Practical Parallel Lempel-Ziv Factorization

Julian Shun *
Carnegie Mellon University
jshun@cs.cmu.edu

Fuyao Zhao *†
Carnegie Mellon University
fuyaoz@cs.cmu.edu

Abstract: In the age of big data, the need for efficient data compression algorithms has grown. A widely used data compression method is the Lempel-Ziv-77 (LZ77) method, being a subroutine in popular compression packages such as gzip and PKZIP. There has been a lot of recent effort on developing practical sequential algorithms for Lempel-Ziv factorization (equivalent to LZ77 compression), but research in practical parallel implementations has been less satisfactory. In this work, we present a simple work-efficient parallel algorithm for Lempel-Ziv factorization. We show theoretically that our algorithm requires linear work and runs in $O(\log^2 n)$ time (randomized) for constant alphabets and $O(n^\epsilon)$ time ($\epsilon < 1$) for integer alphabets. We present experimental results showing that our algorithm is efficient and achieves good speedup with respect to the best sequential implementations of Lempel-Ziv factorization.

1 Introduction

As data sizes increase, compression techniques which reduce the storage requirements of such data become more important. Data compression techniques are widely studied in computer science and fall into two categories—lossless and lossy. For *lossless* methods (e.g. Lempel-Ziv [31, 32], arithmetic coding [26], Huffman coding [15] and Burrows-Wheeler [3]), no information is lost when the data is compressed, while compression with *lossy* methods (e.g. JPEG and MPEG) can result in some information loss.

Lempel-Ziv-77 (LZ77) [31] and Lempel-Ziv-78 (LZ78) [32] form the basis for the family of Lempel-Ziv methods. They are dictionary coders, meaning that the encoder searches a dictionary for matches of substrings of the text, and returns a pointer to the substring's location in the dictionary. In LZ77, the encoder uses a sliding window (implicit dictionary) over the text to search for previous occurrences of substrings. Lempel-Ziv-Storer-Szymanski (LZSS) [29] is a variant of LZ77 that returns a pointer to the dictionary only if the matched substring is "long enough". LZ78 stores an explicit dictionary containing substrings previously seen, and in each iteration searches this dictionary to find the longest substring that exists in the dictionary, and then inserts a new entry into the dictionary. Lempel-Ziv-Welch [30] is a variant of LZ78 that uses a pre-initialized dictionary.

As we are interested in parallel implementations, we focus on LZ77 in this work rather than LZ78, since LZ77 admits efficient parallel solutions, whereas LZ78 was shown to be P-complete (unlikely to have an efficient parallel solution) [10, 11].

LZ77 is a lossless dynamic compression method that has been popular due to its simplicity and computational efficiency. It is a component of the DEFLATE algorithm, which is used in software packages such as gzip and PKZIP. The LZ77 algorithm consists of a compression stage, which computes the Lempel-Ziv factorization (LZ-factorization) of the input string, and a decompression stage, which recovers the original string from the

*Joint first author.

†Part of this work was done while author was an undergraduate student at Central South University.

compressed string. The LZ-factorization can be computed sequentially [27] in linear time with a suffix tree [21], and decompression can be done sequentially in linear time with a scan. The first parallel algorithms for LZ-factorization were described independently by Noar [22] and Crochemore and Rytter [9]. For a string of length n, their algorithms require $O(\log n)$ time and $O(n \log n)$ work on the PRAM, making them not work-efficient. Farach and Muthukrishnan [13] give the first linear-work algorithms for both LZ-factorization and decompression, each requiring $O(\log n)$ time on the PRAM. These parallel algorithms all make use of parallel suffix trees. LZ77 decompression is much simpler and faster than LZ-factorization, so we focus on the latter.

There has been much research done in designing practical sequential algorithms for computing the LZ-factorization. Recently, researchers have proposed the use of suffix arrays instead of suffix trees to obtain faster and more space-efficient algorithms for LZ-factorization [8, 6, 5, 7, 23]. Since suffix arrays can be computed in linear time [17], these LZ-factorization algorithms are also able to run in linear time. The aforementioned sequential algorithms have been shown to perform well in practice.

