Lempel-Ziv Parsing in External Memory*

Juha Kärkkäinen Dominik Kempa Simon J. Puglisi

Helsinki Institute for Information Technology (HIIT) and
Department of Computer Science, University of Helsinki, Finland
{firstname.lastname}@cs.helsinki.fi

Abstract

In the 35 years since its discovery, the Lempel-Ziv factorization (or LZ77 parsing) has become a fundamental method for data compression and string processing. In many applications, computation of the factorization is a time-space bottleneck. However, and despite the increasing need to apply LZ77 to massive data sets (for both storage and indexing), no algorithm to date scales to inputs that exceed the size of RAM. In this paper we describe the first algorithms for computing the LZ77 parsing efficiently using external memory.

1 Introduction

For over three decades the Lempel-Ziv (LZ77) factorization [18] has been a fundamental tool for data compression and today it lies at the heart of popular file compressors (e.g. *gzip* and *7zip*), and information retrieval systems (see, e.g., [7]).

Recently LZ77 has become the basis for several compressed full-text self-indexes [15, 9, 10]. These indexes are designed to support efficient storage and fast searching of massive, highly repetitive data sets such as web crawls, whole genome databases, and versioned repositories of source code files and multi-author documents, like Wikipedia.

In order for these LZ77-based indexes to be constructed, whole collection (i.e. unbounded window) LZ77 factorizations need to be computed. However, to our knowledge, all current LZ77 algorithms are designed to be "in-memory" algorithms and require large amounts of RAM to operate. In order to compute the LZ factorization of a text of length n, most algorithms need at least $6n$ bytes of RAM and often much more [1, 13]. Recently, we managed to reduce the memory requirement to less than $2n$ bytes [12]. In this paper, we go beyond the limitations of RAM by using external memory.

Our contribution. We have designed and implemented three external memory algorithms for LZ77 factorization with quite different resource requirements with respect to CPU time, I/O volume, RAM and disk space.

We present the results from an experimental comparison that has been carefully designed to expose the strengths and limitations of the algorithms. It shows that each of the three algorithms has its niche (a combination of input and system parameters), where it is the best of all algorithms.

*This research is partially supported by Academy of Finland through grant 118653 (ALGODAN) and grant 250345 (CoECGR).

1068-0314/14 $31.00 © 2014 IEEE
DOI 10.1109/DCC.2014.78

Related work. A recent survey [1] and even more recent papers [12, 13] outline the many algorithms for LZ77 factorization, all of which operate in RAM. The performance of the best algorithms has been investigated in recent experimental comparisons [12, 13]. According to those studies, the most efficient in-memory algorithm under limited memory conditions is LZscan, which we have included in our experiments as a point of reference. There is also a recent parallel algorithm for LZ77 [17] that provides a substantial practical speed-up but needs lots of RAM.

2 Basic Notation

Throughout we consider a string $\mathsf{X} = \mathsf{X}[1..n] = \mathsf{X}[1]\mathsf{X}[2]\ldots\mathsf{X}[n]$ of $|\mathsf{X}| = n$ symbols drawn from the alphabet $[0..\sigma-1]$. For $i = 1,\ldots,n$ we write $\mathsf{X}[i..n]$ to denote the *suffix* of X of length $n-i+1$, that is $\mathsf{X}[i..n] = \mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[n]$. We will often refer to suffix $\mathsf{X}[i..n]$ simply as "suffix i". Similarly, we write $\mathsf{X}[1..i]$ to denote the *prefix* of X of length i. $\mathsf{X}[i..j]$ is the *substring* $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j]$ of X that starts at position i and ends at position j. By $\mathsf{X}[i..j)$ we denote $\mathsf{X}[i..j-1]$.

The *suffix array* [16] SA_X (we drop subscripts when they are clear from the context) of a string X is an array $\mathsf{SA}[1..n]$ which contains a permutation of the integers $[1..n]$ such that $\mathsf{X}[\mathsf{SA}[1]..n] < \mathsf{X}[\mathsf{SA}[2]..n] < \cdots < \mathsf{X}[\mathsf{SA}[n]..n]$. In other words, $\mathsf{SA}[j] = i$ iff $\mathsf{X}[i..n]$ is the j^{th} suffix of X in ascending lexicographical order. Let $\mathsf{lcp}(i,j)$ denote the length of the longest-common-prefix of suffix i and suffix j. For example, in the string $\mathsf{X} = \mathit{ccccatcat}$, $\mathsf{lcp}(1,4) = 2 = |\mathit{cc}|$, and $\mathsf{lcp}(5,8) = 3 = |\mathit{cat}|$. The *longest-common-prefix (LCP) array* [14], $\mathsf{LCP}_\mathsf{X} = \mathsf{LCP}[1..n]$, is defined such that $\mathsf{LCP}[1] = 0$, and $\mathsf{LCP}[i] = \mathsf{lcp}(\mathsf{SA}[i], \mathsf{SA}[i-1])$ for $i \in [2..n]$.

