Parallel and Memory-efficient Burrows-Wheeler Transform

Shinya Hayashi, Kenjiro Taura
Graduate School of Information Science and Technology
The University of Tokyo
Tokyo, Japan
Email: {hayashi, tau}@eidos.ic.i.u-tokyo.ac.jp

***Abstract*—In large scale applications of genome analysis, compressed indexes are very useful to avoid the memory usage imposed by suffix arrays. FM-Index is one of the most important such compressed indexes. While it is very memory efficient, its construction requires much larger working memory during construction than the final index itself. Specifically, the Burrows-Wheeler transform (BWT), which is computed on its way to constructing FM-Index, requires a suffix array, which is larger than BWT and FM-Index. To address this, there have been many efforts to compute the Burrows-Wheeler in a smaller space, one of the fastest of which achieves a linear time complexity. Since it is unlikely that any sequential algorithm achieves a sublinear time complexity, the next logical step is a parallel algorithm that achieves a sublinear critical path with a space requirement smaller than suffix arrays. In this paper, we propose such an algorithm. It is based on a divide-and-conquer algorithm, which can be efficiently executed with task parallel models. We evaluated our algorithm on human genome data and obtained almost the same speed with BWT-IS, which is the known fastest algorithm.**

***Keywords*-Burrows-Wheeler transform; suffix array; parallel computing;**

I. Introduction

Suffix array[1] is one of the most commonly used index structures in computational biology and text processing. Suffix array is able to search a text for arbitrary substrings. However, for the original text of n characters, its suffix array consumes $\Omega(n \log n)$ bits of memory, which is $(\log n / \log \sigma)$ times larger than the text itself, where σ is the alphabet size.

To address the above problem, *compressed indexes* have been studied, among which FM-Index[2] has been drawing much attentions [3][4][5][6]. FM-Index has enough information to restore the original text, so the original text is no longer required when one searches the text. In addition, FM-Index can be compressed to the k-th order empirical entropy of the text.

While FM-Index has a number of useful properties, its efficient construction has been challenging. FM-Index construction algorithm first computes a string called Burrows-Wheeler transform[7] (BWT). A straightforward construction algorithm directly following the definition would construct suffix array, requiring $O(n \log n)$ intermediate space. BWT construction algorithm that does not require $O(n \log n)$ space has been proposed, the fastest of which achieved time complexity of $O(n)$ [4]. Since it is unlikely that any sequential algorithm achieves a sublinear time complexity, the next logical step is to seek a parallel algorithm that achieves $o(n)$ critical path and $o(n \log n)$ space complexity. This paper proposes such an algorithm.

This paper is organized as follows. Section II provides theoretical backgrounds related to our algorithm. Section III explains related work. Section IV describes our divide-and-conquer BWT construction algorithm, then section V its parallelization. Section VI shows experimental evaluation of our method on genome sequence. We will conclude in section VII.

II. Theoretical Background

A. Suffix Array

Given a text (string) T, we denote by $T[i]$ its i-th character (i starting from zero) and denote by $T[i, j]$ the substring starting at $T[i]$ and ending at $T[j]$ (so it has $(j-i+1)$ characters). As is customary, we consider a special sentinel character, often denoted as '$', immediately follows the original text. With this convention, given a text T of $(n+1)$ characters including the sentinel, a *suffix* T_i is the substring starting at $T[i]$ and ending at $T[n]$. In other words, $T_i = T[i, n]$.

The *suffix array, SA,* of T is a permutation of $\{0, 1, \ldots, n\}$, whose j-th element $SA[j]$ is the starting offset of T's lexicographically j-th suffix. That is, $SA[j] = i$ means that, among all suffixes of T, T_i's lexicographical rank is j. The sentinel character '$' is considered to be smaller than any other character. Table I shows an example of suffix array.

Suffix array is an index structure by which one can efficiently find any substring in the text; thus it is applicable to genome sequences, texts of some Asian languages like Japanese, and binary data which cannot be easily delimited into natural words. Memory space required for suffix array is $\Omega(n \log n)$ bits, which is $\log n / \log \sigma$ times larger than the original text, where σ is the size of the alphabet. The problem is severe for strings of small alphabet sizes such as genome sequences.

B. Burrows-Wheeler Transform

To address the memory usage problem of suffix array, compression techniques are effective. In particular, LF-

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

mapping we introduce in the next subsection and its index structure FM-Index [2] are very useful. Construction of FM-Index involves constructing the *Burrows-Wheeler transform*, or BWT in short, whose definition is given below.

Definition 1 (Burrows-Wheeler transform): Suppose T denotes the original text of length $(n+1)$ including the sentinel and *SA* the suffix array of it. Then, Burrows-Wheeler transform of T, T^{BWT}, is a permutation of T defined as follows.

$$T^{BWT}[i] = \begin{cases} T[SA[i]-1] & (\text{if } SA[i] > 0), \\ T[n] & (\text{if } SA[i] = 0), \end{cases} \quad (1)$$

We show an example of BWT in Table I. In this table, the text is $T =$ "mississippi$", and the BWT is $T^{BWT} =$ "ipssm$pissii".

