University of Dublin
TRINITY COLLEGE

A 12*n* Byte Approach to Speeding up Qsufsort

BY CHIA LAU

B.A (Mod.) Computer Science
Final Year Project, April 2020
Supervisor: Jeremy Jones

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland

DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

30/04/2020

Name

Date

Abstract

Suffix arrays are a space-efficient data structure used for string processing. Perhaps the most important application of suffix arrays is in the computation of the Burrows-Wheeler Transform which has uses in data compression and more recently bio-informatics.

This paper suggests changes to *qsufsort*, a suffix array construction algorithm, with the aim of speeding it up. The modifications proposed maintain qsufsort's worst case time complexity of $O(n \log n)$. Speed-up is achieved by trading space for running time and exploiting radix sorting for faster integer sorting. This speed-up is demonstrated experimentally on real world data.

1 Introduction

Given a string S, the Burrows Wheeler Transform (BWT) returns a permutation of S. This permutation is determined by first sorting all the rotations of the input string and then returning the last letter of each rotation in sorted order:

Example: Burrows Wheeler Transform

S	rotations of S	sorted rot. of S	BWT(S)
	abacus$	$abacus	
	bacus$a	abacus$	
	acus$ab	acus$ab	
abacus$	cus$aba	bacus$a	s$baauc
	us$abac	cus$aba	
	s$abacu	s$abacu	
	$abacus	us$abac	

$ is a symbol denoting the end of the string.

The original use of the BWT was for compressing data[5]. The transform has two useful properties for this purpose. Firstly, the transform is reversible. The other useful property is that it tended to group identical characters together. Other compression techniques could exploit this.

More recently, the transform has been used for sequence alignment[3]. Given the BWT, one can count the number of occurrences of a length m substring of S in $O(m)$ time, independent of the length of S. The genomes used in sequence alignment problems are often millions to billions of bytes longs. As such, both time and space efficiency become important. The most common method used to compute a BWT efficiently is to construct a suffix array.

A suffix array is an array of integers, each representing a suffix of a string, where lexicographically smaller suffixes appear first.

Example: Suffix Array

S	suffixes		sorted suffixes		suffix array
	0	abacus$	6	$	
	1	bacus$	0	abacus$	
	2	acus$	2	acus$	
abacus$	3	cus$	1	bacus$	6 0 2 1 3 5 4
	4	us$	3	cus$	
	5	s$	5	s$	
	6	$	4	us$	

From the suffix array, the BWT can be quickly determined.

Example: BWT Construction from Suffix Array

S	A : Suffix Array	BWT(S)							
abacus$	6 0 2 1 3 5 4	i	0	1	2	3	4	5	6
		S[A[i]-1]	s	$	b	a	a	u	c

Larsson and Sadakane's suffix array construction algorithm[6], also known as qsufsort, is an algorithm which constructs suffix arrays in $O(n \log n)$ time.

2 Preliminaries

Counting Sort

Counting sort is an integer sorting algorithm that is not comparison based. The algorithm instead groups elements into k buckets and requires each array element must have an integer key value in the range $[0, k)$.

```
COUNTINGSORT(A[n])
1   COUNT[k] ← {0,0,...,0}
2   POS[k], OUTPUT[k]
3   for i ← 0 to n-1
4       COUNT[key(A[i])] += 1
5       POS[0] = 0
6   for i ← 1 to k-1
7       POS[i] = POS[i-1] + COUNT[i-1]
8   for i ← 0 to n-1
9       OUTPUT[POS[key(A[i])]] = A[i]
10      POS[key(A[i])] += 1
11  return OUTPUT
```

Consider an input array A of length n. Counting sort begins by counting the number of occurrences of each key in A and stores these values in an array, *COUNT*. The first *COUNT*[0] positions of the output array, *OUTPUT*, should contain elements whose key value is 0. These positions are bucket 0. Bucket 1 will use the next *COUNT*[1] positions in *OUTPUT*, bucket 2 will use the following *COUNT*[2] and so on. *POS*[*i*] is initialised to the first position of bucket i. For the rest of the algorithm, *POS*[*i*] holds the first unoccupied position within bucket i. The final loop places each element into its bucket's first unoccupied position. This yields a sorted array. Note that the sorting is stable: elements of equal key value maintain their relative input order.

Both the first and third loop perform $O(n)$ operations. The middle loop has $O(k)$ operations. Altogether, counting sort has a time complexity of $O(n+k)$. The space complexity is also $O(n+k)$ with *COUNT* and POS using $O(k)$ space and *OUTPUT* using $O(n)$.

In practice, counting sort is often not performant as the maximum key value, k, may be too large.

Radix Sort

Radix Sort addresses counting sort's maximum key value problem by sorting by the key one digit at a time. The digit could be of any radix or base. In practice, powers of 2 are used for the radix so that digit extraction can use bit operations instead of multiplication and division. The order of the sorting can either be MSD (most significant digit to least significant) or LSD (least significant digit to most significant).

The radix sorting algorithm I will be discussing is an in-place MSD radix sorting algorithm which is sometimes referred to as the american flag sort[8]. I will be referring to this algorithm as usflagsort as shorthand.

```
USFLAGSORT(A[n], d)
1   COUNT[b] ← {0,0,...,0}
2   POS[b], HEAD[b], TAIL[b]
3   for i ← 0 to n-1
4       COUNT[digit(key(A[i]), d)] += 1
5   POS[0] = HEAD[0] = 0
6   for i ← 1 to b-1
7       HEAD[i] = HEAD[i-1] + COUNT[i-1]
8       TAIL[i-1] = POS[i] = HEAD[i]
9   for bucket ← 0 to b-1
10      for i ← POS[bucket] to TAIL[bucket]-1
11          j = digit(A[i], d)
12          while j ≠ i
13              swap(A[i], A[POS[j]])
14              POS[j] += 1
15              j = digit(A[i], d)
16  if d ≠ least significant digit
17      for bucket ← 0 to b-1
18          if COUNT[bucket] ≠ 0
19              USFLAGSORT(A[HEAD[bucket]...TAIL[bucket]-1], d+1)
```

Like counting sort, usflagsort begins by counting frequencies. However, *COUNT*'s size is no longer equal to the maximum key size k but is now equal to the radix b. *digit* is a function which extracts the dth most significant digit from its first argument.

The beginning, the end and the first unoccupied position of each bucket is needed. Arrays *HEAD*, *TAIL* and *POS* respectively store this information.

To avoid the use of an extra output array, usflagsort swaps elements into their correct positions. For a position i in A, it is either in the correct position or it is not. If it is, the algorithm moves onto the next position. Otherwise, $A[i]$ and the first unoccupied position of its bucket, $POS[j]$, swap values. $POS[j]$ then has a correctly placed element whereas $A[i]$ may or may not be correct. So this swapping process is repeated. Effectively, the first incorrectly placed element swaps itself into the correct bucket and this is repeated until all elements are correctly placed.

This process is indifferent to the relative order of equal key values. Usflagsort is not stable, it has traded stability for a lower space complexity.

