Bachelor Thesis

Parallel Multiway LCP-Mergesort

Andreas Eberle

Published: 2014/05/15

Supervisor: Prof. Dr. Peter Sanders
Dipl.-Inform. Timo Bingmann

Institute of Theoretical Informatics, Algorithmics II
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

Abstract

In this bachelor thesis, multiway LCP-Merge is introduced, parallelized and applied to create a fully parallel LCP-Mergesort, as well as NUMA optimized pS^5. As an advancement of binary LCP-Mergesort, a multiway LCP-aware tournament tree is introduced and parallelized. For dynamic load balancing, one well-known and two new strategies for splitting merge work packages are utilised. Besides the introduction of fully parallel multiway LCP-Mergesort, further focus is put on NUMA architectures. Thus 'parallel Super Scalar String Sample Sort' (pS^5) is adapted to the special properties of these systems by utilising the parallel LCP-Merge. Moreover this yields an efficient and generic approach for parallelizing arbitrary sequential string sorting algorithms and making parallel algorithms NUMA-aware. Several optimizations, important for practical implementations, as well as comprehensive experiments on two current NUMA platforms, are then reported and discussed. The experiments show the good scalability of the introduced algorithms and especially, the great improvements of NUMA-aware pS^5 with real-world input sets on modern machines.

Zusammenfassung

In dieser Bachelorarbeit wird ein mehrwegiger LCP-Merge eingeführt, parallelisiert und für den Aufbau eines parallelen LCP-Mergesorts, sowie einer NUMA optimierten pS^5 Implementierung, angewandt. Als Weiterentwicklung des binären LCP-Mergesortes, wird ein mehrwegiger LCP-fähiger Tournament Tree eingeführt und parallelisiert. Zur Aufteilung der Arbeitspakete, welche für eine dynamische Lastverteilung benötigt wird, werden eine bekannte, sowie zwei neu eingeführte Strategien, genutzt. Neben der Einführung eines parallelisierten LCP-Mergesortes, wird der weitere Fokus auf NUMA Architekturen gelegt. Im Zuge dessen, wird 'parallel Super Scalar String Sample Sort' (pS^5), durch Anwendung des parallelen LCP-Merges, auf die besonderen Eigenschaften dieser Systeme angepasst. Zusätzlich führt dies zu einem effizienten und generischen Ansatz um sequentielle Sortieralgorithmen zu parallelisieren und bereits parallele Algorithmen um NUMA-Fähigkeit zu erweitern. Weiterhin werden einige Optimierungen, welche für praktische Implementierungen wichtig sind, sowie ausgiebige Experimente auf zwei aktuellen NUMA Plattformen, erläutert und diskutiert. Die Experimente belegen mit realistischen Eingabedaten die gute Skalierbarkeit der vorgestellten Algorithmen und besonders die enormen Verbesserungen des pS^5 Algorithmus auf NUMA Systemen.

Acknowledgement

My thanks goes to Prof. Dr. Peter Sanders and my advisor Timo Bingmann for giving me the opportunity to work on the interesting subject of sorting strings in parallel. Not only is it a subject of growing importance, but also an interesting algorithmic problem to solve.

Many thanks go to Valentin Zickner for all the coffee drinking sessions and the many important discussions, on and off-topic, as well as for many advices regarding LaTeX.

Moreover, I want to thank Valentin, but also Katja Leppert, Joachim Lusiardi and Katharina Huber for proofreading this thesis and giving me valuable input to improve it further.

I would also like to give thanks to my parents and brothers for all their support. Especially, I want to thank my older brother Christian Eberle, who has not only been a great inspiration through out all my life, but also was the one introducing me to the world of computer programming.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

With the digital age, more and much larger amounts of data arise. Structuring, evaluating and analysing this volume of data is a task of growing importance and difficulty. However, the basic algorithms needed to do this, have been known and used for years. With many of them requiring sorting data and merging results, it is quite comprehensible that sorting is one of the most studied algorithmic problems in computer science but nonetheless still of great interest.

Although the simplest sorting model assumes *atomic* keys, sorting strings lexicographically and merging sorted sequences of strings is required by many algorithms important for today's applications. Examples relying on string sorting range from MapRedcue tools and databases over some suffix sorters to BigData analysis tools and much more. In contrast to atomic keys, strings can be seen as arrays of atomic keys, which leads to a larger computational complexity for string sorting. This is why it is very important to exploit the structure of keys to avoid repeated costly work on entire strings.

Even though there is a large amount of work on sequential string sorting, only little work has been done to parallelize it. But as nowadays the only way to gain wins from Moore's law, is to use parallelism, all performance critical algorithms need to be parallelized. However, with first parallel sorting algorithms available, new challenges arise. As the amount of available memory on modern many-core systems grows, *non uniform memory access* (NUMA) architectures become more common. Curiously, although increased main memory sizes reduce the need for external sorting algorithms on the one hand, NUMA systems induce varying main memory access times, thus making it necessary to apply external sorting algorithm schemes to in-memory implementations.

As a result, it is much more important to maximize efficiency of memory accesses on NUMA systems. Exploiting known *longest common prefixes* (LCPs) when merging strings, can be used to skip over already considered parts of them, which reduces memory accesses. Merging sequences of strings with their according LCP information is an intuitive idea and Ng and Katsuhiko [NK08] already introduced a binary LCP-aware merge sort but no multiway implementation was found. However, as our NUMA systems currently have two, four and eight NUMA nodes, this is required to prevent unnecessary memory operations.

Moreover, an efficient multiway LCP-aware merge allows to improve current sequential and parallel merge sort implementations, possibly making them competitive to currently faster algorithms. Especially for input sets with long average LCPs, this implementation could outperform others.

1.1. Contributions of this Bachelor Thesis

As the first step of this work, *LCP-Mergesort*, initially presented by Ng [NK08], will be redefined to improve comprehensibility of the newly presented algorithms based on it. As Ng only showed an average case analysis, the worst case computational complexity of LCP-Mergesort will be analysed.

With the goal to create a fully parallel LCP-aware merge sort implementation, Ng's binary LCP-Merge algorithm is extended and a K-way LCP-aware tournament tree introduced. This tournament tree is independently usable for merging K sorted sequences of strings with associated LCP information. Furthermore, a parallel K-way LCP-Merge and the resulting fully parallel K-way LCP-Mergesort is presented. Additionally, a com-

mon algorithm for splitting the merge problem is adapted and a completely new one presented.

Since we want to improve practical applications, it is of great importance to consider real hardware architectures and optimizations required by them. Additionally it is important that these algorithms do not just achieve good theoretical results, but can really improve practical runtimes. Therefore we implemented our newly presented parallel LCP-Merge and LCP-Mergesort with three different splitting procedures. Furthermore, the parallel sorting algorithm pS^5 of Timo Bingmann [BS13] will be improved for NUMA architectures by exploiting the properties of K-way LCP-Merge.

In order to evaluate the presented algorithms, they will be compared with existing parallel string sorting implementations like original pS^5. To allow examination of the degree of parallelism, not just runtimes but also speed ups of different algorithms are reviewed.

1.2. Structure of this Bachelor Thesis

Section 2 gives an overview of used notations and existing algorithms. Whereas Ng's LCP-Mergesort is the basis for this work, Bingmann's *Parallel Super Scalar String Sample Sort* is a reference as one of the fastest parallel string sorters.

In Section 3 binary LCP-Mergesort is redefined and multiway LCP-Merge, as well as multiway LCP-Mergesort are introduced. Moreover, a proof of the upper bound of binary LCP-Mergesort's runtime is provided.

Furthermore, Section 4 focuses on implementation details of the newly presented algorithms in order to improve their practical performance even further.

The performance of the resulting C++ implementations is evaluated in Section 5, where speed up factors and runtimes of various variants and algorithms are compared.

Finally, a summation of the results and an outlook to future work, is given in Section 6.

2. Preliminaries

A set $S = \{s_1, ..., s_n\}$ of n strings of total length $N = \sum_{i=1}^{n} |s_i|$ is our input. A string s is a one-base array of characters from the *alphabet* $\Sigma = \{1, ..., \sigma\}$. The length of a string s or any arbitrary array, is given by $|s|$ and the i^{th} element of an array a is accessed via $a[i]$. On the alphabet Σ we assume the canonical ordering relation '$<$' with $1 < 2 < ... < \sigma$. Likewise for strings we assume the lexicographical ordering relation '$<$' and our goal is to sort the strings of the given input sequence S lexicographically. For indicating the end of strings, our algorithms require strings to be *zero-terminated*, meaning $s[|s|] = 0 \notin \Sigma$, which however can be replaced by any other end-of-string convention.

With the *length of the distinguishing prefix* D, denoting the minimum number of characters to be inspected to establish lexicographic ordering of S, there is a natural lower bound for string sorting. More precisely, for sorting based on character comparisons, we get the lower bound of $\Omega(D + n \log n)$, whereas string sorting based on an integer alphabet can be achieved in $\Omega(D)$ time.

Because sets of strings are usually represented as arrays of pointers to the beginning of the string, there is an additional indirection when accessing a string character. This generally causes a cache fault on every string access, even during linear scanning of an array of strings. Therefore a major difference of string sorting in comparison to atomic sorting, is the lack of efficient scanning.

