Burrows-Wheeler Transform Through the Use of Parallel Merge Sort

Angelo Christian Matias, Kurt Neil Aquino, and Roger Luis Uy
College of Computer Studies, De La Salle University, 2401 Taft Ave., Manila, Philippines
{angelo_matias, kurt_aquino, roger.uy}@dlsu.edu.ph

1 INTRODUCTION

The Burrows–Wheeler Transform, also known as BWT for short, is an algorithm used to prepare data for use with data compression techniques invented by Michael Burrows and David Wheeler in 1994. It is done by sorting all rotations of an input text into alphabetical or lexicographical order and taking the last column of the sorted rotations as the output.

A compression technique which utilizes the BWT in order to create a compressed full-text substring index is the Full-text index in Minute space, or FM-Index for short. This compression and search algorithm, invented by by Paolo Ferragina and Giovanni Manzini, is used to efficiently find the number of occurrences of a pattern within the compressed text, as well as locate the position of each occurrence.

This paper compares the runtime and overall performance of the "standard" implementation of the BWT algorithm in Python and its equivalent "parallel" implementation in java through the use of a multithreaded merge sort.

2 IMPLEMENTATION

For the standard implementation, the researchers made use of Python's vast built-in library in order to perform flexible list and array manipulation techniques which the BWT algorithm relies on.

As mentioned in *Section 1*, the first two steps of the BWT algorithm, namely:, listing all of the possible rotations of the input string, and then sorting the list of rotations in alphabetical or lexicographical order, can be performed in Python in a single line, as can be seen in *Table 2.1*.

What the implemented function essentially does is that it creates a sorted list of the indices of the suffixes or rotations of the input string. Python's standard implementation of its `sorted()` function makes use of Timsort which is a hybrid stable sorting algorithm, derived from a mix of both merge sort and insertion sort.

As for getting the last column of the sorted rotations, a simple list and index manipulation can be performed by referencing the generated sorted index rotations list with the original input string.

```
rotations = sorted(range(len(input)),
key=lambda i: input[i:])

bwt = [input[i - 1] if(i > 0) else
input[len(input) - 1] for i in rotations]
```

Table 2.1. Python code for generating the BWT from the sorted rotations

The parallel implementation of the BWT was performed through the use of multithreading in Java. The rotations of the input text were sorted through the use of a multithreaded merge sort. As a divide and conquer algorithm, mergesort divides the problem into subproblems that are independent of each other thus allowing them to be done in parallel.

```
ParallelMergeSort(input,indices,threadCount)
  if(threadCount >= 1)
     mergeSort(input, indices)
  else
  {
     if(indices.length >= 2)
        left = new
Thread(mergeSortThread(input,leftData,threadC
ount/2)).start()
       right = new
Thread(mergeSortThread(input,rightData,thread
Count/2)).start()

       left.join()
       right.join()

       merge(input, leftData, rightData,
dataSet)
   }
```

Table 2.2. Pseudocode for the parallel merge sort algorithm.

For the standard implementation of the FM-Index, the compression technique can be performed by the following steps: Create an array with the resulting BWT, sort the array lexicographically, append each of the characters of the original BWT to the left of the

sorted array, repeat until the substrings being sorted has the same length with the pattern being searched. This can be done in Python as follows:

```
fm_index = sorted(bwt)
for i in range(1, len(substring)):
       fm_index = sorted(fm_index)
       fm_index = [x + y for x, y in
zip(bwt, fm_index)]
```

Table 2.3. Python code for generating the FM-Index

Again, this makes use of Python's versatile built-in array and list operations. Finding the number of occurrences of a pattern within the compressed text, as well as locate the position of each occurrence is as simple as calling the functions:

```
substring_count = fm_index.count(substring)
substring_index = [i for i, j in
enumerate(fm_index) if j == substring]
```

Table 2.4. Python code for counting the number of occurrences and locating the positions of a substring

Considering that most of the FM-index algorithm's computations are heavily dependent towards the results of the computations performed in its previous iterations, doing such computations in parallel would result in redundancies that would make it inefficient and possibly slower than the standard implementation. This applies for both of the algorithm's "count" and "locate" functions as well as it is much more efficient to refer to the previous columns than to manually count and locate the indices of the substring being searched from the start.

