

Sonali Patil

4 May 2020

Parallel Computing

Parallel Computing Final Project

The Parallelization of the Boyer-Moore Horspool Algorithm for Text Searching

Problem

Text searching with large amounts of text can take a great deal of time when comparing each character within the text being searched, with each character in the text that is being searched for. This is known as approximate string matching, which is a manner of finding substrings that match a certain pattern within a string. As the number of comparisons can be drastically high based on how large the text, or the main string being searched is, the runtime for such a search program can be immensely long. Other problems that also increase the runtime is finding the approximate position in which a pattern match is found. The position of a pattern match could potentially not be known, causing the system to have to search and find the index where the match occurred and return it to the user. This adds additional time for the search to complete in finding a given piece of text in a large amount of content. Ultimately, this can be a big problem when implementing text searching for a circumstance, such as a search engine or filtering spam, therefore making it essential to find a quicker solution.

While text searching can take a significant amount of time given the method previously mentioned, the speed of text searching in programs has improved with the utilization of string searching algorithms. String algorithms function through working to find the location where substrings occur within a larger string, or text, piece. Examples of such algorithms include the Rabin-Karp algorithm, Knuth-Morris-Pratt algorithm, Bitap algorithm, Two-way

string-matching algorithm, Backward Nondeterministic DAWG Matching, Backward Oracle Matching, and FM-index algorithm. However, the standard benchmark for practical string-search literature, as identified by Andre Hume and Daniel Sunday in Fast String Searching, is the Boyer-Moore algorithm. A simplification of this algorithm is the Boyer-Moore Horspool algorithm, or Horspool's algorithm, published by Nigel Horspool in 1980.

As all other string searching algorithms, the Boyer Moore Horspool algorithm serves as an algorithm for finding substrings within strings through comparing each character within the substring to find the matching characters in the string and return the match. This algorithm is known to be the fastest way to search for substrings and is usually utilized in search bars, auto-corrects, and Big Data searches. The algorithm works to reduce the number of CPU cycles and search time necessary to find a given text. The execution time is linear as it looks at the size of the string being searched which makes it have a lower execution time factor in comparison to that of other search algorithms. The worst-case performance of the Horspool algorithm is also less than other search algorithms as it looks to have a runtime of $O(mn)$ where m is the length of the text being searched for and n is the text being searched. For these reasons, the Boyer Moore Horspool algorithm could be considered one of the best, if not best, algorithms for texting searching.

The Horspool algorithm works to compare an order of characters within a substring with those within a string to find a potential match. When the order of characters from the substring don't match those in the string, the substring jumps to search from the next position in the pattern found through the value indicated within a Bad Match Table. The Bad Match table is created through calculating the value of each character within the substring with the formula Value

=length of substring – index of each letter in the substring – 1. The last character and characters not within the substring are given the value of the length of the substring. To compare the order of characters in a substring with those in a string, you start from the index of the last character in the substring. If the character matches with its string position character, then the preceding characters are compared as well. If it does not match, or any preceding characters do not match, the value corresponding to the last character is found through the Bad Match Table and the substring skips that many given number of spaces indicated. This is done until a total match is found and the substring is located within the string.

While the Boyer Moore Horspool algorithm, a string searching method, does reduce the runtime on finding a match of a substring within a larger string, or text piece, the operation can also be made faster through parallelization.

Design

In terms of parallelization, a program or system will look to process select data in parallel, for the purpose of reducing runtime as calculations can occur simultaneously. By parallelizing the Boyer Moore Horspool algorithm, searching for a given substring within a large amount of text can be done in a shorter amount of time. This is done through dispersing the given data among available processors on a computer where pthreads look to handle multiple flows of work during overlapping durations of time.

To parallelize the Boyer Moore Horspool algorithm for text search (string matching) over large volumes of text, utilizing OpenMP to break large volumes of text will work to improve the scalability of the search algorithm. By breaking up the text and having separate threads handle

each chunk of the separated code, the program can help reduce the run time of searching for a substring within enormous string counts.