Our algorithms are targeted for shared memory systems supporting p processing elements or hardware threads on $\Theta(p)$ cores. Additionally some algorithms and optimizations are specially targeted for *non uniform memory access* (NUMA) systems, also providing p hardware threads on $\Theta(p)$ cores. However, the p hardware threads are equally divided onto m NUMA nodes, each having fast direct access to local memory and slower access to remote memory via an interconnect bus system. Due to the NUMA architecture, costs of memory accesses across NUMA nodes are much higher and therefore need to be avoided.

Figure 1 illustrates a NUMA architecture with $m = 4$ NUMA nodes and $p = 16$ cores. Whereas the cores p0, p4, p8 and p12, belonging to NUMA node 0, have fast access to local *Memory 0*, remote access to the memories of nodes 1, 2 and 3 is much slower.

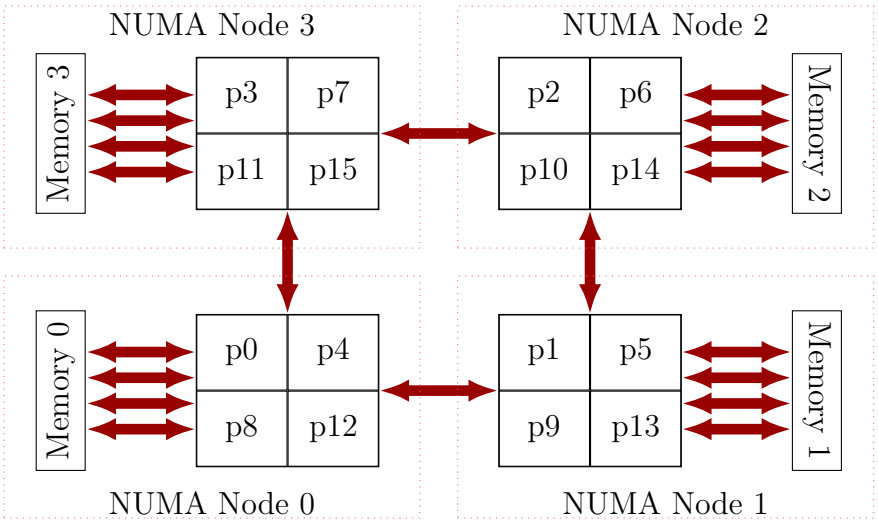


Figure 1: NUMA architecture with $m = 4$ NUMA nodes and $p = 16$ cores.

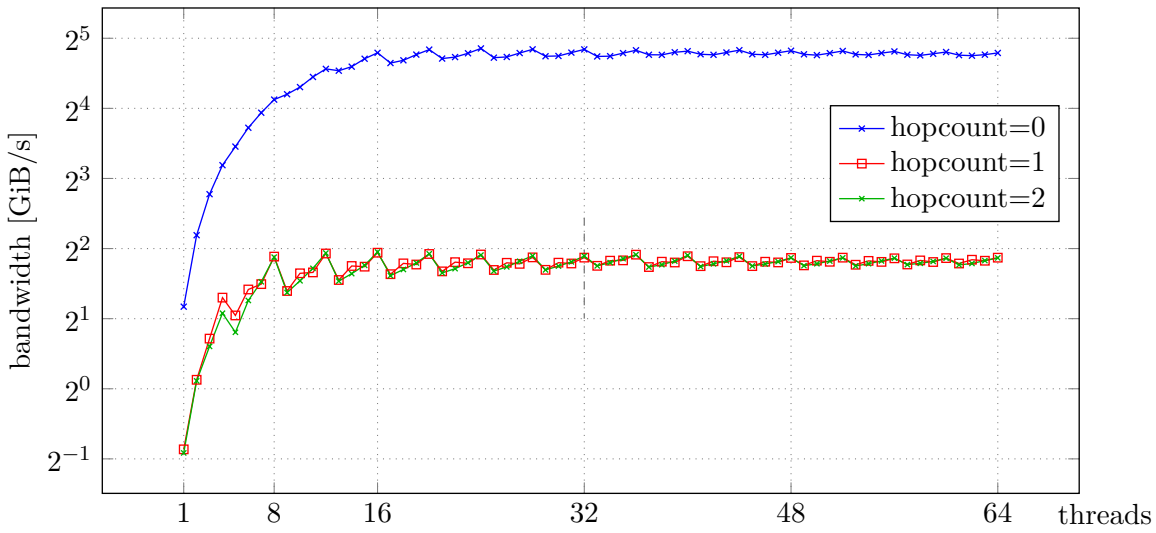


Figure 2: Memory bandwidth for accessing NUMA memory on IntelE5. See Table 1 on page 45 for the exact hardware specification.

This behaviour can be examined in Figure 2, showing the memory bandwidth achieved by the given number of threads, when linearly reading 64 bit values from a memory area, which is equally segmented onto all NUMA nodes. The curves show the memory bandwidth over the available threads when only accessing the memory on the NUMA node that is exactly *hopcount* steps away. Therefore a thread running on NUMA node n will solely write to the memory of node $(n + hopcount) \mod m$. The figure clearly shows the tremendous gap in bandwidth between accessing the local NUMA memory ($hopcount = 0$) and accessing the other node's memories ($hopcount = 1$ or $hopcount = 2$). Since sorting mostly requires read operations, the performance of write operations isn't displayed here. However, for write operations, a further slowdown is experienced, when reading from the memory positioned farthermost away ($hopcount = 2$) in comparison to reading from a direct neighbour node.

More information on `pmbw`, the tool used for creating the measurements of Figure 2, can be found at `http://panthema.net/2013/pmbw/`. For these tests, the NUMA branch of `pmbw` has been used to test the performance of the function `ScanRead64PtrSimpleLoop`.

2.1. Notation and Pseudo-code

To describe the algorithms presented in this paper, we chose a tuple pseudo-code language, combining array manipulation, mathematical set notation and Pascal-like control flow. Ordered sequences are written like arrays using square brackets $[x, y, ...]$ and '+' is extended to also concatenate arrays. Neither arrays nor variables are declared beforehand, so $A[3] := 4$ defines an array A and assigns 4 to the third position, as array indexes are counted from 1 to $|A|$, being the length of the array. An example for powerful expressions possible with this pseudo-code language is the following definition: $A := [(k, \exp(i * \frac{k*\pi}{2}))|k \in \{0, 1, 2, 3\}]$, specifying A to be an array of the pairs $[(0, 1), (1, i), (2, -1), (3, -i)]$.

In order to avoid many special cases, we use the following sentinels: 'ϵ' is the empty string, being lexicographically smaller than any other string, '∞' is the character or string, which is larger than any other, and '$\perp$' as symbol for undefined variables.

Furthermore, for arrays s and t, let the symmetric function $\text{lcp}(s, t)$ denote the length of

S	s_1	s_2	s_3	s_4	$\cdots$	s_n
H	$\perp$	$\text{lcp}(s_1, s_2)$	$\text{lcp}(s_2, s_3)$	$\text{lcp}(s_3, s_4)$	$\cdots$	$\text{lcp}(s_{n-1}, s_n)$

(a) Structural view

S	`aab`	`aacd`	`aacd`	`bac`	`bacd`	`bbac`
H	$\perp$	2	4	0	3	1

(b) Exemplary configuration

Figure 3: Structure of string sequence S with associated LCP array H.

the *longest common prefix* (LCP) of s and t. Thus, for one-based arrays, the LCP value denotes the last index where s and t equal each other, whereas at index $\text{lcp}(s,t)+1$, s and t differ, if that position exists. Based on that, $\text{lcp}_X(i)$ is defined to be $\text{lcp}(X[i-1], X[i])$ for an ordered sequence X. Accordingly, the *associated LCP array* $H = [\perp, h_2, ..., h_n]$ of a sorted string sequence $S = [s_1, ..., s_n]$ is defined as $h_i = \text{lcp}_S(i) = \text{lcp}(S[i-1], S[i])$. Additionally, for any string s, we define $\text{lcp}(\epsilon, s) = 0$ to be the LCP to the empty string ϵ.

Figure 3a shows the structure of a string sequence and how its corresponding LCP array is calculated. Furthermore Figure 3b illustrates the LCP array for the example string sequence $S = [\texttt{aab}, \texttt{aacd}, \texttt{aacd}, \texttt{bac}, \texttt{bacd}, \texttt{bbac}]$.

As the sum of all elements (excluding the first) of an LCP array H will often be used, we define $L(H) = \sum_{i=2}^{n} H_i$ or just L if H is clear in the context. The sum of the distinguishing prefixes D and the sum of the LCP array H are related, but not identical. Whereas D is the sum of the distinguishing prefixes, L only counts the length of LCPs and also misses the length for the first string, leading to $D \geq L$. In the example shown in Figure 5b, we have $L = 2+4+0+3+1 = 10$, whereas $D = 3+3+5+1+4+2 = 18$.

2.2. Existing Sorting Algorithms

To begin with, an overview on existing sorting algorithms is presented. Although there exists a wide range of sorting algorithms, this section focuses on two of them, being essential preliminary work for this thesis. LCP-aware merge sort has been introduced by Waihong Ng in [NK08] and is a basis of this work. Timo Bingmann's pS^5 [BS13] is a parallel string sorting algorithm that achieved great results in previous experiments and will be further optimized by making it NUMA-aware.