To the best of our knowledge, the only parallel implementations of LZ-factorization described in the literature are those of Klein and Wiseman (using CPUs) [19] and Ozsoy and Swany (using GPUs) [24]. Both implementations involve splitting the input string among processors and having each processor independently compute the factorization of its substring. Because in these implementations the processors do not necessarily have access to the entire input string, they do not always compute the same LZ-factorization as would be computed sequentially, and thus can produce larger compressed files. Furthermore, the corresponding papers [19, 24] do not provide any complexity bounds on work and time. Previous work on parallel algorithms for computing the same LZ-factorization as would be computed sequentially do not include any implementations or experiments [13, 22, 9]. The linear-work algorithm of Farach and Muthukrishnan [13] does not lead to a practical implementation, as it involves complicated parallel methods for tree contraction, least common ancestors and Euler tours.

In this work, we present a simple linear-work parallel algorithm for LZ-factorization and practical implementations of it. Our algorithm computes the same factorization as would be computed sequentially. The algorithm is based on parallel suffix arrays [17], finding all nearest smaller values [1], and uses simple parallel routines such as prefix sums and list ranking [16]. Theoretically, our algorithm requires $O(n)$ work and $O(\log^2 n)$ time (randomized) due to the use of suffix arrays, so does not achieve the $O(\log n)$ time bound of Farach and Muthukrishnan [13], but lends itself to a practical implementation. We show experimentally that on 40 cores we achieve speedups between 6.7 and 21.2 compared to running the algorithm on a single core. We also implement a sequential algorithm for LZ-factorization that is faster than previous algorithms [1], and our parallel algorithm achieves a 4.8–15.5 fold speedup on 40 cores over this sequential algorithm.

2 Lempel-Ziv Factorization

Preliminaries. The LZ-factorization of a string $\mathsf{S}[0, \ldots, n-1]$ is $\mathsf{S} = \omega_0\omega_1 \ldots \omega_{m-1}$,

[1] Very recently, a slightly faster sequential algorithm was independently described by Kempa and Puglisi [18].

where $m \leq n$ and for each $0 \leq i < m$, ω_i (called the ith ***factor*** of the string) is either a single character which does not appear in $\omega_0 \ldots \omega_{i-1}$ or is the longest prefix of $\omega_i \ldots \omega_{m-1}$ that also appears starting at a position to the left of ω_i in S. In the example given by Crochemore et. al. [8], the string $abbaabbbaaabab$ has the factorization $\mathsf{S} = \omega_0 \ldots \omega_7$ where $\omega_0 = a$, $\omega_1 = b$, $\omega_2 = b$, $\omega_3 = a$, $\omega_4 = abb$, $\omega_5 = baa$, $\omega_6 = ab$ and $\omega_7 = ab$. To achieve compression, the ω values are not explicitly returned. LZ77 returns a sequence of pairs $(\mathsf{start}_i, \mathsf{prev}_i)$ where start_i indicates the starting position of ω_i in S and prev_i indicates the position of ω_i's left match in S if it exists (coded), and otherwise stores the character at position start_i (uncoded). For decompression, each pair i can reconstruct its factor by looking at prev_i and either directly copying prev_i if it is a character or copying $(\mathsf{start}_{i+1} - \mathsf{start}_i)$ characters starting at the position stored in prev_i (we consider the value of start_0 to be 0 and start_m to be n). The sequence of pairs returned for the string $abbaabbbaaabab$ is $[(0, a), (1, b), (2, 1), (3, 0), (4, 0), (7, 2), (10, 0), (12, 10)]$.

Throughout our discussion, we denote the LZ-factorization by an array LZ of size m where $\mathsf{LZ}[i]$ stores only the start_i value of the pair. To obtain the LZ77 representation, the prev_i value of the pair can easily be computed given the previous occurrence array, defined later in this section. This can easily be modified to return the LZSS representation [29].

