Cache-efficient sorting for Burrows-Wheeler Transform

Advait D. Karande Sriram Saroop

December 2003

Abstract

The expensive activity during compression in the Burrows-Wheeler Transform (BWT) is sorting of all the rotations of the block of data to compress. Most of the fast suffix sorting algorithms suggested to encounter this problem suffer from adverse memory effects. We present cache-efficient suffix sorting algorithms that can overcome this overhead. We analyze the performance of Sadakane's algorithm that uses a cache-oblivious Distribution sort instead of Quick sort. We then present a fast cache-efficient suffix sorting algorithm based on a Divide and Conquer strategy and evaluate its effectiveness. A recent breakthrough in the field of suffix sorting has been the emergence of a linear time suffix sorting algorithm. We present improvisations to the algorithm and compare its performance against contemporary suffix sorting implementations.

1 Introduction

Lossless data compression based on Burrows-Wheeler Transform (BWT) has received much attention since its introduction [1]. The *bzip2* software is an industrial strength compression tool based on this transform. BWT involves sorting of all rotations of the block of data to be compressed. This problem is usually solved by transforming it into an easier problem of suffix sorting. Suffix array data structure is widely used to perform this task, because of its lower memory requirements than other alternatives such as suffix trees. Yet, suffix array construction is a computationally intensive activity, which dominates the compression time.

Several successful attempts have been made to improve the asymptotic behavior of BWT. The original implementation incurs the worst case complexity of $O(N^2 log(N))$, if there is a high degree of repetitiveness in the input file. The number of characters, N, is typically quite large and such worst case behavior is unacceptable. An algorithm introduced by Larsson and Sadakane for suffix sorting, has a better worst case bound of $O(N(logN))$ [2]. Recently, linear time algorithms have been proposed [3][4], but little is known about their practicability.

The practical aspects of the suffix array construction, however, have received little attention. The locality of reference and hence the cache performance is one such factor. On modern architectures with multiple levels of memory hierarchy, the cache behavior and the memory access pattern of an algorithm play an important role in its overall performance.

The objective of this project is to devise ways of making suffix sorting for Burrows-Wheeler transform cache-efficient. In this paper, we present three approaches that can potentially lead to an improved cache behavior for suffix sorting.

- *The use of cache-oblivious distribution sort in suffix sorting.* It performs a factor of 1.76 to 3.76 slower than the quick-sort based approach for relatively small block sizes. However, we expect it to outperform quick-sort and merge-sort based approaches when the block size is increased to extents that accomodate paging. (Section 2)
- *A divide and conquer algorithm to exploit the spatial locality of reference.* It has $O(N^2)$ running time and takes 8N extra space. It displays better cache behavior than asymptotically faster algorithms such as Larsson-Sadakane. (Section 3)

- *Improvements to a linear time suffix array construction algorithm invented recently by Aluru et al [3].* We find that our implementation performs as well as contemporary fast suffix sorting implementations. It displays the linear behavior as per expectations. We also analyze the trade-offs that need to be considered in order to come up with better implementations.(Section 4)

We conclude this paper in Section 5 by summarizing the main results and providing pointers for future work based on this project.

2 Incorporating Cache-oblivious Distribution sort

2.1 Motivation

Sadakane's algorithm for fast suffix-sorting involves passes that sort the suffix according to a given key value. Each of these passes can be modelled as an integer-sorting problem. Cache-oblivious sorting of strings has not been dealt with in the past because of the implicit difficulty caused due to the low locality of references involved. However, cache-oblivious integer sorting algorithms have been developed in the past. The modelling of Sadakane's suffix-sorting as passes of integer array sorting facilitates the usage of cache-oblivious integer-sorting algorithms.

2.2 A Discussion of the Algorithm

The cache-oblivious distribution sort is described in [5]. The key aspects of the algorithm are described below. The distribution sorting algorithm uses $O(n \lg n)$ work to sort n elements, and it incurs $\Theta(1 + (n/L)(1 + \log_Z n))$ cache misses if the cache is tall. This algorithm uses a $bucket - splitting$ technique to select pivots incrementally during the distribution.

Given an array A of length n, the cache-oblivious distribution sort sorts A as follows:

1. Partition A into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$. Recursively sort each subarray.
2. Distribute the sorted subarrays into $q \leq \sqrt{n}$ buckets $B_1, B_2, \ldots, B_q$ of size $n_1, n_2, \ldots, n_q$, respectively, such that for $i = 1, 2, 3, \ldots, q-1$, we have
   (a) $\max\{x|x\epsilon B_i\} \leq \min\{x|x\epsilon B_{i+1}\}$,
   (b) $n_i \leq 2\sqrt{n}$.
3. Recursively sort each bucket.
4. Copy the sorted buckets back to array A.

Strategy

The crux of the algorithm is the Distribution step, which is Step 2 of the above algorithm. There are two invariants in the algorithm. Firstly, each bucket holds at most $2\sqrt{n}$ elements at any time, and any element in bucket B_i is smaller than any element in bucket B_{i+1}. Secondly, every bucket has a *pivot* associated with it. The pivot is a value that is greater than all elements in the bucket. Initially, there exists only one empty bucket with pivot ∞. At the end of Step 2, all elements will be in buckets and both the conditions stated in Step 2 will hold. State information for a subarray includes the index *next* of the element which is to be copied next. The other component of the state information is the index *bnum* of the bucket into which the next element is to be copied. The basic strategy is to copy the element at position *next* into bucket *bnum*. We increment *bnum* until we find a bucket for which the pivot is greater than the element. However, this basic strategy has a poor caching behavior. Hence, a recursive approach is used to distribute the subarrays into buckets. The distribution step is accomplished by a recursive procedure, DISTRIBUTE(i, j, m) which distributes elements from the ith through $(i+m-1)$th subarrays into buckets starting from B_j. Given the precondition that each subarray $r = i, i+1, \ldots, i+m-1$ has its $bnum[r] \geq j$, the execution of DISTRIBUTE(i, j, m) enforces the postcondition that $bnum[r] \geq j + m$. Step 2 invokes DISTRIBUTE$(1, 1, \sqrt{n})$. The DISTRIBUTE procedure is shown below:

```
DISTRIBUTE($i, j, m$)
if $m = 1$
   then COPYELEMS($i, j$)
   else DISTRIBUTE($i, j, m/2$)
        DISTRIBUTE($i + m/2, j, m/2$)
        DISTRIBUTE($i, j + m/2, m/2$)
        DISTRIBUTE($i + m/2, j + m/2, m/2$)
```