Example: Swapping Behaviour of Usflagsort

A	»71«	82	81	82	84	59	04	52
	Bucket 0	Bucket 5		»Bucket 7«	Bucket 8			
Swap 71 and 82								
A	»82«	82	81	71	84	59	04	52
	Bucket 0	Bucket 5			»Bucket 8«			
Swap 82 and 84								
A	»84«	82	81	71	82	59	04	52
	Bucket 0	Bucket 5				»Bucket 8«		
Swap 84 and 59								
A	»59«	82	81	71	82	84	04	52
	Bucket 0	»Bucket 5«					Bucket 8	
Swap 59 and 82								
A	»82«	59	81	71	82	84	04	52
	Bucket 0		Bucket 5				»Bucket 8«	
Swap 82 and 04								
A	04	59	»81«	71	82	84	82	52
			Bucket 5					»Bucket 8«
Swap 81 and 52								
A	04	59	52	71	82	84	82	81

Above, each bucket's unshaded regions is shaded in red. Green positions are sorted. The elements here are sorted by their first digit. After swapping is complete, the subarrays $\{04\}$, $\{59, 52\}$, $\{71\}$ and $\{82, 84, 82, 81\}$ are each recursively sorted, this time by their second digit. A simple micro-optimisation is to only recursively sort buckets if they have two or more elements: single element arrays are already sorted.

The main body of the algorithm (lines 1-15) takes $O(n+b)$ work. The first and second loop are simple loops with $O(n)$ and $O(b)$ operations respectively. The third loop (lines 9-15) is $O(n)$ as the innermost portion (lines 13-15) occurs at most $n-1$ times: each swap will correctly place at least 1 element and the final swap must correctly place both elements. There are $\log_b k$ passes through the data due to recursion. As b is constant, this results in $O(n \log k)$ time complexity.

$O(b)$ space is needed for *COUNT*, *POS*, *HEAD*, *TAIL*. Extra auxiliary space is required on the stack for each level of recursion. In total, $O(b \log k)$ space is needed. As the space requirements are sublinear, usflagsort is an in-place sorting algorithm.

Regions Sort

Parts of usflagsort are trivially parallelisable. The $O(n)$ counting loop in lines 3-4 of usflagsort can be parallelised by assigning each thread a different partition of A to count. Results can then be aggregated afterwards. The recursion step on line 19 is embarrassingly parallel. There are no data dependencies between buckets so buckets can be recursively sorted independently. The $O(n)$ swapping loop of lines 9-15, however, is trickier to parallelise.

Regions sort[7] is a parallel in-place MSD radix sort which solves the issue of parallelising the swapping loop.

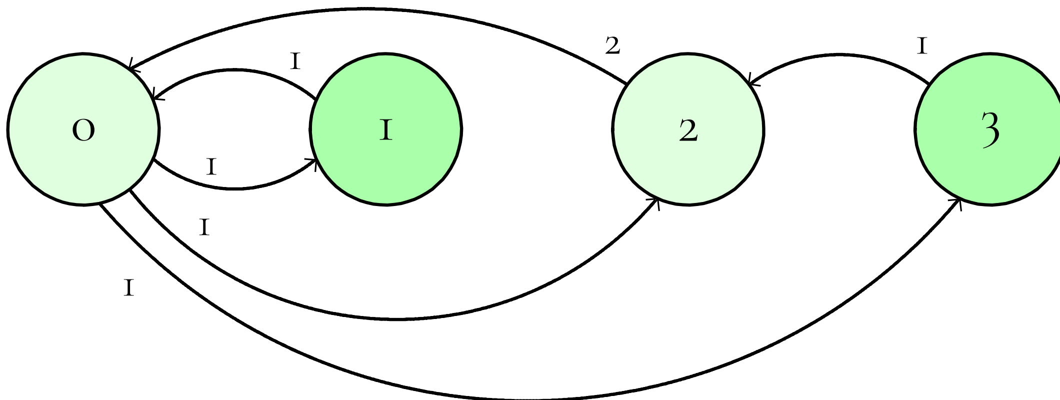
I will begin by introducing some terminology. *Blocks* are the contiguous subarrays formed by splitting the input array A into many smaller pieces. They should be approximately equally sized. The algorithm begins by executing a level of usflagsort (with no recursion step) on each block independently in parallel. Each block is then locally sorted into buckets and the size of each bucket is recorded. By summing bucket sizes, *country* sizes can be determined. Countries are also contiguous subarrays of A. In a globally sorted A, country i should solely contain elements whose current radix is i. Lastly, *regions* are contiguous subarrays whose elements all belong to a single country and a single block bucket.

Example: Blocks, Countries, Regions

A	00	01	01	02	03	11	20	31	11	02	13	21	00	21	21	02
Blocks	0				1				2				3			
After Locally Sorting																
A	00	01	01	02	03	11	20	31	02	11	13	21	00	02	21	21
Block Buckets	0:0				1:0	1:1	1:2	1:3	2:0	2:1		2:2	3:0		3:2	
Countries	0								1			2				3
Regions	0→0				0→0	0→1	0→2	0→3	1→0	1→1		2→2	2→0		2→2	3→2

The blocks, countries and regions of a sample array are shown above. $x{:}\,y$ denotes bucket y of block x. 0:0 has 4 elements, 1:0 and 2:0 have 1 each and 3:0 has 2 elements. Summing the x:0 bucket sizes for each x gives the size of country 0. This is repeated for all other countries. The intersections of block buckets and countries form the regions. Region $x \to y$'s elements are in country x currently but should be in country y.

Example: Regions Sort Graph



A weighted directed graph can be created from the regions. The previous example's regions form the above graph. Here, the four vertices represent the four countries. Each edge is a region $x \to y$ where the edge's source is x, the edge's destination is y and the edge's weight is equal to the number of elements within the region. For example, the edge from vertex 2 to vertex 0 with a weight of 2 represents region $2 \to 0$ and its two elements, 00 and 02. There could be multiple edges from vertex 2 to vertex 0: the graph is not simple. In the previous table, there were two $0 \to 0$ regions. These are not displayed because all edges of the form $x \to x$ can be safely ignored: its elements are already situated in the correct country. If a graph contains no edges after ignoring $x \to x$ edges, then it is already sorted.

To globally sort the array, regions sort uses 2-path processing. 2-path processing begins by selecting a country x. It then matches the edges directed towards x with the edges directed away from x, splitting edges if the sizes don't align. An example of this process is provided below (for a different graph). Each cell contains an edge/region $a \to b$ and the cell's width is proportional to the number of elements. Note that the sum of the weights of outgoing edges and the sum of the weights of incoming edges are equal: if there are m elements within country x which are incorrectly placed, then due to how countries are constructed, there must also be m elements which belong in x that currently are not. After splitting, matched regions are equally sized and can swap elements.