We denote suf_i of string S to be the suffix of S starting at position i. The ***suffix array*** [20] of a string $\mathsf{S}[0, \ldots, n-1]$ is an array SA of length n such that SA is a permutation of the integers $\{0, \ldots, n-1\}$ and $\mathsf{suf}_{\mathsf{SA}[0]} \prec \mathsf{suf}_{\mathsf{SA}[1]} \prec \ldots \prec \mathsf{suf}_{\mathsf{SA}[n-1]}$, where $\prec$ means lexicographically less than. We use $\mathrm{lcp}(s_i, s_j)$ to denote the length of the longest common prefix (lcp) between strings s_i and s_j. The ***longest common prefix*** array is an array LCP of length n such that $\mathsf{LCP}[0] = 0$ and for $i > 0$, $\mathsf{LCP}[i] = \mathrm{lcp}(\mathsf{suf}_{\mathsf{SA}[i-1]}, \mathsf{suf}_{\mathsf{SA}[i]})$. The ***longest previous factor*** of an index i in S, is equal to the maximum value of $\mathrm{lcp}(\mathsf{suf}_i, \mathsf{suf}_j)$, for all $j < i$. We denote LPF to be the longest previous factor array, where $\mathsf{LPF}[i]$ stores the longest previous factor of index i (0 if none). We use $\mathsf{prevOcc}$ to denote the ***previous occurrence array***, where $\mathsf{prevOcc}[i]$ stores the starting location of the longest previous factor of suf_i in S (-1 if none). Figure 1 shows the SA, LCP, LPF, $\mathsf{prevOcc}$ and LZ arrays for the string $abbaabbbaaabab$.

i	$\mathsf{S}[i]$	$\mathsf{SA}[i]$	$\mathsf{LCP}[i]$	suf_i	$\mathsf{LPF}[\mathsf{SA}[i]]$	$\mathsf{prevOcc}[\mathsf{SA}[i]]$	$\mathsf{LZ}[i]$
0	a	8	0	$aaabab$	2	3	0
1	b	9	2	$aabab$	3	3	1
2	b	3	3	$aabbbaaabab$	1	0	2
3	a	12	1	ab	2	10	3
4	a	10	2	$abab$	2	0	4
5	b	0	2	$abbaabbbaaabab$	0	-1	7
6	b	4	3	$abbbaaabab$	3	0	10
7	b	13	0	b	1	7	12
8	a	7	1	$baaabab$	3	2	–
9	a	2	3	$baabbbaaabab$	1	1	–
10	a	11	2	bab	2	2	–
11	b	6	1	$bbaaabab$	4	1	–
12	a	1	4	$bbaabbbaaabab$	0	-1	–
13	b	5	2	$bbbaaabab$	2	1	–

Figure 1: SA, LCP, LPF, $\mathsf{prevOcc}$ and LZ for $\mathsf{S} = abbaabbbaaabab$.

The ***all nearest smaller values (ANSV)*** problem is defined as follows: for each element

in a sequence of elements from a total ordering, find the closest smaller element to the left and the closest smaller element to the right of it (if there is no smaller element, then report it). An algorithm for ANSV returns two arrays LN and RN where LN$[i]$ (RN$[i]$) contains the index of the nearest smaller element to the left (right) of element i (-1 if none).

Throughout this paper, we will use the parallel random-access machine (PRAM) model of parallel computation [16], where ***work*** W refers to the number of operations performed by an algorithm and ***time*** T refers to the number of time steps required by an algorithm. We will assume that the model supports concurrent reads and concurrent writes (CRCW PRAM). If the number of processors available is P, then by Brent's work-time scheduling principle, the total running time will be proportional to $O(\frac{W}{P} + T)$ [16].

Algorithm. Our parallel algorithm is based on the sequential algorithm described by Crochemore, Ilie and Smyth (henceforth CIS) [8], which first computes the LPF array. Computing the LZ-factorization can then be computed with a single pass over the LPF array [6]. The psuedocode for computing LZ from LPF is shown below [6].

LPFtoLZ(LPF, n)

```
1: LZ[0] = 0
2: i = 0
3: while LZ[i] < n do
4:     LZ[i + 1] = LZ[i] + max(1, LPF[LZ[i]])
5:     i = i + 1
6: return LZ
```

We first describe Farach and Muthukrishnan's method of parallelizing **LPFtoLZ** given the LPF array as an input [13]. Their method creates a size $n + 1$ array of pointers, next, where next$[i] = \min(i + \max(\mathsf{LPF}[i], 1), n)$ for $i < n$ and next$[n] = -1$. Following the indices (pointers) starting at next$[0]$ until reaching a value of -1 is sufficient to determine the indices in LZ. Using a parallel list ranking algorithm [16] with the value at index 0 set to 1 and the remaining values set to 0, the result is an array of flags indicating which indices are in the LZ-factorization. This can be done in $O(n)$ work and $O(\log n)$ time. A prefix sums [16] is then done on the array of flags to get the start values for the elements in the LZ-factorization, and this can also be done in $O(n)$ work and $O(\log n)$ time.