The base case occurs when only one bucket needs to be distributed. We then call the subroutine COPYELEMS(i,j) that copies all elements from subarray i to bucket j. If bucket j has more than $2\sqrt{n}$ elements after the insertion, it is split into two buckets of size at least $\sqrt{n}$. The deterministic median-finding algorithm is used for the partitioning of the bucket. The distribution sort described above is provably optimal in terms of number of cache-misses incurred $= \Theta(1 + (n/L)(1 + \log_Z n))$.

2.3 Testing Environment

The tests were performed on Sunfire (sf3.comp.nus.edu.sg). The technical specifications are shown in Table 1.

Table 1: Testing Environment used

Parameter	*Value*
Processor	Ultra Sparc III
Number of processors	8
Operating System	Solaris V8.0
Processor Speed	750MHz
Compiler	gcc 3.2.2
L1 Cache size	64KB 4-way Data, 32KB 4-way Instruction
L2 Cache size	8MB
RAM	8GB

2.4 Performance

We implemented Sadakane's algorithm for suffix sorting that used distribution sort for single character comparisons, and compared it with implementations of Sadakane's algorithm that used Merge Sort and Quick Sort respectively. The last two columns in Table 2, $R_{Distri/Merge}$ and $R_{Distri/Qsort}$ are the ratios of times taken by Distribution sort versus Merge sort and Quick sort respectively. The ratios were calculated for different files which varied from a random string like *rand.txt* that provides an example of a fairly good case to worse cases exemplified by a large amount of repetition like the genome chromosome sequence - *gene.txt*. Table 2 shows that in the above cases, Distribution sort performs at worst a factor of 3.76 slower than Quick sort and a factor of 3.452 slower than Merge sort. Considering the fact that Quick sort and Merge Sort do not suffer from as much memory management overhead as Distribution sort, this factor is acceptable. Moreover for the cases considered above, the cache misses may not be as dominating a factor as the extra housekeeping needed to store state information and splitting the buckets. These factors are discussed in the next subsection.

Table 2: Performance of Distribution sort compared against Quick sort and Merge sort

$FileName$	$Size$	Time(in sec.)			$R_{Distri/Merge}$	$R_{Distri/Qsort}$
		$Qsort$	$MergeSort$	$DistriSort$		
cia.txt	52KB	0.081	0.100	0.170	1.700	2.099
linux.txt	80KB	0.130	0.140	0.251	1.793	1.930
bin.txt	111KB	0.210	0.231	0.370	1.602	1.762
binop.txt	550KB	1.061	1.102	1.873	1.699	1.765
rome.txt	1618KB	2.594	2.804	9.634	3.436	3.714
rand.txt	4MB	4.775	5.201	17.954	3.452	3.760
gene.txt	8MB	32.166	53.232	100.224	1.883	3.116

Overheads incurred

The implementation of the distribution sort incurs a lot of memory management overhead unlike quick sort or even merge sort. The bucket splitting involves dynamic memory allocation using *mallocs*. This deteriorates the performance of the implementation in terms of the observed timing. The cache is not ideal, and hence the required cache-obliviousness may not be observed perfectly. Moreover, it is possible that the run-time is not dominated by the cache-misses. The extra memory management in terms of maintaining correct state information for the buckets and the subarrays contribute adversely to the observed timings. We initially spilt the original array into $\sqrt{n}$ subarrays of size $\sqrt{n}$ each. The splitting into square roots causes rounding errors, and we need to draw out special cases in order to produce correct results, which again results in an overhead. Hence, the above mentioned factors contribute to the degradation of the observed performance. However, if we increase the file sizes to arbitrary lengths, page faults occur. In this paging scenario, distribution sort can potentially perform better than merge sort or quick sort. This is an aspect which needs to be explored further.

3 A divide and conquer algorithm for suffix sorting

This algorithm extends the basic divide and conquer strategy of the merge-sort algorithm used for sorting numbers. Larger suffix arrays are formed by merging smaller suffix arrays formed recursively. In addition it has two other important aspects.

- Right-to-left processing: The relative order among the suffixes near the end of the block can be determined using a small number of character comparisons. The suffixes on their left (towards the beginning of the block) can in turn use this order to determine the relative order among themselves. The algorithm makes use of this intuition. The suffixes are sorted from right to left in a given block of data.
- Storing of match-lengths: The match length between of any two suffixes is the length of their common prefix. If two suffixes have a large common prefix, then determination of the relative order between them requires a large number of comparisons. Comparisons of many other suffixes sharing a portion of this common prefix would also involve the same comparisons. We want to avoid this duplication of work by storing the match lengths of the suffixes whose relative order has been determined.

We begin by defining the data structures used by the algorithm.

3.1 Data structures and notations

- $T[0..N-1]$: The original block of characters to be transformed
- $SA[0..N-1]$: The suffix array which finally contains the sorted order of suffixes

- $ML[0..N-1]$: An array to store the match lengths between the sorted consecutive suffixes. $ML[i]$ is the match length between suffix $SA[i]$ and $SA[i-1]$. $ML[0] = 0$
- last_match_len: The match length of the two leftmost suffixes in the sorted part.
- $POS[0..1]$: It stores the relative position of the two leftmost sorted suffixes