Example: Globally Sorting a Country via 2-path Processing

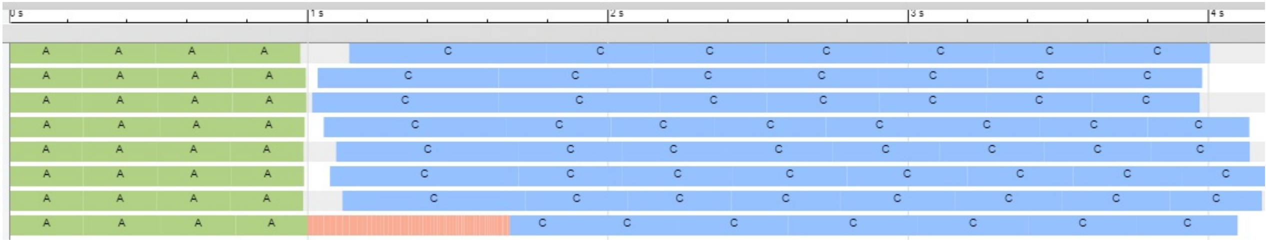
Outgoing Edges	0→1		0→3		0→6
Incoming Edges	2→0	3→0		5→0	
After Splitting					
Outgoing Edges	0→1	0→1	0→3	0→3	0→6
Incoming Edges	2→0	3→0	3→0	5→0	5→0

The effect of swapping the elements of an incoming edge $y \to x$ with the elements of an outgoing edge $x \to z$ is that the elements of $y \to x$ become correctly placed. There are then two cases: $y = z$ and $y \neq z$. If $y = z$, then elements of $x \to z$ will also be correctly placed and both edges can be removed from the graph. Otherwise, $x \to z$ elements will be incorrectly moved into y by the swap. To update the graph in this case, the $x \to z$ edge is changed to a $y \to z$ edge and edge $y \to x$ is removed.

After swapping all matched edges and updating the graph, all edges connected to country x will have been removed. By repeating this process for each other country, all edges will be removed: the array will be sorted by the current digit.

When a country is processed and all of its edges are removed, the rest of the graph will not interact with it further. This allows the processed country to be processed independently: it can be recursively sorted by the next digit in a separate thread. As more countries are processed, more threads become active and more parallelism emerges. This is the method by which regions sort parallelises the swapping loop of usflagsort.

Figure: Early Recursion of Regions Sort



CPU usage per core over time whilst regions sorting 10^9 unsigned integers is shown above. The local sorting of the blocks is shown in green. The orange tasks represent 2-path processing. The blue tasks are the tasks of the recursive calls.

Qsufsort: Prefix Doubling

Larsson and Sadakane's algorithm, or qsufsort, improves upon an older suffix array construction algorithm by Manber and Myers[11]. Both algorithms use an idea known as prefix doubling to achieve a worst case time complexity of $O(n \log n)$.

In prefix doubling algorithms, an array of suffixes I is in h-order if it is sorted by the the h characters of each suffix. The suffix array in h-order is then partitioned into groups of identical symbols.

A single integer can be used to represent a suffix rather than the characters of that suffix. A suffix beginning at index i of the string will have value i in an array I. Array V is an array storing the values I should be sorted by. To sort I into 1-order, $V[i]$ is initialised to contain the first character of suffix i.

Example: h-order and groups

i	0	1	2	3	4	5	6	7
$I[i]$	0	1	2	3	4	5	6	7
$V[I[i]]$	e	n	t	e	n	t	e	$
$I[i]$ in 1-order	7	0	3	6	1	4	2	5
Suffixes in 1-order	$	entente$	ente$	e$	ntente$	nte$	tente$	te$
New $V[I[i]]$	0	3	3	3	5	5	7	7

Consider a group $f \ldots g$ whose first element is at index f and whose last element is at index g. This group is identified with the number g. Elements within V will be overwritten with the group number to keep track of the groups. The above example illustrates this. The range $1 \ldots 3$ forms group 3 as the suffixes in 1-order in that range all begin with e.

The key insight to prefix doubling is that if I is sorted by $V[I[i]+h]$ for each index i, then the array will be in $2h$-order. When $h \geq n$, then all characters of each suffix has been considered in the sorting and the suffix array has been constructed. So by repeatedly doubling the h-order, a suffix array is constructed in $O(\log n)$ passes.

Qsufsort's innovation is that it only sorts unsorted groups. Unsorted groups are groups that contains two or more elements. Single element groups are referred to as sorted groups as they can no longer swap positions with other elements - they are already in their final sorted position. Once there are no more unsorted groups, the suffix array construction is complete. Larsson and Sadakane found this approach to be a magnitude faster in practice than the Manber and Myers algorithm.

Example: Prefix Doubling

`i`	0	1	2	3	4	5	6	7
Input string	e	n	t	e	n	t	e	$
$h=0$								
`V[I[i]]`	*e*	*n*	*t*	*e*	*n*	*t*	*e*	$
`I[i]`	0	1	2	3	4	5	6	7
$h=1$								
`V[I[i]]`	0	3	3	3	5	5	7	7
`I[i]`	7	0	3	6	1	4	2	5
Suffix starting at `I[i]`	$	entente$	ente$	e$	ntente$	nte$	tente$	te$
`V[I[i]+h]`		5	5	0	7	7	3	3
$h=2$								
`V[I[i]]`		1	3	3	5	5	7	7
`I[i]`		6	0	3	1	4	2	5
Suffix starting at `I[i]`		e$	entente$	ente$	ntente$	nte$	tente$	te$
`V[I[i]+h]`			7	7	3	1	5	0
$h=4$								
`V[I[i]]`			3	3	4	5	6	7
`I[i]`			0	3	4	1	5	2
Suffix starting at `I[i]`			entente$	ente$	nte$	ntente$	te$	tente$
`V[I[i]+h]`			4	0				
$h=8$								
`V[I[i]]`			2	3				
`I[i]`			3	0				
Suffix starting at `I[i]`			ente$	entente$				
Suffix Array								
`I[i]`	7	6	3	0	4	1	5	2

Above, the suffix array of the string *entente* is constructed via qsufsort's prefix doubling. The end of string symbol, $, is lexicographically smaller than every other character: it is given a value of zero. Unsorted groups have been marked with colour. At each stage, $I[i]$ is sorted by $V[I[i]+h]$. Intuitively, this can be seen as sorting the an unsorted group by the group values of the greyed-out, unconsidered portion of the suffixes. For example, *ntente$* within *entente$* has a group value of 5 whereas *$* of *e$* only has a group value of 0 so it is lexicographically smaller. The group values are updated after each sort. When there are no more unsorted groups, the algorithm terminates and the suffix array is in I.

Qsufsort: Sorting Unsorted Groups

Qsufsort uses a hybrid approach to sorting unsorted groups. For unsorted groups with 7 or fewer elements, the algorithm uses a variant of selection sort. For larger unsorted groups, Bentley-McIlroy's three way quicksort[4] is employed.

In a traditional selection sort, an outer loop iterates i from 0 to the $n-1$ where n is the size of an array A. The inner loop finds a minimal element of $A[i \ldots n-1]$ which is afterwards swapped into $A[i]$. In total, selection sort uses $O(n^2)$ time.

```
BINGOSORT(A[n])
1   i = 0
2   while i < n
3       min = A[i]
4       j = i + 1
5       for k ← i+1 to n-1
6           if A[k] < min
7               min = A[k]
8               swap(A[i], A[k])
9           else if A[k] = min
10              swap(A[j], A[k])
11              j += 1
12      i = j
```

Qsufsort uses a variant of selection sort known as bingo sort[1]. Instead of swapping a single minimal element per iteration of the outer loop, bingo sort moves all minimal elements. Bingo sort has m iterations of the outer loop rather than n where m is the number of distinct elements within A. This results in an average case time complexity of $O(mn)$. Thus, when there are many duplicate values, bingo sort will perform much better than selection sort.

