Jacob Bothell

CS441 – Programming Languages

Brian Hare

5 May 2017

Clojure Mutli-Threaed Merge Sort Study

Merge sort is a powerful powerful tool that can help bring together data and organize it for analyzing or storage. The merge sort, being a powerful tool, could be necissary functionality to have in any language, and the unfamiliarity with a purely functional language this seems like a great oportunity to gain some insight into the workings and implementation details of a functional language project. There are a few things that must be decided first: exact problem, input, and language of implementation.

The area of investigation will be the effect of multi-threading on a merge sort routine implrmented in a functional language. The input in question will need to be something large enough to allow easy measurement, in terms of milliseconds, of the difference in the number of available threads for execution. With these concerns in mind perhaps a random list of say 1 million items this should allow the considerations of our problem to be measured. Language of implementation is an obvious consideration to take into account when solving a problem with a computer. Our language of choice will be Clojure as it is a code used in industrial applications, making it worth while to learn, but it is also a rather easy/approachable functional language as it is documneted rather well.

Now that the problem is defined and the way it will be solved is also known there only exists an algorithm to be discovered that fits into our language of choice and also solves the problem at hand. Although the algorithm for merge sort is well known, the generaly known algorithm may not work directly with our chosen language and may also not be possible to implement. Since Clojure is a list processing language and these lists can be seperated and ‘passed’ to another function very easily this makes this problem somewhat easy to solve since that is what merge sort is. The result of this thingking brings the algorithm:

1. Find the middle (bit shift operation)
2. Split into two lists (split-at)
3. Continue to do this(recursively) until there is only 1 item in the list (ensured by the bit shift operation)
4. Now merge these items back into one list sorting as we go sorting them of course (concat)
   1. This takes advantage of the undoing of the stack (created while splitting the list)

Now there is an algorithm and some useful function calls to the programming… Where does Clojure programming take place? Clojure is implemented with Java and so most Java compliers can compile Clojure successfully. Eclipse is a convenient JVM and has a ‘store’ of sorts that allows the direct download and install of the Clojure language.

Now comes the hardest part of any project getting the data file into the project the easiest for Clojure is ‘slurp’ this is a function call that takes a path to a file and reads each line of the file into a location into a list. This was really the only tricky part of the project along with the include statements and their syntax.

Full code can be found at <https://github.com/JacobBothell/MultiThread-Merge> it is fully documented with input files in the correct places.

This implamenttation of the Clojure merge sort does seem to do rather well a.k.a. its much faster than I expected it to be considering the size of input file. The current running times for each thread pool size in this implementation are as follows:

The results are somewhat to be expected. As more threads are introduced the execution time decreases and it apears that there is some sort of limit that we begin to approach because the execution times are closer together with more threads. This is somewhat more evident in the following graph where each run of the program is graphed

This graph shows that there begins to be a platoue of execution speed for the machine. This could be the result of many factors including but not limited to: context switching, speed of processor, and memory speed; all of these and more would most likely slow the execution speed of the threads and begin to platoue the maximum speed.

The one real consideration that I think would also lead to considerable speed changes between thread pool sizes is the fact that the file is read in for each pool size. This gives the single thread execution a considerable slow down because its execution will include the loading of the file into memory, I think, and the rest of them would not necisarily load the pages into memory. However the executions of the different runs were done back to back in the same REPL and with this one would think that the file may persist in memory but perhaps it does not; this would be a good subject of further research (garbage collection/memory management for Clojure and/or REPL).

This project was an interesting and fun investigation into the world of functional programming that has lead to the expansion of my thinking about how programs can be organized and executed. I do not, however, plan to write much in Clojure simply because it is not my cup of tea, but perhaps with more time I could come to apreciate it.