HPP; Threaded Merge Sort

Casper Smet

February 21, 2020

Contents

1 Designing a threaded merge sort

1.1 Single-thread merge sort

The first step to designing a threaded merge sort, is designing a single-threaded merge sort. Single-threaded merge sort takes the following steps[1]:

1. Recursively split the array into two sub-arrays of semi-equal length [2] until there are n sub arrays of size 1.

2. Jump up one level of recursion, and merge and sort the two sub-arrays l and r. In order to do this, one must compare the items in l and r. Do this recursively until all sub-arrays have been merged.

Merge sort can then be divided into two steps;

- Split, &
- Merge

[1]Based in part on https://medium.com/karuna-sehgal/a-simplified-explanation-of-merge-sort-77089fe03bb2

[2]Unless the array has an even amount of items, this will result in odd-sized sub-arrays

The *Split*-step does not seem to be particularly computationally intensive[3], as its time complexity is only $O(n)$. The *Merge*-step seems to be the more intensive step of the two.

1.2 More threads

1.2.1 When to start more threads

Two options come to mind on when to start threads:

- In the recursive function call
- Before the sorting starts

The first one being the more eloquent (and probably faster) variant, the second one being easiest to implement.

The main challenges with the first variant are, how to dynamically generate threads, how to reuse threads, and how to retrieve and recombine the data from each thread.

The second method does not really have to worry about these issues. The second method splits the data up into k sub-arrays, runs merge sort on each sub-array, and then merges them using a reduce. This is, essentially, data parallelism, the distribution of data across different nodes.

For this assignment, I have chosen to implement the second method.

2 Time complexity

The time complexity of the generic merge sort variant, is $O(n \log n)$. This needs to be altered slightly.

First of all, this variant of merge sort splits the array into k[4] semi-equal sub-arrays. This operation adds a k to our big O.

Next, it performs regular merge sort k times concurrently, essentially dividing each unit of n by k.

Finally, it merges the k sub-arrays sequentially. If each item is compared to each item, this adds another n^2, ensuring an exponential big O.

This results in the following time complexity:

$O(k + \frac{n}{k} \log \frac{n}{k} + n^2)$

3 Communication overhead

There is no real communication between threads; the main (driver) thread waits for all k threads to complete until it returns the sorted array.

It merges the sorted sub-arrays in the order that their sorting started, this is a possible bottleneck. For example, if thread three finishes before thread two

[3]Especially when using NumPy

[4]Amount of threads

does, the main thread does not merge the sub-arrays from thread one and three, but waits for thread two.

4 Implementation

The threaded merge sort described earlier was implemented in Python 3.7.6 using the threading module. NumPy was used for arrays and array operations.

See `https://github.com/Casper-Smet/High-Performance_Programming/blob/master/3_threaded_merge.py`.

4.1 Run time analysis

In order to truly test my hypothetical time complexity from **Time complexity**, some run time analysis is necessary.

I generated 100 random arrays of length 10,000, for each amount of threads in $[1, 2, 4, 8]$ threaded merge sort was executed. These were the results:

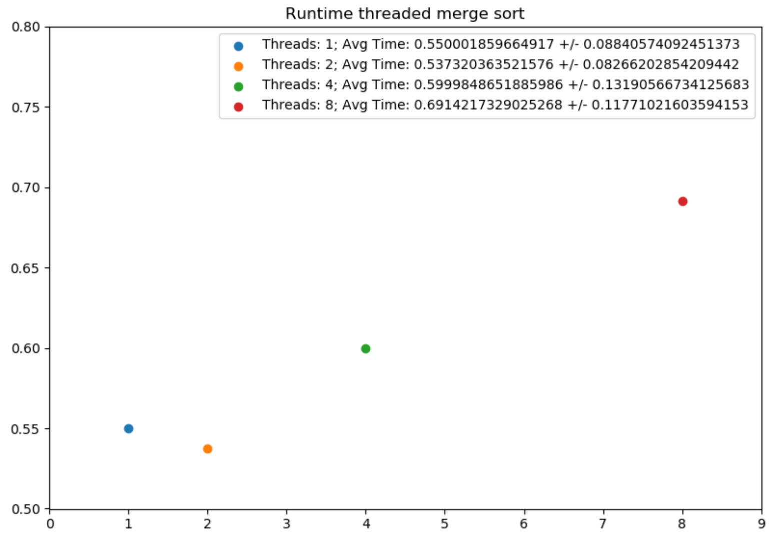

Figure 1: Run time of threaded merge sort

It appears that, only when two threads are used, any time is gained. The two thread version, is essentially method one as described in **More threads**. Two reasons come to mind as for why the versions with more threads are slower than the single-threaded variant.

- The creation, starting, and joining of threads takes time.

- Splitting the array in k sub-arrays is serialised
- The merge step when each thread is finished, is inefficient and serialised

The first reason is not something you can do much about, this is simply a limitation of Python. The second and third reasons are solvable.

In parallel programming operations can be divided into parallelisable and serial (or unparallelisable) operations. Both the second and third reason, are caused by the fact that they are parallelisable, but have not been.

In order to win any real time, both of these must be made into parallel operations. When this goal is reached, you have implemented method one from **More threads.**