For my program, only text files can serve as the first argument for the text being searched. The second argument for my program is the number of threads that the user would like to use to parallelize with. With my parallelization code, I looked to take the entire length of the text, considering it as a string, and divide that text into sections based on the number of threads indicated through user entry. While one may worry that a divide will split a match within the text, the program takes the length of the substring and looks back that amount of characters from where the divides were made. For example, if the substring AB were to be searched within the string EFLKAABBLMAK, the string would first be divided by the number of threads indicated by the user. Assuming that the user would like to use four threads, the string would be split EFL-KAA-BBL-MAK. If the substring were to be searched within any of the sections just like this, no match would be found. Because of the potential for this error, the divide will first move back the length of the substring first and add it to its following section. So the above string would become EFL-FLKAA-AABBL-BLMAK. Now the substring is able to be found despite the threads potentially dividing the match of the substring within the string. From here, Horspool's algorithm runs on each divided section of the string until matches of the substring are found. The positions of found matches are then returned.

Experiments

For my experiment, I chose to look at the speed up of my program to determine whether parallelizing the Boyer Moore Horspool algorithm will reduce the time it takes to search for a

substring within a large amount of text. The data sources that I chose to use for my experiment are various text files of various lengths. These text files include dictionary.txt, sample.txt, shakespear.txt, and small.txt. Within dictionary.txt I searched for the word “dive”, within sample.txt I searched for “abdc”, within shakespear.txt, I searched for “fast”, and within small.txt I searched for “dare”. By searching various text files and searching for words with different occurrence counts, I wanted to see whether there w

To begin, I first started by running all text files with only the use of one thread, or as a serial program to see the amount of time the run would take. After, I ran all the text files again each with 2, 4, 6, and 8 threads and recorded the amount of time the runs took. To calculate the speed-up, I divided the time it took for each multiple thread run by the amount of time it took to run it serially.

Dictionary.txt

Searching for “dive” in dictionary.txt (31 occurrences)

| Number of Threads | 2 | 4 | 6 | 8 |
|----------------------------|-------------|-------------|-------------|-------------|
| T(serial) Time (seconds) | 0.246 | 0.246 | 0.246 | 0.246 |
| T(parallel) Time (seconds) | 0.206 | 0.216 | 0.194 | 0.122 |
| Speed-up Time (seconds) | 1.194174757 | 1.138888889 | 1.268041237 | 2.016393443 |

Sample.txt

Searching for “abdc” in sample.txt (1 occurrences)

| Number of Threads | 2 | 4 | 6 | 8 |
|----------------------------|--------|-------------|--------|--------|
| T(serial) Time (seconds) | 0.0129 | 0.0129 | 0.0129 | 0.0129 |
| T(parallel) Time (seconds) | 0.003 | 0.011 | 0.002 | 0.001 |
| Speed-up Time (seconds) | 4.3 | 1.172727273 | 6.45 | 12.9 |

Shakespeare.txt

Searching for “fast” in shakespeare.txt (199 occurrences)

| Number of Threads | 2 | 4 | 6 | 8 |
|----------------------------|-------------|-------------|-------------|-------------|
| T(serial) Time (seconds) | 2.15 | 2.15 | 2.15 | 2.15 |
| T(parallel) Time (seconds) | 1.96 | 1.88 | 1.89 | 1.43 |
| Speed-up Time (seconds) | 1.096938776 | 1.143617021 | 1.137566138 | 1.503496503 |

Small.txt

Searching for “dare” in small.txt (1 occurrences)

