Concurrent and Parallel Systems Report

# Testing

The results of my testing are:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Thread amount | Writer thread amount | Reader thread amount | Results per a Loop | Requests received | Overall result |
| 2 | 1 | 1 | 0.748872 | 2000 |  |
|  |  |  | 0.689270 | 2000 |  |
|  |  |  | 0.688747 | 2000 |  |
|  |  |  | 0.685620 | 2000 |  |
|  |  |  | 0.681718 | 2000 |  |
|  |  |  | 0.670233 | 2000 |  |
|  |  |  | 0.146475 | 2000 |  |
|  |  |  | 0.145281 | 2000 |  |
|  |  |  | 0.152871 | 2000 |  |
|  |  |  | 0.147246 | 2000 | 0.475633 |
|  |  |  |  |  |  |
| 3 | 1 | 2 | 1.444068 | 3000 |  |
|  |  |  | 1.361720 | 3000 |  |
|  |  |  | 1.363540 | 3000 |  |
|  |  |  | 0.284590 | 3000 |  |
|  |  |  | 0.279652 | 3000 |  |
|  |  |  | 0.278569 | 3000 |  |
|  |  |  | 0.281089 | 3000 |  |
|  |  |  | 0.279833 | 3000 |  |
|  |  |  | 0.279911 | 3000 |  |
|  |  |  | 0.281788 | 3000 | 0.613476 |
|  |  |  |  |  |  |
| 3 | 2 | 1 | 0.740600 | 3000 |  |
|  |  |  | 0.692475 | 3000 |  |
|  |  |  | 0.697867 | 3000 |  |
|  |  |  | 0.695597 | 3000 |  |
|  |  |  | 0.703765 | 3000 |  |
|  |  |  | 0.686547 | 3000 |  |
|  |  |  | 0.161824 | 3000 |  |
|  |  |  | 0.164152 | 3000 |  |
|  |  |  | 0.160054 | 3000 |  |
|  |  |  | 0.167064 | 3000 | 0.486994 |
|  |  |  |  |  |  |
| 4 | 2 | 2 | 1.408408 | 4000 |  |
|  |  |  | 1.363587 | 4000 |  |
|  |  |  | 1.362636 | 4000 |  |
|  |  |  | 0.294518 | 4000 |  |
|  |  |  | 0.297281 | 4000 |  |
|  |  |  | 0.297033 | 4000 |  |
|  |  |  | 0.299120 | 4000 |  |
|  |  |  | 0.296629 | 4000 |  |
|  |  |  | 0.294334 | 4000 |  |
|  |  |  | 0.297196 | 4000 | 0.621074 |
|  |  |  |  |  |  |
| 4 | 3 | 1 | 0.713052 | 4000 |  |
|  |  |  | 0.707879 | 4000 |  |
|  |  |  | 0.654045 | 4000 |  |
|  |  |  | 0.699903 | 4000 |  |
|  |  |  | 0.721423 | 4000 |  |
|  |  |  | 0.699052 | 4000 |  |
|  |  |  | 0.176560 | 4000 |  |
|  |  |  | 0.175547 | 4000 |  |
|  |  |  | 0.175476 | 4000 |  |
|  |  |  | 0.177787 | 4000 | 0.490072 |
|  |  |  |  |  |  |
| 4 | 1 | 3 | 2.010643 | 4000 |  |
|  |  |  | 1.958276 | 4000 |  |
|  |  |  | 0.406488 | 4000 |  |
|  |  |  | 0.405957 | 4000 |  |
|  |  |  | 0.403969 | 4000 |  |
|  |  |  | 0.406092 | 4000 |  |
|  |  |  | 0.407383 | 4000 |  |
|  |  |  | 0.404922 | 4000 |  |
|  |  |  | 0.405364 | 4000 |  |
|  |  |  | 0.405952 | 4000 | 0.721505 |
|  |  |  |  |  |  |
| 5 | 3 | 2 | 1.457945 | 5000 |  |
|  |  |  | 1.399949 | 5000 |  |
|  |  |  | 1.339149 | 5000 |  |
|  |  |  | 0.309269 | 5000 |  |
|  |  |  | 0.306476 | 5000 |  |
|  |  |  | 0.309166 | 5000 |  |
|  |  |  | 0.306877 | 5000 |  |
|  |  |  | 0.304170 | 5000 |  |
|  |  |  | 0.307689 | 5000 |  |
|  |  |  | 0.309191 | 5000 | 0.634988 |
|  |  |  |  |  |  |
| 5 | 2 | 3 | 2.146400 | 5000 |  |
|  |  |  | 2.120166 | 5000 |  |
|  |  |  | 0.429859 | 5000 |  |
|  |  |  | 0.429863 | 5000 |  |
|  |  |  | 0.428278 | 5000 |  |
|  |  |  | 0.431528 | 5000 |  |
|  |  |  | 0.432053 | 5000 |  |
|  |  |  | 0.432757 | 5000 |  |
|  |  |  | 0.430573 | 5000 |  |
|  |  |  | 0.427266 | 5000 | 0.770874 |
|  |  |  |  |  |  |
| 6 | 3 | 3 | 2.110604 | 6000 |  |
|  |  |  | 2.082605 | 6000 |  |
|  |  |  | 0.443588 | 6000 |  |
|  |  |  | 0.436924 | 6000 |  |
|  |  |  | 0.439430 | 6000 |  |
|  |  |  | 0.437020 | 6000 |  |
|  |  |  | 0.440351 | 6000 |  |
|  |  |  | 0.441824 | 6000 |  |
|  |  |  | 0.442330 | 6000 |  |
|  |  |  | 0.443834 | 6000 | 0.771851 |
|  |  |  |  |  |  |