Suffix S_i denotes the suffix starting at $T[i]$. Similarly, suffix $SA[i]$ denotes the suffix starting at position T[SA[i]]. The main structure of the algorithm is similar to that of merge-sort as shown below. We define the precedence relation, $\prec$, on the suffixes as follows. $S_i \prec S_j$ if and only if suffix S_i is lexicographically smaller than S_j.

```
sort(unsigned char T[], int SA[], int ML[], int p, int q){
    **if** $q - p \leq 1$
        **then** compare suffixes $S_p$ and $S_{p+1}$
        store the order in POS[]
        update last_match_len
        update $SA[p..q]$ and $ML[p..q]$
        return

    mid = (p+q+1)/2;
    //sort the right half
    sort(T, SA, ML, mid, q);
    //sort the left half
    sort(T, SA, ML, p, mid-1);
    //merge the two halves
    merge(T, SA, ML, p, q);
}
```

The algorithm recursively processes the two halves of the string - first the right half and then the left half. Now, at the lowest level of recursion, the length of the string to be processed is at most 2 (i.e. $q - p \leq 1$). The two suffixes S_p and S_{p+1} are compared and their relative order is determined. The order is stored in the POS array. If $S_p \prec S_{p+1}$ then $POS = < p, p+1 >$ and if $S_{p+1} \prec S_p$ then $POS = < p+1, p >$.

Lemma 1 *If $q - p \leq 1$, the relative order of the two suffixes S_p and S_{p+1} and their match length can be completely determined by string $T[p..q+1]$ and POS.*

Proof The base case arises when $q = N-1$. Here $T[p..q]$ is a rightmost substring of the original block. It is the first short string to be processed and the POS array and last_match_len do not contain any information. However, by construction, $T[N-1]$ is the lowest character in the alphabet that does not appear in the string. Hence $T[N-2]$ and $T[N-1]$ cannot be equal. Therefore their relative order is $S_{p+1} \prec S_p$; and the match length equal to zero.

If $q \neq N-1$ then suffix S_p can be represented as a concatenation of $T[p..q]$ and $T[q+1..N-1]$. Similarly, suffix S_{p+1} is a concatenation of $T[p+1..q+1]$ and $T[q+2..N-1]$. We first compare $T[p..q]$ with $T[p+1..q+1]$. If they are not equal their relative order and the match length is evidently found. Otherwise, the order is determined by the relative order between $T[q+1]$ and $T[q+2]$ However, this order is already known, and has been stored in the POS array. Similarly, if $T[p..q]$ with $T[p+1..q+1]$ are equal, then the match length between S_p and S_{p+1} is equal to $(last\_match\_len + q - p + 1)$. Hence the result follows. $\square$

Thus, at any time, the POS array and the last_match_len variable store the information about the relative order and the match length of the two leftmost suffixes of the block, whose relative order has been determined. Hence they propagate this information from right to left at the lowest level of recursion.

The relative orders and match lengths are also stored in SA and ML arrays, to be used at all the levels of recursion and they are updated in a bottom-up manner while merging of two suffix arrays.

The merging phase involves merging of the left and right halves of the suffix array formed recursively.

```
merge(unsigned char T[], int SA[], int ML[], int p, int q){
    mid = (p+q+1)/2;
    create arrays SA1[0..mid-1] and SA2[0..q-mid]
    copy array SA[p..p+mid-1] into SA1[0..mid-1]
    copy array SA[p+mid..q] into SA2[0..q-mid]

    create arrays ML1[0..mid-1] and ML2[0..q-mid]
    copy array ML[p..p+mid-1] into ML1[0..mid-1]
    copy array ML[p+mid..q] into ML2[0..q-mid]

    prev_match_len = 0
    last_insert = LEFT
    i=j=k=r=0;
    **while** ($i < mid$ AND $j < q - mid$)
        **if** last_insert = LEFT
                **then** r = min(prev_match_len, ML1[i])
                **else** r = min(prev_match_len, ML2[j])
        compare suffixes T[SA1[i]+r] and T[SA2[j]+r]
        **if** $SA1[i + r] \prec SA2[j + r]$
                **then**
                SA[k] = SA1[i]
                **if** prev_insert = LEFT
                    **then** ML[k] = ML1[i]
                    **else**  ML[k] = prev_match_len
                i=i+1
                last_insert = LEFT
                **else**
                SA[k] = SA2[j]
                **if** prev_insert = RIGHT
                    **then** ML[k] = ML2[j]
                    **else**  ML[k] = prev_match_len
                j=j+1
                last_insert = RIGHT
        update prev_match_len
        k=k+1
    **if** $i < mid$
        **then** copy SA1[i..mid] into SA[k..q]
    **if** $j < q - mid$
        **then** copy SA2[j..q-mid] into SA[k..q]
}
```

The routine merges two suffix arrays $SA1$ and $SA2$ into SA. Its structure is similar to the merge routine used in any merge sort. Figure 1 provides an illustration of the algorithm. The next two sections describe the algorithm, especially the merging process in detail.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	b	a	b	c	a	b	a	b	a	b	a	b	d	d	\0
SA	0	0	0	0	5	4	7	6	15	9	11	8	10	12	14	13
ML	0	0	0	0	0	0	0	0	0	0	2	0	3	1	0	1

POS <5, 4>

last_match_len = 0

Figure 1: A snapshot of the data structures as the divide and conquer algorithm builds a suffix array SA for block of data T. Suffixes T_8 to T_{15} have been sorted and their order has been stored in $SA[8..15]$. Two suffix arrays of length two (SA[4..5] and SA[6..7]) have been created in the sort phase. The next step would be two merge these two suffix arrays into SA[4..7]. POS array stores the relative order of T_4 and T_5. The *last_match_len* variable stores the match length between the same.

