CS221 Lab19 – Multithreading and Mergesort

Paul Haskell

INTRODUCTION

For this lab, you'll only have to write about 15 lines, of code, but they are some tricky lines! You will take a single-threaded program and change it so it uses multiple threads. Hopefully this will make the program even faster.

Merge sort

In CourseInfo's **Lab19** directory, you will find a program **merge.c** and a few helper files. This program generates a bunch of random integers and uses the "merge sort" algorithm to sort them from smallest to largest. Merge sort is a really fast, efficient algorithm. But we want to make it even faster!

Your program

The C language does not have multithreading support built in. Luckily, our Git BASH environment gives us the "POSIX library" that does support multithreading.[1]

Your program should be called **fastmerge.c** , and you can use the helper files also. Start with **merge.c** . Review the code to find what steps you want to run in parallel. Where can you find two expensive operations that happen back-to-back, on independent data?

- Please recall from class how to run a method in a new thread, in parallel with the operations of the current thread.
- Also how recall how to have our current thread wait for another thread to finish. For the parallel merge sort, where do we need to wait?

Give things a try—does the sort still work properly?

Details

There's a pretty good chance that you did everything perfectly up to now, but your program still failed. Merge sort is recursive, and if you start creating new threads recursively, you will probably create **a lot** of threads. Computer operating systems limit the number of threads in any one process, and Merge sort might violate that limit.

What do we do?

- When you want to run two operations in parallel, do you need to create two new threads or only one new thread? (Don't forget about the thread that is already running.)

[1] POSIX is a standard that defines capabilities Operating Systems should provide to programs. Most OS's follow "most of" the standard.

- Even that won't help us with a big sort. Please extend your code so that it keeps track of how many threads have been created but not yet destroyed…
- …and only create a new thread if the total number of threads in the current process is ≤ 10. What to do if we cannot create a new thread? Just do all the operations in the current thread!

Now does your program run? Please test it by running it several times.

One more thing…

Please use the `clock()` method to measure how many clock ticks are required to complete your top-level sort. Write that down, then change the maximum number of threads that your program is allowed to create. Try values from 1 through 10, and write down the number of clock ticks for each. Save the results in a file called **timing.txt**. It should look like

```
1 15
2 14
3 16
…
```

with all the "max thread counts" from 1 to 10, and your actual clock tick values. Interesting, huh? (For each maximum number of threads, you might want to run several times, keeping the smallest value.)

CONCLUSION:

This lab shows a common use of multithreading—to speed up a costly computation by running parts of it in parallel. Hopefully you got a sense of how simple the code is, and how complicated the design can be.

Please push your **fastmerge.c** and **timing.txt** to GitHub before the project deadline. The deadline for completion of Lab19 is 11:59pm Monday April 14.

Task	Score, points
fastmerge.c compiles, runs operations in parallel, and gives correct results	15
Software and design quality, as judged by the grader	10
timing.txt has proper format and reasonable values	5