More algorithms can be found in [BES14] and [BS13], including but not limited to *Multikey quicksort*, *MSD radix sort*, *Burstsort*, *Sample sort* and *Insertion sort*.

2.2.1. LCP-Mergesort by Waihong Ng

LCP-Mergesort is a string sorting algorithm introduced by Waihong Ng and Katsuhiko Kakehi [NK08]. It calculates and reuses the LCP of sorted sub-problems to speed up string sorting. Ng's binary LCP-Mergesort is redefined in more detail in Section 3.1. As part of this section, the worst case computational complexity of LCP-Mergesort will be shown to be in $\mathcal{O}(n \log n + L)$. Later, LCP-Mergesort's basic step `LCP-Compare`

will be reused as a fundamental part of the new parallel `K-Way-LCP-Merge` algorithm presented in this bachelor thesis.

A parallelized version of Ng's binary LCP-Mergesort has been developed by Nagaraja Shamsundar [Sha09]. The basic idea is to run instances of binary LCP-Mergesort on every thread for subsets of the input strings. As soon as two threads finished their work, their sorted result sequences are merged together sequentially. Whenever another thread finishes (and no other thread is currently merging with the output sequence), its sequence is sequentially merged with the output sequence. However, since the final merging is done sequentially, only the sorting of the sequences is parallelized.

2.2.2. pS^5 by Timo Bingmann

Parallel Super Scalar String Sample Sort (pS^5) introduced by Timo Bingmann and Peter Sanders [BS13] is a parallelized version of S^5, designed to make use of the features of modern many-core systems, having individual cache levels but relatively few and slow memory channels. The S^5 algorithm is based on *sample sort* and preliminary results can be found in the bachelor thesis of Sascha D. Knöpfle [Knö12]. Parallel S^5 uses three different sub-algorithms depending on the size of subsets of the input strings. Whereas for large subsets, a sequential S^5 implementation is used, medium-sized inputs are sorted with *caching multikey quicksort*, which itself is internally applying *insertion sort* as base case sorter. In Section 4.4 our new parallel `K-Way-LCP-Merge` algorithm is used to improve the performance of pS^5 even further on NUMA systems.

3. Parallel Multiway LCP-Mergesort

Starting with the basic components, this section introduces a parallel multiway LCP-Merge algorithm, usable for easier parallelization of sorting algorithms. Moreover, as a direct application, a parallel multiway LCP-Mergesort will be introduced. Based on that, in Section 4 the parallel multiway Merge is used for implementing a NUMA-aware version of pS^5 and more.

3.1. Binary LCP-Mergesort

LCP-Merge is a string merging algorithm introduced by Ng and Kakehi [NK08]. By utilizing the longest common prefixes of strings it is possible to reduce the number of needed character comparisons. As Ng and Kakehi show in their paper, this leads to an average complexity of $\mathcal{O}(n \log n)$ for string Mergesort, using the given `LCP-Merge`.

Preceding the proof of $\mathcal{O}(n \log n)$ complexity, this section focuses on reformulating `LCP-Merge` and explicitly defining its comparison step `LCP-Compare`. Since these steps are fundamental parts of the following work, a rather verbose specification is used. This not only allows an easier reuse of the code in later parts but also helps to visualize the proof of computational complexity.

3.1.1. LCP-Compare

`LCP-Compare` is the basic LCP-aware comparison step used in all algorithms presented in this work. It is a replacement for standard string comparison function, which usually iterates over the characters of a string until a mismatch is found. In order to improve runtime, `LCP-Compare` exploits the longest common prefixes calculated in previous steps.

Like shown in Algorithm 1, `LCP-Compare` takes two strings s_a and s_b and the corresponding LCPs h_a and h_b to calculate the sort order of s_a and s_b, as well as the $\text{lcp}(s_a, s_b)$. The given LCPs h_i need to be the LCPs of their string s_i with a third common, lexicographically smaller string. Therefore there must be a string p with $p \leq s_i$ and $h_i = \text{lcp}(p, s_i)$ where $i \in \{a, b\}$.

Figure 4 visualizes the input parameters of `LCP-Compare` and their relation to the common predecessor p. In Figure 4 it is assumed that $h_a = \text{lcp}(p, s_a) < \text{lcp}(p, s_b) = h_b$. In this situation no characters need to be compared, since the lexicographical order can be calculated solely depending on the LCPs: let $y = s_a[h_a + 1]$ and $x = p[h_a + 1]$ be the distinguishing characters of p and s_a. Due to the precondition $p \leq s_a$ and the definition of LCPs, we do not just know $x \neq y$ but also $x < y$. However, due to $h_b > h_a$, we further know the distinguishing characters of s_a and s_b to be y and $x = s_b[h_a + 1] = p[h_a + 1]$ which leads to the conclusion $s_b < s_a$.

In order to effectively calculate the sort order and LCP of s_a and s_b, `LCP-Compare` differentiates three main cases:

Case 1: If both LCPs h_a and h_b are equal, the first $h_a = h_b$ characters of all three strings p, s_a and s_b are equal. In order to find the distinguishing characters of s_a and s_b, the strings need to be compared starting at position $h_a + 1$. This is done by the loop in line 3. With the distinguishing character found by the loop, the sort order can be determined. Additionally the $\text{lcp}(s_a, s_b) = h'$ is inherently calculated in the loop as a by-product.

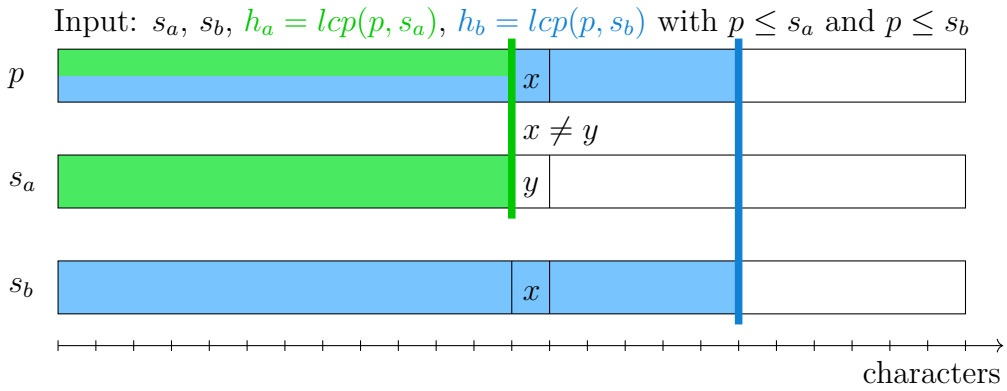


Figure 4: Illustration of case 2 of LCP-Compare with $h_a < h_b$.

Case 2: If $h_a < h_b$, as shown in Figure 4, the first h_a characters of the three strings p, s_a and s_b are equal. Because h_a and h_b are the LCPs to the common predecessor p, the characters at index $h_a + 1$ are the distinguishing characters between s_a and s_b. Due to $p < s_i$ and $h_a < h_b$ follows $p[h_a + 1] = s_b[h_a + 1]$ and $p[h_a + 1] < s_a[h_a + 1]$. This results in $s_b[h_a + 1] < s_a[h_a + 1]$ and therefore $s_b < s_a$.

Case 3: If $h_a > h_b$, the same arguments as in case 2 can be applied in symmetrically.

Algorithm 1 combines these observations to construct `LCP-Compare`, the basic step of `LCP-Mergesort` and the later introduced `K-Way-LCP-Merge`. The three distinct cases from above, being the basic parts of `LCP-Compare`, can be seen in lines 1, 7 and 8, whereas the character comparison loop can be found in line 3.

To be able to use `LCP-Compare` for Binary `LCP-Merge` and `LCP-Mergesort` but also for `K-Way-LCP-Merge`, the function is written in a rather generic way. That's why the caller has to specify the values a and b as keys, identifying the given strings s_a and s_b. Furthermore, `LCP-Compare` does not return the ordered input strings, but w, $l \in \{a, b\}$, and h_w, h_l the corresponding LCPs, so that $s \leq s_w \leq s_l$ and respectively $h_w = \text{lcp}(p, s_w)$ and $h_l = \text{lcp}(p, s_l)$.

Algorithm 1: LCP-Compare

`Input`: (a, s_a, h_a) and (b, s_b, h_b), with s_a, s_b two strings, h_a, h_b corresponding LCPs; assume $\exists$ string p with $p \leq s_a$ and $p \leq s_b$, so that $h_a = \text{lcp}(p, s_a)$ and $h_b = \text{lcp}(p, s_b)$.

```
1 if h_a = h_b then                                   // Case 1: LCPs are equal
2     h' := h_a + 1
3     while s_a[h'] ≠ 0 & s_a[h'] = s_b[h'] do        // Execute character comparisons
4         h'++                                        // Increase LCP
5     if s_a[h'] ≤ s_b[h'] then return (a, h_a, b, h')   // Case 1.1: s_a ≤ s_b
6     else return (b, h_b, a, h')                        // Case 1.2: s_a > s_b
7 else if h_a < h_b then return (b, h_b, a, h_a)      // Case 2: s_a > s_b
8 else return (a, h_a, b, h_b)                        // Case 3: s_a < s_b
```

`Output`: (w, h_w, l, h_l) where $\{w, l\} = \{a, b\}$ with $p \leq s_w \leq s_l$, $h_w = \text{lcp}(w, s)$ and $h_l = \text{lcp}(s_w, s_l)$

3.1.2. Binary LCP-Merge and Binary LCP-Mergesort

Based on `LCP-Compare`, `LCP-Merge` is given in Algorithm 2. The algorithm takes two sorted sequences of strings S_1 and S_2 and their LCP arrays H_1 and H_2 to calculate the combined sorted sequence S_0 with its LCP array H_0.