A second reason for qsufsort's choice of bingo sort is that it conveniently locates the new groups formed after sorting. This is useful as qsufsort integrates the task of updating groups into sorting. For bingo sort, a function similar to the pseudocode below is called after line 11 as after each inner loop completes, the range $[i, j)$ contains elements of equal value.

```
UPDATEGROUP(I[n], V[n], lo, hi)
1   for i ← lo to hi
2       V[I[i]] = hi
```

Bentley-McIlroy's three way quicksort is employed for larger arrays. This is a quicksort variant whose partitioning splits an array into three parts: a left partition consisting of elements strictly less than the pivot element, a right partition of elements greater than the pivot and a centre partition holding all elements equal to the pivot.

Bentley-McIlroy's quicksort uses the ninther[10] as the pivot element. Consider an array of nine or elements. Nine elements are chosen from the array and split into three groups of three. The ninther is the median of the medians of each group. The true median would be the ideal pivot element as the left and right partitions would be similarly sized - it prevents imbalanced recursion. However, calculating the true median is costly. The ninther provides a cheap, low-effort estimate.

```
QUICKSORT(A[n])
1   pivot ← chosen pivot element of A
2   i = p = 0
3   j = q = n-1
4   while true
5       while i ⩽ j and A[i] ⩽ pivot
6           if A[i] = pivot
7               swap(A[p], A[i])
8               p += 1
9           i += 1
10      while i ⩽ j and A[j] ⩾ pivot
11          if A[j] = pivot
12              swap(A[q], A[j])
13              q -= 1
14          j -= 1
15      if i > j
16          break
17      else
18          swap(A[i], A[j])
19          i += 1
20          j -= 1
21   size = min(i-p, p)
22   swap(A[0...size-1], A[i-size...i-1])
23   size = min(q-j, n-q-1)
24   swap(A[i...i+size-1], A[n-size...n-1])
25   if i - p > 0
26       QUICKSORT(A[0...i-p-1])
27   if q - j > 0
28       QUICKSORT(A[n-q+j...n-1])
```

Bentley-McIlroy's quicksort keeps track of four positions whilst partitioning the array (i,j,p,q above). i and p start from index 0 and move upwards. j and q starts at the final index and move downwards. Throughout the algorithm, i stores an index for which all indices less than i contain elements whose value is less than or equal to the pivot. Similarly, elements right of j are greater than the pivot. Elements left of p and elements right of q are equal to the pivot. Partitioning completes when i crosses j.

Example: Three Way Quicksort Partitioning

Suppose `pivot`=3														
A	»2«	3	5	7	1	1	1	3	1	7	1	9	2	»3«
	i, p													j, q
Scan `i` right until `A[i]`≰`pivot`														
A	3	»2«	»5«	7	1	1	1	3	1	7	1	9	2	»3«
		p	i											j, q
Scan `j` left until `A[j]` ≱ `pivot`														
A	3	»2«	»5«	7	1	1	1	3	1	7	1	9	»2«	3
		p	i										j, q	
Swap `A[i]` and `A[j]`. Update `i` and `j` afterwards.														
A	3	»2«	2	»7«	1	1	1	3	1	7	1	»9«	»5«	3
		p		i								j	q	
Repeat previous three steps														
A	3	»2«	2	1	»1«	1	1	3	1	»7«	7	9	»5«	3
		p			i					j			q	
Repeat again. No more repeats are needed as `j` < `i`														
A	3	3	»2«	1	1	1	1	2	»1«	»7«	7	9	»5«	3
			p						j	i			q	
	=pivot					<pivot					>pivot			=pivot
Swap $=_{pivot}$ partitions into centre														
A	2	1	»2«	1	1	1	1	3	»3«	»3«	7	9	»5«	7
			p						j	i			q	
				<pivot					=pivot			>pivot		

In this example, the less-than partition is highlighted in blue, the greater-than partition is highlighted in red and the equals partition in green. The first six rows demonstrate the work performed by Lines 4-20. These lines split the array into four partitions with two equals partitions at the extremities. Lines 21-24 swap these equals partitions into the centre as shown in the final row of the example. At the end, $<_{\text{pivot}}$ and $>_{\text{pivot}}$ are recursively sorted.

Three way quicksort has an average case time complexity of $O(n \log n)$. It is an unstable, in-place sorting algorithm with an average case space complexity of $O(\log n)$ due to the stack space used by recursion. Like bingo sort, three way quicksort is quicker when there are many duplicate values. In three way quicksort's case, a best case time complexity of $O(n)$ is achieved when all values are identical. Groups are also easily extracted from the algorithm. Each call of quicksort will find one group: the group whose values are equal to the pivot. The extents of this group is the range $[i-p, n-q+j)$. This group's values can then be updated from within the algorithm. Every other group will be extracted from the recursive calls.

Qsufsort: Input Transformation

To speed up the first few rounds of prefix doubling, qsufsort uses a technique it calls input transformation. This takes the input string in an array V, transforms its elements and returns an integer.

```
TRANSFORM(V[n])
1  seen[256] ← {false,false, . . . ,false}
2  symbol[256]
3  for i ← 0 to n-1
4      seen[V[i]] = true
5  base = 1
6  for i ← 1 to 255
7      if seen[i] == true
8          symbol[i] = base
9          base += 1
10 m ← largest x that satisfies base^x < 256
11 mult ← base^(m−1)
12 k = 0
13 V[n-1] = 0
14 for i ← n-2 downto 0
15     k = ⌊k / base⌋ + mult × symbol[V[i]]
16     V[i] = k
17 return m
```

The first step of input transformation is to compact the input alphabet (lines 3-9). The input string is passed into the function via V. The lexicographically smallest letter in V is mapped to a symbol value of 1, the next smallest is mapped to 2 and so on. Symbol value zero is reserved for the end of string symbol (\$). For example, in the string *entente*, *e* is the smallest letter so $symbol["e"] = 1$.

Example: Input Transformation, Compacting the Alphabet

i	0	1	2	3	4	5	6	7
V[i]	e	n	t	e	n	t	e	$
symbol[V[i]]	1	2	3	1	2	3	1	0

In the pseudo-code of *TRANSFORM*, *base* stores the number of distinct characters in the input string including \$. If *base* is small enough, $V[i]$ can be transformed into a combination of multiple compacted symbol values like so:

$$V[i] = \sum_{k=1}^{m} S[I[i+k-1]] \text{base}^{m-k}$$

The transformed value would contain the first m characters starting from index i. Any indices which overflow the array, i.e. $S[I[x]]$ for $x \geq n$, will use a symbol value of 0.

Example: Input Transformation, Continued

i symbol[V[i]]	0 1	1 2	2 3	3 1	4 2	5 3	6 1	7 0
New V[i]	123_4 27	231_4 45	312_4 54	123_4 27	231_4 45	310_4 52	100_4 16	000_4 0

Input transformation of the string *entente* continues above with $m=3$. The transformed $V[i]$ values can be calculated in $O(n)$ time independent of m. This is visible in lines 13-16 of the pseudo-code. Transformed values are calculated from right to left starting with $V[n-1]=0$. To calculate $V[i-1]$, the lowest digit of $V[i]$ is removed and $symbol[V[i-1]]$ is prepended. For example, to calculate the transformed value of $V[4]$, 310 loses its 0 and 2 is placed in the front for $V[4]=231_4=45$.

