Trevor Klinkenberg

CS461 Programming Languages

Clojure Project

This Clojure project was designed to carry out Google’s page rank algorithm using concurrency by multithreading the computation. Using multithreading observations can be made on the benefits and drawbacks of multithreading depending on how many threads are used concurrently during the execution. At a certain point in multithreading the number of threads allocated for execution outnumber the hardware allocated threads and performance declines due to the overhead management.

In my program I began by allocating two arrays of 10,000 for the page rank computation. One array would hold the output links for each “website” and the other would contain the running computation of the page rank for each site. The file was read in and split on each individual line producing a 10,000-size vector. The page rank array was then initialized with each value set to one as dictated by the algorithm. Next the output array was populated by taking the page link vector, splitting it by spaces and counting the number of tokens. The token counts on each line were used to populate the output link array for use in computation later. Next a multi-threaded loop was used to calculate the page rank. The page list vector is broken down into individual tokens on each line, these tokens are used to get the output list for calculating the page rank for that page using the formula : PR(A) = .15 + .85 \* (PR(A) / OL(A) ) where PR(A) is the existing page rank of page A and OL(A) is the output list of page A. Finally, this algorithm is executed multiple times with different thread allocations and put into a timing call to get the execution time of each. These timing records are outputted to a file with a descriptor of the amount of threads used. The multi-threading was used in this section since it was the most computational heavy part of the code. The use of multi-threading in this section is acceptable as each thread can independently update different page ranks without affecting the overall results. Reflecting back on this execution however I realize there is a need for locks on the data however as a just in case if two threads were to try and update the same piece of data simultaneously. This could introduce a possible race condition that would skew the overall results. However during my testing I did not observe this as the case.

My results showed a decrease in timing until it reached more threads then the hardware I had existed. I ran this program on server hardware that had 12 cores and 32 threads. Once the thread count was increased to 64 I noticed a decrease in performance as reflected in the graph below.

There is a sharp increase during the 8-thread execution that I believe to be an anomaly due to possible background processes existing on the server hardware. The server hardware is a Virtual Machine host and it is possible that one of the virtual machines may have taken some hardware allocation at this point in time.

This project overall was a good introduction into functional languages and the implementation on concurrency. The use of immutable data in a functional language was difficult to overcome at first alongside understanding the syntax of Clojure in general. However, once these concepts are understood it demonstrates the computational power of a functional language in brute force number crunching. Without the overhead of managing side effects of function calls the language can focus its computational power on number crunching. This lends itself extremely well to concurrent execution since the functions do not have side effects and the data is not easily changed. For future projects I could see this being useful for manipulating large datasets quickly with multi-threaded execution.