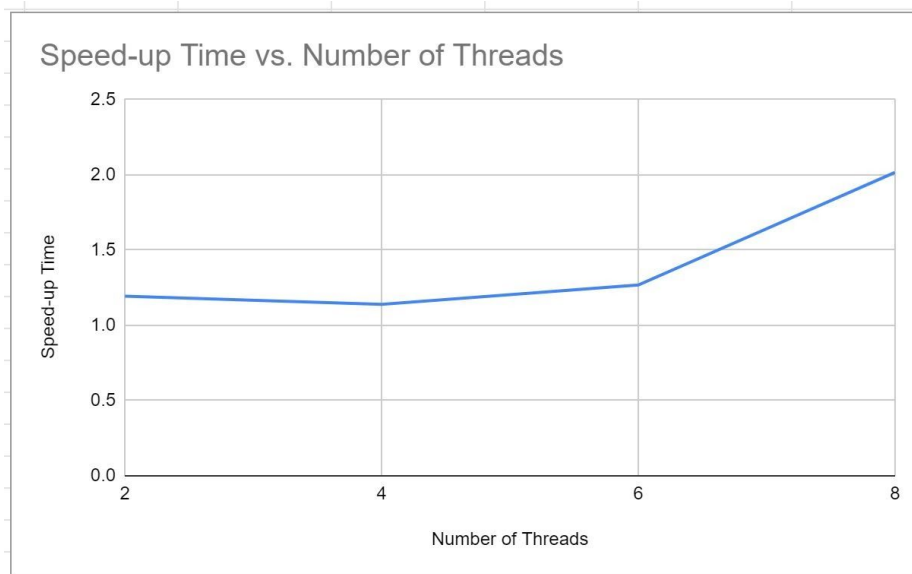
| Number of Threads | 2 | 4 | 6 | 8 |
|----------------------------|----------|----------|----------|----------|
| T(serial) Time (seconds) | 0.000133 | 0.000133 | 0.000133 | 0.000133 |
| T(parallel) Time (seconds) | 3.30E-05 | 2.50E-05 | 2.80E-05 | 1.90E-05 |
| Speed-up Time (seconds) | 4.03E+00 | 5.32E+00 | 4.75E+00 | 7.00E+00 |

Analysis

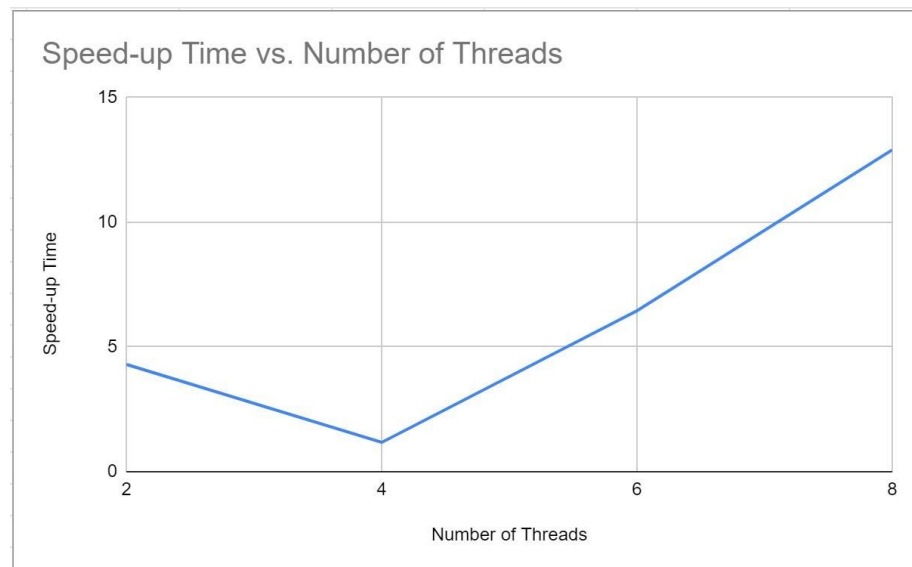
Based on the experiments I saw that the speed-up value for all the tests were greater than one, showcasing that parallelism had given improvement to the runtime of searching for a string within the text content.

