OPTIMIZING WORD COUNT USING OPENMP

RADHIKA BAILURKAR

2019430003

SUSHAMA GARUD

2019430006

SHREYAM SHAH

2019430011
MTECH CSE

INTRODUCTION

Finding specific words from a text is a very trivial task for a person to do. But for a computer it isn't. A simple sequential approach would be to check each and word with the specific word and if it is matching then we found that word. But even that is not a very good solution with a very large text file. We found an algorithm that optimizes the approach. We are now optimizing it further by applying the HPC concepts using OpenMP

programming.

REQUIREMENTS

1. Ubuntu Linux system.
2. Openmp libraries.
3. Gnuplot visualizer.

SERIAL ALGORITHM

The serial algorithm works as follows:

It first converts the text that we want to find into ASCII values. We take a random prime number to hash the search text as well as the document text. We find the hash value of the search text and store it.

Then we take each word, convert it into ASCII value and hash it using the same technique used to hash the text.

Then we check if the hash matches with the search text hash. If it does, we check it letter by letter if it matches or not.

PARALLEL ALGORITHM

We have achieved parallelism as follows:

Since the hashing of each of the words is an independent task, once approach we thought was to give each thread a single word to hash and check if it matches. But that is not a good approach because the number of threads should not be very high. A good approach would be that we divide the text evenly amongst a specific number of threads and each thread would hash the text given to it.

We tested this approach with different numbers of threads from 1-60 with increments of 10, to get the optimal number of threads that would be suitable.

Another problem that we faced was the race condition amongst threads to update the

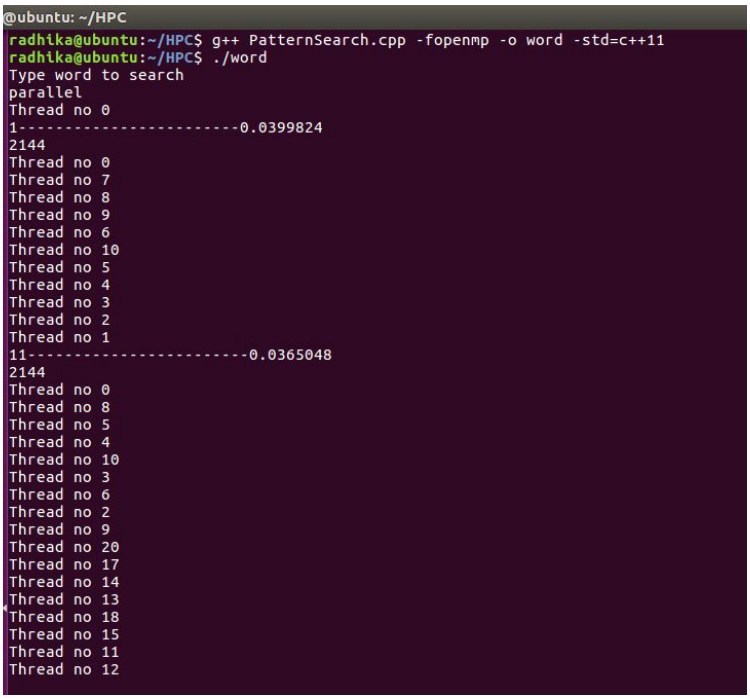
total count found. To avoid that we used a reduction variable available in OpenMP to reduce the count to a single variable.

The number of threads is varied by using *omp_set_num_threads()* method.

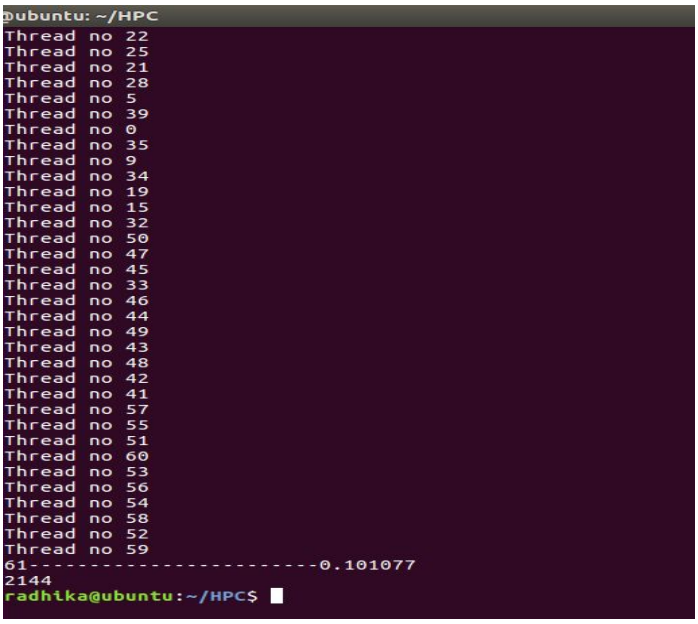
DATA

Test input data was provided which was fed into the algorithm to search words and their respective occurrences.