In qsufsort, m is chosen to be the largest x satisfying $\text{base}^x<256$. Larger m values are desired. After sorting the $0-$order suffix array $I[i]$ by the transformed symbols of V, $I[i]$ will be in $m-$order. Prefix doubling can then begin from m instead of one.

The reason for restricting the left hand side to below 256 is that an $O(n)$ sorting algorithm, counting sort, can be used for the initial sort rather than an $O(n \log n)$ quicksort. Counting sort's $O(n+k)$ space requirements are not an issue as array I can be used for the $O(n)$ part of the space requirements. This is an option because $I[i]=i$ for the initial sort so the index can be used instead of the contents.

The conditions which allow the use of counting sort are lost beyond the first sort. Firstly, the values of elements of I are no longer related to the index so I can no longer be used for the $O(n)$ space requirements. And secondly, the maximum V value after groups are updated will be $n-1$ which makes the $O(k)$ portion of counting sort's complexity significant.

Qsufsort: In 8n bytes

Lightweight is a term used to describe certain suffix array construction algorithms. Lightweight algorithms use a total of $5n$ bytes of space: n bytes for an input *char* array and $4n$ bytes for an output *int* array. Minimal space usage is important for suffix array construction algorithms. Datasets of the applications of suffix arrays are generally very large - millions to billions of bytes. A lower memory requirement would allow larger suffix arrays to be constructed in main memory.

Qsufsort is not lightweight but it comes close - requiring only $8n$ bytes. Unlike lightweight suffix array construction algorithms, the input is not taken as a *char* array but as an *int* array. Qsufsort exploits this difference to store information required for prefix doubling.

There are three lists of data qsufsort must keep track of. The first is an array I which contains the suffix array in $h-$order. The second is an array V of symbol/group values to sort I by. The final list is a list of unsorted groups. Some mechanism is needed to efficiently retrieve the location and size of each unsorted group.

A $12n$ byte method would be to use a third array, L, to keep track of the unsorted groups:

Example: Prefix Doubling with Length Array

i	0	1	2	3	4	5	6	7
Input string	e	n	t	e	n	t	e	$
$h=0$								
V[I[i]]	*e*	*n*	*t*	*e*	*n*	*t*	*e*	$
I[i]	0	1	2	3	4	5	6	7
$h=1$								
L[i]	-1	3			2		2	
V[I[i]]	**0**	3	3	3	5	5	7	7
I[i]	7	0	3	6	1	4	2	5
$h=2$								
L[i]	-1	-1	2		2		2	
V[I[i]]		**1**	3	3	5	5	7	7
I[i]		6	0	3	1	4	2	5
$h=4$								
L[i]	-2		2		-1	-1	-1	-1
V[I[i]]			3	3	**4**	**5**	**6**	**7**
I[i]			0	3	4	1	5	2
$h=8$								
L[i]	-2		-1	-1	-4			
V[I[i]]			**2**	**3**				
I[i]			3	0				
Suffix Array								
I[i]	7	6	3	0	4	1	5	2

When a new sorted group i is created via *UPDATEGROUP*, $L[i]$ is set to -1. $-x$ denotes a contiguous range of length x containing only sorted groups. When a new unsorted group $i \ldots j$ is created, $L[i]$ is instead set to $j-i$, the size of the group. L can then be traversed to find each unsorted group. Starting with $\text{pos}=0$, $L[\text{pos}]$ is read. This value is either negative or positive. If it is positive, then an unsorted group has been found with size $L[\text{pos}]$ and starting index *pos*. This unsorted group can then be sorted into $2h-$order. Regardless of the sign, the magnitude of $L[\text{pos}]$ is added to *pos* to obtain the next group. These steps are repeated to get every unsorted group until $\text{pos} \geq n$.

A simple optimisation to speed up traversal is to combine streaks of sorted groups. At $h=2$ of the example, $L[0]$ and $L[1]$ are both negative with values of -1. This chain of negative values can be simplified to what is seen at $h=4$: $L[0]$ is set to -2.

Observing the bolded values in the example, it can be noted that $V[I[i]]==i$ for a sorted group i. This observation allows I values of sorted groups to be overwritten: it can be recovered later with a for loop over i assigning $I[V[i]]=i$. The negative values of L which represent sorted regions can be stored in I instead.

The remainder data of L, the unsorted group sizes, is also unnecessary. These sizes can be calculated from V. An unsorted group starting at index i has size $V[I[i]]-i+1$. Thus, L is redundant and can be removed entirely.

Example: Prefix Doubling in 8*n* bytes

`i`	0	1	2	3	4	5	6	7
Input string	e	n	t	e	n	t	e	$
$h=0$								
`V[I[i]]`	*e*	*n*	*t*	*e*	*n*	*t*	*e*	$
`I[i]`	0	1	2	3	4	5	6	7
$h=1$								
`V[I[i]]`	**0**	3	3	3	5	5	7	7
`I[i]`	-1	0	3	6	1	4	2	5
Unsorted group size		3-1+1=3			5-4+1=2		7-6+1=2	
$h=2$								
`V[I[i]]`		**1**	3	3	5	5	7	7
`I[i]`	-1	-1	0	3	1	4	2	5
Unsorted group size			3-2+1=2		5-4+1=2		7-6+1=2	
$h=4$								
`V[I[i]]`			3	3	**4**	**5**	**6**	**7**
`I[i]`	-2		0	3	-1	-1	-1	-1
Unsorted group size			3-2+1=2					
$h=8$								
`V[I[i]]`			**2**	**3**				
`I[i]`	-2		-1	-1	-4			
Suffix Array recovered with `I[V[i]]=i` loop								
`I[i]`	7	6	3	0	4	1	5	2

3 The Modifications

Trading Space for Time

Qsufsort spends most its time sorting unsorted groups. This sorts elements x in I with a costly key function of $f(x) = V[x+h]$. Below, this key function is compared with the identity key function $x \rightarrow x$.

Table: Key function benchmarks

Key function	Time to sort 10^3 elements	Time to sort 10^6 elements
$f(x) = x$	55 μs	87,382 μs
$f(x) = V[x]$	97 μs	183,876 μs

Three way quicksort was used in the key function experiment above. Like in qsufsort, this quicksort resorts to selection sort for smaller arrays. The elements of the arrays were random integers in the range $[0, 10^7)$. Here, h is fixed to zero. Using a dynamic h would only slow qsufsort's key function further.

The identity key function is roughly twice as fast. If qsufsort as a whole was two times quicker, its performance would be on par with Manzini and Ferragina's suffix array construction algorithm[2] which was the quickest of the suffix sorting algorithms in this comparison paper[9].

The second key function unavoidably needs to do more work as it must dereference an extra pointer. Another factor that aggravates performance is that it is a cache unfriendly key function - successive calls of $V[x]$ may not spacially local. This leads to more cache misses and things only get worst when constructing larger suffix arrays.