Scalability

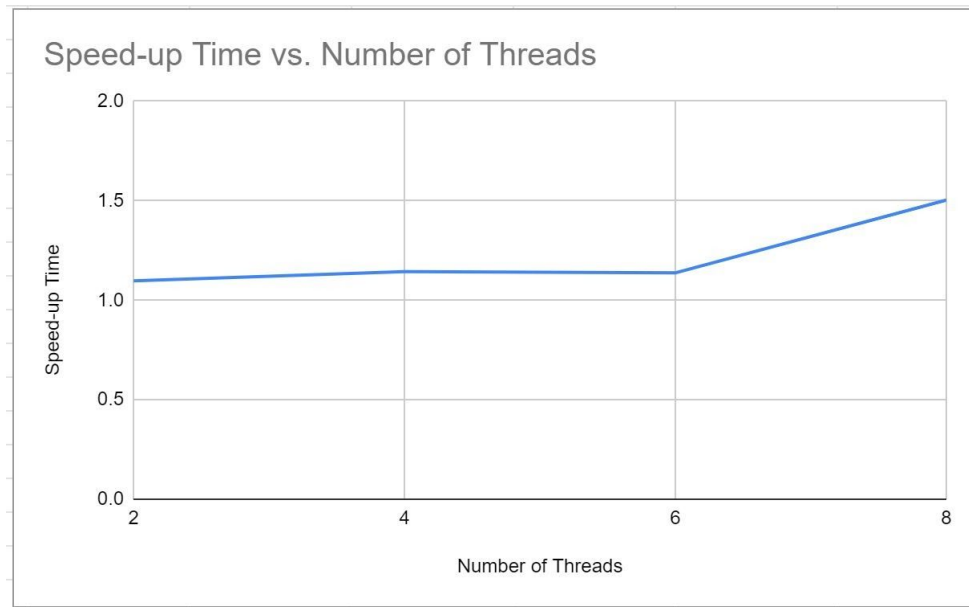
Dictionary.txt- Searching for “dive” in dictionary.txt



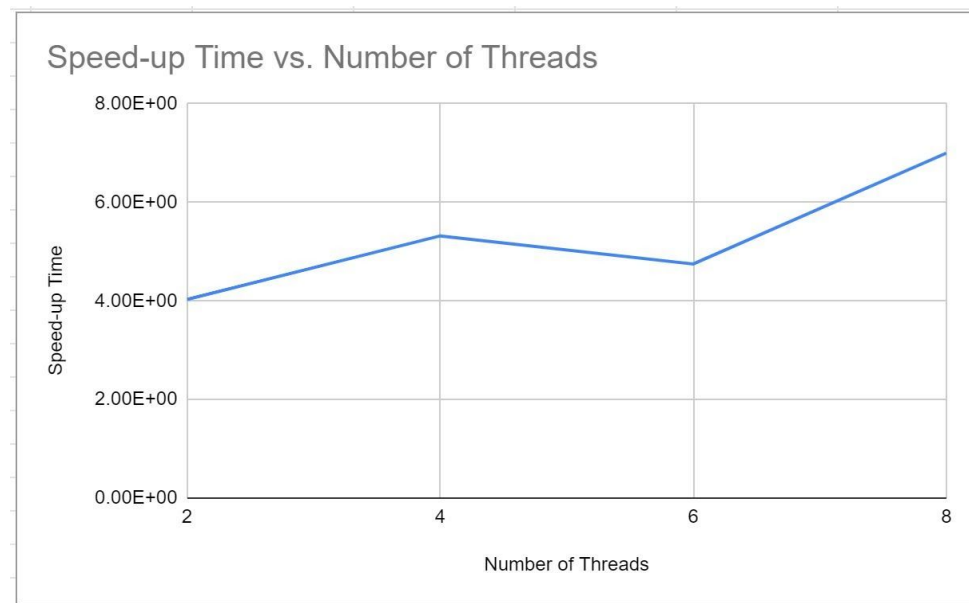
Sample.txt- Searching for “abdc” in sample.txt



Shakespeare.txt- Searching for “fast” in shakespeare.txt



Small.txt- Searching for “dare” in small.txt



Scalability: As the number of threads increases the speed-up increases for all the given text searches, where some hit their max efficiency while others simply continue to grow as a result of their speed-up vs. number of threads relationship. Ultimately, the program stands to be neither strongly or weakly scalable because of being able to handle ever increasing problem sizes.

