Parallel Algorithms for Burrows-Wheeler Compression and Decompression[☆]

James A. Edwards, Uzi Vishkin

University of Maryland, College Park, Maryland

Abstract

We present work-optimal PRAM algorithms for Burrows-Wheeler compression and decompression of strings over a constant alphabet. For a string of length n, the depth of the compression algorithm is $O(\log^2 n)$, and the depth of the the corresponding decompression algorithm is $O(\log n)$. These appear to be the first polylogarithmic-time work-optimal parallel algorithms for any standard lossless compression scheme.

The algorithms for the individual stages of compression and decompression may also be of independent interest: 1. a novel $O(\log n)$-time, $O(n)$-work PRAM algorithm for Huffman decoding; 2. original insights into the stages of the BW compression and decompression problems, bringing out parallelism that was not readily apparent, allowing them to be mapped to elementary parallel routines that have $O(\log n)$-time, $O(n)$-work solutions, such as: (i) prefix-sums problems with an appropriately-defined associative binary operator for several stages, and (ii) list ranking for the final stage of decompression.

Keywords: parallel, PRAM, Burrows-Wheeler, compression

1. Introduction

A *lossless compression function* is an invertible function $C(\cdot)$ that accepts as input a string S of length n over some alphabet Σ and returns a string $C(S)$ over some alphabet Σ' such that, in some statistical model, fewer bits are required to represent $C(S)$ than S. A *lossless compression algorithm* for a given lossless compression function is an algorithm that accepts S as input and

[☆]Partially supported by NSF grant CCF-0811504

Email addresses: `jedward5@umd.edu` (James A. Edwards), `vishkin@umiacs.umd.edu` (Uzi Vishkin)

produces $C(S)$ as output; the corresponding *lossless decompression algorithm* accepts $C(S)$ for some S as input and produces S as output.

In their seminal paper [1], Burrows and Wheeler describe their eponymous lossless compression algorithm and corresponding decompression algorithm; its operation is reviewed in Section 2. The *Burrows-Wheeler (BW) Compression problem* is to compute the lossless compression function defined by the algorithm of [1], and the *Burrows-Wheeler (BW) Decompression problem* is to compute its inverse. The algorithms of [1] solve the BW Compression problem in $O(n \log^2 n)$ serial time and the BW Decompression problem in $O(n)$ serial time. Later work reduced a critical step of the compression algorithm to the problem of computing the suffix array of S, for which linear-time algorithms are known, so both problems can now be solved in $O(n)$ serial time.

1.1. Contributions

The primary contributions of this paper are an $O(\log^2 n)$-time, $O(n)$-work PRAM algorithm for solving the BW Compression problem and a $O(\log n)$-time, $O(n)$-work PRAM algorithm for solving the BW Decompression problem. These algorithms appear to be the first polylogarithmic-time work-optimal parallel algorithms for any standard lossless compression scheme. Also, the algorithms for the individual stages of compression and decompression may be of independent interest:

- We present a novel $O(\log n)$-time, $O(n)$-work PRAM algorithm for Huffman decoding (Section 3.2.1).
- This paper also provides original insights into the BW compression and decompression problems. The original serial algorithms for these problems were presented in such a way that their potential parallelism was not readily apparent. Here, we reexamine them in a way that allows them to be mapped to elementary parallel routines. Specifically:
  - most of the compression and decompression stages can be cast as prefix-sums problems with an appropriately-defined associative binary operator,
  - the final stage of decompression can be reduced to the problem of list ranking.
  - both of these problems have known $O(\log n)$-time, $O(n)$-work solutions.

1.2. Related Work

It is common in practice to partition the input string into uniformly-sized blocks and solve the BW Compression problem separately for each block using a serial algorithm. Because the sub-problems of compressing the blocks are independent of one another, they can be solved in parallel. However, this does not solve the BW Compression problem for the original input and thus is not a parallel algorithm for solving it. It is worth noting that our parallel-algorithmic approach is orthogonal to the foregoing block-based approach; the two approaches could conceivably be combined in applications that require the input to be partitioned into blocks by applying our algorithm to each block separately.

A commonly-used, serial implementation of the block-based approach noted above is bzip2 [2]; the algorithm it applies to each block is based on the original BW compression algorithm of [1]. Bzip2 allows changing the block size (100-900 kB); larger blocks provide a smaller compressed output at the expense of increased run time. There are also variants of bzip2, such as pipeline bzip [3], that compress multiple blocks simultaneously. However, these variants do not achieve speedup on single blocks while our approach does. There exists at least one implementation of a linear time serial algorithm for BW compression, bwtzip [4] However, bwtzip is a serial implementation that emphasizes modularity over performance, unlike the focus of this paper.

There are applications where BW compression would be useful but is not currently used because of performance. One such application is JPEG image compression. JPEG compression consists of a lossy compression stage followed by a lossless stage. The work [5] considered replacing the currently-used lossless stage with the BW compression algorithm. For high-quality compression of "real-world" images such as photographs, this yielded up to a 10% improvement, and for the compression of "synthetic" images such as company logos, the improvement was up to 30%. The author cites execution time as the main deficiency of this approach.

The newer JPEG-2000 standard allows for lossless image compression. Also, unlike JPEG, it employs wavelet compression, which analyzes the entire image without dividing it into blocks. Because of this, it is possible that BW compression would provide an even greater improvement for JPEG-2000 images, analogous to the improved compression of bzip2 with larger block sizes; however, we are not aware of a study similar to the one for JPEG mentioned above. The white paper [6] suggests that Motion JPEG-2000 is a good format for archival of video, where lossless compression is desired in order to avoid introducing visual artifacts. In order to play such a video back at its correct frame rate, the decoder must run fast enough to decode frames in real time.

We are not aware of prior work on running BW compression or decompression in parallel on general-purpose graphics processing units (GPGPU); however the survey paper [7] explains some of the issues for compression; decompression is not discussed. The author gives an outline of an approach for making some parts of the algorithm parallel and claims that the remaining parts would not work well on GPUs due to exhibiting poor locality.