Algorithm 2: Binary LCP-Merge

`Input`: S_1 and S_2: two sorted sequences of strings, H_1 and H_2: the corresponding LCP arrays; assume $S_1[|S_1|] = S_2[|S_2|] = \infty$

1 $i_0 := 1, i_1 := 1, i_2 := 1$
2 $h_1 := 0, h_2 := 0$ *// Invariant:* $h_k = \text{lcp}(S_k[i_k], S_0[i_0 - 1]), k \in \{1, 2\}$
3 `while` $i_1 + i_2 < |S_1| + |S_2|$ `do` *// Loop over all input elements*
4 $\quad (w, \perp, l, h') :=$ `LCP-Compare`$(1, S_1[i_1], h_1, 2, S_2[i_2], h_2)$
5 $\quad (S_0[i_0], H_0[i_0]) := (S_w[i_w], h_w)$
6 $\quad i_w{+}{+}$, $i_0{+}{+}$
7 $\quad (h_w, h_l) := (H_w[i_w], h')$ *// re-establish invariant*

`Output`: S_0: sorted sequence containing $S_1 \bigcup S_2$; H_0: the corresponding LCP array

Like a usual merging algorithm, the loop in line 3 of Algorithm 2 iterates as long as there are any elements in S_1 or S_2 left. During each iteration, the two current strings of the sequences are compared (line 4), the lexicographically smaller one is written to the output sequence (line 5) and the indexes of the output sequence and the sequence with the smaller element are increased (line 6).

In contrast to these common steps, `LCP-Merge` uses `LCP-Compare` instead of a usual string comparison and stores the LCP value of the winner in the output LCP array H_0. This is important for the later `LCP-Mergesort` implementation, since further `LCP-Merge` steps also require valid LCP arrays of their input sequences. The LCP value of the loser, which is calculated by `LCP-Compare`, is stored in a local variable and used for the next iteration.

The loop invariant, given in line 2, ensures that `LCP-Compare` can be applied. However, because it can only be applied after the first iteration, `LCP-Compare`'s preconditions must be checked for the first iteration. This means, the passed LCP values h_1 and h_2 need to refer to a common lexicographically smaller string p. As we initialize h_1 and h_2 with 0 in line 2, setting $p = \epsilon$ fulfills these requirements.

During any iteration, the winner string is written to the output sequence with its corresponding LCP value being assigned to the equivalent position of the LCP array

Algorithm 3: Binary LCP-Mergesort

`Input`: S sequence of sorted strings; assume $S[|S|] = \infty$

1 `if` $|S| \leq 1$ `then` *// Base case*
2 $\quad$ `return` $(S[1], 0)$
3 `else`
4 $\quad l_{1/2} := |S|/2$
5 $\quad S_1 = \{S[1], S[2], ..., S[l_{1/2}]\}$, $S_2 := \{S[l_{1/2} + 1], S[l_{1/2} + 2], ..., S[|S|]\}$
6 $\quad (S'_1, H'_1) :=$ `LCP-Mergesort`(S_1), $(S'_2, H'_2) :=$ `LCP-Mergesort`(S_2)
7 $\quad$ `return` `LCP-Merge`(S'_1, H'_1, S'_2, H'_2)

`Output`: S_0: sorted sequence containing $S_1 \bigcup S_2$; H_0: the corresponding LCP array

in line 5. In order to restore the invariant, the local LCP values are updated in line 7. Whereas the winner's new local LCP value is loaded from the winner's input LCP array, the loser's one is taken from the result of `LCP-Compare`. Therefore the invariant holds true for the winner, due to the definition of LCP arrays and for the loser, due to the postcondition of `LCP-Compare`.

With the given binary `LCP-Merge` algorithm, binary `LCP-Mergesort` can be implemented as shown in Algorithm 3.

3.1.3. Computational Complexity of Binary LCP-Mergesort

Although `LCP-Mergesort` was introduced first by Ng and Kakehi [NK08], they did not provide a worst case analysis. However, their average case analysis shows the computational complexity of `LCP-Mergesort` to remain $\mathcal{O}(n \log n)$ on average, whereas the complexity of standard recursive string Mergesort tends to be greater than $\mathcal{O}(n \log n)$.

In this section the worst case computational complexity of LCP-Mergesort will be analysed and shown to be in $\mathcal{O}(n \log n + L)$

Clearly the number of string comparisons of `LCP-Mergesort` (i.e. calls of `LCP-Compare`) is equal to the number of comparisons of Mergesort with *atomic* keys and therefore in $\mathcal{O}(n \log n)$. However, in difference to Mergesort with *atomic* keys, `LCP-Compare` needs to compare strings, which in general requires more than a single comparison to determine the sort order. In the following the number of comparisons required in each case of `LCP-Compare` shall be counted:

Whenever `LCP-Compare` is called, there need to be integer comparisons of two LCPs to determine the case to select. The three cases can be determined with a maximum of two integer comparisons, resulting in an asymptotically constant cost for this step. Following this, cases two and three do not require any more calculations and can immediately return the result.

However, in case one, the character comparing loop (line 3 of Algorithm 2) is executed starting with the character at position $h' + 1$. If both characters are found to be equal, h' is increased by one and as it is later set to be the new LCP of the loser (line 7) the overall LCP value is increased by one, respectively. Because of LCP values never getting dropped or decremented, this case may only occur L times in total, with L being the sum of all LCPs. If the characters are not equal, the loop is terminated and the result can be returned. Like before, the three comparisons in lines 3, 5 and 6 are counted as one ternary comparison. Since this case terminates the loop, it occurs exactly as often as case 1 is entered. However, this is limited by the times `LCP-Compare` is called, which is in $\mathcal{O}(n \log n)$. But as this is only an upper bound, for most string sets, cases two and three (see Section 3.1.1) reduce the number of times case one is entered.

In conclusion, `LCP-Mergesort`'s computational complexity is shown to have the following upper bound, where c_i is the number of integer and c_c the number of character comparisons:

$$\begin{aligned} &\mathcal{O}((n \log n)c_i + (L + n \log n)c_c)) \\ &= \mathcal{O}(n \log n)c_i + \mathcal{O}((n \log n + L))c_c \\ &= \mathcal{O}(n \log n + L) \text{ comparisons.} \end{aligned}$$

□

In their average case analysis, Ng and Kakehi [NK08] show, the total number of character comparisons to be about $n(\mu_a - 1) + P_\omega n \log_2 n$ where μ is the average length of

the distinguishing prefixes and P_ω the probability of entering case one in `LCP-Compare` (Algorithm 1). Assuming $P_\omega = 1$ and $\mu_a = \frac{D}{n}$ their result matches the worst-case up to the minor difference between D and L.

3.2. K-Way LCP-Merge

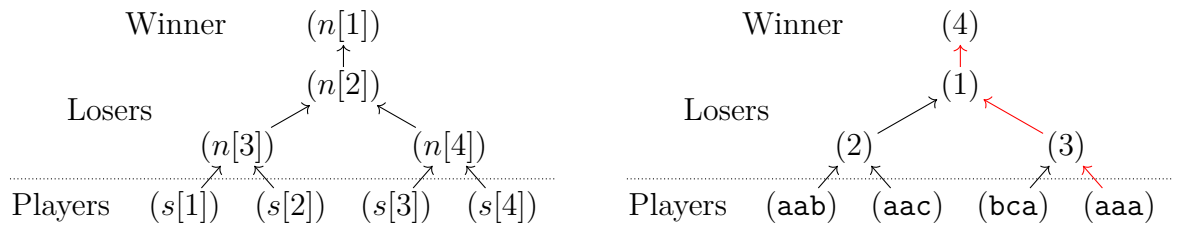
In order to improve cache efficiency and as preliminary work for parallel multiway LCP-Mergesort and NUMA optimized pS^5, a K-way LCP-Merge was developed. A common and well-known multiway merging method is to use a binary comparison to construct a tournament tree, which can be represented as a binary tree structure [Knu98]. Although this allows efficient merging of multiple streams of sorted inputs, no implementation of a LCP-aware tournament tree was found in literature.

3.2.1. Simple Tournament Tree

Multiway merging is commonly seen as selecting the winner of a tournament of K players. This tournament is organized in a binary tree structure with the nodes representing a match between two players. Although there also is the possibility to represent a tournament tree as *winner tree*, for our implementations, a *loser tree* is more intuitive. Therefore, the "loser" of a match is stored in the node representing the match, whereas the "winner" ascends to the parent node and faces the next game. With this method repeatedly applied, an overall winner is found and usually placed on top of the tree in an additional node. We do not consider the players as parts of the actual tournament tree, since they are only used here to ease comprehensibility and not needed in actual code. Therefore the tournament tree has exactly K nodes and the nodes reference every player exactly once.

