Homework 5 - 7

Peter Gifford, Ren Wall, Madison Hanson, Kyle Brekke

due: 6 December 2019

This fifth homework assignment is due on 6 December 2019, and should be submitted to BOTH Gradescope and D2L.

In this homework, you must investigate the question: *how does the optimal number of threads to use compare across two different programming languages?* You should pick a algorithm that uses threads on multiple cores/processors, and either implement it or find implementations in two different languages. You should design how to compare the "optimal" number of threads (do you fix your input size n? if so, at what value and why? what computer do you run it on?) The deliverable is a polished write-up summarizing your findings. It should probably have the following components:

- Description of the problem the algorithm defines.
- Description of the algorithm, most likely using pseudocode.
- Any references used! Links to git repos, for example. Give credit where credit is due!
- Description of your experimental set-up (what computer? how many cores?)
- Description of methods for comparison.
- Most likely a table or graph to demonstrate your findings.

Note: Since two of the first $n = 5$ homeworks are dropped, some individuals might not submit this homework. As such, you are welcome to combine / change groups for this last assignment, if needed.

1 Merge Sort

This is the commonly used merge sort algorithm. This is a divide and conquer algorithm that takes a list of unsorted values and returns a sorted list in a speedy O(nlogn)

2 Pseudocode

Below is the generic version of merge sort with an added description of where the threads will be generated based on the java implementation found in the washington.edu code listed below.

```
procedure MergeSort(a, threadCount) ▷ a - list of values to sort, threadCount - amount of thread to use
    if threadCount ≤ 1 then
        mergeSortNoThread(a)
    elsea.length ≥ 2
        left ← a[0, a.length/2]
        right ← a[a.length/2, a.length]
        MergeSort(left, threadCount/2)                    ▷ Create thread to start this
        MergeSort(right, threadCount/2)                   ▷ Create another thread
        Merge(left, right, a)
    end if
end procedure
procedure MergeSortNoThreads(a)                           ▷ a - list of values to sort
    if a.length ≥ 2 then
        left ← a[0, a.length/2]
        right ← a[a.length/2, a.length]
        MergeSortNoThreads(left, threadCount/2)
        MergeSortNoThreads(right, threadCount/2)
        Merge(left, right, a)
    end if
end procedure
procedure Merge(left, right, a)          ▷ left, right - lists of values to be combines, a - total list to be referenced
    i1, i2 ← 0
    for i ← 0, 0 < a.length, i ← i + 1 do
        if i2 ≥ right.length||(i1 ≤ left.length&&left[i1] ≤ right[i2]) then
            a[i] ← left[i1]
            i1 ← i1 + 1
        else
            a[i] ← right[i2]
            i2 ← i2 + 1
        end if
    end for
end procedure
```

3 Links to original code

https://www.geeksforgeeks.org/merge-sort-using-multi-threading/

https://gist.githubusercontent.com/georgepsarakis/7a7cdaedeaacdb46124fbb38047cacfe/raw/00bc71cb607e45bb9ffa7165e743
merge-sort.py

https://courses.cs.washington.edu/courses/cse373/13wi/lectures/03-13/MergeSort.java - Used as reference for sudo code then actual code did not work.

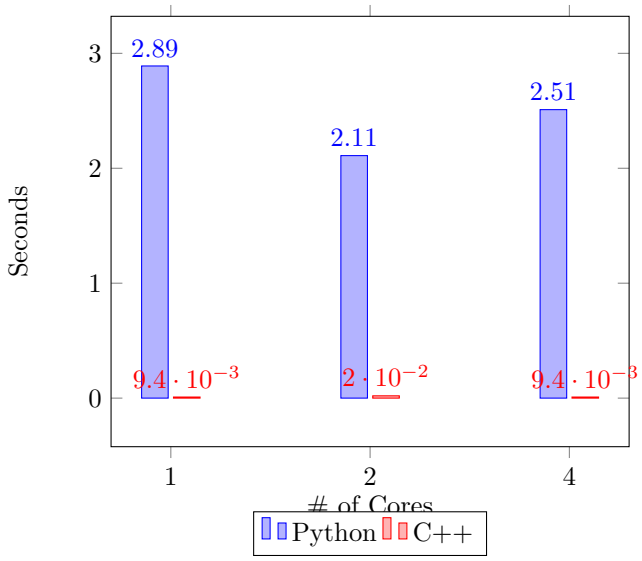
4 Experiment setup

We set up this experiment by first choosing one computer to run everything on. We choose to use a macbook pro 2015 with a 2.7 GHz Dual-Core Intel Core i5 and 8 GB of RAM. Once this was setup we setup the code. Then we took the code provided by the sources above and set them up with system clocks so that we could record how fast the operations take. We ran the algorithms on a set for one hundred thousand random numbers.

5 Methods Used

Using system timers we tracked the time it takes for a merge of the same numbers to execute. We then adjusted the amount of threads to one, two, and four threads because this is how many cores my computer has and we want even numbers because this is splitting the values evenly. Using the random number generators from the respective languages, c++ and python, we ran each set of thread amount five times and averaged the resulting times to get a final average times.

6 Tables



Looking at these results they are very odd. With python having more cores does decrease the amount of time from a single core. However once it is split into 4 cores it takes more time. We assume this is because the code breaks the operations into more threads at each merge step so two threads were being used to do one recursive step and two more threads are doing all rest of them. This combined with the time it takes to start a new thread are most likely the reason the time went up using more cores.

For the C++ code the time increased then went back down most likely because of the thread start cost versus the amount of work they can put in which took it down once there were more cores being applied.