Clojure PageRank Program

Final Project by Samuel Feye

Table of Contents

Introduction 2
Methods 2
Explanation of methods 2
Multithreading 3
Findings 3
Timing Report 4
Graph 4
Conclusion 5

Introduction

For our final project in the UMKC Computer Science course Programming Languages: Design and Implementation I was tasked with implementing Google's PageRank algorithm using the functional programming language Clojure. I received an explanation of Google's algorithm, a text file containing 10,000 dummy page links and a rubric for grading comprehension.

This project brought with it many challenges and questions to be explored. Learning a new language can be very intimidating in and of itself, but learning my first functional language seemed to be the most difficult obstacle to overcome. In addition to that, learning new concepts such as lazy evaluation, concurrency, and multithreading turned out to enhance my understanding of materials we covered in class. I believe that this project was assigned with those questions in mind so that I could gain additional hands-on experience.

In my program I was able to compute the page rank by reading in the text file and iteratively decipher the ranks. This was done by setting the initial page ranks equal to 1 and using a damping factor of 0.85. We were only required to iterate our ranks 1,000 times in hopes of converging and were required to explore the concept of multithreading to observe and record the variance it created on runtime. This report will be an explanation of my programming process throughout my attempt at completing the tasks described above.

Methods

Explanation of methods

In the attempt to compute the page ranks of the pages given in the file, I first tried many unsuccessful avenues without doing much exploration of the Clojure programming language. In the beginning I relied solely on the "Clojure Cheatsheet" document which provided very little insight into the language since I had little knowledge of the basics. After doing more extensive research on functional programing habits, I ditched my overly complex functions and global variables for a step by step programming approach.

In my program I started by creating 4 global variables; damping-factor: a float with value of 0.85, one-minus-damping, **links-coll**: a sequence of hash-maps mapping the id to the following links, and **ranks-coll**: an atom zipmap of ids mapped to ranks. The **ranks-coll** variable is the most important as it holds all of the initial ranks which are updated after each step is completed. I initialized the zipmap by creating 10,000 keys and setting the values to 1.

I then proceeded to create a systematic approach to calculating the ranks. Reading in the file came logically first, the function is called read-lines: it takes in the file string and reads the file line by line producing a lazy sequence of lines. Then the function link-map separates each line by white spaces using clojure.string/split saving the first element as the id and the rest as links. This then returns a hash-map mapping the two together which I conjoin into the **link-coll** sequence. The methods: update-rank-comp, line-ranks-comp, sum-ranks-comp, apply-damp-comp all take in a parameter id in order to compute the ranks. update-rank-comp takes the current page rank and divides that by the count of links to that page. line-ranks-comp updates the ranks for each page linked to an id. sum-ranks-comp then sums all of those updated ranks and finally apply-damp-comp multiplies by 0.85 and adds 0.15. Files: *pages.txt* and *outputs.txt* can be found in the resources folder in the zipped source code

Then came the issue of iterating through the ids and associating the new ranks to the **rank-coll**. This was done in the methods rank-step which iterates 10,000 times through the zipmap calculating a new rank in each iteration. If the rank has not converged (difference is greater than 0.00001) then that new rank is associated into the zipmap. If the id in **rank-cool** has converged then that item is removed from a sequence and will be ignored during further iterations. In the method number-of-iteration I call rank-step 1,000 times as per directed.

Multithreading

I incorporated multithreading in the number-of-threads method in my project. The method takes in an integer to specify the number of threads to use. Then it sets a variable equal to 1,000 divided by number of threads to determine how many iterations each thread will do (I ceiling that number to assure I will have at least 1,000 total). After that I define a future, **fs**, and for loop by number of threads. In each iteration I use future to call number-of-iterations and then end with a doseq to dereference all of the threads. Dereferencing at the end of the method allows for timing the method and outputting the final ranks.