Figure 5a shows the structure of a simple tournament tree with $K = 4$. As visualized, in a node v of the tournament tree, the *index of the input stream* $n[v]$ of the corresponding match's loser, rather than the actual string, or a reference of it, is stored. In the exemplary configuration, shown in Figure 5b, the strings `aab`, `aac`, `bca` and `aaa` compete to become the overall winner. The winner's path P from its player's node to the top is shown in red colours, because it will be of importance for selecting the next winner.

However, before the first winner can be selected, an initial round needs to be played with all players starting from the bottom of the tree. Since the winners, in this case the lexicographically smaller strings, of the first level ascend to the level above, the next matches are played. After the topmost level is reached, the first overall winner is found and therefore is the smallest string. During this initial round all matches, represented by the nodes of the tree, need to be played exactly once. As the tree contains exactly K



(a) Structural view

(b) Example with red winner Path P

Figure 5: Structure of simple tournament tree with $K = 4$.

nodes, K comparisons need to be executed. The initialization phase is further illustrated with an example of a LCP-aware tournament tree in Figures 8 to 11.

After the initial round is finished, only $\log_2 K$ matches need to be played to determine the next winner and therefore the next string to be written to the output. This can be achieved by first replacing the current winner player with the next string on its corresponding input sequence. In order to find the winner of the new set of players, all games along the red path P in Figure 5b of the former winner, must be replayed. Thus the new player needs to play the first match starting at the bottom of the tree with the former loser of that match. Again, whoever loses the match stays at that node representing the match, whereas the winner ascends to the next level. Since the binary tree has $\lceil \log_2 K \rceil$ levels, the new overall winner is found with $\lceil \log_2 K \rceil$ comparisons. The steps for replaying the tournament after removing the current winner, are also further illustrated in the example of a LCP-aware tournament tree in Figures 11 to 13.

Repeatedly applying this process until all input streams are emptied, realises the K-way merge. Assuming sentinels for empty inputs, special cases can be avoided. Furthermore, K can be assumed to be a power of two, since missing sequences can easily be represented by empty streams. Hence, the tournament tree can be assumed as perfect binary tree. Due to using one-base arrays, traversing the tree upwards, that means, calculating the parent p of a node v, can effectively be done by calculating $p = \lceil \frac{v}{2} \rceil$. This leads to a very efficient implementation to find the path from a player's leaf to the root of the tree.

3.2.2. LCP-Aware Tournament Tree

In this section, focus is put on extending the simple tournament tree, described in the section before, to a LCP-aware tournament tree. First of all, to reduce the number of character comparisons done during the matches, we use `LCP-Compare` (see Section 3.1.1) to exploit input sequences' LCP arrays. Because we want to prevent character comparisons we already know to be equal, we also store a LCP value $h[v]$ in the node alongside the index to the losers input sequence. The value stored in $h[v]$ is the LCP of the participants of the match of node v.

Figure 6 visualizes the structure of the new LCP-aware tournament tree. Additionally to winner, loser and player nodes already shown in Figure 5 the input and output sequences have been added as well. These will be useful in the example illustrated in Section 3.2.3.

As pictured in Figure 6, the nodes of the LCP-aware tournament tree now contain the

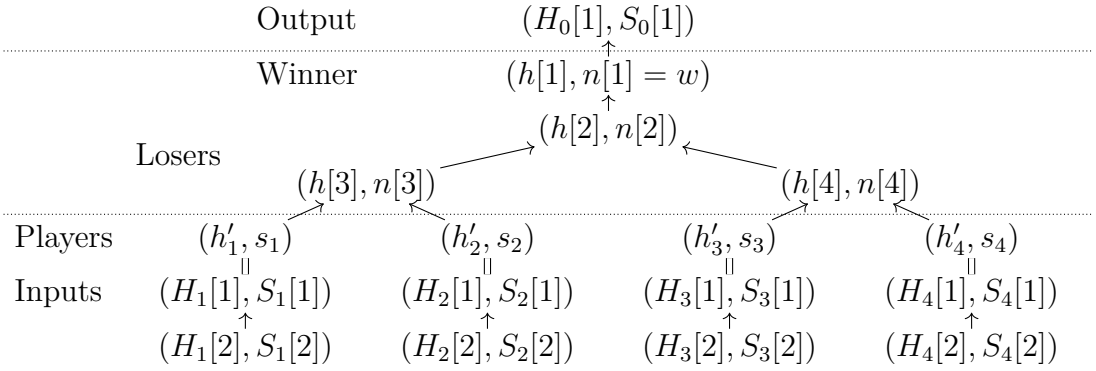


Figure 6: Structure of LCP tournament tree with in and output sequences, $K = 4$.

LCP value $h[v]$ alongside $n[v]$, the index to the input sequence of the corresponding match's loser. The players of the tournament are the first elements of the remaining input sequences. Since we now describe the process, which will be summarized in Algorithm 4, and to emphasize their position as participants of the tournament, they are referred to as players and kept in an additional array. Just like with the simple tournament tree of Figure 5, only the winner and loser nodes are actually part of the tree. Therefore the LCP-aware tournament tree has exactly K nodes.

As well as the standard tournament tree, the LCP-aware tournament tree also needs to be initialized first. Like mentioned before, `LCP-Compare` is used to replace the standard compare operation. However, `LCP-Compare` does not just need two strings as parameters, but also two LCPs to a common lexicographically smaller string. For the process of tree initialization, these LCPs are always 0 and the common base string is ϵ. Therefore the preconditions of `LCP-Compare` are fulfilled and it can be applied to compare the given strings like a normal string comparison procedure.

In order to extract the second winner, we need to make sure, the preconditions of `LCP-Compare` are fulfilled after the first initial round has been completed. Let $w = n[1]$ be the index of the input sequence of the current overall winner, which is to be removed. Exactly as with the simple tournament tree, it is clear, that w won all matches along the path P from its leaf to the top. Therefore all LCP values $h[v]$, stored in the nodes along this path, are given by $h[v] = \text{lcp}(s_{n[v]}, s_w)$ and it is true that $s_w \leq s_{n[v]}$, $\forall v \in P$. Let s'_w be the successor of the input sequence with index w. Then the definition of LCP arrays specifies the corresponding LCP of the input sequence to be $h'_w = \text{lcp}(s_w, s'_w)$ and $s_w \leq s'_w$. Combining these observations one can determine that all strings that might get compared by `LCP-Compare`, i.e. that are along path P, have the common predecessor s_w and all the used LCP values refer to s_w. Therefore the correctness of the preconditions of `LCP-Compare` is ensured.

Likewise it needs to be shown that after n winners have been removed, the next one can also be removed and the matches had been replayed as described. However, the exact same argument can be applied again and so merging K sequences with `K-Way-LCP-Merge` works as desired. Pseudo code of `K-Way-LCP-Merge` can be seen in Algorithm 4.

To refine the calculations done in Algorithm 4, we will first focus on the implementation of the initialization phase realized by the loop in line 2. The functionality of the loop is based on viewing the tournament tree as a perfect binary odd-even-tree like shown in Figure 7, where the colours visualize the parity of the indexes written in the nodes.

During the initialization phase, the loop iterates over all players, starting from index $v = 1$ and lets them play as many matches as there are currently available. Therefore in the first iteration of the loop the string of player $k = 1$ is to be positioned in the tree. Due to line 4, this results in $v = K + k$ being odd. Therefore the inner loop is not called and the index of the string is directly written to the odd node with index $v = \frac{K+k}{2} = 5$ in Figure 7.

In the second iteration with $k = 2$, the inner loop in line 5 is played once as $v = 10$ is even before the first iteration and odd the next time. However, the comparison is done with the odd node $v = \frac{10}{2} = 5$. After the inner loop finished, the index of the previous game's winner is written to the next parent node.