3.2 Computation of match lengths

The routine computes the match lengths for the new suffix array. The variable *prev_match_len* stores the match lengths between the suffixes from $SA1$ and $SA2$ compared in the previous iteration of the loop. The variable *prev_insert* keeps track of which suffix array the suffix inserted in the SA in the previous iteration came from. These two variables help us to compute minimum match length between the two suffixes $SA1[i]$ and $SA2[j]$ being compared in the current iteration. For example, let the previous insertion into SA be from the left suffix array $SA1$. In other words, $SA[k-1] = SA1[i-1]$ at the beginning of the current iteration. Therefore the *prev_match_len* variable stores the match length between $SA1[i-1]$ and $SA2[j]$. Since, $ML1[i]$ stores the match length between $SA1[i-1]$ and $SA1[i]$, the match length between the suffixes $SA1[i]$ and $SA2[j]$ must be at least $r = min(prev\_match\_len, ML[i])$. Therefore substrings $T[SA1[i]..SA[i]+r-1]$ and $T[SA2[j]..SA2[j]+r-1]$ must be the same. Here, r is the length of minimun common prefix of the two suffixes. Hence the order between suffixes $SA1[i]$ and $SA2[j]$ can be determined by comparing the suffixes starting at $T[SA1[i]+r]$ and $T[SA2[j]+r]$ respectively. This mechanism avoids the necessity for repeated comparisons.

After determining the order between suffixes $SA1[i]$ and $SA2[j]$, we insert the smaller suffix into $SA[k]$. We also need to store the match length between $SA[k-1]$ and $SA[k]$ in $ML[k]$. If $SA[k-1]$ and $SA[k]$ came from the same parent array then their match length is already known and can be copied from the corresponding match length array. Otherwise, the match length between the two is equal to the value of the *prev_match_len* variable. Finally, the *prev_match_len* variable is updated to store the match length between SA1[i] and SA2[j]. This match length is found while comparing the two suffixes as explained in the next section.

3.3 Comparison of suffixes

In the merge routine the order between the suffixes $SA1[i]$ and $SA2[j]$ is determined by the order between the suffixes starting at positions $T[SA1[i] + r]$ and $T[SA2[j] + r]$. Let $d = SA2[j] - SA1[i]$, the distance between the start points of the two suffixes. Then, in the worst case, this comparison would involve the comparison between characters T[SA1[i]+r..N-d-1] and T[SA2[j]+r..N-1], which is computationally expensive. However, it turns out that this worst case behavior can be avoided based on the data structures that have been built so far. Recall that SA1[0..mid-1] contains the order of the suffixes $S_p, S_{p+1}, \dots, S_{mid-1}$. Similarly, SA2[0..q-mid] stores the order of the suffixes $S_{mid}, S_{mid+1}, \dots, S_q$. Since the algorithm sorts suffixes from right to left, we know that the relative order of some suffixes starting from S_{q+1} is already known. The folowing observation formalizes this intuition based on the divide and conquer nature of the algorithm and right to left processing.

Observation 2 *While merging the suffix arrays for suffixes $S_p, S_{p+1}, \dots, S_{mid-1}$ and $S_{mid}, S_{mid+1}, \dots, S_q$, if $q \neq N-1$, the relative order of at least M suffixes $S_{q+1}, S_{q+2}, \dots, S_{q+M}$ is known and has been stored in $SA[q+1..q+M]$, where $M = q - p + 1$ and $M \leq N - q$*

Note that if $q = N - 1$, the relative order of suffixes $S_p, S_{p+1}, \dots, S_q$ can be completely determined by the substring $T[p..q]$, since the last character $T[N-1]$ is the lowest character in the alphabet, not appearing in the original block. For all other cases, if two substrings of equal length in $T[p..q]$ are found to be equal then the relative order of their corresponding suffixes can be determined using the order of already sorted suffixes.

While comparing the suffixes starting at positions $T[SA1[i] + r]$ and $T[SA2[j] + r]$, we need to handle 3 cases.

Case 1: $SA1[i] + r \leq q$
We first compare $T[SA1[i]+r..q]$ and $T[SA2[j]+r..q+d]$. Depending on their equality there are two subcases.

Case 1a: $T[SA1[i] + r..q] \neq T[SA2[j] + r..q + d]$
In this case, the order has evidently been found. The match length between the two suffixes is the sum of r and the length of the common prefix of $T[SA1[i] + r..q]$ and $T[SA2[j] + r..q + d]$.

Case 1b: $T[SA1[i] + r..q] = T[SA2[j] + r..q + d]$
Now we must determine the order between S_{q+1} and S_{q+d+1}. Start scanning the suffix array from $SA[q+1]$ and find positions a and b in the array such that $SA[a] = q + 1$ and $SA[b] = q + d + 1$. If $a < b$, then the $S_{q+1} \prec S_{q+d+1}$, which implies $S_{SA1[i]} \prec S_{SA2[j]}$. Similarly, if $b < a$, then $S_{SA2[j]} \prec S_{SA1[i]}$. In this case, the match length between $S_{SA1[i]}$ and $S_{SA2[j]}$ is the sum of $q - SA[i] + 1$ and the match length of S_{q+1} and S_{q+d+1}. The match length between S_{q+1} and S_{q+d+1} found by scanning the match length array ML as follows.

$$ml_{ab} = \begin{cases} min(ML[a+1], ML[a+2], \dots, ML[b]) & \text{if } a < b \\ min(ML[b+1], ML[b+2], \dots, ML[a]) & \text{if } b < a \end{cases}$$

Case 2: $SA1[i] + r > q$
In this case, the order between $SA1[i]$ and $SA2[j]$ is same as the order between S_{q+1} and S_{q+d+1}. The order and match length is determined in the same manner as case 1b.

3.4 Timing analysis

In this analysis, for the sake of simplicity of explanation, we assume the block size, N, to be a power of 2, such that $N = 2^c$. Also, we measure the running time of the algorithm in terms of the number of character comparisons. A comparison of two suffixes would involve multiple character comparisons. For example, consider the merging of two suffix arrays, $SA[p..mid-1]$ and $SA[mid..q]$. We define the character comparisons on the portion of the block that is currently being merged as *local comparisons*. Therefore, the character comparisons performed the substring $T[p..q]$ during the merge process are the local comparisons. They correspond to Case 1 described in Section 3.3. If the local portions of the two suffixes are equal, then the sorted portion of the suffix array starting from $SA[q+1]$ is scanned to determine their relative order. The comparisons involved in the scanning operation are termed as *remote comparisons.* We analyse the number of local and remote comparisons separately.