Now what remains is to show how to compute the LPF array. As done in CIS, the suffix array SA is first computed. While CIS computes the LCP array after computing SA, we compute LCP while computing SA. Using the algorithm of Karkkainen and Sanders [17], both SA and LCP can be computed in parallel using $O(n)$ work and $O(\log^2 n)$ time (randomized) for constant-sized alphabets and $O(n)$ work and $O(n^\epsilon)$ for $\epsilon < 1$ on integer alphabets on the CRCW PRAM.

After computing SA and LCP, we apply the following lemma due to Crochemore et. al. [7], which states that any LPF$[i]$ can be computed using an ANSV computation and range minima queries on SA and LCP. To deal with boundary cases, we assume $\mathsf{suf}_{\mathsf{SA}[-1]}$ evaluates to the empty string (and therefore has an lcp of 0 with any other string).

Lemma 2.1 *Let* LN$[i]$ *and* RN$[i]$ *be the left and right nearest smaller neighbors of element* i *in* SA. *Then* $\mathsf{LPF}[i] = \max(\mathrm{lcp}(\mathsf{suf}_{\mathsf{SA}[i]}, \mathsf{suf}_{\mathsf{SA}[\mathsf{LN}[i]]}), \mathrm{lcp}(\mathsf{suf}_{\mathsf{SA}[i]}, \mathsf{suf}_{\mathsf{SA}[\mathsf{RN}[i]]}))$.

Berkman, Schieber and Vishkin [1] show that ANSVs can be computed in $O(n)$ work and $O(\log \log n)$ time on the CRCW PRAM. It can be shown that for any $0 \leq i < j < n$,

$\mathrm{lcp}(\mathsf{suf}_{\mathsf{SA}[i]}, \mathsf{suf}_{\mathsf{SA}[j]}) = \min_{i<k\leq j} \mathsf{LCP}[k]$, so using range minima queries, we can compute the lcp values and hence the LPF values [17]. $\mathsf{prevOcc}[i]$ is set to $\mathsf{LN}[i]$ if $\mathsf{suf}_{\mathsf{SA}[\mathsf{LN}[i]]}$ has a longer lcp with $\mathsf{suf}_{\mathsf{SA}[i]}$, and $\mathsf{RN}[i]$ otherwise. Range minima queries can be performed in $O(1)$ work and time, and requires $O(n)$ work and $O(\log n)$ time for preprocessing [16].

In the example shown in Figure 1, to determine $\mathsf{LPF}[7]$ and $\mathsf{prevOcc}[7]$ (corresponding to suffix $baaabab$), we look at its left nearest smaller value in SA, which is 4, and its right nearest smaller value, which is 2, and then select the one corresponding to the suffix with a larger lcp with $baaabab$, which is of length 3. Therefore, $\mathsf{LPF}[7] = 3$ and $\mathsf{prevOcc}[7] = 2$.

We now describe two variants of our algorithm, differing only in how LPF is computed. Our first variant (PLZ1) uses Lemma 2.1 directly. It builds a range minima query table on the LCP array for constant-time queries and then in parallel does range minima queries to compute each $\mathsf{LPF}[i]$. The n queries require a total of $O(n)$ work and $O(1)$ time.

Our second variant (PLZ2) uses as a component the sequential algorithm of Crochemore et. al. [6], which takes the ANSVs as input and does a single pass over the string to compute the LPF array. Their crucial observation is that $\mathsf{LPF}[i] \geq \mathsf{LPF}[i-1] - 1$, and using this dependence they derive a linear-work algorithm for computing LPF.