C. LF-Mapping

LF-Mapping is an auxiliary mapping that allows us to extend a suffix by one character towards the head of the text. Specifically, LF(i) is defined as follows.

$$\text{LF}(i) = j \text{ such that } \begin{cases} SA[j] = SA[i] - 1 & (\text{if } SA[i] > 0), \\ SA[j] = n & (\text{if } SA[i] = 0). \end{cases}$$

That is, given suffix S_k, whose lexicographical rank is i, LF(i) is the lexicographical rank of suffix S_{k-1} (S_k extended by one character ahead). Such a mapping could be straightforwardly constructed by first building the suffix array and then finding, for each i, the index j satisfying $SA[j] = SA[i] - 1$, but this obviously requires as much memory as suffix array.

Remarkably, one can calculate LF(i) by a more compact data structure with the following observation due to [7].

Lemma 1 (LF-Mapping): Let T be the original text and L its BWT (i.e., $L = T^{BWT}$). Denote by $C(c)$ the number of characters in T that are lexicographically smaller than c, and denote by $\text{rank}_c(B, i)$ the number of times c appears in $B[0, i]$ (the prefix of B up to the character $B[i]$). Then the LF-mapping can be calculated as follows.

$$LF(i) = C(L[i]) + \text{rank}_{L[i]}(L, i) - 1 \quad (2)$$

Table I
AN EXAMPLE OF SUFFIX ARRAY.

	suffixes	sorted suffixes	suffix array	T^{BWT}
0	mississippi$	$	11	i
1	ississippi$	i$	10	p
2	ssissippi$	ippi$	7	s
3	sissippi$	issippi$	4	s
4	issippi$	ississippi$	1	m
5	ssippi$	mississippi$	0	
6	sippi$	pi$	9	p
7	ippi$	ppi$	8	i
8	ppi$	sippi$	6	s
9	pi$	sissippi$	3	s
10	i$	ssippi$	5	i
11	$	ssissippi$	2	i

In this paper, we use LF-mapping to reduce the size of intermediate suffix arrays; instead of building a full suffix array, we build a sparsely sampled suffix array. By using LF-mapping, we are able to reconstruct a suffix of arbitrary rank in the original text. This is crucial in merging BWTs built from two substrings, as described in Section IV.

We show an example of LF-Mapping in Table II, where $T =$ "mississippi$". The fourth column is derived from the second by extending each suffix by a character ahead. The column LF(i) is, by its definition, the rank of the suffix in the fourth column. Observe this can alternatively be obtained using the above Lemma, as illustrated in the sixth column. Notice that L is the last characters of the second column.

D. Wavelet matrix

Wavelet matrix[8] is a data structure for efficiently calculating LF-mapping, or more specifically, $\text{rank}_c(B, i)$. Recall that it is the number of occurrences of character c in the prefix $B[0, i]$. Wavelet matrix uses $o(n \log \sigma)$ extra space in addition to the original text and allows us to compute $\text{rank}_c(B, i)$ in $O(\log \sigma)$ time.

Fig. 1 illustrates the wavelet matrix for the alphabet { a, b, …, h } and the input text T="bbaffghccedahhcgbdde". To construct wavelet matrix from the string, we first assign a unique binary number to each character of the alphabet, in the alphabetical order. In our example, we assign "000" to 'a', "001" to 'b', and so on. Building wavelet matrix consists of $\log \sigma$ iterations. In the first iteration, the first (most significant) bits of all characters in the string are taken as a sequence of n bits, then all characters are stably partitioned by their first bits. In our example, all characters in { a, b, c, d } will be partitioned to the left of the string and all characters in { e, f, g, h } to the right. In the second iteration, we perform a similar step against the string obtained in the end of the first iteration, this time using the second (the second most significant) bit of each character. That is, we take the second bits of all the characters as a sequence of n bits; and partition all the characters by their second bits,

Table II
AN EXAMPLE OF LF-MAPPING.

i (rank)	sorted cyclic shifts	L	extended by a character	LF(i)	$C(L(i))$ $+\text{rank}_{L[i]}(L, i)$ -1
0	$mississippi	i	i$mississipp	1	$1+1-1$
1	i$mississipp	p	pi$mississip	6	$6+1-1$
2	ippi$mississ	s	sippi$missis	8	$8+1-1$
3	issippi$miss	s	sissippi$mis	9	$8+2-1$
4	ississippi$m	m	mississippi$	5	$5+1-1$
5	mississippi$	$	$mississippi	0	$0+1-1$
6	pi$mississip	p	ppi$mississi	7	$6+2-1$
7	ppi$mississi	i	ippi$mississ	2	$1+2-1$
8	sippi$missis	s	ssippi$missi	10	$8+3-1$
9	sissippi$mis	s	ssissippi$mi	11	$8+4-1$
10	ssippi$missi	i	issippi$miss	3	$1+3-1$
11	ssissippi$mi	i	ississippi$m	4	$1+4-1$

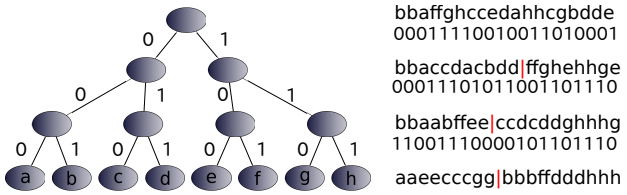

Figure 1. Wavelet matrix

moving { a, b, e, f } to the left and { c, d, g, h } to the right. We repeat this step for all bits ($\log \sigma$ iterations).