Local comparisons

Consider the merging of two suffix arrays of length $N/2$ each at the topmost level of recursion. It involves at most N suffix comparisons. Further, each suffix comparison may involve at most N local comparisons. Note that this is a loose bound, because even for a block with high degree of repetitiveness, the number of comparisons would be reduced by the match length heuristic. In total, N^2 local comparisons are needed.

Now consider the next level of recursion, which involves two merges of two suffix arrays, each of length $N/4$. Here there are at most N suffix comparisons, $N/2$, in each merge. Now, number of local comparisons in each suffix comparison would be at most $N/2$, resulting in at most $N^2/2$ local comparisons.

Similarly, we can calculate the number of local comparisons at each level of recursion. At the lowest level, when 2^{c-2} merges result in as many suffix arrays of length 4, $N^2/2^{c-2}$ local comparisons are required. So, the total number of local comparisons is:

$$\begin{aligned}\text{Total no. of local comparisons, L(N)} &= N^2 + \frac{N^2}{2} + \ldots + \frac{N^2}{2^{c-2}} \\ &= 2(1-(1/2)^{c-1})N^2 \\ &= O(N^2) \qquad (1)\end{aligned}$$

Remote comparisons

Consider the merging of two suffix arrays of length $N/2$ each at the topmost level of recursion. Here there are no remote comparisons required, since the entire block of data is involved in merging, the order between any two suffixes can be determined by only local comparisons.

Now consider the next level of recursion, which involves two merges of two suffix arrays, each of length $N/4$. Again, the merging of suffix array $SA[N/2..(3N/4-1)]$ and $SA[3N/4..N-1]$ does not require any remote comparisons. The next merge between $SA[0..(N/4-1)]$ and $SA[N/4..(N/2-1)]$ may require scanning of $SA[N/2..N]$, if the two suffixes turn out to be equal after the local comparisons. This follows from Observation 2. There are $N/2$ such suffix comparisons. Hence the total number of remote comparisons required is at most $N/2 * N/2$.

At the next level, is one merge of length $N/4$ that requires scanning the sorted part of length $N/2$ and two merges of length $N/4$ that involve scanning sorted parts of length $N/4$ - again from Observation 2. The total number of remote comparisons is at most $N/4 * N$. Proceeding in this manner, the total number of remote comparisons, R(N), can be found as follows.

$$
\begin{aligned}
R(N) &= 0 \\
&\quad + \frac{N}{2} * \frac{N}{2} \\
&\quad + \frac{N}{4} * \frac{2N}{2} \\
&\quad + \frac{N}{8} * \frac{3N}{2} \\
&\quad + \ldots \\
&\quad + \frac{N}{2^{c-2}} * \frac{((c-2)N}{2} \\
&= \frac{N^2}{4} + \frac{2N^2}{8} + \frac{3N^2}{16} + \ldots + \frac{(c-2)N^2}{2^{c-1}} \\
&= N^2(\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \ldots + \frac{c-2}{2^{c-1}}) \qquad (2)
\end{aligned}
$$

$$
\text{2*R(N)} = N^2(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \ldots + \frac{c-2}{2^{c-2}}) \qquad (3)
$$

Subtracting Equation 2 from 3:

$$
\begin{aligned}
\text{R(N)} &= N^2(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^{c-2}} - \frac{c-2}{2^{c-1}}) \\
&= N^2\left((1 - (1/2)^{c-1}) - \frac{c-2}{2^{c-1}}\right) \\
&= O(N^2) \qquad (4)
\end{aligned}
$$

From Equations 1 and 4, the running time of the divide and conquer algorithm is $O(N^2)$.

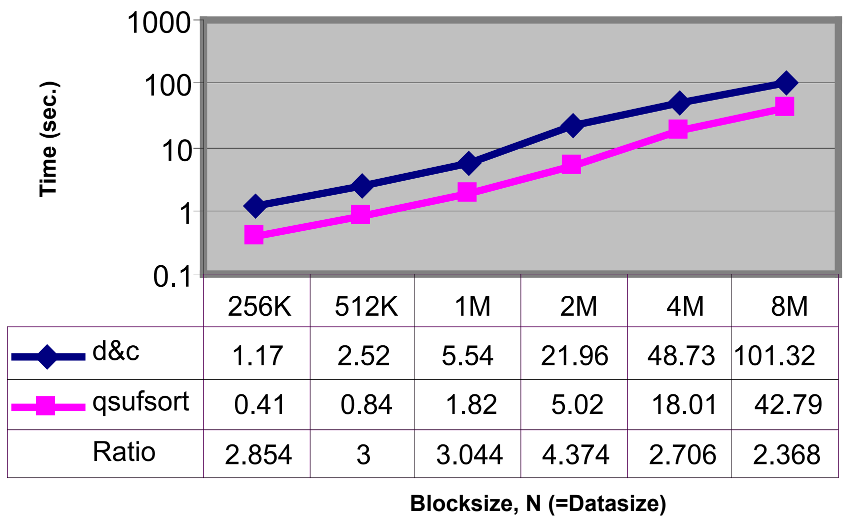
3.5 Memory requirements

The memory efficiency of a suffix sorting algorithm is measured in terms of extra space required besides the original character block and the suffix array. In the above discussion of the algorithm, we have assumed the existence of two temporary arrays to hold the two sorted halves of the suffix array being merged into a single array. Hence we would need two temporary arrays of length $N/2$ each to merge suffixes. Similarly, we would need two temporary arrays of length $N/2$ each for merging match lengths.

However, this requirement can be reduced by a simple modification to the algorithm. At the beginning of the merge routine, only the left sorted half of SA needs to be stored in a temporary array. The right half of SA and the temporary array can then be merged into SA. Similarly, we would need only one temporary array of length $N/2$ for merging match lengths. Besides the temporary arrays, of course, we would need two arrays of size N to store the merged suffixes and match lengths. All temporary arrays and the match length array, ML, are integer arrays. Therefore the total extra space required is $8N$ bytes, for a character data of length N.

