A Survey of Indexing on Graphical Processing Units

A Thesis

Presented to

The Division of Mathematics and Natural Sciences

Reed College

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Arts

Ryan Quisenberry

May 2023

Approved for the Division
(Computer Science)

_____

Jim Fix

ACKNOWLEDGEMENTS

I want to thank a few people.

PREFACE

This is my preface, will write this later.

LIST OF ABBREVIATIONS

Some abbreviations:

BWT Burrows-Wheeler Transform

TABLE OF CONTENTS

ABSTRACT

This is a very succinct, easy to read summary of my findings.

DEDICATION

You can have a dedication here if you wish.

INTRODUCTION

I'm introducing my thesis.

CHAPTER 1

THE SUFFIX ARRAY

CHAPTER 2

THE BURROWS-WHEELER TRANSFORM

The Burrows-Wheeler Transform (BWT) is a transformation on a string S of length N, which results in a permutation L of S. The permutation L is formed by lexicographically sorting the rotations (cyclic shifts) of S and taking the last character of each such rotation. As it turns out, this transformation has a number of exceedingly useful properties. The first property is that L can be used to re-create the original string S, that is to say the BWT is reversible as presented by Burrows and Wheeler [1]. Additionally L is often highly compressible, shown again in [1]. Finally the transformation is useful for memory-efficient sequence searches on S as implemented in the BWA and Bowtie algorithms given by Li and Durbin [2] and Cox *et al.* [3] respectively. These properties of make the Burrows-Wheeler Transform an exceedingly useful method for working with strongly structured and very large strings.

2.1 Construction

The BWT is performed in a few simple steps which we will demonstrate below. These steps are easy to understand but are cumbersome in practice if implemented naively. Luckily there is a solution to this which we describe in section 2.1.2, but first we will describe the full and un-enhanced algorithm for proposes of clarity.

2.1.1 The Burrows-Wheeler Transform Algorithm

The BWT is performed by first appending a unique termination character \$ to the end of S whose lexicographical order is less than any character in S[1]. Next we create a matrix M whose rows consist of the cyclic-shifts of S\$. Finally we sort the rows of M in lexicographical order, as shown in table 2.1, we can then take the resulting final column of M to be the transformed string L.

0	griffin$		$griffin
1	$griffin		ffin$gri
2	n$griffi		fin$grif
3	in$griff	**Sort**	griffin$
4	fin$grif	$\Longrightarrow$	iffin$gr
5	ffin$gri		in$griff
6	iffin$gr		n$griffi
7	riffin$g		riffin$g

Table 2.1: The matrix M for the string S\$ = griffin\$.

As demonstrated in 2.1, the Burrows-Wheeler Transform of the string S\$ = griffin\$ is the string L = nif\$rfig. Though not immediately evident from this example, we will describe how this transform is helpful for data compression in section 2.3 and for memory-efficient sequence searches in section 2.4.

2.1.2 Linear Time Suffix Array Enhancement

By constructing and sorting the matrix M as shown in section 2.1.1, we are a creating a table of size $\mathcal{O}(N^2)$ which would require $\mathcal{O}(N^2)$ time to construct.[2] Additionally, sorting the rows of M would be costly as well and if performed naively could also take $\mathcal{O}(N^2)$ time to perform. If the goal of using the BWT is to compress or sequence S, then it does not make sense to take $\mathcal{O}(N^2)$ time and space to produce L. Luckily both the construction of M and the lexicographical sorting of M can be bypassed by using the Suffix Array Pos of S\$.

Here we give the linear time and space enhancement to the construction described in sec-

[1] This step is not actually required in order to preform the transform but is useful for purposes of clarity and for performing the inverse transform as shown by Manzini [4]. We will show how this technique is helpful in section 2.2.

[2] For simplicity we consider N to be the size of S including the addition of the termination character \$.

tion 2.1. By using the Suffix Array Pos as described in Chapter 1, which can be constructed in linear time as shown by Manber and Myers in [5], we can also perform the Burrows-Wheeler Transformation on a string S in linear time.

