Daniel Ewing

CS 441

Professor Hare

9 December 2018

Clojure Merge Sort Final Project

This short program reads a file of unsigned integers into a list and will sort them using a merge sort algorithm while exploring how parallelization of this process affects runtimes. Writing this code was incredibly difficult as up until this point I had zero experience with functional programming, let alone the Clojure language. Luckily the resources and tutorials on the Clojure website we able to be of assistance. It was there I learned how to populate a list from a file using the slurp function, how to divide a list into two equal sized lists by using the “split-at” function, and how to map subroutines to a specific thread by calling Clojure’s pmap function. With this knowledge I was able to write one variable, the list, and what amounts to three functions to accomplish this task. The actual function count is nine, but one of those is a test function which just sorts and prints the list, and six of the nine functions behave in virtually the exact same manner.

The first function simply slurps a file, and the variable which is created populates itself by calling this “read-file” function. The next function is the “merge-sort” function which takes two lists, and merges them. First three internal variables are created, a new list which will be passed the sorted items, and copies of each list. The function then loops through these lists and conjoins the heads of the copied lists onto the new list if they’re sorted appropriately. Eventually one list will be empty at which point the items from the non-empty list can be concatenated to the end of the sorted list, as they will already be sorted due to the fact that the first time merge-sort is called by another function will be on a list of two items. The rest of the functions behave in a similar manner and include “thirty-two-thread-merge-sort” which is passed a list to sort, and if there are more than two items in the list it splits the lists in half and passes each half to the sixteen-thread-merge-sort function with each half being mapped to a different thread. This function behaves the exact same way and calls eight-thread-merge-sort and so on all the way down to single-thread-merge-sort. This function has a slight difference in its functionality, as it recursively splits the lists into half’s it maps all recursive function calls to a single thread. Once all of the items in the original list have been split into lists of two. The “merge-sort” function finally gets to start acting. Merge sort will sort these small lists into larger lists recursively as the small sorted lists are returned as parameters for their recursive function calls. Once the single thread sort has completed its task, the entirety of the list is sorted if “one-thread-merge-sort” was the parent function, but it not both calls of that function pass their sorted list back up to the two-thread-merge-sort function, which calls merge-sort to sort them, and passes those back up to the four-thread and so on up until the parent is passed the two sorted lists which belong to it, and can then sort them.

Runtime received a drastic improvement on multiple threads as opposed to a single thread, but due to context switching and the overhead that accompanies parallelizing a task this yielded diminishing returns each time the number of threads doubled until there was effectively no benefit. Below is a chart showing the average completion time after running the program five times, along with the time it took to run Clojure’s sorting algorithm. These times do not include file access time as it was subtracted from the runtime in my excel spreadsheet.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Times in milliseconds | | | | | |
| # of threads | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average Times |
| 1 | 7453 | 7256 | 7042 | 6987 | 7154 | 7178.4 |
| 2 | 4323 | 4437 | 3871 | 4056 | 4074 | 4152.2 |
| 4 | 3227 | 2972 | 2646 | 2919 | 2786 | 2910 |
| 8 | 2741 | 2452 | 2484 | 2446 | 2330 | 2490.6 |
| 16 | 2608 | 2258 | 2166 | 2408 | 2106 | 2309.2 |
| 32 | 2582 | 2161 | 2157 | 2251 | 2357 | 2301.6 |
| Clojure Sort | 249 | 202 | 189 | 212 | 164 | 203.2 |
| File Access Time | 160 | 171 | 150 | 150 | 172 |  |

As we can see, even though we have greatly improved runtime, reducing the time it took to roughly a third of running on a single thread, the parallelized merge-sort algorithm can’t hold a candle to the under the hood workings of Clojure’s sorting algorithm. Be that as it may, this project was an excellent look into the workings of functional programming and the Clojure language.