To sum it up, comparisons need to be done at the *parents* of all even nodes (this time including the player nodes). The remaining winner of the last comparison then has to be written to the next parent node, which is done in line 9. To ensure the correctness

Algorithm 4: K-Way-LCP-Merge

`Input`: S_k sorted sequences of strings, H_k the corresponding LCP arrays; assume sentinels $S_k[|S_k|+1] = \infty$, $\forall k = 1, ..., K$ and K being a power of two.

```
1  i_k := 1, h_k := 0, ∀k := 1, ..., K
2  while k = 1, ..., K do                                   // Play initial games
3      s[k] := S_k[1]
4      x := k, v := K + k
5      while v is even & v > 2 do
6          v := v/2
7          (x, ⊥, n[v], h[v]) := LCP-Compare(x, s[x], 0, n[v], s[n[v]], 0)
8      v := ⌈v/2⌉
9      (n[v], h[v]) := (x, 0)
10 j := 1
11 while j ≤ Σ_{k=1}^{K} |S_k| do                           // Loop over all elements in inputs
12     w := n[1]                                           // Index of the winner of last round
13     (S_0[j], H_0[j]) := (s[w], h[1]), j++               // Write winner to output
14     i_w++, s[w] := S_x[i_x]
15     v := K + w, (x, h') := (w, H_w[i_w])                // v index of contested, x index of contender
16     while v > 2 do                                      // Traverse tree upwards and play the games
17         v := ⌈v/2⌉                                      // Calculate index of contested
18         (x, h', n[v], h[v]) := LCP-Compare(x, s[x], h, n[v], s[n[v]], h[v])
19     (n[1], h[1]) := (x, h')                             // Now the tournament tree is complete again
```

`Output`: S_0: sorted sequence containing $S_1 \bigcup S_2$; H_0: the corresponding LCP array

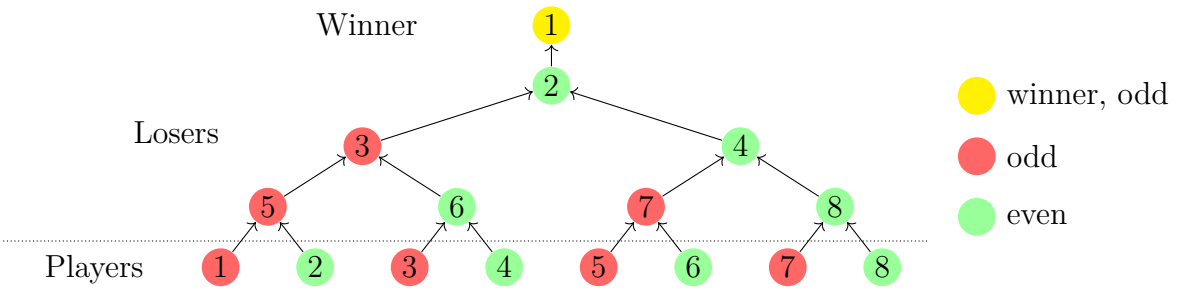


Figure 7: Binary Odd Even Tree with $K = 8$.

of this procedure, all nodes used for comparisons need to be already initialized and the last parent node p_k of the iteration for player k needs to be empty before the run.

From Figure 7 one can easily see that even nodes are always the right child of their parents, whereas odd nodes are always the left child, except for node 2 as node 1 is a special case. Let v_e be the even, v_o the odd child of the parent node v_p. The parent's left sub-tree, with v_o on its top, must already be fully initialized since the initialization starts from the left side and all leafs in that sub-tree have a lower player index. Because the left sub-tree is already initialized, the match of v_o was already played and so its winner's index has been stored in v_p, which therefore is initialized and can be used for comparison with the winner of v_e. When looking at the saving of the last winner in line 9, we need to check that this node is not initialized yet, as otherwise it would be overwritten. Here, a similar argument can be used. Since the last node being compared is an odd node v_o (except for node 2), its complete sub-tree is initialized. However, no

players positioned right of this sub-tree have been worked yet and so the right child of the parent of v_o can not be set yet either.

3.2.3. K-Way LCP Tournament Tree Example

The following example shall be used for further illustration of how a K-Way LCP tournament tree, implicitly used for `K-Way-LCP-Merge` (Algorithm 4), is constructed during the initialization phase and rebuild after the current minimum has been removed. The example uses a tournament tree with $K = 4$ input sequences and its structure is oriented on the structural view of the tree, shown in Figure 6. The four sequences contain the following strings with corresponding LCPs: Sequence 1: `aab` and `aba` with an LCP of 1; Sequence 2: `aac` and `aad` with an LCP of 2; Sequence 3: `bca` and `ca` with an LCP of 0; Sequence 4: `aaa` and `acb` with an LCP of 1.

Figure 8 illustrates the state before the initialization of the tree started. The sorted input sequences with the appropriate LCPs are shown at the bottom, players and the tree's nodes are not initialized yet.

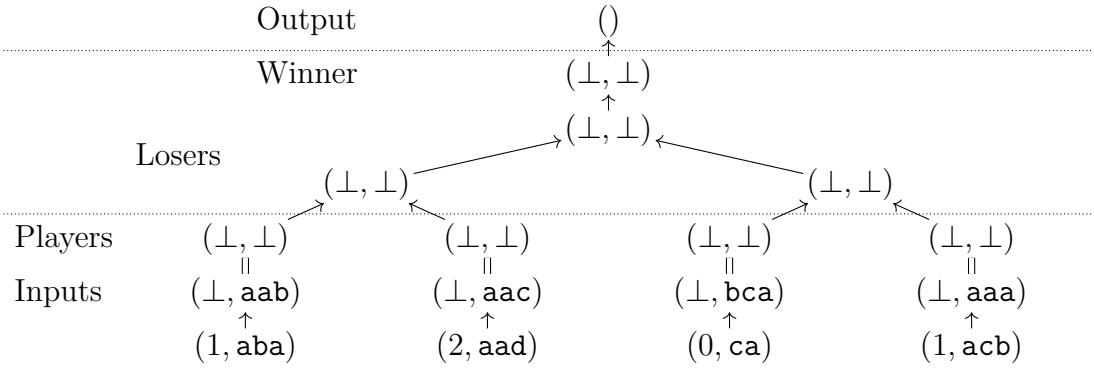


Figure 8: LCP-aware tournament tree example: part 1

Figure 9 shows the state after the first iteration of the initialization loop in line 2 the first player and its *parent* tree node are initialized. The LCP in the tree node has been set to 0, because it is the LCP to the string ϵ, which is a lexicographically smaller common string to all players.

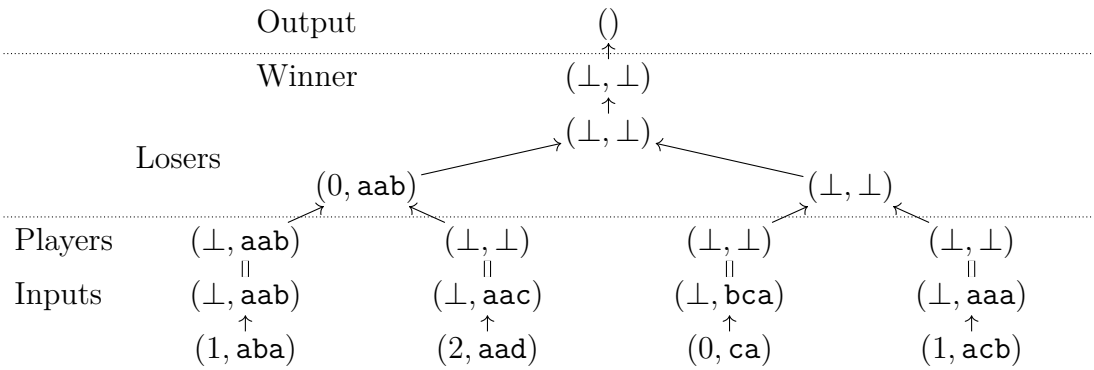


Figure 9: LCP-aware tournament tree example: part 2

.

In Figure 10 the tree's state after the second run of the initialization loop in line 2 is visualized. The string `aab` won the match with `aac` and moved upwards to the next

free position, whereas aac stays at the loser position with its current LCP $h[3]$ being set to 2.

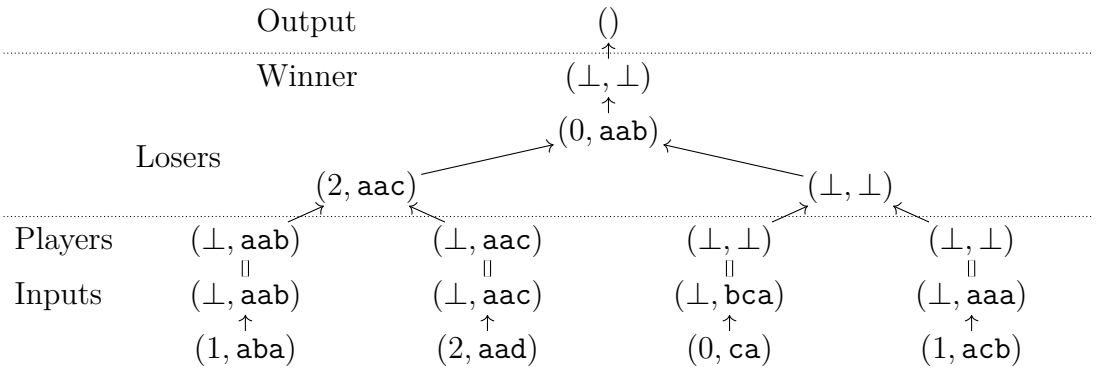


Figure 10: LCP-aware tournament tree example: part 3

The tournament tree's state after the third initialization step is shown in Figure 11. The first string of the third input sequence moved up to its parent node. However, since the stream's index is uneven, the string can directly be placed in the match's node and does not need to be compared, as no other string can be there, yet.