Findings

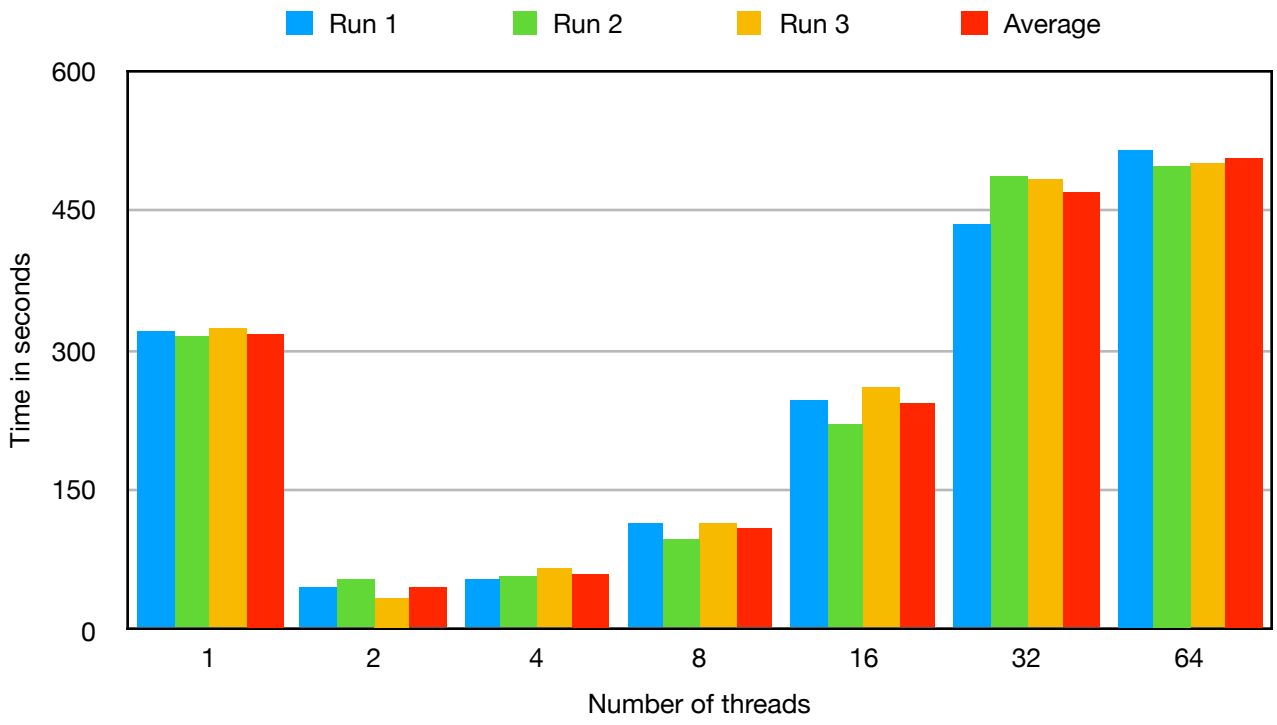
In the final stages of running my project I used the built in timing method to output the duration of the computations using Clojure's spit functionality. The program was ran on a 2015 Mac 2.7 GHz Dual-Core Intel Core i5 with Catalina OS. For lower numbers of threads I found that my program ran much slower than desired and for higher counts of threads my cpu load was maxed out, making it nearly impossible to do anything else while I waited for the results. Running on a single thread, load was around 30% which is similar to the load when I'm browsing the internet. Load percentages were as followed: two threads 60%, four threads

80%, eight threads 100%, sixteen threads 100%, thirty-two threads 100%, and sixty-four threads 100%. The timing results as I increased thread count yielded faster computation times until the eight, sixteen, thirty-two and sixty-four range where load percentages started to max out. The fastest computations were found when utilizing two and four threads. I was able to observe the impact of my program by using third-party software which shows realtime statistics of load, ram, and temperature.

Timing Report

#	1-thread	2-threads	4-threads	8-threads	16-threads	32-threads	64-threads
1	318.8 s	44.3 s	52.5 s	113.7 s	244.6 s	434.9 s	515.9 s
2	313.1 s	53.2 s	56.4 s	95.4 s	221.1 s	486.0 s	498.4 s
3	322.3 s	34.7 s	65.6 s	115.1 s	259.4 s	483.5 s	500.5 s
avg.	318.1 s	44.1 s	58.2 s	108.1 s	241.7 s	468.1 s	504.9 s

Graph



Conclusion

In conclusion my results coincide with the ideal n+1 thread recommendation. Since I was running on a dual core computer it was best to run my program using three threads. This is supported by my results as the two and four thread computations were significantly the fastest.

I was really surprised with these results initially. When I first began tackling this project I just assumed that one thread would be the slowest and sixty-four threads would be the fastest. But I was completely wrong, in fact, computing with two to sixteen threads was faster than a single thread. This is due to the **bottleneck** effect, even though I could have written faster code, I believe that this was the intended result I was to observe. In addition to that I believe we were intended to explore concurrency further by being asked to read and write data simultaneously while multiple threads were running, explore immutable variable, and learning to code functionally.

In regards to other projects, I am way more interested to time them now! I also think it was a really good idea to choose Clojure as the assigned language since it runs on the JVM and a lot of the concurrency methodology is synonymous to Java. I program with Java frequently so now when I run into some tough iterative computations, incorporating concurrency might be a potential option to explore. Overall I enjoyed the project, I thought it was interesting to learn how to program in Clojure and adding concurrency provided a challenge.