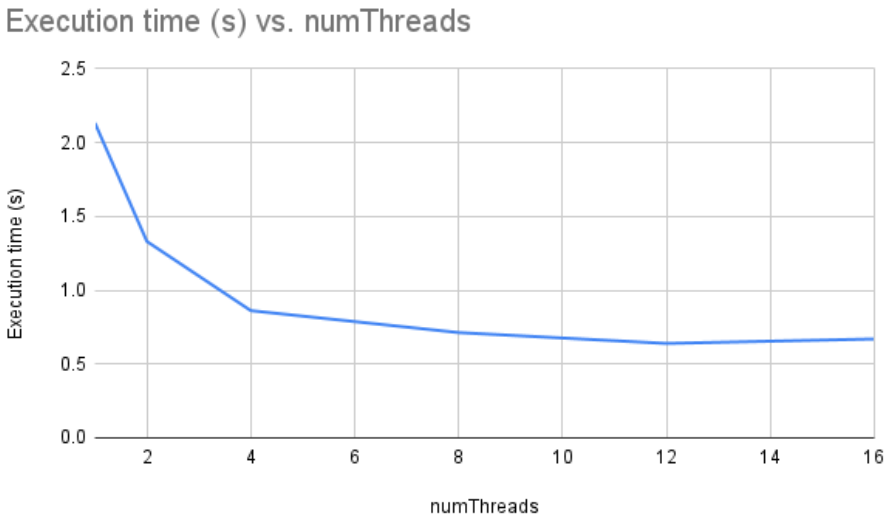
You will write your code in `psort.c` which should compile (and should be compiled) with the following flags: `-Wall -Werror -pthread -O`.
The last one is important: it turns on the optimizer!
In fact, for fun, try timing your code with and without `-O` and marvel at the difference.

Your code will first be measured for both correctness, i.e, ensuring that it sorts input files correctly and performance.

To demonstrate performance, measure your code's execution time (**for only the sort operation**) to perform sort of a large file (say 10^6 B) with varying **numThreads** (=1, =2, =4, =8, =12, =16) on a multi-processor machine **multiple times**. Do NOT include the I/O execution time for this measurement!

Then, plot the average of your results either by hand or using any software to get a graph like below. **Be sure to include the input file size used for sorting**. We expect your execution time to generally

decrease with increasing numThreads. (Food for thought: Can increasing **numThreads** ever increase execution time?)



Include `psort.c` in your CS handin directory and upload the above execution-time vs numThreads graph to canvas.

Hints and Considerations

While this project is open-ended in terms of implementation, here are a few considerations best taken into account while you code.

- **Which algorithm to use.** While parallelization will yield speed up, each thread's efficiency in performing the
  sorting is also of critical importance. Recollect your favourite sorting algorithms (BubbleSort, QuickSort, MergeSort etc.). Which one is better for sorting large files? And which among them is easier to parallelize? Additionally, you can also get some inspiration from papers like the famous [AlphaSort paper](**https://www.cs.cmu.edu/~natassa/courses/15-721/papers/P233.PDF (https://www.cs.cmu.edu/~natassa/courses/15-721/papers/P233.PDF)**)
- **How to access the input/output files efficiently.** On Linux, there are many ways to read from a file, including C standard library calls like `fread()` andraw system calls like `read()`. One particularly efficient way is to use memory-mapped files, available via `mmap()`. By mapping the input file into the address space, you can then access bytes of the input file via pointers and do so quite efficiently. Similarly, how you write the output, and perhaps, how you overlap writing with sorting, can make your sort run faster.
- **Where to run and test your code.** All the CSL machines are multi-processor machines with 12 cores which you can use for this project. Consider generating input files and evaluating your `sort` performance in `/nobackup` folder of these machines because of local I/Os.