The bit sequence obtained at each iteration can be represented by a succinct bit vector in $o(n \log \sigma)$ space and supports $O(\log \sigma)$ time algorithm for calculating $\text{rank}_c(B, i)$. We refer the reader to [8] for details.

III. Related Work

Many authors have shown that BWT can be computed without constructing suffix array for the whole text[3][4][5][9][10]. Particularly, BWT-IS[4] is very fast both in theory and practice; it achieves $O(n)$ time complexity, and uses less memory compared to other work. Their algorithm is based on the suffix array construction algorithm called SA-IS[11]. In SA-IS, the whole suffix array is induced from a carefully sampled suffix array. BWT-IS simulates this algorithm by using BWT only, and induces whole BWT from the sampled one.

There is also a parallel algorithm to compute BWT[12]. This algorithm is based on a sampling sort and parallelized by MapReduce. Suffix array is constructed first and then BWT is computed by equation 1. In this algorithm, a text must be put into a distributed file system, so that each node is able to access the whole text during suffix array construction. The communication cost can be large when the number of compute node increases.

Another way to compute BWT in parallel is to compute suffix array by a parallel suffix array construction algorithm such as pDC3[13] and then compute BWT by equation 1. This algorithm constructs the whole suffix array of the input text, thus the memory usage of $\Omega(n \log n)$ bits is inevitable.

To the best of our knowledge, no algorithms have been proposed that achieve both $o(n)$ (parallel) time complexity and $o(n \log n)$ space complexity.

IV. Divide-and-conquer BWT

A. Overview

Our algorithm for computing BWT is based on a divide-and-conquer strategy; we split the text recursively into short substrings, compute BWT for each of them, and merge them. A divide-and-conquer algorithm is then parallelized with task parallelism (Section V). We outline our algorithm in Fig. 2.

Each recursive call receives a target interval $[i, j]$ (via the parameters "begin" and "end" in the code), which is

```
1  void ComputeBWT(int begin, int end, Node& parent){
2    if(inputSize<=LEAF_SIZE){
3      ComputeLeaf(begin, end, globalBegin, globalEnd, parent);
4      return;
5    }
6    Node left, right;
7    int mid=inputSize/2;
8    // Recursive function call
9    ComputeBWT(begin, begin+mid, left);
10   ComputeBWT(begin+mid, end, right);
11   // Merge left and right child node
12   MergeNode(left, right, parent, left.begin(), left.end(),
13     right.begin(), right.end());
14 }
```

Figure 2. Pseudo code of the outline of our algorithm.

the starting positions of suffixes it is in charge of. That is, given a target interval $[i, j]$, it is responsible for suffixes $T_i, T_{i+1}, \ldots, T_j$ and will build the BWT from them. Since i-th character of a BWT is the previous character of the suffix of rank i, the returned BWT is a permutation of $T[i-1], T[i], \ldots, T[j-1]$.

The crux of the algorithm is how to merge two BWTs built from two adjacent substrings with a small amount of memory. Merging would be straightforward if we materialize the full suffix arrays of both, but it would incur the $\Omega(n \log n)$ space requirement we are trying to avoid. To this end, each recursive call returns, along with a BWT, a *sampled* suffix array[6] and a bit array *mark* indicating which elements of the suffix array are sampled.

Another important point is that comparing two suffixes costs time complexity of $O(n)$ in the worst case. To address this problem, we use difference cover sample, DCS[3]. DCS enables us to limit the number of character comparisons in the worst case. Below, we first explain DCS, and then describe how BWTs, sampled suffix arrays, and mark bit arrays of substrings are computed in leaves of the call tree and then merged in internal nodes.

B. Difference cover sample

Difference cover sample, or *DCS*[14], is a data structure that enables efficient comparison of suffixes. In DCS, one first chooses a set of integers called *difference cover,* or *DC,* from the range $[0, v)$ where v is a parameter which defines the maximum number of character comparisons necessary to compare any pair of suffixes. They are defined as follows.