The *next and previous smaller value* (NSV/PSV) arrays are defined as $\mathsf{NSV}[i] = \min\{j \in [i+1..n] \mid \mathsf{SA}[j] < \mathsf{SA}[i]\}$ and $\mathsf{PSV}[i] = \max\{j \in [1..i-1] \mid \mathsf{SA}[j] < \mathsf{SA}[i]\}$. The array $\mathsf{NPSV}_{\text{text}}[1..n]$ stores the two arrays interleaved and in text order: $\mathsf{NPSV}_{\text{text}}[\mathsf{SA}[i]] = (\mathsf{SA}[\mathsf{NSV}[i]], \mathsf{SA}[\mathsf{PSV}[i]])$.

The *longest previous factor* (LPF) at position i in string X is a pair $\mathsf{LPF}_\mathsf{X}[i] = (p_i, \ell_i)$ such that, $p_i < i$, $\mathsf{X}[p_i..p_i+\ell_i) = \mathsf{X}[i..i+\ell_i)$, and ℓ_i is maximized. In other words, $\mathsf{X}[i..i+\ell_i)$ is the longest prefix of $\mathsf{X}[i..n]$ which also occurs at some position $p_i < i$ in X. There may be more than one potential p_i — in this paper any suffices.

The *LZ77 factorization* (or LZ77 parsing) of a string X is a greedy, left-to-right parsing of X into longest previous factors. More precisely, if the j^{th} LZ factor (or *phrase*) in the parsing is to start at position i, then $\mathsf{LZ}[j] = \mathsf{LPF}[i] = (p_i, \ell_i)$ (to represent the j^{th} phrase), and then the $(j+1)^{\text{th}}$ phrase starts at position $i+\ell_i$. The exception is the case $\ell_i = 0$, which happens iff $\mathsf{X}[i]$ is the leftmost occurrence of a symbol in X. In this case $\mathsf{LZ}[j] = (\mathsf{X}[i], 0)$ (to represent $\mathsf{X}[i..i]$) and the next phrase starts at position $i+1$. When $\ell_i > 0$, substring $\mathsf{X}[p_i..p_i+\ell_i)$ is called the *source* of phrase $\mathsf{X}[i..i+\ell_i)$. We denote the number of phrases in the LZ77 parsing of X by z.

Given two strings Y and Z, the matching statistics of Y w.r.t. Z, denoted $\mathsf{MS}_{\mathsf{Y}|\mathsf{Z}}$, is an array of $|\mathsf{Y}|$ pairs, (p_1,ℓ_1), (p_2,ℓ_2), ..., $(p_{|\mathsf{Y}|},\ell_{|\mathsf{Y}|})$, such that for all $i \in [1..|\mathsf{Y}|]$, $\mathsf{Y}[i..i+\ell_i) = \mathsf{Z}[p_i..p_i+\ell_i)$ is the longest substring starting at position i in Y that is also a substring of Z. Note the subtle link to the LPF array.

3 EM-LPF

Our first external memory LZ factorization algorithm is EM-LPF, an external memory variant of the in-memory algorithm by Crochemore, Ilie and Smyth [4]. The algorithm has three main phases:

1. Compute SA and LCP from X.
2. Compute LPF from SA and LCP.
3. Compute LZ from LPF.

The first phase, computing the suffix and LCP arrays, is a much studied problem, and could be computed with any algorithm. We use the eSAIS [3] implementation by Timo Bingmann[1], which is the fastest implementation we know of capable of computing both arrays for arbitrary inputs larger than RAM.

```
LPF-from-SA-and-LCP(SA[1..n], LCP[1..n])
1:  S ← empty stack
2:  for i ← 1 to n + 1 do
3:      if i = n + 1 then (j, ℓ) ← (0, 0)
4:      else (j, ℓ) ← (SA[i], LCP[i])
5:      while S ≠ ∅ do
6:          (j_s, ℓ_s) ← top(S)
7:          if j < j_s then
8:              pop(S)
9:              if ℓ < ℓ_s then
10:                 (j', ·) ← top(S)
11:                 LPF[j_s] ← (j', ℓ_s)
12:             else
13:                 LPF[j_s] ← (j, ℓ)
14:                 ℓ ← ℓ_s
15:         elsif j > j_s and ℓ_s ≥ ℓ then
16:             pop(S)
17:             if S = ∅ then j' ← 0
18:             else (j', ·) ← top(S)
19:             LPF[j_s] ← (j', ℓ_s)
20:         else break
21:     push(S, (j, ℓ))
22: return LPF[1..n]
```

```
LZ-from-LPF(LPF[1..n], X[1..n])
1:  i ← 1; z ← 0
2:  while i ≤ n do
3:      (p, ℓ) ← LPF[i]
4:      if ℓ = 0 then p ← X[i]
5:      z ← z + 1
6:      LZ[z] ← (p, ℓ)
7:      i ← i + max{ℓ, 1}
8:  return LZ[1..z]
```

Figure 1: The second (left) and the third (right) phase of the EM-LPF algorithm.

Pseudocode for the second and third phases is shown in Figure 1. The logic of the pseudocode is identical to the original description [4]. Two changes are notable, however. First, our pseudocode computes both components of LPF and LZ, not just the length component, as in [4]. Second, in order to avoid non-sequential accesses to SA and LCP, we do not store pointers into those arrays on the stack, but rather actual element values.

For clarity, the pseudocode describes the in-memory algorithm, but it is easy to see that most accesses to large arrays are purely sequential and thus easily implemented via external memory scan operations. The only non-sequential operations are the writes to LPF in the second phase. The external memory version of the algorithm replaces the assignment $\mathsf{LPF}[j_s] \leftarrow (j, \ell)$ by outputting the tuple (j_s, j, ℓ). The LPF array can then be obtained by sorting the output sequence by the first component. Sorting is accomplished using the external memory STXXL library [6].

[1] `http://tbingmann.de/2012/esais/`

The total I/O complexity is the sorting complexity $O((n/B)\log_{M/B}(n/B)))$, where M is the size of RAM and B is the disk block size, both in units of $\Theta(\log n)$-bit words. The algorithm could be implemented to run in just $O(B)$ words of RAM, but the actual implementation uses a bit more, dominated by eSAIS (the in-memory RMQ version, see [3]). The main bottleneck in the practical scalability of the algorithm, however, is likely to be the disk space requirement of eSAIS, which is reported as $54n$ in [3] but we found it to be significantly more in practice, see Section 6.

4 EM-LZscan

Our second algorithm, called EM-LZscan, is an external memory version of the scanning-based, block-oriented LZ77 factorization algorithm LZscan that was introduced in [12].

Conceptually EM-LZscan divides X up into at most $d = \lceil n/b \rceil$ fixed size blocks of length $b = \Theta(M)$: $\mathsf{X}[1..b]$, $\mathsf{X}[b+1..2b]$, ... In the description that follows we will refer to the block currently under consideration as Z, and to the prefix of X that ends just before Z as Y. Thus, if $\mathsf{Z} = \mathsf{X}[kb+1..(k+1)b]$, then $\mathsf{Y} = \mathsf{X}[1..kb]$. To begin, we will assume no LZ factor crosses the boundary of Z (an assumption we will later remove). The outline of the algorithm for processing a block Z is shown below.

1. Compute $\mathsf{MS}_{\mathsf{Y}|\mathsf{Z}}$
2. Compute $\mathsf{MS}_{\mathsf{Z}|\mathsf{Y}}$ from $\mathsf{MS}_{\mathsf{Y}|\mathsf{Z}}$, SA_{Z} and $\mathsf{LCP}_{\mathsf{Z}}$
3. Compute $\mathsf{LPF}_{\mathsf{YZ}}[kb+1..(k+1)b]$ from $\mathsf{MS}_{\mathsf{Z}|\mathsf{Y}}$ and $\mathsf{LPF}_{\mathsf{Z}}$
4. Compute LZ from $\mathsf{LPF}_{\mathsf{YZ}}[kb+1..(k+1)b]$