Unlike PLZ1, PLZ2 does not build a range minima query table for constant-time queries but instead builds a segment tree [12] on the LCP array, an idea which was also investigated by Canovas and Navarro [4]. The segment tree is a binary tree whose leaves store the elements of LCP and internal nodes store the minimum value of its children. It requires $O(n)$ work and $O(\log n)$ time to construct. Range minima queries can be answered by traversing the $O(\log n)$ levels of the tree, hence requiring $O(\log n)$ work and time. PLZ2 then divides the input into $\frac{n}{\log n}$ blocks and computes the LPF values of each block. The longest previous factor of the first element is computed using a range minima query on the segment tree described above, and since $\mathsf{LPF}[i]$ only depends on $\mathsf{LPF}[i-1]$, we can then run the sequential algorithm of Crochemore et. al. [6] to compute the remaining longest previous factors of each block. Since we perform in parallel one query for each of the $\frac{n}{\log n}$ blocks, this leads to a cost of $O(n)$ work and $O(\log n)$ time. Running the linear-work sequential algorithm per block in parallel takes a total of $O(n)$ work and $O(\log n)$ time, since the size of each block is $O(\log n)$. Our motivation for designing PLZ2 was that constructing the segment tree is simpler than constructing the table for constant-time queries, and since we only perform queries on a subset of the elements, we found experimentally that the decreased construction time more than makes up for the increased query times.

The steps for LZ-factorization are summarized below. PLZ1 and PLZ2 differ only in the computation of step 3.

ComputeLZ(S, n)

1: Compute the suffix array, SA, and longest common prefix array, LCP, for S.
2: Compute the left and right smaller neighbor arrays, LN and RN, on SA using an ANSV algorithm.
3: Compute the LPF and prevOcc arrays.
4: **return LPFtoLZ**(LPF, n)

From the above discussion, we see that all the steps require $O(n)$ work, and the time is dominated by the suffix array construction. We have the following lemma:

Lemma 2.2 *Our parallel algorithm for computing the Lempel-Ziv factorization requires $O(n)$ work and $O(\log^2 n)$ time (randomized) for constant-sized alphabets and $O(n)$ work*

and $O(n^\epsilon)$ ($\epsilon < 1$) for integer alphabets on the CRCW PRAM.

Except for the suffix array and lcp computation, our algorithm takes $O(\log n)$ time for arbitrary alphabets, so improvements to the bounds for suffix array and lcp computation can improve our overall bounds as well. Our algorithm is amenable to implementation, as we describe in the next section.

3 Implementations

Parallel. We implemented PLZ1, PLZ2 and a simple variant of PLZ2 that avoids computing the LCP array. For suffix arrays, we used the linear-work and $O(n^\epsilon)$ time (for some constant $\epsilon < 1$) implementation from the Problem Based Benchmark Suite [28]. There are faster sequential suffix array codes when running on one core, but most of them do not compute the LCP array (see, e.g., [25]). We implemented an optimized version of the $O(n \log n)$ work and $O(\log n)$ time ANSV algorithm of Berkman et. al. [1] instead of their much more complicated work-optimal version. For **LPFtoLZ** , we implemented a random sampling-based list ranking algorithm [16] and used the parallel sequence routines from the Problem Based Benchmark Suite [28]. For the range minima query table used for computing the LCP array inside the suffix array algorithm, we used an $O(n \log n)$ work, $O(\log n)$ time construction for constant-time range minima queries. For PLZ1, we built a range minima table on the resulting LCP array using the same construction.

For PLZ2, we set the number of blocks to $\frac{n}{8196}$ in our experiments, which we found to give the best results. We implemented the sequential algorithm of Crochemore et. al. [6], which is used in each block. Our variant of PLZ2, which we call PLZ3, does not compute the LCP array, but instead computes the lcp values of the first element of each block with its nearest smaller neighbors using naive string comparison, and uses this to compute its LPF value. The rest of each block is computed in the same way as in PLZ2.

Sequential. Here we describe a simple sequential algorithm for LZ-factorization which we use in our experiments in Section 4. Our sequential algorithm (LZ-ANSV) first computes the suffix array (without lcps), and then computes the ANSVs on the suffix array sequentially using the stack-based algorithm of Gabow et. al. [14]. It then loops through the suffixes in their original order, and for the positions appearing in the LZ-factorization, it computes the longest previous factor with the suffixes corresponding to the positions of their left and right smaller neighbors in SA using naive string comparison. By incrementing the index of the loop by the length of the longest previous factor after computing it for an element, it bypasses the LPF computation for the elements not appearing in the LZ-factorization. LZ-ANSV requires $O(n)$ work.