Algorithm 1 Linear Time Burrows-Wheeler Transformation

```
1: function BWT(S, Pos, N)
2:     i := 0
3:     while i < N do
4:         if Pos[i] = 0 then
5:             B[i] :=' $'
6:         else
7:             B[i] := S[Pos[i − 1]]
8:         i := i + 1
```

Algorithm 1 avoids constructing the matrix M by exploiting the unique relationship between Suffix Arrays and the Burrows-Wheeler Transform. Since the Suffix Array Pos contains the indices of the sorted suffixes of S, we can note that for each index in Pos, the preceding index would be position of the corresponding character in L. The relationship is given formally as the following,

$$L[i] = \begin{cases} S[Pos[i-1]] & \text{if } i-1 > 0, \\ \$ & \text{else.} \end{cases}$$

This relationship at first may seem counter-intuitive but as it turns out, the list of lexicographically sorted suffixes are of which Pos contains the indices of, is entirely encapsulated in M. This is perhaps best visualized by the following.

i	M	Pos[i]	S[Pos[i-1]]
1	**$**griffin	[8] : **$**	S[7] : n
2	**ffin$**gri	[4] : **ffin$**	S[3] : i
3	**fin$**grif	[5] : **fin$**	S[4] : f
4	**griffin$**	[1] : **griffin$**	S[8] : $
5	**iffin$**gr	[3] : **iffin$**	S[2] : r
6	**in$**griff	[6] : **in$**	S[5] : f
7	**n$**griffi	[7] : **n$**	S[6] : i
8	**riffin$**g	[2] : **riffin$**	S[1] : g

Table 2.2: The relationship between the BWT and the Suffix Array

In the above table we can see that for a given index in Pos, the preceding index is the corresponding character in L.

2.2 The Inverse Burrows-Wheeler Transformation

One of the most remarkable characteristics of the Burrows-Wheeler Transform is that it is reversible. By reversible we mean that if L is the resulting permutation from performing the BWT on S, we can use L to re-create S using only itself as shown by Burrows and Wheeler [1].[3]

2.2.1 The Last First Mapping

The following algorithm is not the algorithm given in the original report by Burrows and Wheeler but instead makes use of some precomputed information. The algorithm below was first presented by Langmead in [6], and it relies on a fundamental fact about the matrix M; the ith occurance of a character in L, corresponds to the ith occurance of that character in the first column of M as well as in the original string S. This property is called Last First Mapping and was proven to be a property of the BWT in Burrow's and Wheeler's original technical report [1].

From this Last First property, we can device a mapping LF[] which when given an index i in L, gives the corresponding index in F for which the occurance of the character at L[i] is the same as the character F[LF(i)]. That is to say if L[i] is the second occurance of the character A in L, which we can denote A_2, then LF[i] will return the index j of A_2 in F. The following visual example perhaps shows this relationship best.

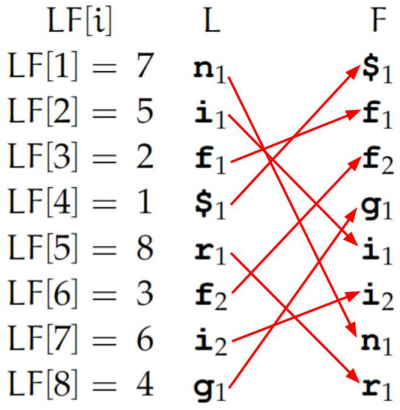

Figure 2.1: The LF mapping of L and F.

We can say that the occurrence of a character c at index i in L is equivalent to the number of occurances of that character in the prefix L[0..i]. We can say this value is the result of the function Occ(c, i) which was shown by Ferragina and Manzini to be able to be calculated in constant time [7]. As it turns out, the Last First Mapping can be calculated easily

[3] Burrows and Wheeler state that their algorithm relies on both L and I, the index of S in the matrix M, but it is trivial to determine I with the addition of the termination character $ to S; I being the index of $ in L.

using the occurance function Occ() and the function C() which when given a character in L, outputs the number of occurrances of lexographically smaller characters in L. For instance using our example of S$ = griffin$, C($) = 0 (this is always the case) and C(n) = 6 since we have $, g, i, f, f, i ≺ n. As it turns out, The Last First Mapping can be computed as the sum of these two functions, [6]

$$LF[i] = C(L[i]) + Occ(L[i]).$$

We will show in the following sub section how using this LF Mapping we can easily reconstruct S from L and F.

2.2.2 The Inverse Transformation Algorithm

Now that we have our LF Mapping, computing S from L and F is simple. We start at the last character in S, which thanks to the addition of our termination character $, will always be the first character n L. We then compute the LF mapping of our character to find its position in F. As we observed earlier in our enhanced BWT, L and F have a special property where characters in the same row directly follow one another in the original string S. That is to say if we have L[i] = i and F[i] = n, then those same characters will appear in S as ...in...$. Using this property we can see that once we have LF[1], we will have the position of our final character in F, and we can begin to build our string backwards by next prepending the character in the corresponding row. We visualize the process below in figure 2.2.