A parallel algorithm for Huffman decoding is given in [8]. However, the algorithm is not analyzed therein as a PRAM algorithm, and its worst case run time is $O(n)$. Our PRAM algorithm for Huffman decoding runs in $O(\log n)$ time.

The rest of the paper is organized as follows. Section 2 describes the principles of BW compression and decompression. Section 3 describes our parallel algorithms for the same along with their complexity analysis. Finally, Section 4 concludes.

2. Preliminaries

We use ST to denote the concatenation of strings S and T, $S[i]$ to denote the i^{th} character of the string S ($0 \leq i < |S|$), and $S[i, j]$ to denote the substring $S[i]...S[j]$ ($0 \leq i \leq j < |S|$); $S[i, j]$ is the empty string when $i > j$.

In their original paper, Burrows and Wheeler [1] describe a lossless data compression algorithm consisting of three stages in the following order:

- a reversible block-sorting transform (BST)[1]
- move-to-front (MTF) encoding
- Huffman encoding.

The compression algorithm is given as input a string S of length n over an alphabet Σ, with $|\Sigma|$ constant. S is provided as input to the first stage, the output of each stage is provided as input to the next stage, and the output of the final stage is the output of the overall compression algorithm. The output S^{BW} is a bit string (i.e., a string over the alphabet $\{0, 1\}$). The

[1]This transform is also referred to by some authors as the Burrows-Wheeler Transform (BWT). We refrain from using this name to avoid confusion with the similarly-named Burrows-Wheeler compression algorithm which employs it as a stage.

corresponding decompression algorithm performs the inverses of these stages in reverse order:

- Huffman decoding
- MTF decoding
- inverse BST (IBST)

See Figure 1. Here, we review the three compression stages.

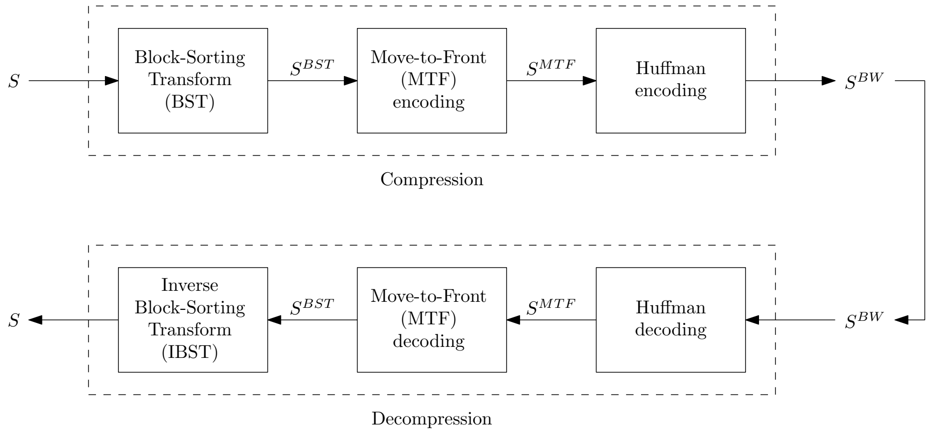

S
Block-Sorting Transform (BST)
S^{BST}
Move-to-Front (MTF) encoding
S^{MTF}
Huffman encoding
S^{BW}
Compression
S
Inverse Block-Sorting Transform (IBST)
S^{BST}
Move-to-Front (MTF) decoding
S^{MTF}
Huffman decoding
S^{BW}
Decompression

Figure 1: Stages of BW compression and decompression.

2.1. Block-Sorting Transform (BST)

Given a string S of length n as input, the BST produces as output S^{BST}, a permutation of S. We assume throughout that S ends with a special character that does not appear anywhere else in S. This can be ensured by adding a new character "\$" to Σ and appending \$ to the end of S before running the algorithm. The permutation is computed as follows (see Figure 2).

1. List all possible rotations of S (each of which is a string of length n).
2. Sort the list of rotations lexicographically.
3. Output the last character of each rotation in the sorted list.

Motivation As explained by Burrows and Wheeler [1], the BST has two properties that make it useful for lossless compression: (1) it has an inverse and (2) its output tends to have many occurrences of any given character in close proximity, even when its input does not. Property (1) ensures that the later decompression stage can reconstruct S given only S^{BST}. Property (2) is exploited by the later compression stages to actually perform the compression.

$$\text{banana\$} \xrightarrow{\text{rotate}} \begin{array}{l} \text{banana\$} \\ \text{anana\$b} \\ \text{nana\$ba} \\ \text{ana\$ban} \\ \text{na\$bana} \\ \text{a\$banan} \\ \text{\$banana} \end{array} \xrightarrow{\text{sort}} \underbrace{\begin{array}{l|l|} \text{\$banan} & \text{a} \\ \text{a\$bana} & \text{n} \\ \text{ana\$ba} & \text{n} \\ \text{anana\$} & \text{b} \\ \text{banana} & \text{\$} \\ \text{na\$ban} & \text{a} \\ \text{nana\$b} & \text{a} \end{array}}_{M} \xrightarrow{\text{output}} \text{annb\$aa}$$

Figure 2: BST of the string "banana\$". The sorted list labeled M can be viewed as a matrix of characters.

2.1.1. Inverse of the BST (IBST)

We start by reviewing a proof that the BST is invertible. The input to the inverse of the BST (IBST) is the output of the BST, S^{BST}, which is the rightmost column in matrix M (as denoted in Figure 2). This rightmost column is replicated in Figure 3(a). For the proof, we first discuss a wasteful way to derive the full matrix M used by the BST stage. This is done using an inductive process. Following step i of the induction, we produce the first i columns of M.