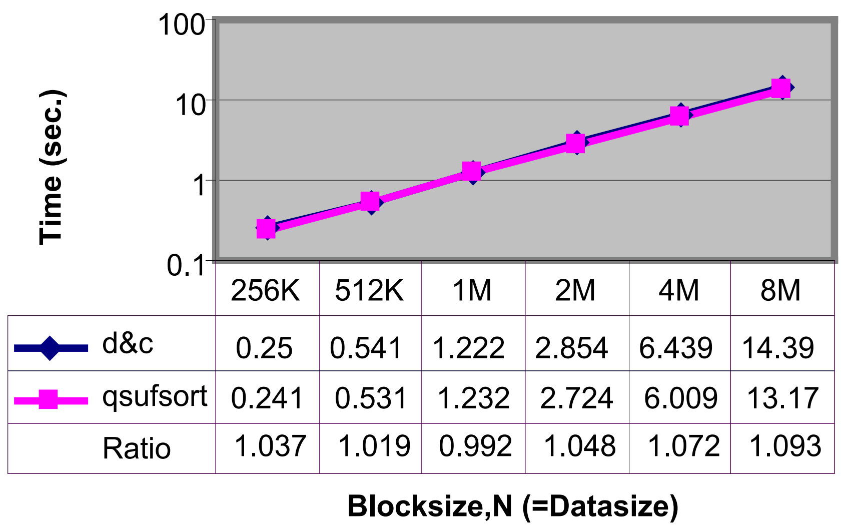
3.6 Performance

We compared the performance of the divide and conquer algorithm with the qsufsort suffix sorting algorithm developed by Larsson-Sadakane [2]. It uses the doubling technique first introduced by Manber-Mayers in [6]. It has a worst case time complexity of $O(Nlog(N))$ and with a good implementation, it is considered as the fastest algorithm for generic suffix sorting.



	256K	512K	1M	2M	4M	8M
◆ d&c	1.17	2.52	5.54	21.96	48.73	101.32
■ qsufsort	0.41	0.84	1.82	5.02	18.01	42.79
Ratio	2.854	3	3.044	4.374	2.706	2.368

Figure 2: Perfomance of divide and conquer algorithm as compared to the qsufsort algorithm on a Sun UltraSPARC III cluster. This experiement has been perfomrmed on Human Genome dataset files.



	256K	512K	1M	2M	4M	8M
◆ d&c	0.25	0.541	1.222	2.854	6.439	14.39
■ qsufsort	0.241	0.531	1.232	2.724	6.009	13.17
Ratio	1.037	1.019	0.992	1.048	1.072	1.093

Figure 3: Perfomance of divide and conquer algorithm as compared to the qsufsort algorithm on a Pentium4 2.4Ghz machine. This experiment has been performed on Human Genome dataset files.

Figure 2 shows the performance of the two algorithms for varying block size, N, on Human Genome dataset on a Sun UltraSPARC III cluster. Even though the qsufsort algorithm outperforms the divide and conquer algorithm, the time taken by the divide and conquer version exceeds the time taken by qsufsort by a factor of 2-5 for sufficiently large blocksize.

Figure 3 shows the result of a similar experiment on a Pentium4 2.4GHz machine. On this machine, the divide and conquer algorithm performs almost as well as the qsufsort algorithm.

Besides the running time, we also analyzed the cache performance of the two algorithms using a cache simulator. The experiment was performed on two different data sets, reuters corpus and protein sequence dataset, using a fixed block size of $N = 1,048,576$. Tables 3 and 4, summarize the cache behavior of the two algorithms. An L1 data cache of size 16KB was used in this experiments. It can be seen that even though the divide and conquer algorithm makes many more data references than the qsufsort algorithm, it still incurs lesser cache misses.

Table 3: Cache Performance: Reuters dataset [a]

	qsufsort	d&c
# data references	525,305K	1,910,425K
L1 data cache misses	14,614K	13,707K
Cache miss ratio	2.7%	0.7%

[a]Input file size: 1MB, Blocksize, N=1MB

Table 4: Cache Performance: Protein sequence dataset [a]

	qsufsort	d&c
# data references	531,201K	2,626,356K
L1 data cache misses	16,323K	12,945K
Cache miss ratio	3.0%	0.4%

[a]Input file size: 890KB, Blocksize, N=1MB

3.7 Discussion

From the performance results it is clear that the qsufsort algorithm by Larsson-Sadakane is still faster for generic suffix sorting. This is expected considering the asymptotic running times of the two algorithms. However, based on the results on different data sets, one can say that the divide and conquer algorithm has a good average case running time. Most importantly, it shows a good cache behavior. Secondly, it does not require any memory parameters to be set, besides the blocksize, N, which is usually determined by the compression algorithm using the BWT. For these reasons, the algorithm is likely to be scalable and is likely to perform well across various memory hierarchies.

4 Linear time construction of suffix arrays

4.1 Motivation

A linear time algorithm to sort all suffixes of a string over a large alphabet of integers is presented in [3]. This could potentially be a major break-through in the field of suffix sorting which has witnessed several improvisations to the $\mathrm{O}(n \lg n)$ algorithm in terms of improving the constants, and coming up with better implementations. The work done in [3] proves the linear bound on the proposed algorithm. With the knowledge gained from previous suffix-array implementations, we improvised on the linear time algorithm and came up with an efficient implementation. We examine the performance of our implementation, and analyze the trade-offs involved.

4.2 Description of the algorithm

Consider a string $T = t_1 t_2 \ldots t_n$ over the alphabet $\Sigma = 1 \ldots n$. We denote the last character of T by \$, assume it to be the lexicographically smallest character. Let $T_i = t_i t_{i+1} \ldots t_n$ denote the suffix of T starting with t_i. For strings α and β, we use $\alpha \prec \beta$ to denote that α is lexicographically smaller than β. We classify the suffixes into two types: Suffix T_i is of type S if $T_i \prec T_{i+1}$, and is of type L if $T_{i+1} \prec T_i$. Since, the last suffix does not have a successor, we can classify it as both type S and type L.

Lemma 3 *All suffixes of T can be classified as either type S or type L in O(n) time.*

Proof Consider a suffix $T_i (i < n)$

Case 1 If $t_i \neq t_{i+1}$, we can determine if T_i is of type S or L by only comparing t_i and t_{i+1}.

