CS441 – Programming Languages – Clojure Merge Sort Project
Eric Wilson

For this project, I decided to use a merge sort algorithm because it seemed like it would lend itself better to being implemented in a strange new functional language. My journey into this project began with frustrating several hour long attempt to pick up the language naturally from various examples. After that failed, I turned to a several hour long YouTube playlist explaining the basics of the Clojure language. This proved extremely helpful and I learned almost everything I would need to write the code required for this project. I felt confident that I at least understood the syntax and basic functions well enough understand examples and to move forward. I felt like this language had a very steep but very short learning curve. I referred to the official Clojure documentation extensively as I proceeded.

Algorithm

My algorithm proceeds as follows: first, I read the text file into a `PersistentList` converting the strings to integers as I go using the `read-string` function. My `mergsort` function takes the list and applies the `merg` function to two halves of the list which are recursively sorted by mapping the `mergsort` function onto them. To split the list, I chose to use `split-at` function because the alternative, partition, has a tendency to drop values from the list. I chose `quot` because it performs pure integer division and either `split-at` or `partition` would return an empty set if it were given a fractional divisor. It was brought to my attention that 1,000,000 is evenly divisible by 32 and all of its factors so this wouldn't have been an issue for this project, but I chose robustness.

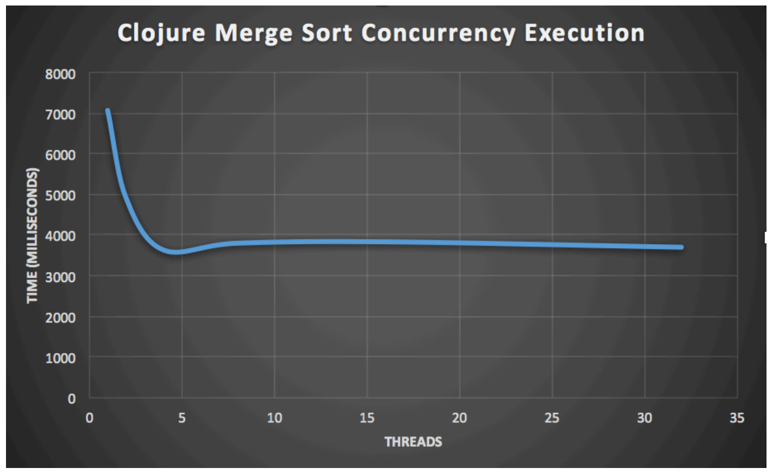
My `merg` function accepts two lists to be merged by moving the least of the two first elements of the lists into a new `PersistentVector`. I chose to use the `recur` function for this because it reuses the stack frame for tail recursion, therefore avoiding an overflow with large sets. It first checks to see if either list is empty. If either one is empty, the other is added onto the end of the new vector. If not, we check to see if the first element of the left list is greater than the first element of the right list. If it is, we recur back into the `loop` given the unaltered left list, all but the first element of the right list, and the new vector with the first element of the right list conjoined onto the end. If the first element of the left list is not greater than the first element of the right list, the opposite `recur` is performed.

To demonstrate concurrency, I borrowed a design idea from a stackoverflow post (cited as a comment in the code) which used a new function for each number of threads wherein each would call the previous one. I thought this was kind of sloppy so I decided to challenge what I had learned about Clojure thus far and turn this idea into a single function that could be given a specified number of threads and would call itself recursively to create the correct number of threads. This function accepts a list of integers and an integer power of 2 to specify the number of threads to be created. It first checks to see if the size of the list is less than two. This would only come up in scenarios where the size of the list was on the same order as the number of threads so it didn't really do anything for the list of one million integers divided into a maximum of 32 threads. This was, again, for robustness. Assuming our list was of sufficient size, we proceed to check if our number of threads is greater than two. If it's not, we simply apply the basic `mergsort` function to a `pmap` (the same as the map function but it splits it into threads) of a halved list. This would happen a maximum of 32 times in the experiment. If the number of threads was greater than 2, that meant that we still had to recur into the `threadulesque-`

`mergsort` again. This was done by applying our `merg` over a two lists onto which we `pmap` `threadulesque-mergsort`. The important part here, is that our function is also `pmap`-ed onto a second parameter for each half. Both of these are the current thread count divided by two. This allows the function to continue branching into new threads for each recurrence and eventually terminate when the proper number of threads is reached.

Results

My implementation was tested using 1, 2, 4, 8, 16, and 32 threads. It was run four times and the average results are displayed in the accompanying graph. The machine it was run on has a dual-core processor with hyperthreading. This effectively gives four cores. The results do, indeed, reflect this. As you can see, the single threaded approach takes a significantly higher amount of time to execute than any of the multi-threaded approaches. The execution time continues to drop until four threads of execution. At this point, the time goes up slightly and then remains relatively constant for the remainder of the experiment. This shows the limitations of concurrency on modern consumer level devices since they typically have between two and eight cores. Higher numbers of threads still run "concurrently" but are forced to share CPU time.



Epilogue

I very much enjoyed this project. In a recent job interview, I was asked about my experience with functional programming and all I could say was that I had some experience with lambda expressions from a course I took in Java but nothing extensive. I believe this project allowed me to refresh and develop my skills and I now have a much greater understanding of functional programming than I did before. I think that UMKC should expose students to functional programming earlier and with more emphasis. I still got the job because I'm awesome and UMKC made me that way. Thank you.

Data

Threads	Avg Time (ms)	Attempt 1	Attempt 2	Attempt 3	Attempt 4
1	7082.217717	6422.063134	7177.41882	7224.65321	7504.735705
2	4913.152455	6939.868219	3932.288614	4771.524878	4008.928107
4	3636.426989	3654.406731	3915.031602	3472.752365	3503.517259
8	3801.671807	3600.101718	3469.789987	4121.757183	4015.038339
16	3836.133351	4408.122749	4044.921324	3402.57543	3488.9139
32	3703.229293	3395.887888	3370.395295	3984.142629	4062.491359