4 Experiments

We experimentally compare the performance of the different implementations of our parallel LZ-factorization algorithm. We are not aware of any existing parallel implementations for computing the same LZ-factorization as would be computed sequentially. Previous parallel algorithms for doing so [13, 22, 9] use parallel suffix trees and are relatively complicated (no implementations are available). We show that the best publicly available parallel suffix tree algorithm [2] is slower than that of our entire LZ-factorization algorithm on most strings; hence it is unlikely that a parallel implementation of LZ-factorization that uses suffix trees will outperform our implementation.

We compare our parallel code with our sequential LZ-ANSV code and the sequential algorithm of Ohlebusch and Gog [23], which they showed to be faster than other sequential algorithms (note that as mentioned in Footnote 1, very recently, a faster algorithm was published by Kempa and Puglisi [18]). We obtained the code from Ohlebusch and Gog, and refer to it as LZ-OG. All of the implementations in our experiments compute pairs containing the starting position and previous occurrence for each factor in the LZ-factorization. For fair comparison, all of the implementations used the same suffix array code [28].

Experimental Setup. We performed experiments on a 40-core Intel machine (with hyper-threading) with 4×2.4GHz Intel 10-core E7-8870 Xeon processors, a 1066MHz bus, and 256GB of main memory. The parallel programs were compiled with Intel's `icpc` compiler (version 12.1.0) using CilkPlus with the `-O2` flag. The sequential programs were compiled using `g++` 4.4.1 with the `-O2` flag.

We used a variety of real-world strings available online (`http://people.unipmn.it/manzini/lightweight/corpus/` and `http://pizzachili.dcc.uchile.cl/texts.html`), XML code from Wikipedia samples (wikisamp*.xml), and artificial inputs. Our artificial inputs are of size 10^7 and include an all identical string (10Midentical), a random string with an alphabet size of 10 (10Mrandom), and a string with an alphabet size of 2 where every $\sqrt{10^7}$th position contains the first character and all other positions contain the second character (10Msqrtn).

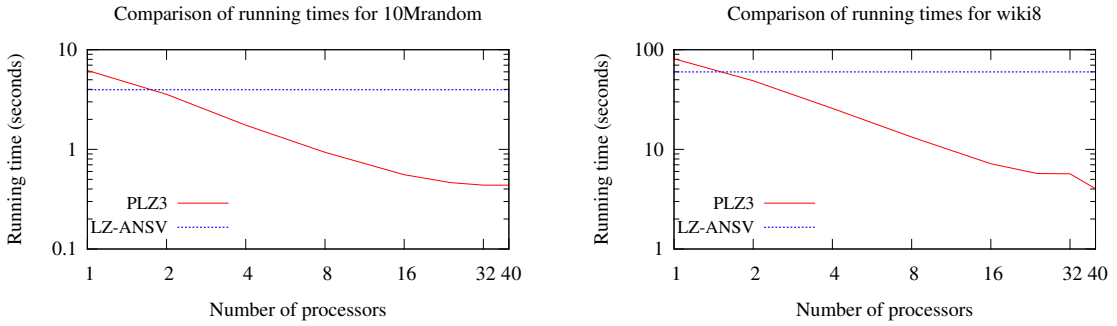
Experimental Results. We first compared the three variants of our PLZ algorithm and found that PLZ3 gives the best absolute performance across the board, both in parallel and sequentially. This is due to the fact that PLZ3 does not need to compute the LCP array (which takes about one-third of the time of the suffix array code), and this more than makes up for the extra time spent in performing naive string comparisons for the first element of each block. The parallel running times on 40 cores with hyper-threading (T_{40}) for all three variants are shown in Table 1.

We compared PLZ3 to the two sequential algorithms LZ-ANSV and LZ-OG, and Table 1 shows a comparison of running times on the input strings. T_1 is the time (in seconds) for running PLZ3 on a single core and the speedup is computed as T_1/T_{40}. The results show that our sequential algorithm (LZ-ANSV) outperforms LZ-OG for all of the input strings. Note that, however, LZ-ANSV does not compute the entire LPF array whereas LZ-OG does, so for applications where the entire LPF array is required, LZ-ANSV will not suffice. On a single core, PLZ3 is 1.3–1.6 times slower than LZ-ANSV. On 40 cores with hyper-threading, PLZ3 achieves 6.7–21.2 times speedup with respect to running the algorithm on one core. It achieves a 4.8–15.5 times speedup with respect to LZ-ANSV.