With regards to the implementation of the substring search, it could be noted that the current implementation consumes a large amount of memory especially on strings of larger sizes. One proposed solution for this is not to generate the entire occurrence table as a whole but instead use it as a "checkpoint" for every n characters in the string and simply calculating the number of occurrences of the character as needed. The work of Labeit et. al. (2017) have also suggested the use of wavelet trees to reduce the memory consumption in generating the FM-index and calculating the number of occurrences of a given substring. Their work also showed that the use of wavelet trees also allows the FM-index, and with it the Burrows-Wheeler transform, to be generated in parallel also through the use of wavelet trees.

3 RESULTS AND ANALYSIS

For the testing and analysis, it should be noted that only the performance and runtime of the BWT algorithm was compared in both "standard" and "parallel" implementations as the sorting algorithm of the FM-Index, as mentioned in *Section 2*, heavily relies on the outputs of its previous instances, which makes it a sequential program, and therefore impossible to run in parallel.

For the parallel implementation, the maximum number of threads were set to 16 as running too many threads may not only be unnecessary but also increase the overhead which could negatively affect the execution time of the program.

Both the "standard" and "parallel" implementations were run 5 times using a machine with an Intel Core i7-6700HQ CPU, with an input string of varying, exponentially increasing lengths: 64, 8192, 16384, and 32768. The results of which, are recorded in Figure 4.1.

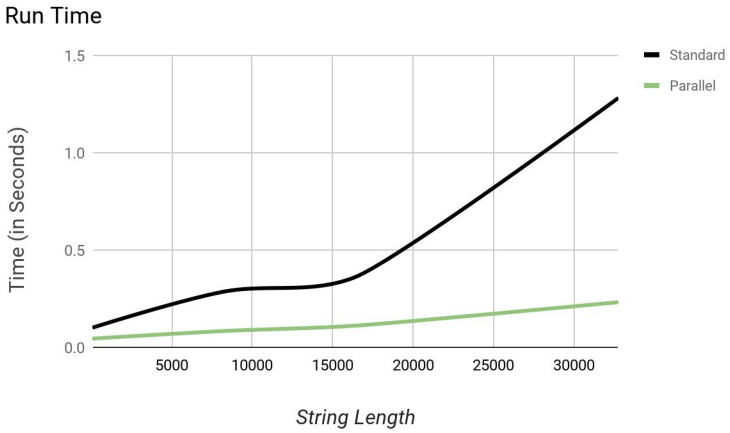

Figure 3.1. Run Times of the Standard and Parallel Implementations of the Burrows-Wheeler Transform.

4 CONCLUSION

From *Figure 3.1,* it can be observed that the "parallel" implementation runs significantly faster than the "standard" implementation, as it maintains a relatively linear runtime, as compared to the "standard's", even as the length of the input string exponentially increases.

The said speedup could be attributed to the optimizations done on the Burrows-Wheeler transform algorithm as well as the use of multithreading to maximize the CPU's computing power. The speedup becomes more apparent on larger length inputs as the overhead costs of starting new threads becomes a smaller fraction of the runtime while more computations are done in parallel.

These results can also be used as a stand-alone comparison of the Tim Sort function used by Python with the multithreaded Merge Sort function implemented in Java, regardless of whether or not the sort functions are being utilized in BWT .

5 REFERENCES

Labeit, J., Shun, J., & Blelloch, G. (2017). Parallel lightweight wavelet tree, suffix array and FM-index construction. Journal of Discrete Algorithms (2017).

Burrows, M., & Wheeler, D. (1994, May 10). A Block-sorting Lossless Data Compression Algorithm - HP Labs. Retrieved October 11, 2017, from http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf.

Gil, J., & Scott, D. (2009, July 7). A Bijective String Sorting Transform - Dogma. Retrieved October 11, 2017, from http://bijective.dogma.net/00yyy.pdf.