Definition 2 (Difference cover[14]): A set $D \subseteq [0, v)$ is a difference cover modulo v if

$$\{(i - j) \bmod v \mid i, j \in D\} = [0, v) \tag{3}$$

Definition 3 (Difference cover sample[14]): A set $C \subseteq [0, n)$ is a difference cover sample modulo v if

$$C = \{i \in [0, n) \mid i \bmod v \in D\} \tag{4}$$

```
1  void ComputeLeaf(int begin, int end,
2    int globalBegin, int globalEnd, Node& leaf){
3    vector<int> sa;
4    // Construct SA
5    leaf.StringSort(begin, end, globalEnd, sa);
6    // Compute BWT
7    for(int i=0; i<leaf.bwt.size(); i++){
8      if(globalBegin==begin && sa[i]==0)
9        leaf.bwt[i]=$;
10     else leaf.bwt[i]=T[begin+sa[i]−1];
11   }
12   if(!leaf.isRootNode() && leaf.isLNode())
13     // Construct wavelet matrix only for left node
14     leaf.ConstructWaveletMatrix();
15   SampleSA(sa, leaf.sampledSA, leaf.mark);
16 }
```

Figure 3. Pseudo code of BWT on leaf substrings.

where D is a difference cover modulo v.

By sorting all suffixes starting at positions in a difference cover sample modulo v, the comparison of *any* two suffixes is guaranteed to finish by comparing at most v characters. This sorting can be done in almost the same way with DC3 algorithm[14], in $O(vn)$ time.

Colbourn and Ling[15] have shown how to construct a DC modulo v of size $O(\sqrt{v})$. Following their scheme, the number of bits required to store the DCS is $O(n \log n/\sqrt{v})$.

In our algorithm, we build a DCS modulo v, with $v = \log^2 n$, prior to the main divide-and-conquer step. The DCS thus takes $O(n)$ space and a single suffix comparison takes time $O(\log^2 n)$.

C. BWT for leaf substrings

We divide the entire interval $[0, n]$ of the input text T until it becomes shorter than a constant threshold. For each of such short strings, we simply build its full suffix array, derive the BWT as well as the sampled suffix array from it. We show its pseudo code in Fig. 3.

Sorting suffixes can be done by any string sorting algorithm. We use multikey quick sort[16]. Note that the time complexity of multikey quick sort in the worst case is $O(n^2)$, but it can be reduced to $O(n \log^2 n)$ by using DCS. Table III shows an example BWT of a range $[2, 8]$ of the string $T =$"mississippi$". As we noted above, the range represents the starting positions of suffixes it should consider, so we are now considering suffixes "ssissippi$", "sissippi$", ..., and "ppi$" and as the BWT, we will return a permutation of $T[1, 7]$, which is "ississi". The table also shows the derived LF-mapping for this range. Note that $L[6]$ and LF(6) are left *undefined*. The suffix in question starts at the head of the range, so its preceding character is out of the range. We can safely leave LF-mapping of such elements undefined, as long as we do not use it, as we will explain below.

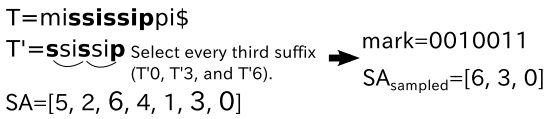

Figure 4. An example of suffix array sampling

After building the full suffix array, we sample elements from it. The sampled positions are recorded in the mark array, so that mark$[i] = 1$ if and only if $SA[i]$ is sampled. A BWT, a sampled suffix array, and a mark bit array thus computed are returned.

We show an example of a sampled suffix array in Fig. 4, where the sampling interval is three; that is, every third suffix is sampled. In the suffix array, the sampled positions 0, 3, and 6 appear at sixth, fifth, and second positions. So the mark array has these positions set, and sampled elements of the suffix array are left in the same order.

There is a trade-off between memory usage and the time to restore an element of suffix array. When sampling interval is s, the average time necessary to compute an element of suffix array is $O(s \log \sigma)$, and the memory usage for suffix array and mark is $O(n \log n/s)$ bits and $O(n)$ bits, respectively. In order to achieve $O(n \log \sigma)$ space, we choose $s \propto \log n$.

D. Merging BWTs

In internal nodes, results from the two recursive calls are merged. We need to merge BWTs, sampled suffix arrays, and mark bit arrays. We show the pseudo code in Fig. 6. In fact, they can be merged in almost the same way, so we explain only how to merge two BWTs below;

Suppose T_l and T_r are adjacent substrings, and their corresponding BWTs T_l^{BWT} and T_r^{BWT}, respectively. The easiest way to obtain the BWT of $(T_l + T_r)$ is to restore each character's offset in the original text and merge them in the manner of merge sort. However, restoring each character's offset requires $\Omega(n \log n)$ bits.

To reduce the memory usage without incurring too much penalty on time complexity, we use the idea of *gap* array[5]. Suppose the lengths of T_l and T_r are m_l and m_r, respectively. We can merge T_l^{BWT}, T_r^{BWT} by a theorem in [5].