Case 2 If $t_i = t_{i+1}$, we find the smallest $j > i$ such that $t_j \neq t_i$.

if $t_j > t_i$, then suffixes $T_i, T_{i+1}, \ldots, T_{j-1}$ are of type S.

if $t_j < t_i$, then suffixes $T_i, T_{i+1}, \ldots, T_{j-1}$ are of type L.

Therefore, all suffixes can be classified using a left to right scan of T in $O(n)$ time. □

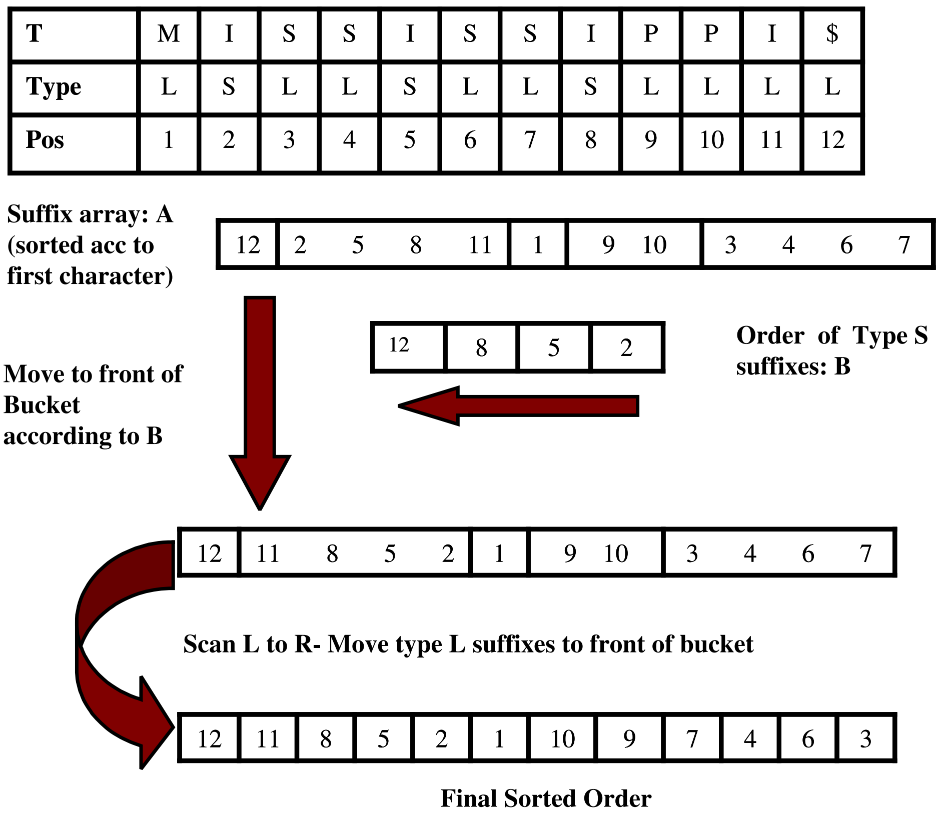
Another important property of type S and type L suffixes is, if a type S suffix and a type L suffix both begin with the same character, the type S suffix is lexicographically greater than the type L suffix. Therefore, among all suffixes that start with the same character the type S suffixes appear after the type L suffixes. Let A be the suffix array, and B be the array of suffixes of type S, sorted in lexicographic order. The final sorted order is obtained from B as follows:

1. Bucket all suffixes of T according to their first character in array A. Each bucket consists of all suffixes that start with the same character. This step takes $O(n)$ time.
2. Scan B from right to left. For each suffix encountered in the scan, move the suffix to the current end of its bucket in A, and advance the current end of the bucket by one position to the left. After the scan of B is completed, all type S positions are in their correct positions in A. The time taken for this step is $O(|B|)$, which is bounded by $O(n)$.
3. Scan A from left to right. For each entry $A[i]$, if $T_{A[i]-1}$ is a type L suffix, move it to the current front of its bucket in A, and advance the front of the bucket by one. This takes $O(n)$ time. At the end of this step, A contains all suffixes of T in sorted order.

The remaining task is to sort all type S suffixes in T. The substring $t_i \ldots t_j$ is called a type S substring if both i and j are type S positions, and every position between i and j is a type L position. For each suffix T_i, define its $S - distance$ to be the distance from its starting position i to the nearest type S position to its left. We first sort all the type S substrings in $O(n)$ time. This is done by a single traversal of $S - distance$ lists. Sorting of the type S substrings generates buckets where all substrings are identical. A final recursive sorting of these buckets yields the final lexicographic ordering of the type S suffixes.

4.3 An example

Figure 4 illustrates obtaining the final sorted order of suffixes.



T	M	I	S	S	I	S	S	I	P	P	I	$
Type	L	S	L	L	S	L	L	S	L	L	L	L
Pos	1	2	3	4	5	6	7	8	9	10	11	12

Suffix array: A (sorted acc to first character)
12 | 2 5 8 11 | 1 | 9 10 | 3 4 6 7
12 | 8 | 5 | 2
Order of Type S suffixes: B
Move to front of Bucket according to B
12 | 11 8 5 2 | 1 | 9 10 | 3 4 6 7
Scan L to R- Move type L suffixes to front of bucket
12 | 11 | 8 | 5 | 2 | 1 | 10 | 9 | 7 | 4 | 6 | 3
Final Sorted Order