The running times of PLZ3 and LZ-ANSV as a function of the number of processors for the 10Mrandom and wikisamp8.xml files are shown in Figures 2(a) and 2(b), respectively. PLZ3 achieves good speedup and outperforms LZ-ANSV with just 2 or more processors. Figure 3 shows the running time of PLZ3 on 40 cores as a function of the input size (random characters). We see that PLZ3 scales gracefully with the size of the input. Figure 4 shows the breakdown of the running time of PLZ3 on several input strings. We note that the suffix array takes about 80% of the time. If the lcps are also computed (as in PLZ1 and PLZ2), then the suffix array time becomes about 1.5 times slower, which explains why PLZ3 improves over PLZ1 and PLZ2 by not computing the LCP array. The LPF computation takes

Text	Size (MB)	LZ-ANSV	LZ-OG	PLZ3 T_1	PLZ3 T_{40}	PLZ3 Speedup	PLZ1 T_{40}	PLZ2 T_{40}	PST T_{40}
10Midentical	10	1.68	1.74	2.35	0.347	6.772	0.553	0.515	0.377
10Mrandom	10	3.97	4.67	6.2	0.437	14.18	0.521	0.55	0.286
10Msqrtn	10	2.14	2.44	3.36	0.401	8.379	0.618	0.611	0.439
chr22.dna	34.6	19.4	22.0	28.9	1.57	18.40	1.9	2.02	1.56
etext99	105	69.9	75.2	99.0	4.8	20.62	5.69	6.18	5.33
howto.txt	39.4	24.0	25.5	33.4	1.82	18.35	2.24	2.36	1.93
jdk13c	69.7	40.4	41.4	54.1	2.86	18.91	3.89	3.95	3.45
pitches	55.8	31.8	34.3	43	2.27	18.94	2.79	2.89	2.47
proteins	210	147	172	203	9.79	20.74	11.6	12.5	11.4
rctail96	115	70.0	72.9	96.5	4.77	20.23	6.19	6.57	5.83
rfc	116	72.8	76.6	100	4.83	20.70	6.19	6.5	5.77
sources	211	140	163	186	9.17	20.28	11.7	12.1	11.4
sprot34.dat	110	69.0	72.2	93.7	4.6	20.36	5.9	6.31	5.57
w3c2	104	63.1	64.7	84.1	4.42	19.02	5.96	6.11	5.5
wikisamp8.xml	100	59.9	61.4	81.2	4.03	20.14	5.51	5.54	4.96
wikisamp9.xml	1000	653	670	894	42.1	21.23	55.1	56.2	59.9

Table 1: Comparison of running times (seconds) of parallel and sequential algorithms on different inputs on a 40-core machine with hyper-threading.



(a) 10Mrandom (b) wikisamp8.xml

Figure 2: Log-log plots of running times on a 40-core machine (with hyper-threading).

less than 20% of the overall time, and the ANSV computation and conversion from LPF to LZ take very little time. The suffix array portion of the code achieves the lowest speedup, so improvements in parallel suffix array code will likely improve our LZ-factorization code.

In Table 1, we also include the 40-core times for a modified version of the best publicly available parallel suffix tree code of Blelloch and Shun [2] (labeled **PST**). We note that for most strings, the parallel suffix tree code alone takes as much time or more time than our entire LZ-factorization algorithm, and since a suffix tree-based LZ-factorization algorithm involves many other procedures (e.g. tree contraction, least common ancestors and Euler tours), it is unlikely that such an algorithm will have a better overall performance.