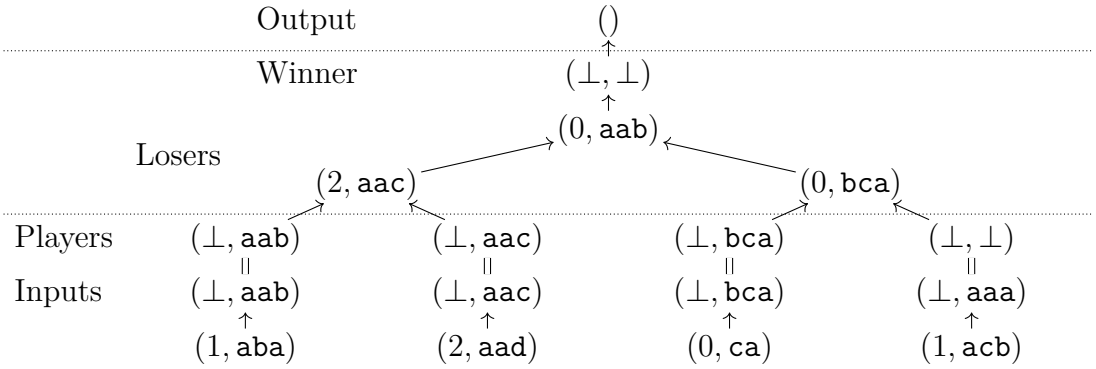


Figure 11: LCP-aware tournament tree example: part 4

Figure 12 shows the fully initialized tree after the fourth initialization step, which is the tree's state, just before the loop in line 11 of Algorithm 4 is entered. During this last step, the string aaa is first compared with bca. Because aaa is lexicographically smaller, it ascends the tree to attend the next match, whereas bca stays at the match's node with the common LCP $\text{lcp}(\texttt{aaa}, \texttt{bca}) = h[4] = 0$. As aaa also wins the match with aab, it is written to the root of the tree and aab stays at the loser position with the new LCP $\text{lcp}(\texttt{aaa}, \texttt{aab}) = h[2] = 2$. The red line illustrates the winner's path to the top of the tree.

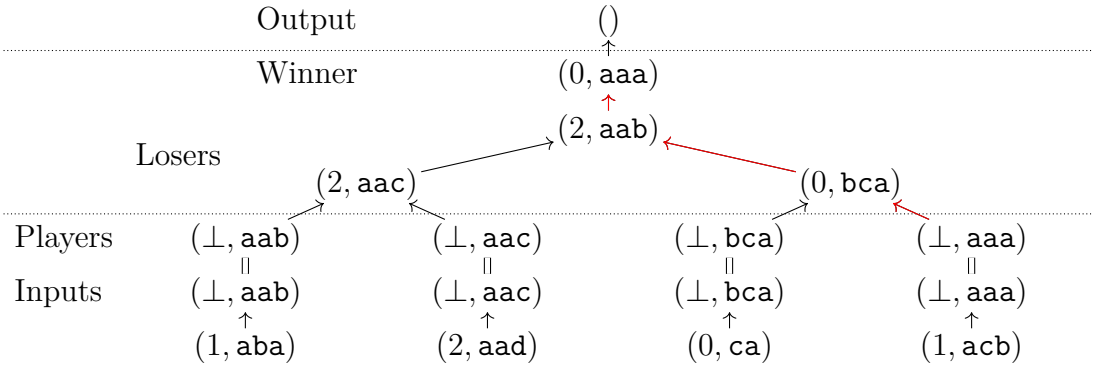


Figure 12: LCP-aware tournament tree example: part 5 with winner path P (red)

The intermediate state after the first winner has been removed and written to the output stream, is displayed in Figure 13. Since the winner's input stream has moved forward, the string `acb` replaces the former winner `aaa`. The LCP of `acb` is taken from the LCP array of the input stream as it directly refers to `aaa`. With this steps done up to line 11 of Algorithm 4, the new set of players is complete and ready to compete with each other.

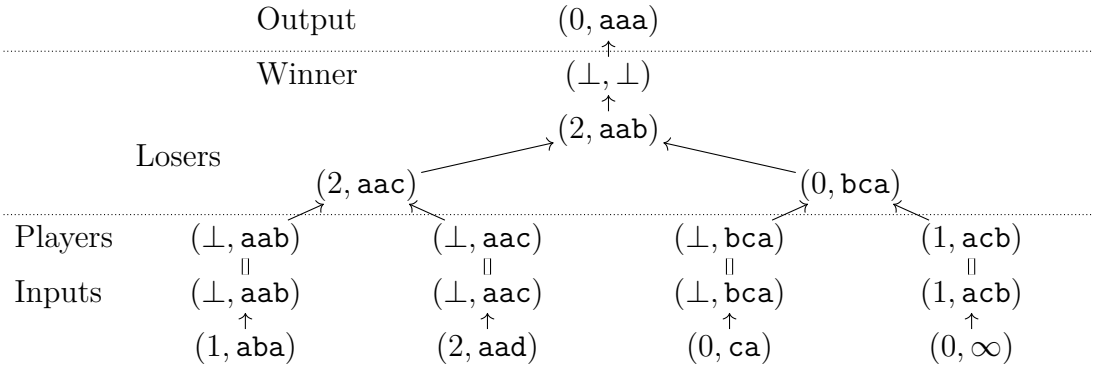


Figure 13: LCP-aware tournament tree example: part 6

After the inner loop in line 16 of Algorithm 4 finishes, the situation shown in Figure 14 is achieved. During the iterations, the following matches were played: `acb` won against `bca` and `aab` won the match with `acb`. Both matches where determined by the LCP values. Therefore not a single character comparison was needed and the effect of exploiting the LCPs in `LCP-Compare` becomes visible.

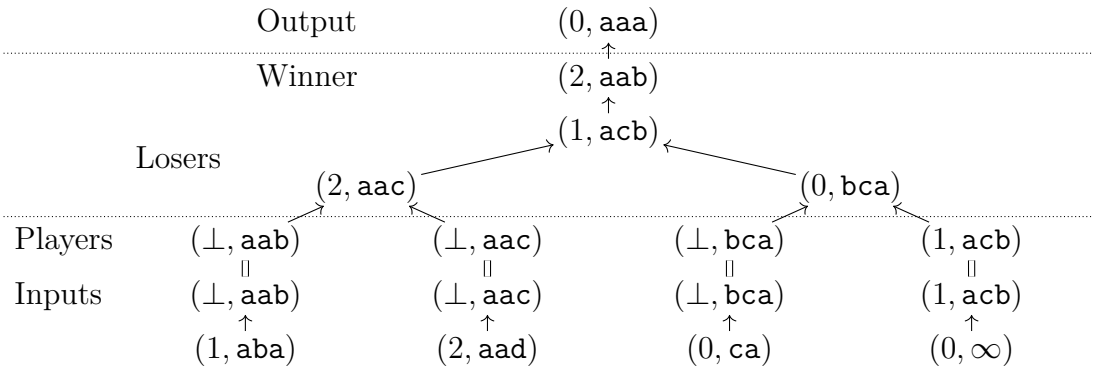


Figure 14: LCP-aware tournament tree example: part 7

3.3. Parallelization of K-Way LCP-Merge

This section focuses on parallelization of K-Way LCP-Merge, merging K sorted input sequences of strings with their corresponding LCP arrays. When trying to solve problems in parallel, a common approach is to split-up the work into sub-tasks, process the sub-tasks in parallel and in the end, put the pieces back together. Applying this to sorting, one can let any sequential sorting algorithm work on parts of the input in parallel. However, merging the resulting sorted sequences can not be parallelized without significant overhead needed to split up the work into work disjoint subtasks [Col88]. Instead of being able to simply cut the input sequences into pieces, the merging problem needs to be divided into disjoint parts, as commonly done in practical parallel merge sort implementations [AS87], [SSP07].

One well-known way to accomplish a partitioning for atomic merge sort, is to sample the sorted input sequences to get a set of splitters. After they have been sorted, they can each be searched (e.g. via binary search) in all the input sequences. The positions found for a splitter define splitting points, separating disjoint parts of the merging problem. This approach is directly adapted to our LCP-aware multiway string merging algorithm in Section 3.3.1. In the following we refer to this splitting method, creating multiple work disjoint parts in a single run, as *classical splitting*.

As a simplification of classical splitting, *binary splitting*, creating only two jobs in a run, is introduced. Here we do not sample and split for several splitters, but for just a single splitter. This approach is explained in more detail in Section 3.3.2.

In Section 3.3.3 a new splitting algorithm is defined. By exploiting LCP arrays of the input sequences to find splitting points, it is possible to almost fully avoid random memory accesses to characters of strings normally causing a significant amount of cache faults.

Another way to split the input sequences of an atomic merge into exactly p equal-sized range-disjoint parts was proposed by Varman et al. [PJV91]. Although their algorithm allows to create equally-sized parts with atomic keys, this approach is not sufficient for string merging. Static load balancing is not an efficient solution, due to the varying cost of an equal number of string comparisons, depending on the length of distinguishing prefixes. Therefore, oversampling (creating more tasks than processing units available) and dynamic load balancing is required. Since the benefit of exact splitting only appears with atomic keys, the algorithm has not been considered any further in this work.

Instead, the same lightweight dynamic load balancing framework as for pS^5 [BS13] is used. Every thread currently executing a merge job, regularly checks if any threads are idle as no jobs are left in the queue. In order to reduce balancing overhead the threads execute this check only about every 4000 outputted strings. If an idle processing unit is detected by a thread, its K-way merge job is further split up into new jobs by applying the heuristic above.

3.3.1. Classical Splitting with Binary Search for Splitters

As described in the previous section, the merge problem can not easily be divided into disjoint sub-tasks. One widely used approach to create range-disjoint parts is to separate the elements of the input sequences by sampled splitters. After sorting these splitters, a binary search can be used to find the splitting positions.