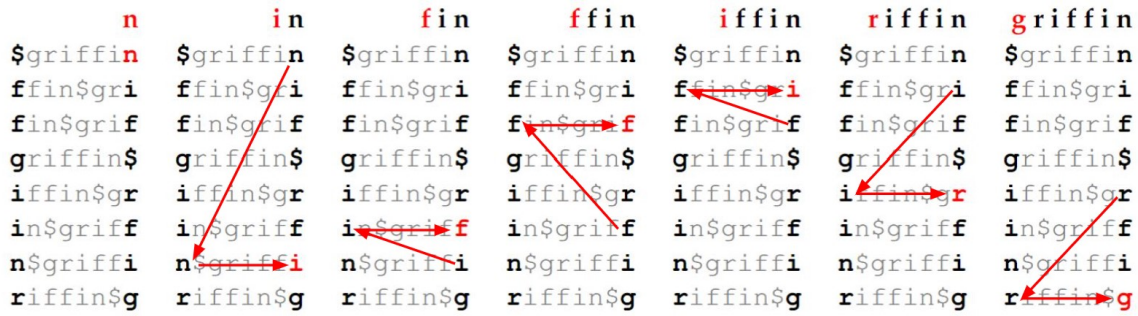

Figure 2.2: A reconstruction of S = griffin from L and F.

As we can see given the LF Mapping we can compute S from L in linear time. It is important to note however that this relies on the function C() being precomputed as a table which is often done in practice and requires little overhead when built into the construction of the Burrows-Wheeler transform.

2.3 Compression

2.4 Sequencing

CHAPTER 3

PREFIX-FREE PARSING

One of the major roadblocks in analyzing exceedingly large sequences of data is the storage space needed to operate on them. While many algorithms are efficient in theory, they rely on the ability to work in internal memory, and when data exceeds the size of this workspace, we see a massive drop-off in performance in practice. This is the motivation for Prefix-Free Parsing. Prefix-Free Parsing, first introduced by Boucher, *et al*. in 2018 [8][1], works to massively reduce the size of data that needs to be operated on at a given point in time, allowing in theory for a Burrows-Wheeler Transform of the input data to be computed in reasonable time for sequences of data previously too large to work with.

This is accomplished by dividing the input into phrases of variable length. Particularly two important data structures are created from dividing the input, a sorted dictionary of the unique phrases in the input, and a parse containing the lexicographical ranks of all the phrases in the order they appear in the input (i.e. the ranks of the phrases in the dictionary). Depending on how the input is divided into these phrases, the parse P, and the dictionary D can be much shorter than our input. As it turns out from these two data structures we can compute the Burrows-Wheeler Transform of the input and thus ideally, allow for the expedient transformation of data sets previously too large to be operated

[1]The original 2018 paper did not include authors Ben Langmead and Taher Mun and also did not present a number of findings in practice, thus for the purposes of this document we cite the 2019 publication of the paper as cited above.

upon in internal memory. In practice we accomplish this by first computing the BWT of P and using that information along with D to reconstruct the BWT of our input.

Boucher, et al. have already shown promising results given good parameters on how the input is divided into the phrases in the parse. The current status of their publication is that they are looking to optimize the algorithm for the parsing in practice as well as expand capacity for and test much larger sets of data. In the following sections we will describe both theoretically and practically how Prefix-Free Parsing accomplishes these results.

3.1 Prefix-free Parsing for Constructing Big-BWTs

For the purposes of demonstrating the Prefix-free Parsing Algorithm we will perform our operations on the following string which mimics a small slice of genetic data we might see when employing such an algorithm in practice;

$$\begin{aligned} \mathsf{T} &:= \texttt{ACCATTCGATTACCAGTACCAGTTCGA}, \\ w &:= 2, \\ \mathsf{p} &:= 3. \end{aligned}$$

The algorithm takes 3 inputs; a text T, a window size w, and a prime number p. We use these inputs to create a number of helpful data structures for creating a Burrows-Wheeler Transform of T. We explain this algorithm given by Boucher, *et al.* in [8] below in section 3.1.1.