Table III
AN EXAMPLE OF LF-MAPPING FOR AN INTERVAL $[3, 7]$ IN THE INPUT STRING "MISSISSIPPI$".

i	sorted suffixes	L	LF(i)
0	ippi$	s	3
1	issippi$	s	4
2	ppi$	i	0
3	sippi$	s	5
4	sissippi$	s	6
5	ssippi$	i	1
6	ssissippi$	-	-

Theorem 1: Assume the suffix $T_r[t, n]$ falls between lexicographically i-th and $(i+1)$-th suffixes of T_l, and $T_r[t-1, n] = cT_r[t, n]$ between the j-th and $(j+1)$-th suffixes. That is,

$$T_l[SA[i], n] < T_r[t, n] < T_l[SA[i+1], n] \tag{5}$$

and

$$T_l[SA[j], n] < T_r[t-1, n] < T_l[SA[j+1], n]. \tag{6}$$

Then j can be computed by the following.

$$j = \begin{cases} C[c] + \text{rank}_c(T_l^{BWT}, i) + b & (\text{if } c = T_l[m_l - 1]), \\ C[c] + \text{rank}_c(T_l^{BWT}, i) & (\text{otherwise}), \end{cases} \tag{7}$$

where b is defined as follows.

$$b = \begin{cases} 1 & (\text{if } T_r[0, n] < T_r[i+1, n]), \\ 0 & (\text{otherwise}). \end{cases} \tag{8}$$

By this theorem, each element of T_r^{BWT} can be inductively inserted into T_l^{BWT}, once we find a position to insert the suffix starting from the end of T_r.

Suppose gap is an array of size $(m_l + 1)$ which counts how many elements are inserted into each gap between two elements of T_l^{BWT}. The array gap would use $O(n \log n)$ bits of memory if it is naively represented; by taking advantage of the fact that elements are non-negative integers whose sum are m_r, however, we can encode them more efficiently. We use p (> 0) bits to each element of gap. An element smaller than $2^p - 1$ is straightforwardly represented in the gap array. If an element equals to or is larger than $2^p - 1$, we store $(2^p - 1)$ in the gap array and make an additional key-value pair mapping the index to the value. We maintain these key-value pairs in a hash table. Creating one such pair uses $2 \log n$ bits of memory. To estimate the memory usage, we must count how many key-value pairs are created in the worst case. Because the sum of all elements of gap is m_r, the number of pairs created in the worst case is $m_r/(2^p - 1)$. In this case, the memory usage is $np + 2m_r \log n/(2^p - 1)$. As $m_r \leq n$, by choosing $p \propto \log \log n$, the memory usage becomes $O(n \log \log n)$.

The position where the suffix starting from the end of T_r is to be inserted is determined by a binary search. Though time complexity of ordinary string binary search is $O(\log^3 n)$ with DCS in the worst case, in our algorithm, an element of suffix array must be computed from sampled one. This can be done quickly by converting T_l^{BWT} into wavelet matrix. Wavelet matrix uses the same number of bits with the original text and enables one to do rank operation in $O(\log \sigma)$ time. Therefore, binary search can be accomplished in $O(\log \sigma \log^4 n)$ time in the worst case. Once the suffix starting from the end of T_r has been inserted, the position to insert longer suffixes into can be determined by inductively applying Eq.7 and Eq.8. Extending a suffix by one character takes $O(n \log \sigma)$ time when the second clause of Eq.7 applies (i.e., no need to compute the "b" term). A string comparison is necessary when c (the extended character) and the last character of T_l are the same and thus the "b" term needs to be computed by Eq.8. The number of character comparisons is $O(\log^2 n)$, thanks to the DCS with $v = \log^2 n$. To summarize, the overall time complexity of gap construction is $O(\log \sigma \log^4 n + n \log \sigma + n \log^2 n)$; the first term for the initial binary search and the second and third for extending suffixes n times. Assuming $\sigma \leq n$, as is the case in any interesting scenario, it is $O(n \log^2 n)$. With the gap array, T_l^{BWT} and T_r^{BWT} can be merged in $O(n \log \sigma)$ time. So can sampled suffix array and mark array. Thus we established that a single merge operation on n characters has the worst case time complexity of $O(n \log^2 n)$.

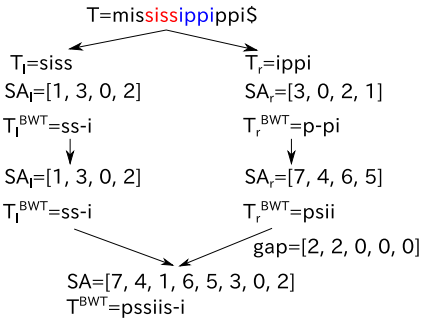

Figure 5. An example of merging two BWTs

We will show an example of a merging step in Fig. 5. In this figure, T_l and T_r are "siss" and "ippi", respectively, but we assume as if it contains the character next to those substrings. SA and BWT is constructed for both of substrings. A character '-' represents an unknown character. Note that we show the full suffix array here for the sake of illustration.

E. Time and space complexity

We have just established that a merging step on n characters takes $O(n \log^2 n)$ time. Since the recursion depth is $O(\log n)$, the total complexity is $O(n \log^3 n)$ in the worst case. A sequential space complexity is $O(n \log \log n)$ plus the recursion depth, which is $O(\log n)$. It is therefore $O(n \log \log n)$.