Here, I propose an approach to reduce the number of $V[x]$ calls by packing more data into a larger I array. This larger I array is a length n array of 64 bit *int*s rather than 32 bits. The upper 32 bits of an element maintains the old purpose: storing suffix array indices. The lower half stores $V[x]$. $V[x]$ can be calculated once for each element, directly before sorting, and saved in the lower 32 bits. This reduces the number of $V[x]$ calls from $O(n \log n)$ to exactly n for sorting an unsorted group.

```
COMBINE(hi: int32, lo: int32)
1   return int64(hi « 32 | lo)

HIGH(i: int64)
1   return int32(i » 32)

LOW(i: int64)
1   return int32(i)
```

Several utility functions are needed. *COMBINE* takes two 32 bit integers and packs them into a single 64 bit integer, *HIGH* returns the most significant 32 bits of a 64 bit integer and *LOW* returns the least significant 32 bits. As $V[x]$ is stored in the lower half, the key function for sorting is *LOW*.

Table: Key function benchmarks. Part 2.

Key function	Time to sort 10^3 elements	Time to sort 10^6 elements
$f(x) = x$	55 μs	87,382 μs
$f(x) = V[x]$	97 μs	183,876 μs
$f(x) = \text{LOW}(x)$	73 μs	97,904 μs

With this approach, the time to sort is now much closer to the identity key function's. The time required for copying $V[x]$ into the lower halves of I is included. The great drawback with this approach is that is requires significantly more memory - suffix array creation with this needs $12n$ bytes.

Some implementation details of integrating this change into qsufsort will now be discussed.

Firstly, nearly all occurrences of $I[i]$ within qsufsort should be replaced with HIGH($I[i]$). This has low performance impact as bitshifting is relatively cheap compared to memory access. The one exception to this replacement is the usage of I to store sorted group length data for group traversal. Sorted elements of I will never have $V[x]$ values loaded into them so the entire 64 bits can be used to store a negative integer. Because sorted groups and unsorted groups are distinguished by the sign of the integer, this

forces the most significant bit, the sign bit, to always be unset for unsorted group elements of I. So, like qsufsort, this $12n$ suffix construction algorithm has a maximum input string size of 2^{31}.

As mentioned prior, values from V are loaded into the lower half of I directly before sorting. This is implemented with a for loop which uses a mixture of *HIGH* and *COMBINE* to overwrite the lower halves of a subarray of I.

```
LOADVALUES(I[lo...hi]: int64, h)
1   for i ← lo to hi
2       x = HIGH(I[i])
3       I[i] = COMBINE(x, V[x+h])
```

In qsufsort, checks to ensure index $x+h$ is not out of bounds are not needed. The reason for this is that the last h suffixes when I is in h-order are in sorted groups as the unique end of string symbol $ will appear in their length h prefixes. For the same reason, a bounds check is not needed here.

A consequence of loading the key into I is that the key value is no longer dynamically updated during sorting. Because group updating is integrated into sorting in qsufsort, it can be the case that elements in h-order are sorted by the newly updated group values which are of $2h$-order. For certain input string distributions, this can improve the time complexity. This benefit is lost with the $12n$ byte approach as updating affects array V but not the lower bits of I that the key function reads from.

Qsufsort with Radix Sorting

Input transformation allows the first few levels of prefix doubling to be skipped. The number of levels skipped is limited by the maximum value of transformed symbols. Qsufsort's chosen maximum value of 255 is due to the use of counting sort for the initial sort. Counting sort become inefficient for larger maximum transformed symbols as its time complexity and space complexity are both $O(n+k)$. This is the exact problem radix sort is designed to solve.

With radix sort, the maximum transformed symbol values can be the maximum integer value. This allows approximately four times as many input string characters to be packed into a transformed symbol value and about two more levels of prefix doubling to be skipped.

```
TRANSFORM(I[n]: int64, V[n])
1   seen[256] ← {false,false,...,false}
2   symbol[256]
3   for i ← 0 to n-1
4       seen[V[i]] = true
5   base = 1
6   for i ← 1 to 255
7       if seen[i] == true
8           symbol[i] = base
9           base += 1
10  m ← largest x that satisfies base^x < INT_MAX
11  mult ← base^(m-1)
12  k = 0
13  I[n-1] = combine(n-1, 0)
14  for i ← n-2 downto 0
15      k = ⌊k / base⌋ + mult × symbol[V[i]]
16      I[i] = combine(i, k)
17  return m
```

As seen in the pseudo-code above, *TRANSFORM* needs only a few minor modifications: increasing the maximum transformed symbol value and changing where these transformed symbols are written to. The modified *TRANSFORM* procedure now writes the transformed symbols to the lower half of I instead of V. This allows the cheaper key function described in the previous section to be used for the initial sort. Initialising the upper half of I elements to the suffix array indices can also be moved here.

The second change, writing to I, depends on using usflagsort as the initial sort. A modified usflagsort for this sort is shown below. The only addition is a loop for updating group values at the final level of recursion.

```
USFLAGSORT(I[n], d)
1   COUNT[b] ← {0,0,...,0}
2   POS[b], HEAD[b], TAIL[b]
3   for i ← 0 to n-1
4       COUNT[digit(key(I[i]), d)] += 1
5   POS[0] = HEAD[0] = 0
6   for i ← 1 to b-1
7       HEAD[i] = HEAD[i-1] + COUNT[i-1]
8       TAIL[i-1] = POS[i] = HEAD[i]
9   for bucket ← 0 to b-1
10      for i ← POS[bucket] to TAIL[bucket]-1
11          j = digit(I[i], d)
12          while j ≠ i
13              swap(I[i], I[POS[j]])
14              POS[j] += 1
15              j = digit(I[i], d)
16  for bucket ← 0 to b-1
17      if COUNT[bucket] ≠ 0
18          if d ≠ least significant digit
19              USFLAGSORT(I[HEAD[bucket]...TAIL[bucket]-1], d+1)
20          else
21              UPDATEGROUP(I[HEAD[bucket]...TAIL[bucket]-1])
```

The use of usflagsort allows the mechanisms around initial sort to change drastically. In qsufsort, counting sort was run on array V with the goal of performing a $h=0$ sort: sorting array I where $I[i]=i$ by $V[i]$. Because the indices and values are equal in I, there is no need to read values from array I - the contents need not be used. This frees the contents of I for use as the output array for the counting sort procedure run on V. A specialised counting sort which outputted indices into I could be used.

With usflagsort, there is no requirement for $O(n)$ extra space. This allows the initial sort to have behaviour similar to that of the unsorted group sorting. Usflagsort can sort the 64 bit array I by its lower 32 bits and populate V with group values. Because usflagsort is a radix sort, the key function can be changed to the identity function. The extraction of the lower 32 bits is not needed if usflagsort starts looking at digits only from the 32nd most significant bit onwards.

A hybrid sorting approach can also be used to sort smaller subarrays quicker. In my implementation, usflagsort switched to insertion sort for $n \leqslant 2^4$ and switched to three way quicksort for $n \leqslant 2^7$. These magic numbers were chosen after some trial and error.