5 Conclusion

We have presented a simple parallel algorithm for Lempel-Ziv factorization, which requires $O(n)$ work and $O(\log^2 n)$ time (randomized) for constant-sized alphabets and $O(n)$ work and $O(n^\epsilon)$ time ($\epsilon < 1$) for integer alphabets. We showed that a practical implementation of our algorithm is fast for a wide variety of input strings, and achieves good speedup on

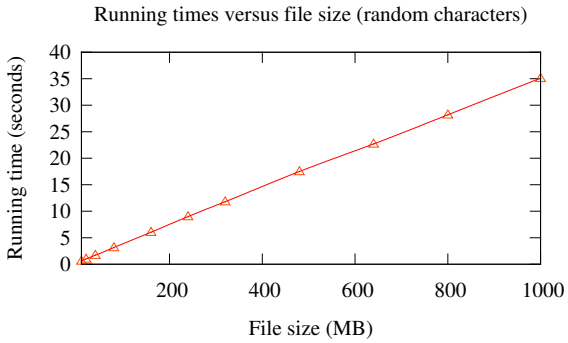


Figure 3: Running time versus input size of PLZ3 on 40 cores.

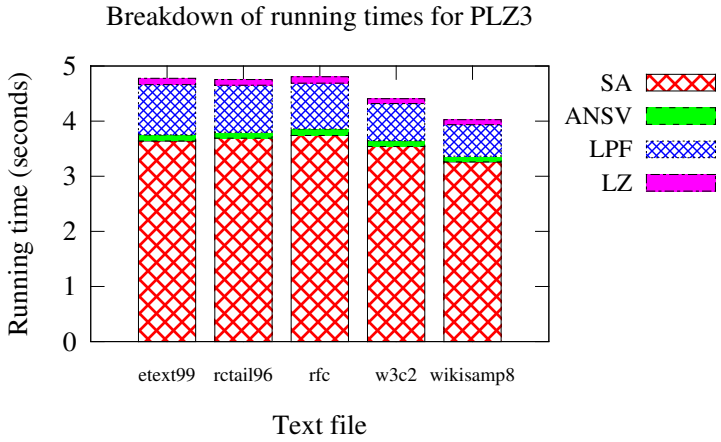


Figure 4: Breakdown of running time of PLZ3 on 40 cores.

a 40-core machine, even relative to the best sequential implementations. Opportunities for future work include integrating our algorithm into popular compression packages and developing practical parallel implementations of other methods in the Lempel-Ziv family.

Acknowledgments. We thank Guy Blelloch and the anonymous reviewers for their helpful comments. This work is partially supported by the National Science Foundation under grant number CCF-1018188, and by Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program.

References

[1] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *J. Algorithms*, 1993.

[2] G. E. Blelloch and J. Shun. A simple parallel cartesian tree algorithm and its application to suffix tree construction. In *ALENEX*, 2011.

[3] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, HP Labs, 1994.

[4] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *SEA*, 2010.

[5] G. Chen, S. Puglisi, and W. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1, 2008.

[6] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 2008.

[7] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń. LPF

computation revisited. In *Combinatorial Algorithms*. 2009.

[8] M. Crochemore, L. Ilie, and W. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In *Data Compression Conference*, 2008.

[9] M. Crochemore and W. Rytter. Efficient parallel algorithms to test square-freeness and factorize strings. *Inf. Process. Lett.*, 1991.

[10] S. De Agostino. P-complete problems in data compression. *Theor. Comp. Sci.*, 1994.

[11] S. De Agostino. Lempel-Ziv data compression on parallel and distributed systems. In *Data Compression, Communications and Processing*, 2011.

[12] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

[13] M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression (extended abstract). In *SPAA*, 1995.

[14] H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *STOC*, 1984.

[15] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 1952.

[16] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[17] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. In *ICALP*, 2003.

[18] D. Kempa and S. J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *ALENEX*, 2013.

[19] S. T. Klein and Y. Wiseman. Parallel Lempel Ziv coding. *Discrete Appl. Math.*, 2005.

[20] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *SODA*, 1990.

[21] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 1976.

[22] M. Naor. String matching with preprocessing of text and pattern. In *ICALP*, 1991.

[23] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *CPM*, 2011.

[24] A. Ozsoy and M. Swany. CULZSS: LZSS lossless data compression on CUDA. In *IEEE International Conference on Cluster Computing*, 2011.

[25] S. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 2007.

[26] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM J. Res. Dev.*, 1979.

[27] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. ACM*, 1981.

[28] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *SPAA*, 2012.

[29] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 1982.

[30] T. Welch. A technique for high-performance data compression. *Computer*, June 1984.

[31] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 1977.

[32] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 1978.