V. Parallel algorithm

A. Task parallel model

In task parallel model, tasks are spawned in arbitrary places in the program, and are automatically and dynamically scheduled to available workers by the runtime system.

This model nicely supports divide-and-conquer algorithms, in which a large problem is recursively divided into smaller problems, whose sub-results are merged into the result of the original problem. Divide-and-conquer algorithms

```
1  void ConstructGap(Node &left, Node &right,
2      int lBegin, int lEnd, int rBegin, int rEnd){
3    int c, idx;
4    // find idx s.t. T_idx < T_rEnd-1 < T_idx+1
5    idx=StringBinSearch(left, lBegin, rEnd-1);
6    gap[idx+1]++;
7    for(int i=right.size()-2; i>=0; i--){
8      c=T[rBegin+i];
9      idx=left.C(c)+left.rank(c, idx)-1;
10     // Eq. 8
11     if(c==T[lEnd-1] &&
12         StringLessThan(rBegin, rBegin+i+1)) idx++;
13     gap[idx+1]++;
14   }
15 }
16 void MergeBWT(Node &left, Node &right, Node &parent){
17   int pBWTIdx=0, rIdx=0;
18   for(int j=0; j<gap[0]; j++){
19     if(right.isSA0(rIdx)) parent.bwt[pBWTIdx++]=T[lEnd];
20     else parent.bwt[pBWTIdx++]=right.bwt[rIdx++];
21   }
22   for(int i=1; i<gap.size(); i++){
23     parent.bwt[pBWTIdx++]=left.bwt[i];
24     for(int j=0; j<gap[i]; j++){
25       if(right.isSA0(rIdx)) parent.bwt[pBWTIdx++]=T[lEnd];
26       else parent.bwt[pBWTIdx++]=right.bwt[rIdx++];
27     }
28   }
29   if(!parent.isRoot() && parent.isLeft())
30     // Construct wavelet matrix only for left node
31     parent.ConstructWaveletMatrix();
32 }
33 void MergeNode(Node &left, Node &right, Node &parent,
34   int lBegin, int lEnd, int rBegin, int rEnd){
35   // gap array construction
36   ConstructGap(left, right, lBegin, lEnd, rBegin, rEnd);
37   // merge three components
38   MergeBWT(left, right, parent);
39   MergeSampledSA(left, right, parent);
40   MergeMark(left, right, parent);
41 }
```

Figure 6. Pseudo code of BWT merging.

are usually expressed using a recursion, so it can be naturally implemented with task parallel models.

B. Parallelizing divide-and-conquer BWT with tasks

The first step toward a parallel BWT construction algorithm is creating a task for each recursive call to `ComputeBWT` (Fig. 2). This leads to an algorithm whose critical path is $O(n \log^2 n)$, which is due to the merging step. It is unsatisfactory, considering a serial algorithm of $O(n)$ time complexity is available. To reduce the critical path further, we also parallelize the merging step.

In order to parallelize a merging step, we should parallelize both construction of the gap array and merging of BWT, sampled suffix array, and mark. To parallelize gap construction, we first split T_r into blocks, and compute

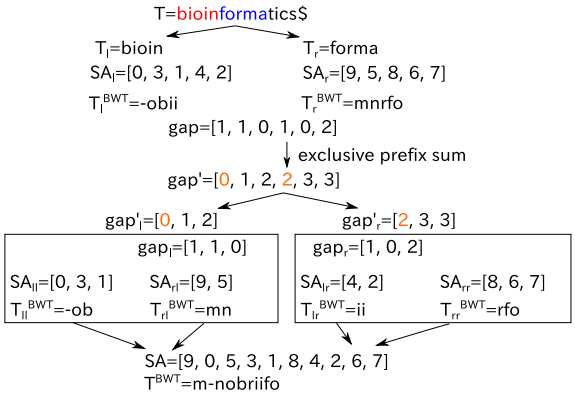

Figure 7. An example of parallel merging BWT

where the suffix starting from the end of each block is inserted into suffix array of T_l. Note that whole suffix array is not available here, so we should use a sampled one. Then, we compute the insert position of precedent suffixes in each block. With a constant block size, the time to construct a gap array for a block is dominated by the binary search. As all blocks can be merged in parallel, the critical path of a parallel merging step is $O(\log \sigma \log^4 n)$.

Next, we should parallelize merging of BWTs, sampled suffix arrays, and mark arrays. This can be done if prefix sum of gap is available. We split T_l^{BWT} and gap into blocks. When we merge i-th block, we compute a prefix sum of gap before the head of i-th block of gap. Suppose t is the result of this prefix sum. We start to merge T_r^{BWT} from the position t. Storing the result of prefix sum for all elements uses $O(n \log n)$ bits of memory, so we actually do not compute prefix sum. Instead, we construct auxiliary data structure to quickly compute any element of it. This data structure is based on the so-called four Russian technique[17]. First gap is split into super blocks of size $O(\log^2 n)$, and we store the sum of all elements which appear before each super block. Second, each super block is split into blocks of size $O(\log n)$, and we store the sum of elements which appear between the head of current super block and the head of each block. Using this data structure, each element of prefix sum of gap can be computed in $O(\log n)$ time on average.