1. To obtain the first column of M, we sort the rows (characters) of S^{BST} (Figure 3(b)). This works because every column in M has the same characters and the rows of M are sorted. If there are multiple occurrences of the same character, we maintain their relative order; this is known as stable sorting.
2. To obtain the first two columns, we perform the following two steps:
   (a) Insert S^{BST} to the left of the first column (Figure 3(c)).
   (b) Sort lexicographically the rows to obtain the first two columns of M (Figure 3(d)). When comparing rows, there are two cases:
      - If two rows begin with different characters, we order them according to their first characters.
      - If two rows begin with the same character, we do not need to compare the second character. The tie of the first character has already been broken by the previous round of sorting. Therefore, we only need to maintain the relative order of these two rows.

   Same permutation observation: We will later make use of the following implied observation: the permutation in all steps i is identical.

3. We repeat step 2 in order to obtain the first three columns (Figure 3(e,f)), the first four columns (Figure 3(g,h)), and so on until M is entirely filled in.

We take the row of M for which \$ is in the rightmost column to be the output of the IBST since the rows of M are rotations of the input string where the first row (the input string itself) was the one for which \$ was the last character.

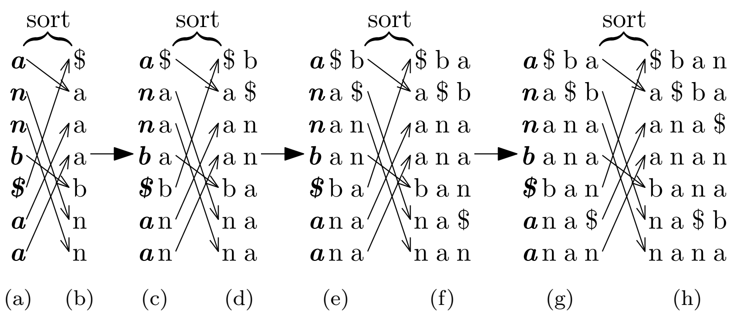


Figure 3: Reconstructing M from the BST of "banana\$". Each round of sorting reveals one more column of M; after k rounds, the first k columns of M are known. The first four rounds are shown. Prior to each round, S^{BST} (shown in bold italics) is inserted as the leftmost column.

The linear time serial algorithm

Next, we economize the above construction to visit only $O(n)$ of its entries, providing an $O(n)$ serial algorithm for unraveling enough of M to reproduce the output; namely, the row of M for which \$ is in the rightmost column.

First, we locate all instances of \$ in M and augment this with the last column of M (Figure 4(a)). Now, by rotating every row so that \$ appears in the rightmost column, it reveals in each column the corresponding character of the input string, as shown in Figure 4(b).

The following pseudocode summarizes our description of the $O(n)$-time serial algorithm (to compute the IBST of S^{BST}).

```
// Input: $S^{BST} = x_1 x_2 ... x_n$.
// Output: $S$, the IBST of $S^{BST}$.

1. Apply a stable integer sorting algorithm to sort the elements of $x_1 x_2 ... x_n$.
   The output is a permutation storing the rank of the $i^{\text{th}}$ element $x_i$ into
   $T[i]$.
2. $L[0] := 0$    // $L[j]$ is the row of \$ in column $j$
3. for $j := 0$ to $n-2$ do
```

$$\begin{array}{ccccccc}
\$ & - & - & - & - & - & a \\
- & \$ & - & - & - & - & n \\
- & - & - & \$ & - & - & n \\
- & - & - & - & - & \$ & b \\
- & - & - & - & - & - & \$ \\
- & - & \$ & - & - & - & a \\
- & - & - & - & \$ & - & a
\end{array}
\xrightarrow[\text{rows}]{\text{rotate}}
\begin{array}{ccccccc}
- & - & - & - & - & a & \$ \\
- & - & - & - & n & - & \$ \\
- & - & n & - & - & - & \$ \\
b & - & - & - & - & - & \$ \\
- & - & - & - & - & - & \$ \\
- & - & - & a & - & - & \$ \\
- & a & - & - & - & - & \$
\end{array}$$

(a) (b)

Figure 4: Reconstructing "banana\$" from its BST given the last column of M and all instances of \$. After rotation, the character in row i of the last column moves to column $n-2-j$, where j is the column index of \$ in row i.

```
3.1. $L[j+1] = T[L[j]]$    // The location (i.e., row) of $ in column $j+1$
     is determined by applying the permutation $T$ to the location of $
     in column $j$
3.2. $S[n-2-j] := S^{BST}[L[j]]$    // Every determination that a $
     appears in row $i$ and column $j$ of $M$ implies one character in the
     output string $S$. This character is computed by a proper shift.
```

Note that we have written all characters of S except the last, which is \$. We skip this character because it is not part of the original input to the compression algorithm.

In Section 3.2.3, we replace step 1 and step 3.1 of the above algorithm with equivalent parallel steps. Step 3.2 is done later separately.

2.2. Move-to-Front (MTF) encoding

Given the output S^{BST} of the preceding stage as input, MTF encoding replaces each character with an integer indicating the number of different characters (not the number of characters) between that character and its previous occurrence. We follow the convention that all characters in Σ appear in some order and precede S^{BST}. This "assumed prefix" ensures that every character in S^{BST} has a previous occurrence and thus that the foregoing definition is valid. MTF encoding produces as output a string S^{MTF} over the alphabet of integers $[0, n-1]$, with $|S^{MTF}| = |S^{BST}|$. See Figure 5.

Let L_i be a list of the different characters in $S^{BST}[0, i-1]$ in the order of their last appearance, taking into account the assumed prefix. That is, compact $S^{BST}[0, i-1]$ by removing all but the last occurrence of every character in Σ and reverse the order of the resulting string to produce L_i. Specifically, let $L_0 = (\sigma_1, ..., \sigma_{|\Sigma|})$ be a listing of the characters of Σ in some predetermined order.

$$\Sigma = \{\$, a, b, n\}$$
$$S^{BST} = (a, n, n, b, \$, a, a)$$

	assumed prefix										
i	0	1	2	3	4	5	6	7	8	9	10
$S^{BST}[i]$	n	b	a	\$	a	n	n	b	\$	a	a
$prev[i]$	-	-	-	-	2	0	5	1	3	4	9
$C[i]$	-	-	-	-	{\$}	{\$,a,b}	{}	{\$,a,n}	{a,b,n}	{\$,b,n}	{}
$\lvert C[i]\rvert$	-	-	-	-	1	3	0	3	3	3	0

$$S^{MTF} = (1, 3, 0, 3, 3, 3, 0)$$

Figure 5: MTF of the string "annb\$aa". $C[i]$ is the set of characters between $S^{BST}[i]$ and its previous occurrence.