Step 1: Computing Matching Statistics. Similar to most algorithms for computing the matching statistics, we first construct some data structures on Z and then scan Y. The main difference to the in-memory algorithm is that Y is not held in RAM but read from disk. For the details of the data structures we refer to [12]. The key properties are linear time construction and space requirement of $29b$ bytes. The scanning of Y is the computational bottleneck of the algorithm in theory and practice. The I/O complexity is $O(|\mathsf{Y}|/(B\log_\sigma n))$ ($\log_\sigma n$ is the number of characters that fit in one machine word) and the time complexity is $O(|\mathsf{Y}|\sigma)$ (where σ, the time for the rank operation, can be reduced to $O(\log(2+(\log\sigma/\log\log n)))$ in theory [2]).

An important optimization, called the skipping trick, speeds up the computation for highly repetitive inputs [12]. It takes advantage of repetition present in Y that was found in the previous stages of the algorithm. Consider an LZ factor $\mathsf{Y}[i..i+\ell)$. Because, by definition, $\mathsf{Y}[i..i+\ell)$ occurs earlier in Y too, any source of an LZ factor of Z that is completely inside $\mathsf{Y}[i..i+\ell)$ could be replaced with an equivalent source in that earlier occurrence. Thus such factors can be skipped during the computation of $\mathsf{MS}_{\mathsf{Y}|\mathsf{Z}}$ without affecting the correctness of the factorization. To implement the skipping trick, we need to know the positions of phrase boundaries. We store the boundaries on disk and scan them in synchrony with Y. This does not affect the I/O complexity of the algorithm as $z = O(n/\log_\sigma n)$. As a practical optimization, we skip, and scan the boundaries of, only phrases of length 40 or more.

Step 2: Inverting Matching Statistics. With the help of $\mathsf{SA_Z}$ and $\mathsf{LCP_Z}$, constructed in RAM, we can *invert* $\mathsf{MS_{Y|Z}}$ to obtain $\mathsf{MS_{Z|Y}}$, which is what we need for LZ77 factorization. Note that a direct computation of $\mathsf{MS_{Z|Y}}$ using the techniques from Step 1 would need $\mathrm{O}(|\mathsf{Y}|)$ words of RAM, which we cannot afford. Again, we refer to [12] for details of the inversion algorithm and give only the key properties. The algorithm accesses each entry of $\mathsf{MS_{Y|Z}}$ (except those skipped by the skipping trick) once, in an arbitrary order, and processes the entry in constant time. Thus we do not need to store $\mathsf{MS_{Y|Z}}$ but can process each entry as soon as it is produced in Step 1. The rest of the computation takes $\mathrm{O}(b)$ time and no I/Os.

Step 3: Computing LPF. There is no I/O involved in this step as all the data structures of this step fit in $\mathrm{O}(b)$ space. We refer to [12] for details.

Step 4: Parsing. The block Z is parsed in RAM using the standard LPF-based parsing (see Fig. 1) but with $\mathsf{LPF_Z}$ replaced by $\mathsf{LPF_{YZ}}[kb+1..(k+1)b]$. This produces the correct global factorization except for the last phrase because, as described below, the beginning of a block is always aligned with the beginning of a phrase.

The last phrase of Z produced by the parsing, call it P, could extend beyond the end of Z in the global parsing of X. If $|\mathsf{P}| \leq b/2$, we start the next block at the beginning of P, and compute the true phrase while processing that block. This can increase the number of blocks, but not more than by a factor of two.

If $|\mathsf{P}| > b/2$, we run a separate procedure to determine that one phrase, and then start the next block at the end of that phrase. The procedure involves scanning X from the beginning up to the end of the phrase using a modified string matching algorithm by Crochemore [5, 11]. If the length ℓ of the phrase is very long, the procedure may involve up to $\mathrm{O}(1 + \ell/M)$ scans of X. This does not change the time or I/O complexity of the algorithm as each extra scan moves the start of the next block $\Omega(M) = \Omega(b)$ steps forward.

Complexity. The total CPU time complexity of EM-LZscan is $\mathrm{O}(n^2\sigma/M)$ and the I/O complexity is $\mathrm{O}(\frac{n \log \sigma}{B \log n} \cdot \frac{n}{M})$. For highly repetitive inputs, the skipping trick can dramatically reduce both the time and I/O complexities, as our experiments show.