This hybrid usflagsort can also be used to sort unsorted groups because sorting unsorted groups is also about sorting array I by the lower halves of its elements and subsequently updating the group values. Usflagsort's better time complexity of $O(n)$ will allow larger unsorted groups to be sorted quicker. However, qsufsort's overall time complexity of $O(n \log n)$ is unchanged as there will still be, in the worst case, up to $\log n$ levels of prefix doubling and each level will have up to $n-h$ total unsorted elements. $n+(n-1)+(n-2)+(n-4)+\cdots(n-n) \approx n \log n-(1+2+4+\cdots+n) \approx n \log n-2n = O(n \log n)$.

Insertion Sort

Insertion sort can be used to sort the smallest subarrays instead of bingo sort. Insertion sort is another in-place comparison based sorting algorithm with $O(n^2)$ average and worst case time complexity. In insertion sort there is a sorted region and an unsorted region. Elements in the unsorted region are shifted leftwards until it is correctly placed. In the example below, the element being shifted is marked in red and the elements it has displaced are shaded in green. The grey region is the unsorted region.

Example: Insertion Sort

$i=1$	1	4	2	8	5	7	1
$i=2$	1	2	4	8	5	7	1
$i=3$	1	2	4	8	5	7	1
$i=4$	1	2	4	5	8	7	1
$i=5$	1	2	4	5	7	8	1
$i=6$	1	1	2	4	5	7	8

Notice that for $i=1$ and $i=3$ no displacement occurred. Insertion sort is an adaptive sorting algorithm. For nearly sorted arrays, insertion sort performs much more efficiently. If no element is more than k positions away from its sorted position, insertion sort completes in $O(kn)$ time. The input distribution for which bingo sort enjoys it best case time complexity of $O(n)$ (every

element has equal value) is similarly $O(n)$ for insertion sort.

When each element has a different value, insertion sort can expect to perform half as many comparisons as bingo sort. With bingo sort, an element at i would be compared with every element at a higher index for a total of $\frac{n^2+n}{2}$ comparisons. Meanwhile, insertion sort (assuming uniformly random data), will on average compare an element at i with half of the elements left of it for $\frac{n^2+n}{4}$ comparisons. Because of this, insertion sort is often more performant than selection sort.

```
INSERTIONSORT(I[n])
1   for i ← 1 to n-1
2       tmp = I[i]
3       j = i-1
4       while j ⩾ 0 and key(I[j]) > key(tmp)
5           I[j+1] = I[j]
6           j -= 1
7       I[j+1] = tmp
8   i = 0
9   while i < n
10      j = i
11      i += 1
12      while i < n and key(I[j]) == key(I[i])
13          i += 1
14      updategroup(I[j...i-1])
```

An implementation of insertion sort with functionality to update group values is shown above. Groups are scanned for in $O(n)$ time from lines 8 to 14. This extra loop's time complexity is dominated by the sorting loop which has an average case time complexity of $O(n^2)$.

Parallelisation of Modified Qsufsort

Here, I discuss some methods to parallelise the various parts of the modified qsufsort.

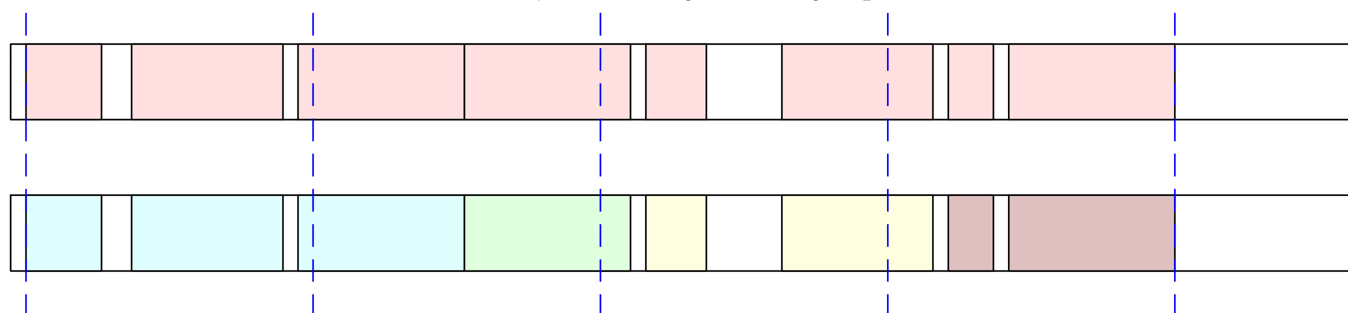
The two $O(n)$ loops of input transformation can be parallelised very easily. The first $O(n)$ loop records which characters appear in the input string. To parallelise this, each thread can be assigned a portion of the input string to scan and their own length 256 array to record their local results. The global results are afterwards aggregated. *TRANSFORM*'s other $O(n)$ loop loads transformed symbol values into I as it iterates through each character in the input string. Again, the input string can be partitioned amongst the threads. An $O(m)$ loop to calculate the rightmost symbol, $\sum_{i=1}^{m} S[I[i + i - 1]] \text{base}^{m-i}$, begins each thread. Whereafter the original $O(n)$ loop can fill in the rest of the partition's values.

By switching the initial sorting algorithm from usflagsort to regions sort, the initial sort can be performed in parallel. The local sorting of the blocks can reuse the same logic of lines 1–15 of usflagsort. Combining sorting and updating groups is also performed in a similar manner as usflagsort: logic to update the groups values is added to the last level of recursion of regions sort. Beyond that, no changes are required for regions sort to function.

Regions sort is not as suitable for sorting unsorted groups. This is because the initial sort concerns sorting one large array whereas unsorted groups are often small and plentiful. The few unsorted groups that are large can be sorted with regions sort. The rest can be sorted using usflagsort.

To parallelise sorting unsorted groups further, one can distribute all the groups of one prefix doubling level amongst each thread. Ideally, each thread is given approximately the same number of elements to reduce idle CPU time due to workload imbalance. Solving this problem without using too much time or memory is quite difficult.

Example: Chunking unsorted groups



Above, illustrates a cheap approach which provides some parallelisation does not ultimately solve the workload imbalance. The coloured portions represent the unsorted groups. The upper array shows all the unsorted groups before they are distributed amongst threads. The bottom array shows the groups split amongst four different colours, each of which represent a thread. The dashed blue line splits the extents of the unsorted groups into four equally sized chunks. Whichever chunk a group's first element is within is the chunk that will process that group. Whilst the work distribution amongst threads can be highly imbalanced, this approach has the advantages of requiring no extra traversals through the groups as well as no extra space requirements.

Sorting multiple unsorted groups is not possible in the original qsufsort because groups are updated dynamically whilst sorting and the sorting key function directly read from the group values array V. This combining of sorting and updating is kept consistent in serial by only updating values from lowest to highest. However, running the same algorithm in parallel violates this order of updates.

With the $12n$ byte modified qsufsort, this is not an issue because the sorting key function does not read from V. If all V values are loaded into the lower 32 bits of each element of each unsorted group before any parallel sorting begins, correctness can be maintained. This loading can also be performed in parallel by distributing the work with the chunking scheme devised above. However, there is a drawback to this idea. The splitting of loading and sorting into two distinct phases creates extra work: two traversals through the groups is now needed per prefix doubling.