For $i > 0$, L_i can be derived from L_{i-1} using the MTF encoding and decoding algorithms described by Burrows and Wheeler [1], which serially construct L_i for all i, $0 \leq i < n$. The MTF encoding algorithm takes S^{BST} as input and produces S^{MTF} as output (see Figure 6 and read from top to bottom):

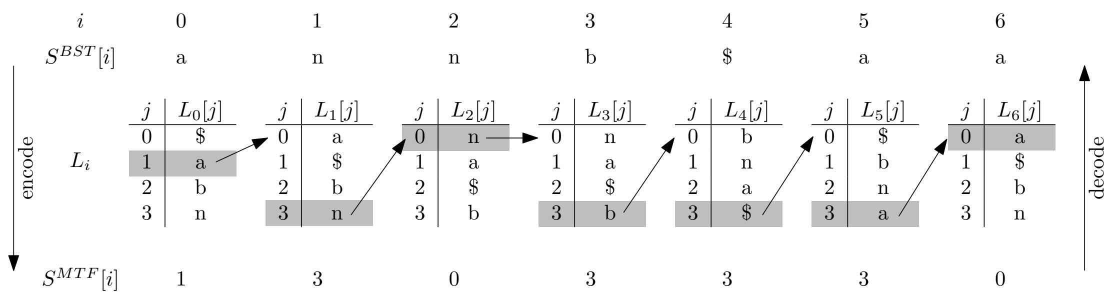
```
1. $L := L_0$
2. for $i := 0$ to $n - 1$ do    // At the beginning of iteration $i$, $L = L_i$
   2.1. Set $j$ to the index of $S^{BST}[i]$ in $L$
   2.2. $S^{MTF}[i] := j$
   2.3. Move $L[j]$ to the front of $L$ (i.e., remove the $j^{\text{th}}$ element of $L$, then
        reinsert it at position 0, shifting elements to make room)
```

Observe during each iteration that j is the number of different characters between $S^{BST}[i]$ and its nearest preceding occurrence. See Figure 6 and observe that S^{MTF} is the same as in Figure 5.

The MTF decoding algorithm takes S^{MTF} as input and produces S^{BST} as output (see Figure 6 and read from bottom to top):

```
1. $L := L_0$
2. for $i := 0$ to $n - 1$ do    // At the beginning of iteration $i$, $L = L_i$
   2.1. $j := S^{MTF}[i]$
   2.2. $S^{BST}[i] := L[j]$
   2.3. Move $L[j]$ to the front of $L$
```

In Section 3, we construct L_i for all i in parallel using the so-called parallel prefix sums routine with appropriately-defined associative binary operators in order to perform MTF encoding (Section 3.1.2) and decoding (Section 3.2.2).



i
$S^{BST}[i]$
L_i
$S^{MTF}[i]$
encode
decode
0 1 2 3 4 5 6
a n n b $ a a
1 3 0 3 3 3 0
j $L_0[j]$: 0 $, 1 a, 2 b, 3 n
j $L_1[j]$: 0 a, 1 $, 2 b, 3 n
j $L_2[j]$: 0 n, 1 a, 2 $, 3 b
j $L_3[j]$: 0 n, 1 a, 2 $, 3 b
j $L_4[j]$: 0 b, 1 n, 2 a, 3 $
j $L_5[j]$: 0 $, 1 b, 2 n, 3 a
j $L_6[j]$: 0 a, 1 $, 2 b, 3 n

Figure 6: MTF encoding and decoding. Observe that $S^{BST}[i] = L_i[S^{MTF}[i]]$. In both the encoder and the decoder, the shaded elements are moved to the front of the list according to the arrows. In the encoder, the shaded element is identified by searching the list L_i for the character $S^{BST}[i]$. In the decoder, the shaded element is chosen to be the one whose index is $j = S^{MTF}[i]$; no searching is necessary.

Motivation The output of MTF encoding is such that two occurrences of a given character that are close together in S^{BST} are assigned low MTF codes because there are few other characters in between. Because of property (2) of the BST, this is likely to occur often, and so smaller integers occur more frequently in S^{MTF} than larger integers. This means that Huffman encoding (or a similar encoding such as arithmetic encoding) may effectively compress S^{MTF} even if this is not the case for S itself. Because both the BST and MTF encoding stages are reversible, S can be recovered from S^{MTF}.

2.3. Huffman encoding

The input to the Huffman encoding stage is the string S^{MTF}, and it produces as output (1) the string S^{BW}, a bit string (recall: a string over the alphabet $\{0, 1\}$) whose length is $\Theta(n)$ and (2) a coding table T, whose size is constant given that $|\Sigma|$ is constant. The goal of Huffman encoding is to assign shorter codewords to characters that occur more frequently in S^{MTF}, thus minimizing the average codeword length. Huffman encoding proceeds in three steps.

1. Count the number of times each character of Σ occurs in S^{MTF} to produce a frequency table F
2. Use F to construct a coding table T such that, for any two characters $a, b \in \Sigma$, if $F(a) < F(b)$, then $|T(a)| \geq |T(b)|$
3. Replace each character of S^{MTF} with its corresponding codeword in T to produce S^{BW}.