5 SE-KKP

Our third algorithm is not fully external but semi-external: it needs to have the whole input text in RAM at one stage of the computation. It is based on the KKP3 algorithm [13] and has three phases:

1. Compute SA from X.
2. Compute $\mathsf{NPSV_{text}}$ from SA.
3. Compute LZ from $\mathsf{NPSV_{text}}$ and X.

The array $\mathsf{NPSV_{text}}$ (just NPSV from now on) has a very similar intermediary role in SE-KKP as LPF has in EM-LPF. The main differences between the algorithms are that SE-KKP does not need to compute LCP in the first phase but needs random access to the text X during the third phase.

In the first phase, we need to compute SA but not LCP and we know that the size of the input text does not exceed the RAM size. Thus, we do not need the LCP computation and the full scalability of eSAIS. On the other hand, we want to be able to process input sizes very close to the RAM size, which excludes any in-memory SA construction algorithm. The best match to our requirements is the FGM algorithm [8] which has $O(n^2/M)$ time complexity and I/O volume, but fairly small constant factors. FGM is originally designed and implemented to compute the Burrows–Wheeler transform (BWT), but can be used for computing SA too [8, Th. 3]. We have modified the BWT implementation by Giovanni Manzini[2] to do this.

```
NPSV-from-SA(SA[1..n])
1:  S ← {0}
2:  for i ← 1 to n + 1 do
3:      if i = n + 1 then j ← 0
4:      else j ← SA[i]
5:      j_s ← top(S)
6:      while j_s > j do
7:          pop(S)
8:          NPSV[j_s] ← (j, top(S))
9:          j_s ← top(S)
10:     push(S, j)
11: return NPSV[1..n]
```

```
LZ-from-NPSV(NPSV[1..n], X[1..n])
1:  i ← 1; z ← 0
2:  while i ≤ n do
3:      (nsv, psv) ← NPSV[i]
4:      ℓ ← lcp(nsv, psv)
5:      if X[i + ℓ] = X[nsv + ℓ] then p ← nsv
6:      else p ← psv
7:      ℓ ← ℓ + lcp(i + ℓ, p + ℓ)
8:      if ℓ = 0 then p ← X[i]
9:      z ← z + 1
10:     LZ[z] ← (p, ℓ)
11:     i ← i + max{ℓ, 1}
12: return LZ[1..z]
```

Figure 2: The second (left) and the third (right) phase of the SE-KKP algorithm. The procedure $\mathsf{lcp}(\cdot,\cdot)$ computes lcp values using brute force character comparisons.

The pseudocode for the second and third phase is given in Figure 2. It is essentially the same as the original description [13, KKP3], to which we refer for the detailed explanation. The main difference is that we use an explicit stack instead of overwriting the suffix array with the stack. Similar to EM-LPF the only non-sequential operations in the second phase are write operations, to NPSV in this case, and the solution is the same too: external memory sorting between the second and third phases. The lcp computation on line 4 of LZ-from-NPSV requires the string X to remain in RAM as the positions nsv and psv can be effectively random.

The algorithm requires the RAM to be large enough to hold the string X. Recall M is the size of RAM in terms of $\Theta(\log n)$-bit integers, while the size of X is $O(n \log \sigma)$ bits. Thus, when the algorithm is usable, we can assume that $M = \Omega(n/\log_\sigma n)$. The time complexity of the algorithm is then $O(n \log_\sigma n)$ and the I/O complexity is $O(n \log_\sigma n/B)$, both dominated by the FGM algorithm.

6 Experimental Results

We implemented the three algorithms described in this paper and performed experiments to find the limits of their scalability in practice. Also included is LZscan [12], the best of prior algorithms under limited RAM.

[2] `http://people.unipmn.it/manzini/bwtdisk/index.html`

Table 1: Files used in the experiments. The value of n/z (the average length of a phrase in the LZ factorization) is included as a measure of repetitiveness.

Name	n	σ	n/z	Description
countries	10 GiB	203	2871	Wikipedia version database[a]
cere	10 GiB	5	3230	Set of Yeast genomes[b]
enwik	6 GiB	209	19.30	English Wikipedia XML[c]
hg	5.85 GiB	31	18.44	2 Human genomes[d,e]

[a] `http://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)`
[b] `http://pizzachili.dcc.uchile.cl/repcorpus.html`
[c] `http://dumps.wikimedia.org/enwiki/`
[d] `http://hgdownload.soe.ucsc.edu/goldenPath/hg19/chromosomes/`
[e] `ftp://public.genomics.org.cn/BGI/yanhuang/fa/`

Experimental Setup. In our experiments we use various prefixes of the files listed in Table 1. The first two files are highly repetitive, which is the forte of LZ-based compression and compressed indexes, and the other two represent "normal" files.