The results provided shows that concurrency speeds up the program over many tests and has a speed up on reading information over writing information. And the final graph shows the increase of every 1000 is not making much difference between the time taken and how many requests when you use more threads.

# Implementation

My implementation used two main data structures, being Maps and Thread pooling. I used Maps as my way to store the data passed from the clients POST requests. It uses two Maps, one for the data (key and value) and one for data a key has. For the READ requests the Map just gets iterated over intel till it finds its key plus its position. When POSTing in my server the post thread will assign a number on the end of the category passed for easy search with the READ threads And I used thread pooling because it is easy to manage the threads being used with thread pooling over manual threading. Thread pooling also reduce the number of applications threads which speeds up the program.

The reason I chose Maps as my storage data structure because its very fast and iterating through the data structure. This is because you only must look up for keys which speeds up the search function and reduces the time in a critical section. So, this reduces deadlocks for the program over using vectors or lists. The map is also ordered by the keys which means you can find the values in key order and leave the critical section early if the key is used many times, as you can sort the key to the start of the map. And lastly Maps have a reasonable insertion time of O(Log(N)) when insertion time for a vector is O(n) so if the value being inserted is in the middle of the map it will be faster than a vector but insertion for a linked list is faster with a O(1) but has the other problems stated of a slower iteration time when compared to a Map. To improve my data storage data structure, I could use a Linked Map which is not a primary data structure used in C++ but is used in Java. So using a list<pair<K,V>> and a unordered\_map<k, list::iterator<pair<K,V>>> you can get the same functionality as a Linked Map. The list will be used to keep the data store and for a quick look up you use the unordered\_map to iterator over the list and find if the key is stored in the list and what location it is stored.

The reason I chose Thread pooling as my is because it easy manages the threads the CPU can handle. It reuses threads to get the work done which means you don’t have to waste time and memory creating and destroying threads. A thread pool only creates a new thread if it did not have a thread to execute the new queued task. For some of the tasks I could used single threads since thread pools don’t work well with log running operations and can lead to thread starvation, which will slow the program down since the threads will have to wait for the threads in the pool to free up before stating its operation.