The output S^{BW} of the Huffman encoding stage is the output of the overall compression algorithm. See Figure 7.

$$T = \begin{array}{c} 0 \rightarrow 10 \\ 1 \rightarrow 11 \\ 3 \rightarrow \ 0 \end{array}$$

$$S^{MTF} = (1, 3, 0, 3, 3, 3, 0)$$

$$S^{BW} = 11\ 0\ 10\ 0\ 0\ 0\ 10$$

Figure 7: Huffman table and encoding of S^{MTF} (spaces added for clarity). Recall that this is, in fact, the compression of the original string "banana\$".

3. Parallel Algorithm

Given an input string of length n, the original decompression algorithm [1] by Burrows and Wheeler runs in $O(n)$ serial time, as do all stages of the compression algorithm except the (forward) BST, which requires $O(n \log^2 n)$ serial time according to the analysis of [9]. More recently, linear-time serial algorithms [10, 11] have been developed to compute suffix arrays, and the problem of finding the BST of a string can be reduced to that of computing its suffix array (see Section 3.1.1), so Burrows-Wheeler (BW) compression and decompression can be performed in $O(n)$ serial time. The linear-time suffix array algorithms are relatively involved, so we refrain from describing them here and instead refer interested readers to the cited papers.

The parallel BW compression and decompression algorithms follow the same sequence of stages given in Section 2, but each stage is performed by a PRAM algorithm rather than a serial one. There are notable differences between the algorithms for compression and decompression, so we describe them separately.

3.1. Compression

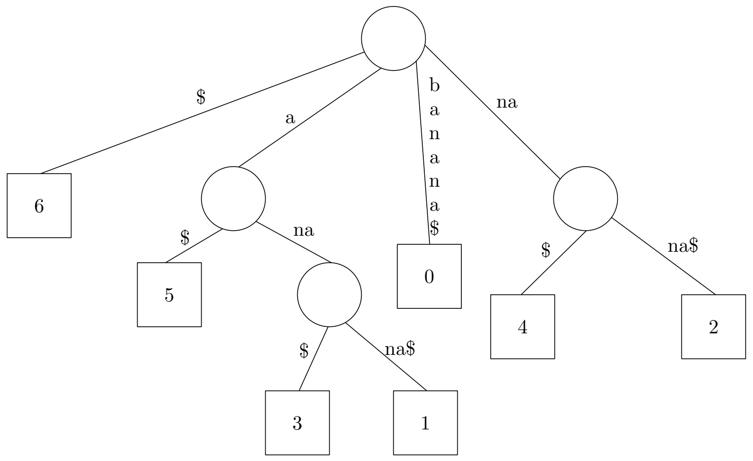
The input is a string S of length n over an alphabet Σ, where $|\Sigma|$ is constant. The overall PRAM compression algorithm consists of the following three stages.

3.1.1. Block-Sorting Transform (BST)

The BST of a string S of length n can be computed as follows. Add a character \$ to the end of S that does not appear elsewhere in S. Sorting all rotations of S is equivalent to sorting all suffixes of S, as \$ never compares equal to any other character in S. Such sorting is equivalent to computing the suffix array of S, which can be derived from a depth-first search (DFS) traversal of the suffix tree of S (see Figure 8). The suffix tree of S can be computed in $O(\log^2 n)$ time and $O(n)$ work using the algorithm of [12]. The order that leaves are visited in a DFS traversal of the suffix tree can be computed using the Euler tour technique [13] within the same complexity bounds, yielding the suffix array of S. Given the suffix array SA of S, we derive S^{BST} from S in $O(1)$ time and $O(n)$ work as follows:

$$S^{BST}[i] = S[(SA[i] - 1)_{\bmod n}], 0 \leq i < n$$

Overall, computing the BST takes $O(\log^2 n)$ time using $O(n)$ work.



i	0	1	2	3	4	5	6
$S[i]$	b	a	n	a	n	a	\$
$SA[i]$	6	5	3	1	0	4	2
$S[SA[i]-1]$	a	n	n	b	\$	a	a

Figure 8: Suffix tree and suffix array (SA) for the string S = "banana\$".

3.1.2. Move-to-Front (MTF) Encoding

Computing the MTF number for every character in the BST output S^{BST} amounts to finding L_i for all i, $0 \leq i < n$, as defined in Section 2.2. We accomplish this using prefix sums with an associative binary operator $\oplus$ as follows. We first define the function $MTF(X)$, which captures the local contribution of the substring X to the lists L_i. Then, we use $\oplus$ to merge this local information pairwise, finally producing all the L_i for the overall string S^{BST}.

Let $MTF(X)$ be the listing of the different characters in X in the reverse order of last occurrence in X; this is the empty list when X is the empty string. We refer to $MTF(X)$ as the *MTF list of* X. For example, given the string $X =$"$\underline{\text{b}}$ana$\underline{\text{na}}$", the last occurrence of each character is underlined, and reversing the order of these characters yields the list $MTF(X) =$(a, n, b). The index of a character in $MTF(X)$ is equal to the number of different characters that follow its last occurrence in X. For example, in "$\underline{\text{b}}$ana$\underline{\text{na}}$", zero different characters follow the last "a", one follows the last "n" (i.e., "a"), and two follow the last "b" (i.e., "n" and "a"). When X is a prefix of S^{BST}, this definition coincides with that of L_i.

Denote by $x \oplus y$ the list formed by concatenating to the end of y the list formed by removing from x all elements that are contained in y. Note that $\oplus$ is an associative binary operator. Observe the following:

- For a string consisting of a single character c, $MTF(c) = (c)$, the list containing c as its only element.
- For any two strings X and Y, $MTF(XY) = MTF(X) \oplus MTF(Y)$.

Our goal is to compute all of the lists L_i. This is equivalent to computing the MTF lists of all the prefixes of S^{BST}, taking into account the assumed prefix. By the above observations, this amounts to a prefix sum computation over the array A, where $A[i]$ is initialized to the singleton list $(S^{BST}[i])$. Because $|\Sigma|$ is constant, and the lists produced by the $\oplus$ operator have no more than $|\Sigma|$ elements, the $\oplus$ operator can be computed in $O(1)$ time and work. Therefore, we can compute the prefix sums in $O(\log n)$ time and $O(n)$ work by the standard PRAM algorithm for computing all prefix sums with respect to the operation $\oplus$.