We performed experiments on a machine equipped with a 3.16GHz Intel Core 2 Duo CPU with 6144KiB L2 cache, 4GiB of main memory, and two 320GiB hard drives. Only a single thread of execution was used. The OS was Linux (Ubuntu 12.04, 64bit) running kernel 3.2.0. The compiler was `g++` version 4.6.4 with `-O3 -static -DNDEBUG` options. All reported runtimes are wallclock (real) times. The implementations are available at `http://www.cs.helsinki.fi/group/pads/`.

Experiment 1. First we studied the scalability of the two proper external memory algorithms, EM-LPF and EM-LZscan. Scalability of EM-LPF is primarily limited by the disk space requirement of eSAIS. The test machine was equipped with two 320GiB disks with 480GiB of effectively usable disk space. The reported peak disk usage of eSAIS is $54n$ [3] implying a limit of 8.8GiB for the maximum input text that could be processed with EM-LPF. However, we were able to process texts only up to 6GiB[3].

In contrast, EM-LZscan needs little extra disk space, but slows down rapidly as the input grows much bigger than available RAM. To expose this behaviour within the 6GiB input size limit of EM-LPF, we restricted (with the Linux boot option `mem`) the physical amount of RAM in our test machine to 2GiB and designated exactly 1.5GiB to be used by each algorithm. For EM-LPF this is not a serious restriction.

The results are given in Fig. 3. The results for the two non-repetitive files (enwik and hg) show the quadratic time complexity of EM-LZscan, while EM-LPF scales nicely until it runs out of disk space. For the highly repetitive files, EM-LZscan is much faster because of the skipping trick, and would handle even larger inputs effectively. On the same machine but without the manual RAM limitation, we factorized a 40.5GiB version of the countries file using EM-LZscan in 2.3 days.

Experiment 2. In the second set of experiments, we study LZ factorization when the input text fits into the main memory but is too big to use standard techniques that require at least $6n$ bytes of RAM [13]. The main competitors in this context are

[3]For larger files, we get an STXXL error message "External memory block allocation error".