We show an example of parallel merging algorithm in Fig. 7. Here, we explain how to execute merging by using two threads. In this figure, after computing suffix array, BWT, and gap, we compute prefix sum of gap. Then, gap is split into two parts in the same way with T_l. The first element of gap'_l is 0, so the thread which handles the left part of merge starts to read SA_r and T_r^{BWT} from the zeroth element. On the other hand, the first element of gap'_r is 2, so the thread which handles the right part of merge starts to read SA_r and T_r^{BWT} from the second element.

Wavelet matrix and DCS construction must be also parallelized. As for wavelet matrix, binary sequence of each level

in wavelet matrix must be constructed in parallel. In fact, this can be done naively with critical path being $O(1)$, thus the overall time for wavelet matrix construction is $O(\log \sigma)$. DCS construction is also able to be paralellized. As we mentioned in section IV-B, it is constructed based on DC3. So, DCS construction can be parallelized almost in the same way with a known method to parallelize DC3[13].

C. Time and space complexity

We have seen in the previous section that the critical path of a single merging step for n characters is $O(\log \sigma \log^4 n)$. With the recursion depth $O(\log n)$, the critical path of the entire parallel algorithm is $O(\log \sigma \log^5 n)$. Memory usage is $O(n \log \log n)$ plus total size of stacks. With the recursion depth $O(\log n)$ and with the work stealing scheduler [18], each of p workers consumes at most $O(\log n)$ stack space. Thus the space usage is $O(n \log \log n + p \log n)$, which is $O(n \log \log n)$ if $p \in o(n)$, a reasonable assumption.

VI. Evaluation

The proposed method was implemented in C++. We parallelized our algorithm with MassiveThreads[19], a task parallel library developed by our group.[1] It has tasking primitives similar to those found in Cilk[18] (spawn and sync), OpenMP tasking (task and taskwait), and Intel threading building block (`task_group` class) and implements a work-stealing scheduler[20].

Evaluation was performed on a shared-memory machine that has four Intel(R) Xeon(R) E7540 2.00GHz chips. Each chip has six physical cores and each physical core supports two virtual cores (a.k.a. hardware threads). The machine has 24 physical cores in total and equips 256GB memory. We compared the performance of our method with BWT-IS and a straightforward BWT computation method. The latter first constructs a suffix array by SA-IS, the fastest known algorithm for constructing suffix arrays and then derives the BWT from it. We used a human genome, HG19, as input data, which we get from UCSC web site[21]. We used chromosomes 1 thru 4.

We first show how the performance changes with the number of threads, using chr1 as the input data. Table V shows the elapsed times of the three methods. SA-IS and BWT-IS are not parallelized, so they are executed only with a single thread. The column named DCS, w/o DCS, and total

Table IV
Input data.

name	size[MB]	description
chr1	243	First chromosome of human genome.
chr1-2	480	Concatenation of first and second chromosomes.
chr1-3	672	Concatenation of first to third chromosomes.
chr1-4	858	Concatenation of first to fourth chromosomes.

[1] http://code.google.com/p/massivethreads/

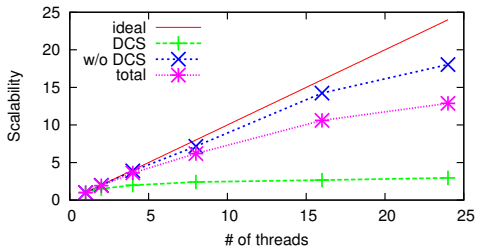

Figure 8. Scalability

represents the execution time of DCS, that of computation other than DCS, and the sum of them, respectively. The exuecution time of our algorithm is similar to BWT-IS even when we used 24 threads. To investigate, we show in Fig. 8 the scalability of DCS construction phase, the phases other than DCS construction, and all the phases. The phases other than DCS scaled relatively well, achieving 18.0x speedup with 24 threads; speedup of DCS is relatively poor (approximately 3.0x), hindering the total speedup. In DCS construction, the most time-consuming task is a parallel string sorting using v-characters prefix. We are planning to improving this part by using more a sophisticated method like [22].

Memory usage is shown in Table VI. Our method uses much memory space than BWT-IS though it is better than the naive one. This result is consistent with the theoretical value.

Next, we investigated how performance changes with the size of input data. We used 24 threads for our algorithm. Fig. 9 and 10 show the execution time and memory usage of the three methods. Our method and BWT-IS show almost the same performance. As shown in Fig. 9, DCS construction still occupies a large part of execution time. If we can

Table V
Execution time vs. the number of threads.

# of threads	our algorithm			BWT-IS[s]	naive (SA-IS)[s]
	DCS[s]	w/o DCS[s]	total[s]		
1	109.9	1288.4	1398.4	105.8	205.6
2	72.4	653.2	725.6	-	-
4	55.4	331.6	386.9	-	-
8	45.7	180.0	225.7	-	-
16	41.4	90.5	131.9	-	-
24	37.2	71.4	108.6	-	-

Table VI
Memory usage vs. the number of threads.