The prefix sums algorithm works in two phases:

1. Adjacent pairs of MTF lists are combined using $\oplus$ in a balanced binary tree approach until only one list remains (see Figure 9).
2. Working back down the tree, the prefix sums corresponding to the rightmost leaves of each subtree are computed using the lists computed in phase 1 (see Figure 10).

3.1.3. Huffman Encoding

The PRAM algorithm for Huffman encoding follows readily from the description in Section 2.3.

1. Construct F using the integer sorting algorithm outlined in [14], which sorts a list of n integers in the range $[0, r-1]$ in $O(r + \log n)$ time using $O(n)$ work. Because $r = |\Sigma|$ is constant, this takes $O(\log n)$ time and $O(n)$ work.

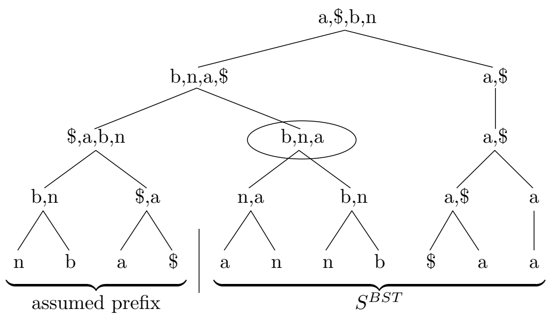


Figure 9: Phase 1 of prefix sums: Computing local MTF lists for “annb$aa” using the operator ⊕. Each node in the tree is the ⊕-sum of its children. For example, the circled node is (n, a) ⊕ (b, n).

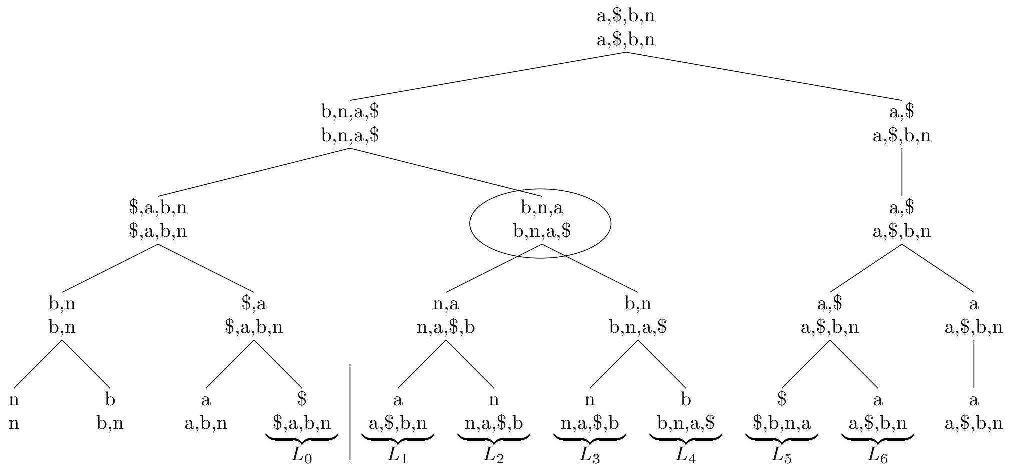


Figure 10: Computing the prefix sums of the output of the BST stage, “annb$aa”, with respect to the associative binary operator ⊕. The top line of each node is copied from the tree in Figure 9. The bottom line of a node V is the cumulative ⊕-sum of the leaf nodes starting at the leftmost leaf in the entire tree and ending at the rightmost child of V (i.e., the prefix sum up to the rightmost leaf under V). For example, the circled node contains the sum of leaves corresponding to the prefix “nba$annb”. Observe the correspondence of the labeled lists with Figure 6.

2. Use the standard heap-based serial algorithm to compute the Huffman code table T. Since $|\Sigma|$ is constant, this takes $O(1)$ time and work.
3. (a) Compute the prefix-sums of the code lengths $|T(S^{MTF}[i])|$ into the array U ($O(\log n)$ time, $O(n)$ work).
   (b) In parallel for all i, $0 \leq i < n$, write $T(S^{MTF}[i])$ to S^{BW} starting at position $U[i]$ ($O(1)$ time, $O(n)$ work).

The overall Huffman encoding stage runs in $O(\log n)$ time using $O(n)$ work. The above presentation proves the following theorem:

Theorem 1. *The algorithm of Section 3.1 solves the Burrows-Wheeler Compression problem for a string of length n over a constant alphabet in $O(\log^2 n)$ time using $O(n)$ work.*

3.2. Decompression

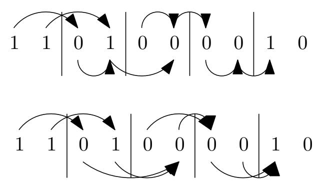
The input is a string S^{BW} produced by the compression algorithm of Section 3.1. The decompression algorithm outputs the corresponding original string S by applying the inverses of the stages of the compression algorithm in reverse order as follows.

3.2.1. Huffman Decoding

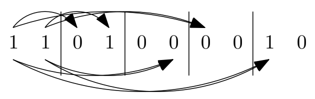
The main obstacle to decoding S^{BW} in parallel is that, because Huffman codes are variable-length codes, we do not know where the boundaries between codewords in S^{BW} lie. We cannot simply begin decoding from any position, as the result will be incorrect if we begin decoding in the middle of a codeword. Thus, we must first identify a set of valid starting positions for decoding. Then, we can trivially decode the substrings of S^{BW} corresponding to those starting positions in parallel.

Our algorithm for locating valid starting positions for Huffman decoding is as follows. Let l be the length of the longest codeword in T, the Huffman table used to produce S^{BW}; l is constant because $|\Sigma|$ is. Without loss of generality, we assume that $|S^{BW}|$ is divisible by l. Divide S^{BW} into partitions of size l. Our goal is to identify one bit in each partition as a valid starting position. The computation will proceed in two steps: (1) initialization and (2) prefix sums computation.