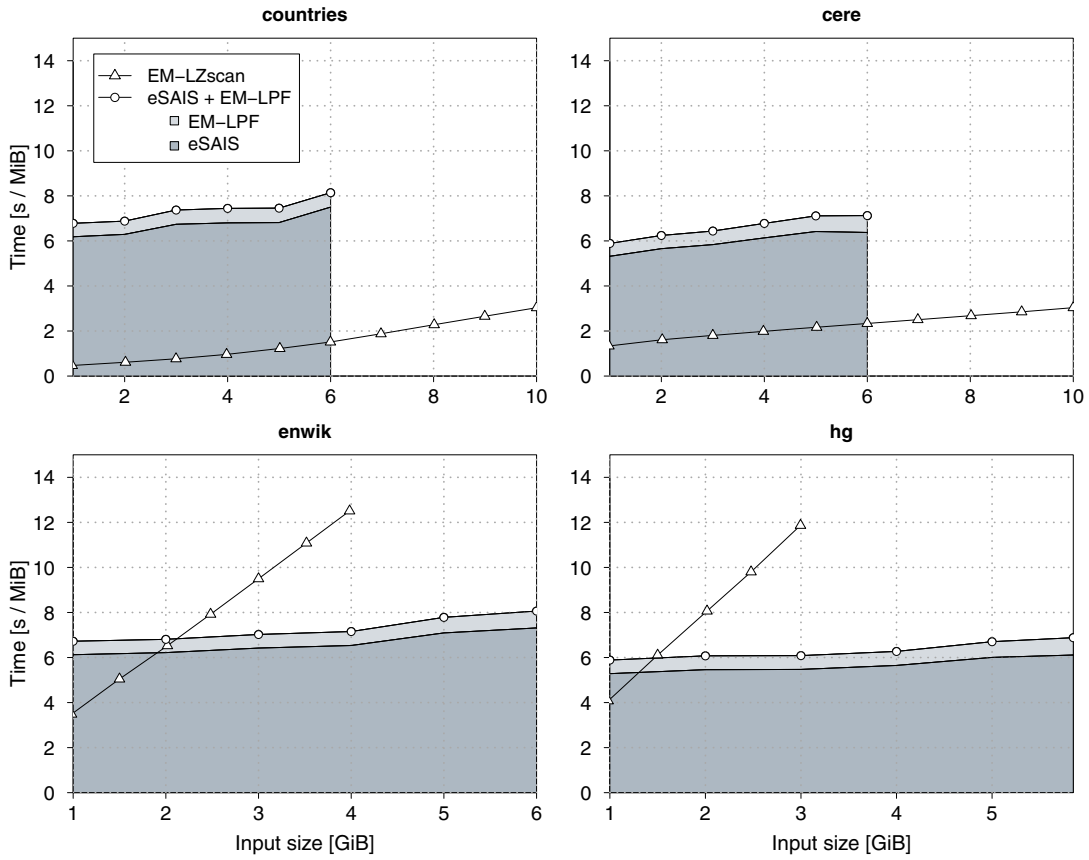

Figure 3: Comparison of external memory LZ77 parsing algorithms. For EM-LPF we separate the total factorization time into two components: time spent for precomputing SA and LCP arrays using eSAIS (dark grey), and the time for computing the actual LZ77 parsing using EM-LPF (light grey).

EM-LZscan and SE-KKP as well as LZscan [12]. We restricted the RAM used by the algorithms to at most 3GiB.

The results are given in Fig. 4. For non-repetitive files, SE-KKP outperforms its competitors. EM-LZscan is faster than its in-memory version, because not having to keep the text in memory frees RAM for other data structures. For highly repetitive data, LZscan and EM-LZscan are again much faster because of the skipping trick.

Discussion. For highly repetitive files, EM-LZscan is the best algorithm through the range of our experiments. For non-repetitive data, it is beaten by SE-KKP for smaller files and by EM-LPF for large files. Thus all three of our algorithms are the best under some conditions.

Our test machine is quite modest by modern standards and machines with much more RAM and disk space are commonly available. An interesting question is which of the two proper external memory algorithms scales better with improved hardware. Currently, the price of 1TiB of disk space is roughly the same as the price of 16GiB of RAM. An extra 1TiB of disk would allow eSAIS/EM-LPF to process about 16GiB

