Merge Sort is arguably the most practical and efficient comparison-sorting algorithm. Quick Sort can theoretically be much faster, but it’s not consistent. Quick Sort could be on the order of n^2, while Merge Sort will always be on the order of n \* lg n.

For each recursive call, the work is split equally among two other recursive calls with half of the previous one’s input.

n

n/2 n/2

n/4 n/4 n/4 n/4

n/8 n/8 n/8 n/8 n/8 n/8 n/8 n/8

…

1 1 1 1 1 …. 1 1 1 1 1

If we make the assumption that n is a multiple of 2, then we can solve for an exact value of the height of this tree:

n / 2h = 1 n = 2h h = log2n

So we now know the height of the tree, but we also need to know the amount of work one each level to find the total work performed. If you sum up the values per level, it is always n. So the total work done is n \* log2n.

This recurrence tree could also be represented by the following relation:

Θ(1) if n = 1

T(n) = 2 \* T(int(n/2)) + Θ(n) if n > 1

As previously proven, the fastest a comparison-sort algorithm can be is n \* lg n. However, that doesn’t mean we cannot reduce the runtime on a practical level. There are many ways we could reduce the runtime of Merge Sort, while of course still maintaining it’s overall complexity of n \* lg n. What if we used the power of parallel processing to divide the work needed for Merge Sort among several processes that can work simultaneously? A general guideline is to create one process per CPU in your machine. In this case, my machine has 4 CPU’s, so I created 4 processes that worked in parallel. I feel the need to clarify the difference between threads and processes. Threads are more simple and small “subprocesses”, they are simply an individual flow of execution. An important note is that threads share the exact same memory space, which will be important to note for later. A process is rather large, they are also an individual flow of execution, but multiple processes do not share a memory space.[[1]](#endnote-1) In the case where we are, in essence, trying to have multiple mergeSort algorithms running at the same time, we need to use processes instead of threads. The reason for this is because the most widely used Python interpreter, CPython, implements what is known as a Global Interpreter Lock (GIL). The purpose of the GIL is to prevent multiple threads from accessing or altering the same piece of data at the same moment, which could lead to issues. In essence, the GIL ensures that only one thread is running at the same time, which is the exact opposite of what we are trying to do with this algorithm.[[2]](#endnote-2)

Using the multiprocessing module, we can subdivide the array we wish to merge sort into several segments, that will each be sorted by an individual process. Afterwards, we will have some amount of individually sorted sub-arrays. So, now we pair them up and merge them back together until we get back to a single list, also can be done by multiple processes. Here is a diagram to describe the flow of execution:

[3, 7, 5, 9, 10, 4, 2, 6, 1, 8]

[3, 7, 5] [9, 10, 4] [2, 6, 1] [8]

Process 1 Process 2 Process 3 Process 4

Merge Sort Merge Sort Merge Sort Merge Sort

[3, 5, 7] [4, 9, 10] [1, 2, 6] [8]

Processes are terminated, only 1 main process now

( [3, 5, 7], [4, 9, 10], [1, 2, 6], [8] )

Process 1 Process 2

([3, 5, 7], [4, 9, 10]) ([1, 2, 6], [8])

Merge Merge

[3, 4, 5, 7, 9, 10] [1, 2, 6, 8]

Processes terminated, only 1 main process now

([3, 4, 5, 7, 9, 10], [1, 2, 6, 8])

Merge

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Done!

Pseudocode:

def multiMergeSort(array):

dataset = subdivideArray(array, numCPU) #Split the array into subarrays based on the number of CPU’s in the machine

pool = multiprocessing.Pool(number of CPU’s) #Creates a number of processes all grouped together so we can utilize them easier.

dataset = pool.map(mergeSort, dataset) #Each sub-array of dataset will be mergeSorted by a unique process simultaneously

if len(dataset) is odd: #Want to pair up sub-arrays, so if odd number of sub-arrays, pop one out of the dataset and save it for the end.

extra = dataset.pop(-1)

while len(dataset) > 1: #Keep going until all sub-arrays have been merged into a single array.

j = 0

for i in range(len(dataset) / 2):

dataset[j] = (dataset[j], dataset[j + 1]) #Condense adjacent sub-arrays into one tuple, easily be passed to merge as left-right sides.

dataset.pop(j + 1) #Can discard this value now, shortening the length of dataset.

j += 1

dataset = pool.map(merge, dataset) #Merge the pairs of subarrays into a single, sorted array.

#Repeat until there is only one sorted array left in dataset.

final = dataset[0] #No need for 2D array of 1 element

if extra is not None: #If there was extra, merge to final

final = merge((final, extra))

return final

The multiprocessing module makes this task quite easy. The use of the Pool object, which basically instantiates a certain number of processes all at once. The map() method of the Pool object allows us to pass in a function and some iterable value, in our case a list of subarrays, and perform that function on each element, distributing that work over all of the processes in the Pool object currently.[[3]](#endnote-3) I made the code generalized such that it will function perfectly on machines that have a different number of CPU’s than my own. It is important to note that the version of Merge Sort I have implemented here is out-of-place, meaning that a new array is instantiated to represent the final sorted product, instead of working within the bounds of memory of the original array. An out-of-place Merge Sort is much slower than an in-place Merge Sort, here is a graph of time comparisons for different sized arrays:

A screenshot of a computer

Description automatically generated

When the size of the array is small, the time disparity is negligible. But as n becomes large, at n = 3,000,000 for example, there is a 17 second time difference between these two versions of Merge Sort. With all of this in mind, you may ask, why did I choose to implement a Multi-processing version on the out-of-place version, the slower one, instead of the in-place version? The reason is quite complicated, but it boils down to the intricacies of how parameters and return values work in the multiprocessing library. The Pool object discussed previously, along with the map() method that takes in a function and an iterable value. The nuance to this is the function you pass to map() can only have a single parameter, the way to circumvent this issue for functions with multiple parameters is to place all of the parameters in a tuple, then unpack them inside the function . So let’s compare the parameters and return values of both the in-place and out-of-place versions of Merge-Sort:

**Out-ofPlace:**

mergeSort(array)- takes single array to be mergeSorted as parameter. Returns a fully sorted array, either because the size is 1 or by calling the merge() function

merge(leftSide, rightSide)- takes 2 arrays to be merged together, can easily be turned into 1 parameter by the tuple (leftSide, rightSide). Returns a fully sorted array combined from the 2 subarrays.

**In-Place:**

mergeSort(array, p, r)- the overarching array, and the beginning and end of the section that we are currently examining. Can be combined with a tuple . No return value.

merge(array, p, q, r)- the overarching array, the beginning, middle, and end of the left and right halves to be examined. Can be combined with tuple. No return value

The lack of a return value is why multiprocessing in-place merge sort has problems. I think that when we pass mergeSort a tuple like (array, p, r), or even a list, it might create a new memory address for the array. The version of array being accessed in mergeSort may be identical to the original array, but it is not the same address. And, there are no pointers to this new array so we can’t access it when mergeSort() ends. The value of array remains unchanged before and after mergeSort() ran. I know there is a way to do this, I imagine a solution using global variables perhaps, but I think getting stuck in this caveat isn’t worthwhile for the scope of this project.

With that out of the way, I will strictly be comparing out-of-place Merge Sort to its multiprocessing variant, in-place will be excluded from this analysis. We can experimentally confirm that this multiprocessing variant of Merge Sort is still on the order of n \* lg n. The following are graphs of the sample ratios for both merge sort and multiprocessed merge sort:

Graphical user interface, chart, application

Description automatically generated

One can observe a pretty apparent convergence of the values as n becomes large for both variants of merge sort. So this experimentally confirms that they are both on the order of n \* lg n, but how can we compare the runtimes of these two algorithms? Let’s look at a graph comparing the runtimes:

A screenshot of a computer

Description automatically generated

It can be observed that as n becomes large, the disparity between the normal and multiprocessed variant is significant. At n = 3,000,000, the multi-processed variant is approximately 25 seconds faster. But as you can see, close to n = 0, there is a point where the multi-processed variant is slower than the regular version. This is because the multiprocessed variant has a lot of overhead. You have to create the pool of processes and subdivide the array before you can start the sort itself, which comes with a cost. So, let’s find the exact point in which the faster runtime of the multiprocessed variant outweighs its overhead and surpasses the regular merge sort in efficient runtime. Based on the data, we can see that at n = 100,000, the multi-processed variant is already 0.15 seconds faster. So we can perform a bisection method to find an approximate value of n for which the 2 runtimes are approximately equal. The bisection method program I wrote comparing the runtime of regular and multiprocessed merge sort returned a value of n=67,589, for when multiprocessed becomes faster than regular. Since we are dealing with small values of n, there are a lot of small random things that can make the runtime change, so I decided to run the same value of n 5 times and average them together to get a more accurate time. To see this n = 67,589 in another perspective, here is a graph of the runtimes of merge sort and multiprocessed merge sort from n = 40,000 to n = 90,000, jumping by 200 between each data point.

Chart

Description automatically generated

This graph has some pretty ugly spikes. This is due to, as mentioned earlier, extremely random situations that determine the exact execution time of a program. These spikes aren’t apparent for larger n because the changes are so insignificant, but here those spikes are important. If I were to perform a linear regression algorithm on both of these data sets, I think their intersection would be somewhere between 65000-70000, just as the bisection method program predicted. To summarize this, if the size of the array that needs to be sorted is less than 65,000, use the regular merge sort algorithm. Otherwise, use the multiprocessing variant.

In conclusion, the runtime of a multiprocessing merge-sort is significantly faster than a variant using a single process, despite still being on the order of n \* lg n. I determined that for a machine with 4 CPU’s, a multiprocessing merge sort is faster when the array size exceeds 65,000. This is a very specific set of conditions, but it is good proof of concept. Perhaps by experimenting with altering the number of CPU’s, the number of processes created, different Python interpreters that don’t use GIL or different languages altogether, a general formula could be created to describe the relationship between merge sort and its multiprocessed variant. I would like to analyze the flow of execution in the multiprocessed variant, perhaps leading to a formula relating the runtime of regular merge sort to the runtime with multiprocessing on a certain number of CPU’s and processes. For example, when the 4 processes are created and mergesort individual subarrays simultaneously, that part of the normal runtime would theoretically be ¼ of the single process merge sort. But those 4 processes aren’t created and used throughout the entire program, so it would have to be a summation of separate parts. But this analysis is a good candidate for future research.

1. https://stackoverflow.com/questions/3044580/multiprocessing-vs-threading-python [↑](#endnote-ref-1)
2. https://python.land/python-concurrency/the-python-gil [↑](#endnote-ref-2)
3. https://docs.python.org/3/library/multiprocessing.html#multiprocessing.pool.Pool [↑](#endnote-ref-3)