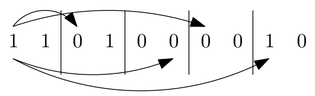
For the initialization stage, we consider every bit i, $0 \leq i < |S^{BW}|$, in S^{BW} as if it were the first bit in a string to be decoded, henceforth S_i^{BW}. In parallel for all i, we decode S_i^{BW} (using the standard serial algorithm) until we cross a partition boundary, at which point we record a pointer from bit i to the stopping point. Now, every bit i has a pointer $i \rightarrow j$ to a bit j in



(a) Step 1: initialization.



(b) Step 2: prefix sums.



S^{BW}	$\underline{1}1$	$\underline{0}$ 10	$\underline{0}$	$\underline{0}$ 0	$\underline{1}0$
S^{MTF}	1	3, 0	3	3, 3	0

(c) Pointers from bit 0, corresponding to valid staring positions in S^{BW} (underlined).

Figure 11: Huffman decoding of S^{BW} (from Figure 7).

the immediately following partition, and if i happens to be a valid starting position, then so is j. See Figure 11(a).

For the prefix sums stage, we define the associative binary operator $\oplus$ to be the merging of adjacent pointers (that is, $\oplus$ merges $A \to B$ and $B \to C$ to produce $A \to C$). See Figure 11(b). The result is that there are now pointers from each bit in the *first* partition to a bit in every other partition. Finally, we identify all bits with pointers from bit 0 as valid starting positions for Huffman decoding (see Figure 11(c)); we refer to this set of positions as V. All this takes $O(\log n)$ time and $O(n)$ work.

The actual decoding is straightforward and proceeds as follows.

1. Employ $|S^{BW}|/l$ (which is $O(n)$) processors, assign each one a different starting position from the set V, and have each processor run the serial Huffman decoding algorithm until it reaches another position in V in order to find the number of decoded characters. Do not actually write the decoded output to memory yet. This takes $O(1)$ time because the partitions are of size $O(1)$.
2. Use prefix sums to allocate space in S^{MTF} for the output of each processor. ($O(\log n)$ time, $O(n)$ work)

3. Repeat step (1) to actually write the output to S^{MTF}. ($O(1)$ time, $O(n)$ work)

These three steps, and thus the entire Huffman decoding algorithm, take $O(\log n)$ time and $O(n)$ work.

3.2.2. Move-to-Front (MTF) Decoding

The parallel MTF decoding algorithm is similar to the parallel MTF encoding algorithm but uses a different operator for the prefix sums step. MTF decoding uses the characters of S^{MTF} directly as indices into the MTF lists L_i; recall from Section 2.2 that L_i is the listing of characters in backward order of appearance relative to position i in S^{BST}. Therefore, for every character in S^{MTF}, we know the effect of the immediately preceding character on the L_i (see Figure 12(b)). We want to know, for every character in S^{MTF}, the cumulative effect of all the preceding characters as shown in Figure 12(d).

Formally, $S^{MTF}[i]$ defines a permutation function mapping L_i to L_{i+1}; this function reproduces the effect of iteration i of the serial algorithm on L_i (i.e., it moves $L_i[S^{MTF}[i]]$ to the front of the list). Denote by $P_{i,j}$ the permutation function mapping L_i to L_j. Given $P_{0,1}$, $P_{1,2}$, $P_{2,3}$, etc., we want to find $P_{0,1}$, $P_{0,2}$, $P_{0,3}$, etc.. We can do this using prefix sums with function composition as the associative binary operator (see Figure 12(c)). A permutation function for a list of constant size can be represented by another list of constant size, so composing two permutation functions takes $O(1)$ time and work. Therefore, the prefix sums computation, as well as the overall MTF decoding algorithm, takes $O(\log n)$ time and $O(n)$ work.

3.2.3. Inverse Block-Sorting Transform (IBST)

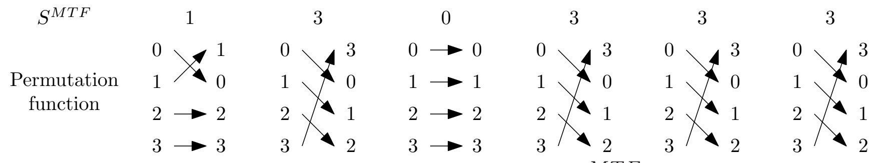
We derive our algorithm from the serial IBST algorithm given in Section 2.1.1. In step 1, we use the integer sorting algorithm of [14] to sort the characters of S^{BST}. Because $|\Sigma|$ is constant, the characters have a constant range, and so this step takes $O(\log n)$ time and $O(n)$ work.

The key difference from the serial algorithm is step 3.1 (in the pseudocode). Recall that this step computes, for each column j of M, the row $L[j]$ containing \$. In our parallel algorithm, we will be guided by the following "inverse clue": for the \$ character of each row i of M, we figure out (all at once—using list ranking) its column.

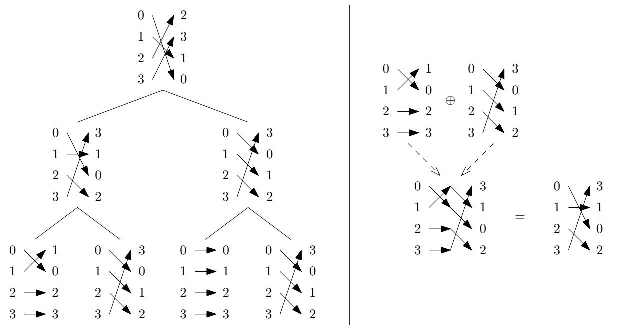
This is done by a reduction to the list ranking problem. The input to the list ranking problem is a linked list represented by an array comprising the elements of the list. Every entry i in the array points to another entry containing its successor $next(i)$ in the linked list. The list ranking problem finds for each element its distance from the end of the list.