4 Evaluation

The modifications to qsufsort described in this paper were implemented in C++ with g++ version 7.5 as the compiler. For the parallelisation, the OpenMP v4.5 was used. The testing machine contained an Intel i5-8300H CPU running at 2.30 GHz and had four physical cores (each with hyperthreading).

The benchmarks used were as follows:

Table: Benchmark Descriptions

Name	Bytes	Mean LCP	Max LCP	Alphabet Size	Source
world192.txt	2,473,400	23	559	94	Canterbury Corpus
bible.txt	4,047,392	14	551	63	Canterbury Corpus
E.coli	4,638,690	17	2,815	4	Canterbury Corpus
chr22.dna	34,553,758	1973	199,999	4	Manzini Corpus
howto	39,422,105	253	70,720	197	Manzini Corpus
alla.txt	50,000,000	24,999,999	49,999,999	1	Generated Data
random.txt	50,000,000	3	8	64	Generated Data
etext99	105,277,340	1,073	286,352	146	Manzini Corpus
chr9.fa	138,394,717	14,999,999	813,619	10	Genome Reference Consortium

The sources for most of these benchmarks were the Canterbury Corpus, Manzini Corpus and the Genome Reference Consortium. The Canterbury Corpus are a set of benchmarks designed to evaluate lossless compression algorithms and are among the data Larsson and Sadakane used to check qsufsort against real world data. The Manzini Corpus are a set a benchmarks created to test Manzini and Ferragina's suffix array construction algorithm[2]. Whilst both of these datasets contained a DNA file, both of these files are relatively small compared to real life genome data. A larger DNA file was obtained from the Genome Reference Consortium. The last two files, *alla.txt* and *random.txt*, were files I generated to test the algorithm on certain input distributions. The former is a file containing the letter *a* repeated 50 million times. The latter contains 50 million characters taken uniformly randomly from a pool of 64 characters.

The LCP, or Longest Common Prefix, array of a suffix array is another array which holds the LCP of each of the adjacent sorted suffixes. The LCP itself is the equal to the first index i for which the ith character of the first string does not match the ith character of the second string. For example, *zebra* and *zero* have an LCP of 2.

The mean LCP column shows the mean value of the LCP array and max LCP shows the maximum entry. These values are related to how much prefix doubling is needed to construct the suffix array. Worst case inputs will have large LCP values. Real world data tends to have lower LCP values.

The alphabet size counts the number of distinct characters within the input file. This affects how many input characters can be packed into a transformed symbol.

Results

Table: Benchmark Results

Name	Qsufsort	Algorithm A		Algorithm B		Algorithm P	
world192.txt	0.314 s	0.241 s	1.30×	0.234 s	1.34×	0.143 s	2.19×
bible.txt	0.665 s	0.417 s	1.59×	0.387 s	1.71×	0.266 s	2.50×
E.coli	0.811 s	0.302 s	2.68×	0.295 s	2.74×	0.295 s	2.74×
chr22.dna	8.138 s	3.432 s	2.37×	3.440 s	2.36×	2.896 s	2.81×
howto	8.743 s	6.035 s	1.44×	5.730 s	1.52×	3.787 s	2.30×
alla.txt	3.755 s	5.798 s	0.64×	10.280 s	0.36×	4.354 s	0.86×
random.txt	7.887 s	3.306 s	2.38×	3.225 s	2.44×	3.045 s	2.59×
etext99	29.768 s	20.411 s	1.45×	18.915 s	1.57×	11.311 s	2.63×
chr9.fa	34.689 s	17.195 s	2.01×	17.702 s	1.95×	12.046 s	2.87×

Elapsed time and relative speedup are shown above. The times recorded are averages of three trials. The *Qsufsort* column shows the performance of Larsson's implementation which is publically available. This is used as the baseline for the relative speedup calculations.

Algorithm A in the table uses 12*n* bytes for a cheaper key function and usflagsort as the initial sort. Like qsufsort, it uses three way quicksort to sort unsorted groups. It is an improvement for all but one benchmarks. *Algorithm B* replace quicksort with usflagsort for sorting unsorted groups. This performs marginally better than *Algorithm A* for 6 benchmarks, marginally worse for 2 and substantially worse for *alla.txt*.

Algorithm P is the parallel qsufsort being run with 8 threads on a 4 core machine. It uses regions sort for the initial sort and usflagsort for sorting unsorted groups. Whilst it is the fastest on most benchmarks, for certain benchmarks *A* and *B* are not far behind despite being serial. The table below investigates how the number of threads affects *Algorithm P*'s performance.

Table: Parallel Thread Scaling

Name	1 Thread	2 Threads		4 Threads		8 Threads	
world192.txt	0.284 s	0.196 s	1.44×	0.144 s	1.97×	0.143 s	1.98×
bible.txt	0.492 s	0.368 s	1.33×	0.271 s	1.81×	0.266 s	1.84×
E.coli	0.531 s	0.371 s	1.43×	0.314 s	1.69×	0.295 s	1.80×
chr22.dna	5.207 s	3.622 s	1.43×	3.032 s	1.71×	2.896 s	1.79×
howto	7.930 s	5.283 s	1.50×	4.209 s	1.88×	3.787 s	2.09×
alla.txt	9.727 s	6.499 s	1.49×	4.796 s	2.02×	4.354 s	2.23×
random.txt	5.726 s	3.686 s	1.55×	3.238 s	1.76×	3.045 s	1.88×
etext99	25.801 s	16.735 s	1.54×	12.905 s	1.99×	11.311 s	2.28×
chr9.fa	22.799 s	16.101 s	1.41×	12.786 s	1.78×	12.046 s	1.89×

Parallelisation mainly benefits the first few rounds of prefix doubling where workload imbalance is often less of a problem. Also note that Algorithm P on 1 thread runs about 26% slower than Algorithm B.

5 Conclusion

Although the proposed changes to qsufsort do not lower the time complexity, it does successfully speed it up in practice and allows for groups to be sorted in parallel. The experiments shows that up to a 2x speedup can be obtained using these modifications. However, this is achieved at the expense of space.

Future work includes optimising the parallel algorithm to scale better with threads perhaps by using smarter work distribution methods and improving the algorithm's handling of its worst benchmark, *alla.txt*.

Bibliography

[1] P. Black. "bingo sort", in dictionary of algorithms and data structures. 2008.
[2] P. Ferragina G. Manzini. Engineering a lightweight suffix array construction algorithm. 2004.
[3] Li H. and Durbin R. Fast and accurate short read alignment with burrows-wheeler transform. 2009.
[4] M. McIlroy J. Bentley. Engineering a sort function. 1993.
[5] D.J. Wheeler M. Burrows. A block-sorting lossless data compression algorithm. 1994.
[6] K. Sadakane N. Larsson. Faster suffix sorting. 1999.
[7] E. Fan, J. Shun, O. Obeya, E. Kahssay. Theoretically-efficient and practical parallel in-place radix sorting. 2019.
[8] M. McIlroy, P. McIlroy, K. Bostic. Engineering radix sort. 1993.
[9] A. Turpin, S. Puglisi, W. F. Smyth. The performance of linear time suffix sorting algorithms. 2005.
[10] J. Tukey. The ninther, a technique for low-effort robust (resistant) location in large samples. 1978.
[11] G. Myers U. Manber. Suffix arrays: a new method for on-line string searches. 1993.