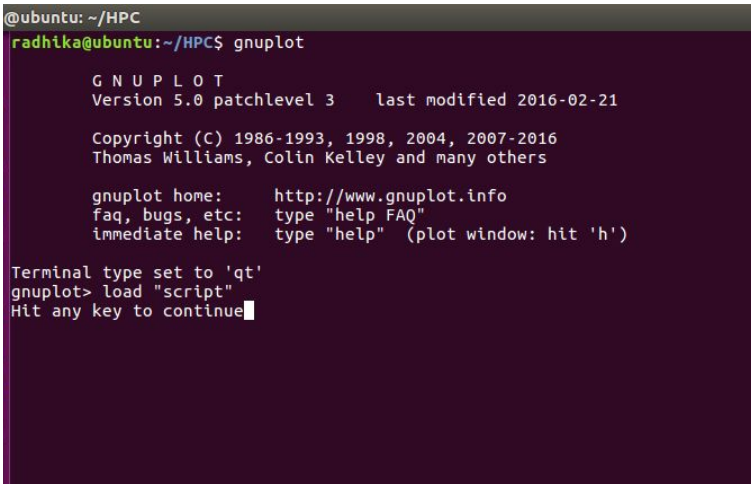
RESULTS



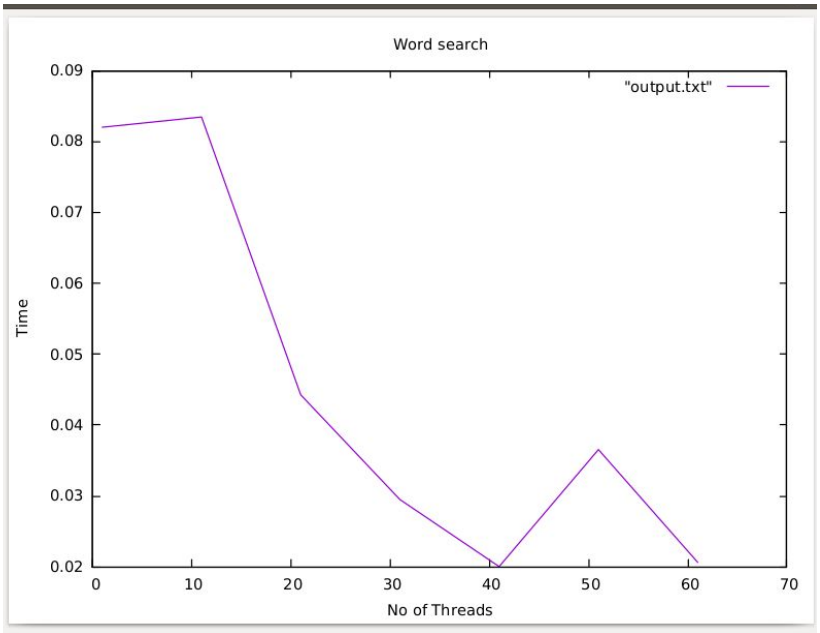
First we will compile and run our program to check whether any errors are present. Then after successful execution, it will tell the user to input the word to search. it will first run in a sequential manner on a single thread. As we increase the number of threads from here on, it will run on multiple threads, ie 11, 21,31,41,etc.



As we increase the number of threads from here on, it will run on multiple threads, ie 11, 21,31,41,51 and 61. Each thread is associated with their respective timestamps and number of occurrences of words found.



Here we have implemented gnuplot. Gnuplot is a portable command-line driven graphing utility for Linux, OS/2, MS Windows, OSX, VMS, and many other platforms.



The above graph tells us about the distribution of Time taken versus Number of Threads which was plotted with the help of output generated.

WORK DONE BY INDIVIDUAL MEMBERS

Sushama Garud: Code for reading the text from a file, taking all necessary inputs from the user, setting the number of thread logic.

Shreyam Shah: Implementation of logic for word count and applying parallel programming concepts.

Radhika Bailurkar: Code for timer's logic. Saving the data on the output file. Plotting of the output graph using GNUPlot.

CONCLUSION

Thus, from the output and the graph, we can say that by keeping the number of threads between 20-60, we get significant reduction in the running time as compared to the sequential algorithm.