# of threads	proposal[s]	BWT-IS[s]	naive(SA-IS)[s]
1	976.8	483.0	2207.9
2	1158.0	-	-
4	1195.6	-	-
8	1202.8	-	-
16	1182.1	-	-
24	1193.8	-	-

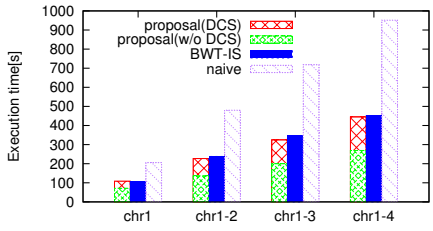

Figure 9. Data size vs. execution time.

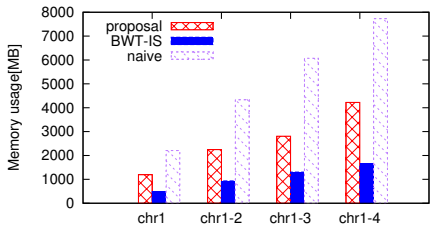

Figure 10. Data size vs. memory usage.

improve the scalability of DCS construction, the overall performance will be much better.

VII. Conclusion

In this paper, we proposed a new algorithm to compute Burrows-Wheeler transform. Our algorithm is based on a divide-and-conquer algorithm; thus it is easily parallelized in manner of task parallel model. We also proposed a parallelized merging step and achieved $O(\log \sigma \log^5 n)$ worst case critical path and $O(n \log \log n)$ space complexity. Our experimental results have shown that our method is as fast as BWT-IS, which is the fastest known algorithm to compute BWT in small space.

The absolute performance was not as good as we desired; one of our future efforts is naturally to improve its performance, particularly of the DCS construction. Reducing memory usage is another direction. Though our method uses less memory than a method that builds a suffix array, it still needs larger space than BWT-IS. One possibility is to compress bit sequences for the wavelet matrix.

References

[1] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SODA '90*, pp. 319–327, 1990.

[2] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," *FOCS '00*, pp. 390–398, 2000.

[3] J. Kärkkäinen, "Fast BWT in small space by blockwise suffix sorting," *Theoretical Computer Science*, vol. 387, no. 3, pp. 249–257, 2007.

[4] D. Okanohara and K. Sadakane, "A linear-time Burrows-Wheeler transform using induced sorting," *SPIRE '09*, pp. 90–101, 2009.

[5] P. F. T. Gagie and G. Manzini, "Lightweight data indexing and compression in external memory," *Algorithmica*, vol. 63, no. 3, pp. 707–730, 2012.

[6] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys*, vol. 39, no. 1, 2007.

[7] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler, "A block sorting lossless data compression algorithm," Tech. Rep., 1994.

[8] F. Claude and G. Navarro, "The wavelet matrix," *Proc. SPIRE'12*, pp. 167–179, 2012.

[9] R. A. Lippert, C. M. Mobarry, and B. P. Walenz, "A space-efficient construction of the burrows-wheeler transform for genomic data," *Computational Biology*, 2005.

[10] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu, "A space and time efficient algorithm for constructing compressed suffix arrays," *Algorithmica*, pp. 23–36, 2007.

[11] G. Nong, S. Zhang, and W. H. Chan, "Linear suffix array construction by almost pure induced-sorting," *Data Compression Conference*, pp. 193–202, 2009.

[12] R. K. Menon, G. P. Bhat, and M. C. Schatz, "Rapid parallel genome indexing with MapReduce," *MapReduce '11 Proceedings of the second international workshop on MapReduce and its applications*, pp. 51–58, 2011.

[13] F. Kulla and P. Sanders, "Scalable parallel suffix array construction," *Parallel Computing*, vol. 33, pp. 605–612, 2007.

[14] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linearwork suffix array construction," *Journal of the ACM*, vol. 53, pp. 918–936, 2006.

[15] C. J. Colbourn and A. C. Ling, "Quorums from difference covers," *Information Processing Letters*, vol. 75, pp. 9–12, 2000.

[16] J. L. Bentley and R. Sedgewick, "Fast algorithms for sorting and searching strings," *In Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pp. 360–369, 1997.

[17] V. Arlazarov, E. Dinic, M. Konrod, and I. Faradzev, "On economic construction of the transitive closure of a directed graph," *Sov.Math. Dokl*, pp. 1209–1210, 1975.

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. of PPoPP '95*, 1995, pp. 207–216.

[19] J. Nakashima and K. Taura, "MassiveThreads: A thread library for high productivity languages," *Concurrent Objects and Beyond From Theory to High-Performance Computing. (to appear).*

[20] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[21] "UCSC Genome Bioinformatics," http://genome.ucsc.edu/.

[22] T. Bingmann and P. Sanders, "Parallel string sample sort," *arXiv preprint arXiv:1305.1157*, 2013.