1 3 0 3 3 3 0

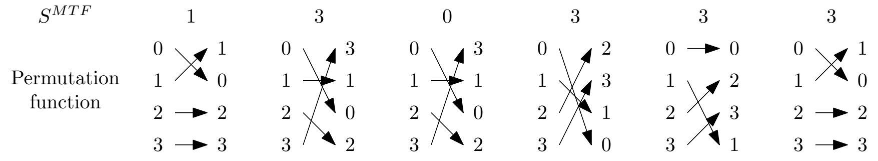
(a) S^{MTF} (from Figure 11).



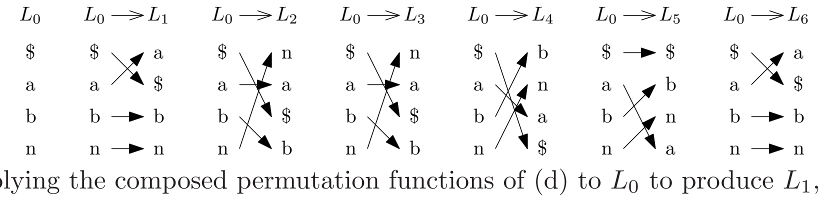
(b) Initialization: the permutation function defined by $S^{MTF}[i]$ moves element i to the front of its input list.



(c) (Left) Prefix sums: composition of permutation functions using a balanced binary tree (here, we show the tree for the first four elements).
(Right) Computing the ⊕-sum of the leftmost two leaves of the tree. The result is the parent of the two leaves.



(d) Output of prefix sums: composed permutation functions.



(e) Applying the composed permutation functions of (d) to L_0 to produce L_1, L_2, etc.

Figure 12: MTF decoding of S^{MTF} from Figure 11: construction of L_i in parallel using composed permutation functions. The last character of S^{MTF} is not used in this construction because the corresponding list L_7 is not needed. Observe the correspondence of the labeled lists in (e) with Figure 6.

In matrix M, every row and every column have a single occurrence of the character \$. Using the permutation T, the \$ character in row i and column j points to \$ in some other row $i_1 = T[i]$ and column $j+1$. The \$ of row i will occupy entry i in the input array for our list ranking problem. The ranking of the linked list will provide for each element its column. This will produce the combinations of row and column for all the \$ characters. We use the list ranking algorithm of [15] to rank the linked list in $O(\log n)$ time and $O(n)$ work.

Overall, the IBST takes $O(\log n)$ time and $O(n)$ work.

The above presentation proves the following theorem:

Theorem 2. *The algorithm of Section 3.2 solves the Burrows-Wheeler Decompression problem for a string of length n over a constant alphabet in $O(\log n)$ time using $O(n)$ work.*

4. Conclusion

This paper presents the first fast optimal PRAM algorithms for the Burrows-Wheeler Compression and Decompression problems. This is particularly significant since PRAM parallelism has been all but absent from lossless compression problems. In addition to this algorithmic complexity result, this paper provides new insight into how BW compression works. It also suggests that elementary parallel routines such as prefix-sums and list ranking may be more powerful than meets the eye.

5. Acknowledgement

Helpful discussions with Prakash Narayan are gratefully acknowledged.

References

[1] M. Burrows, D. J. Wheeler, A block-sorting lossless data compression algorithm, Tech. rep., Digital Systems Research Center (1994).

[2] J. Seward, bzip2, a program and library for data compression, `http://www.bzip.org/`.

[3] J. Gilchrist, A. Cuhadar, Parallel lossless data compression based on the Burrows-Wheeler transform, in: Proc. Advanced Information Networking and Applications, 2007, pp. 877 –884. doi:10.1109/AINA.2007.109.

[4] S. T. Lavavej, bwtzip: A linear-time portable research-grade universal data compressor, `http://nuwen.net/bwtzip.html`.

[5] Y. Wiseman, Burrows-Wheeler based JPEG, Data Science Journal 6 (2007) 19–27.

[6] I. Gilmour, R. J. Dávila, Lossless video compression for archives: Motion JPEG2k and other options (2006).
URL `http://www.media-matters.net/docs/WhitePapers/WPMJ2k.pdf`

[7] A. Eirola, Lossless data compression on GPGPU architectures, `http://arxiv.org/abs/1109.2348v1` (2011).

[8] S. T. Klein, Y. Wiseman, Parallel Huffman decoding with applications to JPEG files, The Computer Journal 46 (5) (2003) 487–497. arXiv:http://comjnl.oxfordjournals.org/content/46/5/487.full.pdf+html, doi:10.1093/comjnl/46.5.487.

[9] J. Seward, On the performance of BWT sorting algorithms, in: Data Compression Conference, 2000. Proceedings. DCC 2000, 2000, pp. 173 –182. doi:10.1109/DCC.2000.838157.

[10] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (6) (2006) 918–936. doi:10.1145/1217856.1217858.

[11] G. Nong, S. Zhang, W. H. Chan, Linear suffix array construction by almost pure induced-sorting, in: Proc. Data Compression Conference, IEEE, 2009, pp. 193–202. doi:10.1109/DCC.2009.42.

[12] S. C. Sahinalp, U. Vishkin, Symmetry breaking for suffix tree construction, in: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, STOC '94, ACM, New York, NY, USA, 1994, pp. 300–309. doi:10.1145/195058.195164.

[13] R. E. Tarjan, U. Vishkin, Finding biconnected components and computing tree functions in logarithmic parallel time, in: Proceedings of the 25th Annual Symposium on Foundations of Computer Science, 1984, pp. 12–20. doi:10.1109/SFCS.1984.715896.

[14] R. Cole, U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, Inf. Control 70 (1) (1986) 32–53. doi:10.1016/S0019-9958(86)80023-7.

[15] R. Cole, U. Vishkin, Faster optimal parallel prefix sums and list ranking, Information and Computation 81 (3) (1989) 334 – 352. doi:10.1016/0890-5401(89)90036-9.