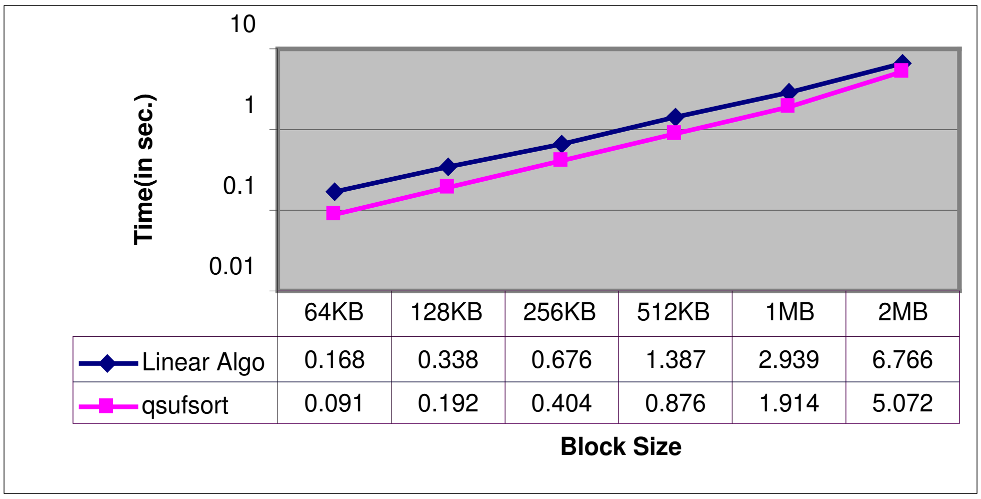
Figure 4: An example of how to obtain the final sorted ordering of the suffixes

4.4 An implementation strategy

In order to sort the type S suffixes in T, we have to sort all type S substrings. This sorting generates buckets where all the substrings in a bucket are identical. We generate a new string T' by scanning T from left to right and for each type S position in T writing the bucket number of the type S substring from that position to T'. Then we sort T' recursively. The ordering of T' corresponds to the sorted order of the type S suffixes. However, if a bucket contains only one type S substring, the position of the corresponding type S suffix in the sorted order is already known. Let $T' = b_1 b_2 \ldots b_m$. Consider the maximal substring $b_i \ldots b_j (j < m)$ such that each $b_k (i \leq k \leq j)$ contains only one type S substring. We can then shorten T' by replacing each such maximal substring $b_i \ldots b_j$ with its first character b_i. Since $j < m$, the bucket corresponding to '$' is never dropped. We can compute the shortened version of T' without having to compute T' first and then shorten it. This method has the effect of eliminating some of the suffixes of T'. It also modifies each suffix that contains a substring that is shortened.

4.5 Performance

The tests were performed on Sunfire (sf3.comp.nus.edu.sg).



	64KB	128KB	256KB	512KB	1MB	2MB
◆ Linear Algo	0.168	0.338	0.676	1.387	2.939	6.766
■ qsufsort	0.091	0.192	0.404	0.876	1.914	5.072

Figure 5: Performance of Linear time algorithm as compared to qsufsort

It can be seen from Figure 5 that as expected, the time taken by the linear algorithm is linearly proportional to the block size. It performs as well as the fastest available suffix sorting algorithm available : *qsufsort*, for larger block sizes.Our implementation is still crude as compared to the refined implementations of the $O(n \log n)$ algorithms like qsufsort. Some of the trade-offs are discussed below.

Trade-offs

The current implementation suffers from some memory-management overhead. We presently use 3 integer arrays of size n, and 3 boolean arrays(2 of size n, and 1 of size $n/2$). This gives rise to a total space of $12n$ bytes plus $2.5n$ bits. This is greater than the $8n$ bytes used by Manber and Myers' suffix sorting algorithm [6] which takes $O(n \log n)$ time. Hence, we find that a trade-off between space and time exists. However, with more efficient implementations, the advantages in terms of linearity of time would outweigh problems caused due to the extra space requirements. In fact we found that certain optimizations which included storing the reverse correspondences between buckets and suffixes led to a marked improvement in performance.

5 Conclusions and Future Work

Increasing the Block size in the Burrows-Wheeler transform provides better compression. But, the cost incurred for the sorting of the rotated strings proves to be the bottleneck in achieving better compression ratios in a shorter time. We suggest techniques for fast suffix sorting that would ease this bottleneck.

We implemented a cache-oblivious Distribution sort based suffix sorting. We found that it incurs memory management overheads, and performs a factor of 1.76 to 3.76 slower than the Quick sort based approach, and a factor of 1.6 to 3.45 slower than the Merge sort based approach. However, we would expect our approach to be more effective when the Block sizes are increased to such an extent that the paging effects become more significant. We proceeded to develop an $O(N^2)$ divide and conquer algorithm that is cache-efficient and requires no memory parameters to be set. Even though it is found to be slower than the asymptotically superior algorithms such as qsufsort by a factor of upto 4.3 for sufficiently large block sizes, it shows a much better cache behaivor. We then present issues dealing with the linear time suffix sorting algorithm. We examine the tradeoffs between space and time. The linear time algorithm performed as expected, and was comparable to the fastest suffix sorting available- qsufsort.

Several issues still remain to be investigated. The divide and conquer algorithm that we have introduced essentially processes the block of data from right to left. This sequential nature of the algorithm will not allow it to be easily parallelizable. A variant of the algorithm should be developed that offers parallelism, and yet maintains the good cache behavior of the original algorithm. We have tried to make our implementation of the linear time algorithm cache-efficient. Nevertheless the implementation is still crude and needs to be refined. A more thorough analysis of the experimental data is needed in order to explain the experimental results satisfactorily. Experiments should further be performed to analyze the effects of arbitrary block sizes. New structures like the suffix cactus [7] offer an interesting point of view to the family of suffix structures. Employing these structures to facilitate fast suffix sorting is another area which can be looked into.

References

[1] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm.," Tech. Rep. 124, Digital Systems Research Center, Palo Alto, 1994.

[2] N. J. Larsson and K. Sadakane, "Faster suffix sorting," 1999.

[3] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," 2003.

[4] J. Kärkkäinen and P. Sanders, "Simple linear work suffix array construction," in *Proc. 13th International Conference on Automata, Languages and Programming*, Springer, 2003.

[5] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science, FOCS '99*, 1999.

[6] U.Manber and G.Myers, "Suffix arrays: a new method for on-line search," in *SIAM Journal on Computing*, pp. 22:935–48, 1993.

[7] J. Karkkainen, "Suffix cactus: A cross between suffix tree and suffix array," in *Combinatorial Pattern Matching*, pp. 191–204, 1995.