Homework 2 Report

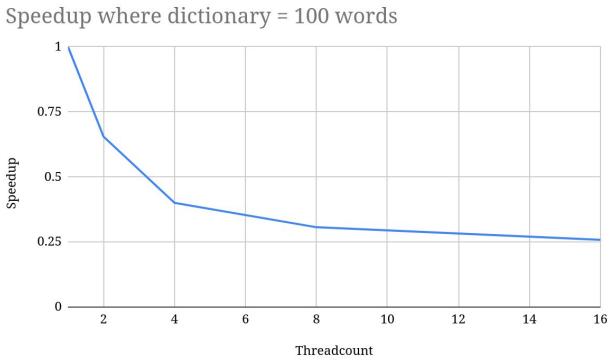
Jacob Hedén Malm
jacmalm@kth.se
980405-1499

For this homework I chose to attempt the palindromes assignment. At first, I planned on reading in all of the words, storing them in a set, and then iterating over all of the words in the set and checking if they are palindromes, or if the reverse of the words could be found in the dictionary. If they could be found, or if they were palindromes, I wanted to store both words in a new data structure, and delete both words from the dictionary set.

I wanted to use a set because I was going to be doing a lot of lookup, insert, and remove operations, and a set is very efficient for doing all of these. The standard implementation of a set in C++ is a red black tree, which guarantees logN search time. I ran into some troubles with this implementation because I could not iterate over the set by index accesses, as the iterator to a set is bidirectional and not a random access iterator. This meant that I could not parallelize the iteration over the set using omp parallel for, and instead had to use tasks. My solution was to create one untied task for iterating over the set. It was untied as this would be the most intensive task by far, and needed to be stopped and continued as other tasks lagged behind. Inside of the for loop, I spawned a new task for each iteration. This task consisted of doing the checking and potential insertion of the word in question.

One problem I ran into with this implementation was that the execution time of my algorithm would go up the more threads I added. I believe this is due to the fact that the smaller tasks that were spawned inside of the for loop were too fine grained, and did not justify the overhead of spawning a new thread. This implementation can be found in the folder marked “DEPRECATED”.

This led me to rewrite the program. I moved from using sets to using vectors to store all the words. I sorted the vector after insertion was done, (not sure why this had to be done, as the input data appeared sorted, but I did not find all the palindromes if I did not sort it), and used a regular for loop, accessing elements by vector[i]. This for loop was parallelized using pragma omp parallel for. I searched for elements in the vector using binary search. This program worked much better when I parallelized it. The execution times for word lists of different lengths can be found below. As we can see, the execution time slows down the more threads we use when the wordlist is 100 words. This is because the wordlist is so small, that the overhead in spawning new threads is not worth it, and a



sequential problem does the best job. As we can see with dictionaries of larger size, the speedup is significant with more threads.

I am assuming that there is a “peak” in speedup at 8 threads, because the hardware architecture of the computer had more natural support for this number. Perhaps the cluster size was 4 cpus with capability of 8 software threads. This would also explain the smaller peak at 4 threads.