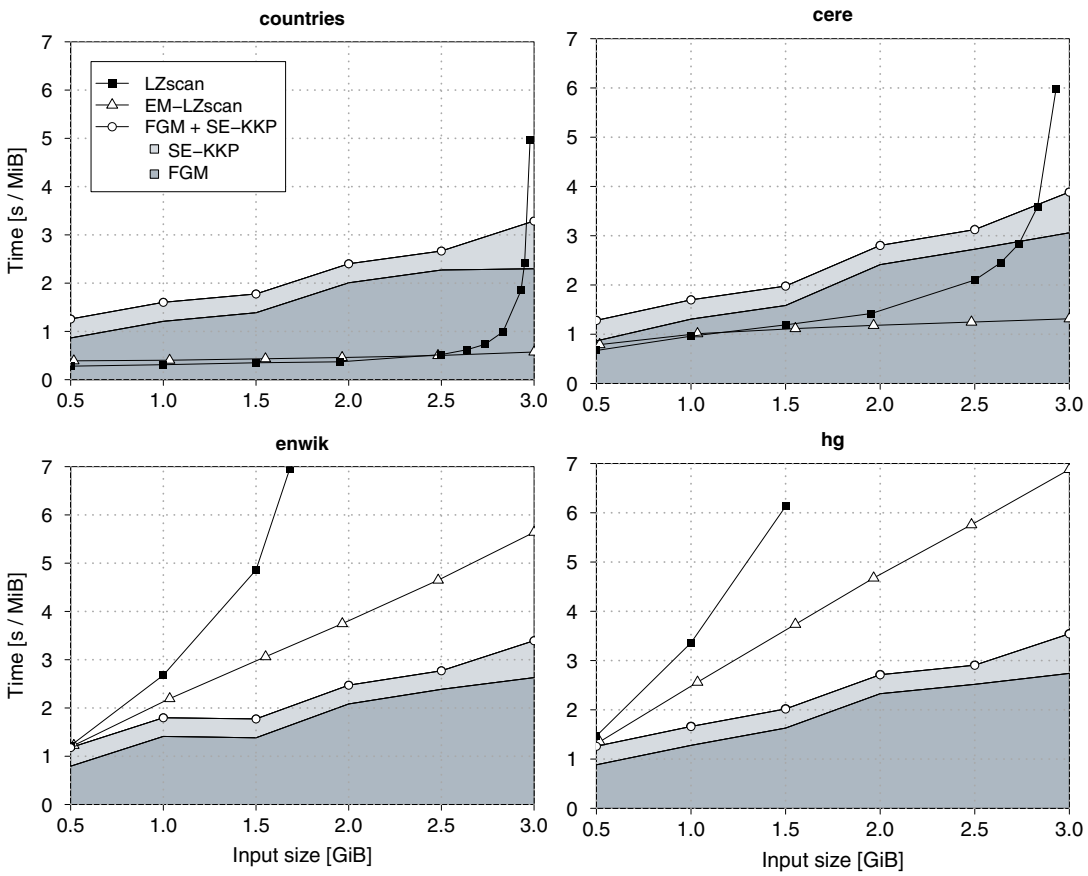

Figure 4: Comparison of semi-external LZ77 parsing algorithms. For SE-KKP we separately show the time to precompute SA using FGM (dark grey), and the actual LZ77 parsing time using SE-KKP (light grey).

larger inputs, while an extra 16GiB of RAM would extend the scalability of EM-LZscan by about the same for non-repetitive data and much more for highly repetitive data. Furthermore, the limit of scalability of EM-LZscan is soft: larger files can be factorized given more time. Thus there are reasons to call EM-LZscan more scalable.

7 Concluding Remarks

We have described three very different methods for LZ factorization in external memory and have shown that in practice the method of choice depends on the size and type of input as well as the RAM size and disk space of the system.

A different path to making LZ77 factorization scalable is to distribute or parallelize the computation. Shun and Zhao [17] recently studied LZ factorization in the multicore setting. Of the three algorithms we describe, the work of EM-LZscan seems most amenable to distribution (each node processes a different block, or set of adjacent blocks), whereas the distributability of EM-LPF and SE-KKP will depend largely on that of the SA construction algorithm used.

References

[1] Al-Hafeedh, A., Crochemore, M., Ilie, L., Kopylova, E., Smyth, W., Tischler, G., Yusufu, M.: A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. ACM Comput. Surv. 45(1), 5:1–5:17 (2012)

[2] Belazzougui, D., Navarro, G.: New lower and upper bounds for representing sequences. In: Proc. ESA, pp. 181–192 (2012)

[3] Bingmann, T., Fischer, J., Osipov, V.: Inducing suffix and LCP arrays in external memory. In: Proc. ALENEX. pp. 103–112 (2013)

[4] Crochemore, M., Ilie, L., Smyth, W.F.: A simple algorithm for computing the Lempel-Ziv factorization. In: Proc. DCC. pp. 482–488 (2008)

[5] Crochemore, M.: String-matching on ordered alphabets. Theor. Comput. Sci. 92, 33–47 (1992)

[6] Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Softw., Pract. Exper. 38(6), 589–637 (2008)

[7] Ferragina, P., Manzini, G.: On compressing the textual web. In: Proc. WSDM. pp. 391–400 (2010)

[8] Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica 63(3), 707–730 (2012)

[9] Gagie, T., Gawrychowski, P., Puglisi, S.J.: Faster approximate pattern matching in compressed repetitive texts. In: Proc. ISAAC. pp. 653–662 (2011)

[10] Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: A faster grammar-based self-index. In: Proc. LATA. pp. 240–251 (2012)

[11] Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Crochemore's string matching algorithm: Simplification, extensions, applications. In: Proc. PSC. pp. 168–175 (2013)

[12] Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lightweight Lempel-Ziv parsing. In: Proc. SEA. pp. 139–150 (2013)

[13] Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Linear time Lempel–Ziv factorization: Simple, fast, small. In: Proc. CPM. pp. 189–200 (2013)

[14] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. CPM. pp. 181–192 (2001)

[15] Kreft, S., Navarro, G.: Self-indexing based on LZ77. In: Proc. CPM. pp. 41–54 (2011)

[16] Manber, U., Myers, G.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)

[17] Shun, J., Zhao, F.: Practical parallel Lempel-Ziv factorization. In: Proc. DCC. pp. 123–132 (2013)

[18] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory 23(3), 337–343 (1977)