3.1.1 The Construction Algorithm

We begin our algorithm by prepending a start character # and appending a string of termination characters $\$^w$ to our text T to form the string $\#\mathsf{T}\$^w$ which we denote T'. The characters # and $ must be lexicographically less than any character in the original text T.[2] Next we slide a length-w window over T' and construct a set of strings E. We from a member string of E whenever the Rabin-Karp fingerprint [9] of the length-w window modulo p is 0.[3] We also add the starting character and termination string to the collection to form our resulting set, here we give an example using our inputs from above.

$$\begin{aligned} \mathsf{T}' &= \texttt{\#ACCATTCGATTACCAGTACCAGTTCGA\$\$}, \\ \mathsf{E} &= \{\texttt{\#, \$\$, CG, TT, CA, AC}\}. \end{aligned}$$

Once we have our set E, we construct a dictionary D and a parse P. Our dictionary is constructed by making another pass over our modified input T'. We add a unique ele-

[2]The start and termination characters do not need to be lexicographically different so long as the satisfy this property [8].

[3]This step can be performed using any reasonable hashing method for which a reasonable set E can be constructed but here we follow the choice used by Boucher, *et al.* in [8].

ment d to D whenever whenever we encounter a substring of T′ with the following properties:

1. An element of E is a proper prefix of d.
2. An element of E is a proper suffix of d.
3. No other elements of E are substrings of d.

That is to say D is the unique collection of all substrings of T′ which start and end with an element of E and which contain no other substring in E. Since D is a unique collection, on a reasonable scale, the dictionary should become much smaller than the input string, which is important to note for when we construct our Burrows-Wheeler Transform. Below in figure 3.1 we show how this mapping might conserve space in the resulting dictionary. This conveniently results in the resulting dictionary being much smaller than the input for reasonable sized and well-structured inputs.

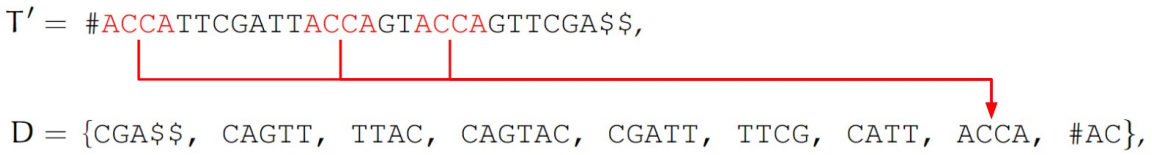

Figure 3.1: Placement of a slice of T′ into D.

Importantly however we do still require information about every dictionary element we encounter, even if the element has already been added to D. This information is the parse P. P is the in-order lexicographical ranks of all the elements we encountered in our pass over T′. For instance as we encounter the following elements from T′, we can also note their lexicographical rank in relation to all other elements as follows.

#AC	0
ACCA	1
CATT	4
TTCG	8
CGATT	6
TTAC	7
ACCA	1
CAGTAC	2
ACCA	1
CAGTT	3
TTCG	8
CGA$$	5

Table 3.1: The parse P of T′.

APPENDIX A

THE FIRST APPENDIX

REFERENCES

[1] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Digital Equipment Corporation, Technical Report 124, May 1994.

[2] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics (Oxford, England)*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009.

[3] A. J. Cox, M. J. Bauer, T. Jakobi, and G. Rosone, "Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform," *Bioinformatics*, vol. 28, no. 11, pp. 1415–1419, Jun. 2012.

[4] G. Manzini, "The Burrows-Wheeler Transform: Theory and Practice," in *Mathematical Foundations of Computer Science 1999*, G. Goos, J. Hartmanis, J. van Leeuwen, M. Kutyłowski, L. Pacholski, and T. Wierzbicki, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, vol. 1672, pp. 34–47.

[5] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, Oct. 1993.

[6] B. Langmead, "Highly Scalable Short Read Alignment with the Burrows-Wheeler Transform and Cloud Computing," Master's thesis, University of Maryland., 2009.

[7] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. Redondo Beach, CA, USA: IEEE Comput. Soc, 2000, pp. 390–398.

[8] C. Boucher, T. Gagie, A. Kuhnle, B. Langmead, G. Manzini, and T. Mun, "Prefix-free

parsing for building big BWTs," *Algorithms for Molecular Biology*, vol. 14, no. 1, p. 13, Dec. 2019.

[9] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, Mar. 1987.

[10] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, 2009.