Efficiency: As the number of cores used to run the program increases, the efficiency decreases, This is seen as the speed up continued to increase when more cores were utilized.

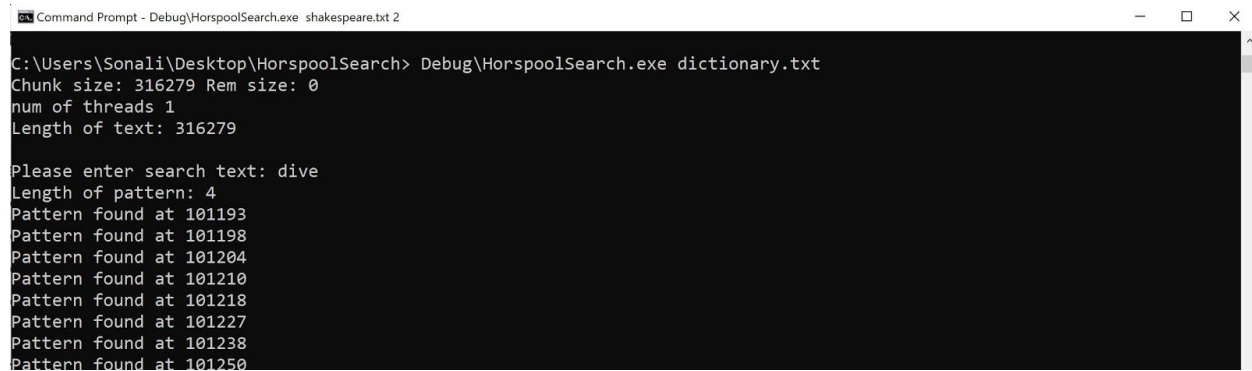
Solution: Based on these results, it can be concluded that by parallelizing the Boyer Moore Horspool algorithm the speed up increases showcasing a quicker runtime and performance in comparison to that of searching for a string serially. The quality of this solution, however, can be questioned as when I ran the program on two different computers with the same number of cores, as well as that of a virtual machine, I got differing speed-up times. For one computer, the speed-up was slower, whereas for another it was faster. When I used the virtual machine with 8 cores, the speed-up again increased to where I chose that as my testing environment. However, I think different testing environments may have an affect on how the program runs, bringing into question the quality of the solution.

Future Improvements: To grow upon the Boyer Moore Horspool algorithm and my program, other parallelization methods can also be considered. For example, looking into splitting the text based on number of lines or number of words (whitespace being counted as separation) can also be tested for whether their speed up is better than that of the current solution. Other string searching can also be considered such as the KMP algorithm which can have a

better runtime than Harpool's algorithm given certain circumstances. Overall, I hope to look further into the manners of improving text searching by using parallelization based on these results and more research.

Implementation

The program can be compiled through the command line or on a virtual machine. To compile through the command line, begin by rebuilding the sln solution located inside of the HorspoolSearch folder. Then open up the command prompt and locate the HorspoolSearch folder within your system. Cd inside the folder and type Debug\HorspoolSearch.exe along with the file you wish to read and the number of threads you would like to use.



```
Command Prompt - Debug\HorspoolSearch.exe shakespere.txt 2
C:\Users\Sonali\Desktop\HorspoolSearch> Debug\HorspoolSearch.exe dictionary.txt
Chunk size: 316279 Rem size: 0
num of threads 1
Length of text: 316279

Please enter search text: dive
Length of pattern: 4
Pattern found at 101193
Pattern found at 101198
Pattern found at 101204
Pattern found at 101210
Pattern found at 101218
Pattern found at 101227
Pattern found at 101238
Pattern found at 101250
```