The basic principle behind this algorithm is that an arbitrary string can be used to split up a sequence of strings into two range-disjoint pieces. To do so with given splitter

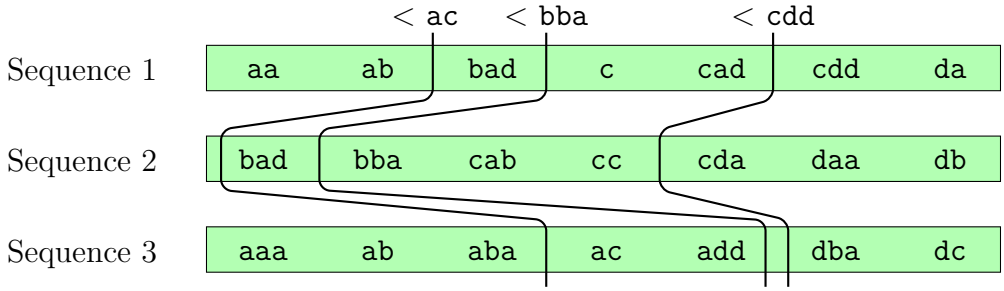


Figure 15: Splitting of three input sequences with splitters `ac`, `bba` and `cdd`.

string s, we define the splitting position of sequence k to be

$$p_k = \underset{1 \leq n \leq |S_k|}{\arg\min} \{s \leq S_k[n]\} - 1, \forall 1 \leq k \leq K$$

Then, the complete merge operation can be split up into two disjoint parts, the first containing all sequences $S'_k = (S_k[0], ..., S_k[p_k])$, the second containing the sequences $S''_k = (S_k[p_k + 1], ..., S_k[|S_k|])$. By definition, all strings of sequences S'_k are lexicographically smaller than the splitter string s. Therefore a job can be created to merge input sequences S'_k and write the output directly to the positions $1 \leq n \leq \sum_{k=1}^{K} |S'_k|$ of the output sequence. Another independent job can be created to merge the sequences S''_k and write the output to positions $\sum_{k=1}^{K} |S''_k| + 1 \leq n \leq \sum_{k=1}^{K} |S_k|$. As these job's input and output data is range-disjoint, it can easily be parallelized.

As modern multi-core systems have many cores, we need to create more than just two jobs. This can be achieved by sampling multiple splitters from the input sequences and sorting them. Binary search can then be used to find all splitting positions and so multiple range disjoint jobs can be created in a single run. Figure 15 illustrates the splitting of three input sequences by the three splitters `ac`, `bba` and `cdd`. As the figure shows, the new merge jobs may also contain empty input sequences. More on practical optimizations resulting from this can be found in Section 4.2.

Algorithm 5 shows an implementation of the classical splitting algorithm taking K sorted input sequences to create M independent merge jobs. The loop in line 2 samples splitters from every input sequence which are sorted in line 6. Because the input sequences are already sorted, the splitters can be sampled equidistantly. As a result of that, the splitters of the different streams only need to be merged, instead of completely sorted.

The foreach loop in line 8 creates the actual merge jobs. To do so, the inner loop in line 10 iterates over all input sequences and searches the splitting position p. Afterwards, the found splitting position is used to split the input sequence into two disjoint sequences. Whereas the first sequence is used to create a new merge job, the second is to be split up further in the next iteration. In line 13, the separated sequences S'_k are combined to one merge job that is completely independent from the others. After all splitters have been found, the remaining parts of the input sequences build the last merge job in line 14.

In order to discuss the runtime of Algorithm 5, the three main steps of the algorithm need to be considered. Since the splitter sampling done in lines 2 to 5 generates exactly $M - 1$ splitters, each sampled in $\mathcal{O}(1)$, this step can be accomplished in $\mathcal{O}(M)$ time. Merging the K sorted sequences of splitters in line 6 can be done with simple multiway string merging in $\mathcal{O}(\Delta D)$ time, with ΔD being the sum of the distinguishing prefixes of

Algorithm 5: Classical Splitting

```
Input: S_k sorted sequences of strings with 1 ≤ k ≤ K and N the number of desired
       merge jobs; assume M = x · K + 1
1  m' := ⌊M/K⌋                          // Calculate number of splitters per input sequence
2  for 1 ≤ k ≤ K do                     // Loop over all input sequences
3      dist := ⌈|S_k|/(m' + 1)⌉
4      for 1 ≤ i ≤ m' do                // Sample m' splitters from sequence k
5          splitters_k[m] := S_k[i * dist]   // Build array of equidistant splitters
6  splitters := merge(splitters_1, ..., splitters_K)   // Merge sorted arrays of splitters
7  m := 1
8  for 1 ≤ i < M do                     // For each splitter create a disjoint job
9      s := splitters[i]
10     for 1 ≤ k ≤ K do                 // Search splitter s in all input sequences
11         p := Binary-Search(S_k, s) − 1   // Binary search position, so that S_k[p] < s
12         S'_k := (S_k[1], ..., S_k[p]), S_k := (S_k[p + 1], ..., S_k[|S_k|])   // Create new sequences
13     J_m = {S'_1, ..., S'_K}, m++     // Create merge job containing the new sequences
14 J_M = {S_1, ..., S_K}                // Create merge job with remaining sequences
Output: M merge jobs J_n = {S_{m,k}} with 1 ≤ m ≤ M and 1 ≤ k ≤ K so that
        S_k = ∪_{m=1}^{M} S_{m,k} and ∅ = ∩_{m=1}^{M} S_{m,k}, ∀1 ≤ k ≤ K
```

all splitters. In the last step in lines 8 to 14, binary search is used to find the splitters in all input sequences. As $M-1$ splitters need to be found in K input sequences of length $|S_k|$, the runtime is limited by $\mathcal{O}(K \cdot M \cdot \log |S_{max}|)$, where $S_{max} = \arg\max_{S_k} |S_k|$ is the longest input sequence. Combining these observations, the runtime of Algorithm 5 is shown to be in $\mathcal{O}(\Delta D + K \cdot M \cdot \log |S_{max}|)$.

3.3.2. Binary Splitting

Binary splitting follows the same principle as classical splitting by using a splitter string to separate the sequences into work disjoint parts. In contrast to classical splitting, only one splitter string is sampled and therefore only two jobs are created in one splitting run.

However, to utilize all processing units, we need to create more than just two jobs. To achieve this, every merge job checks, if there are any idle threads and splits itself up further whenever more jobs are needed. For fast job creation on start-up, this check is executed directly when a merge job's execution is started. Moreover, for fast reaction during later execution, the check is repeated regularly.

In comparison to classical splitting, binary splitting introduces more overhead because more splitting runs need to be executed. However, a run of binary splitting finishes much faster than a run of classical splitting, because much less work is required. This enables binary splitting to respond faster to idling threads, reducing wasted processing time. Moreover, since the merge jobs of binary splitting immediately start splitting up further, the splitting process is inherently parallelized, whereas classical splitting is mostly sequential.

Another aspect is that binary splitting can directly react to the need for new jobs, whereas classical splitting produces more jobs than initially required to reduce the

number of splitting runs. This results in binary splitting producing less jobs than classical splitting, partly compensating the higher splitting costs per created job.

3.3.3. Splitting by LCP Level

Although classical splitting is shown to have good theoretical runtime and low constant runtime factors, in practice, it might hit performance penalties, as it uses mostly random memory accesses. Almost all memory accesses are made by binary search for splitters where strings of a very wide range of memory are accessed. Furthermore, access of string characters also incurs costly cache faults resulting in unpredictable access times, especially on NUMA architectures. Additionally, classical splitting currently does not exploit the LCP arrays of the input sequences. Therefore we developed a splitting algorithm trading random memory accesses against linear scanning memory accesses of the LCP array to reduce the number of character comparisons to a minimum. The basic principle of LCP splitting is to find independent areas by merging the top of the LCP interval trees [AKO04] of the K input sequences.

For LCP splitting, we consider all occurrences of a global minimum l of the LCP array. For sequence S_k we define the M positions p_i, $i \in \{1, ..., M\}$ to be the positions having the minimum LCP l. When additionally defining $p_0 = 1$ and $p_{M+1} = |S| + 1$, these positions divide the input sequence into disjoint areas $a_{k,i} = [p_i, p_{i+1})$ with $i \in \{0, 1, ..., M\}$. Due to the definition of LCP arrays, all strings in the input sequence must have a common prefix of length l and within the areas $a_{k,i}$, there is a common prefix of at least length $l + 1$ (as otherwise the area would have been splitted). Therefore splitting the input sequence at positions with global minimum LCP l generates disjoint areas containing only strings with a distinct common prefix with a length of at least $l + 1$. The only remaining task is to match these areas of all K input sequences to create merge jobs. Following the previous observations, all strings in such a region, have an equal character at position $l + 1$. Furthermore, between any strings of two different regions, those characters are the distinguishing characters. Therefore only the characters at position $l + 1$ need to be compared to find matching regions between different input sequences.

Figure 16 shows a sorted sequence of strings with its corresponding LCP array visualized as red lines on appropriate height. In the example, the minimum LCP is $l = 2$ and can

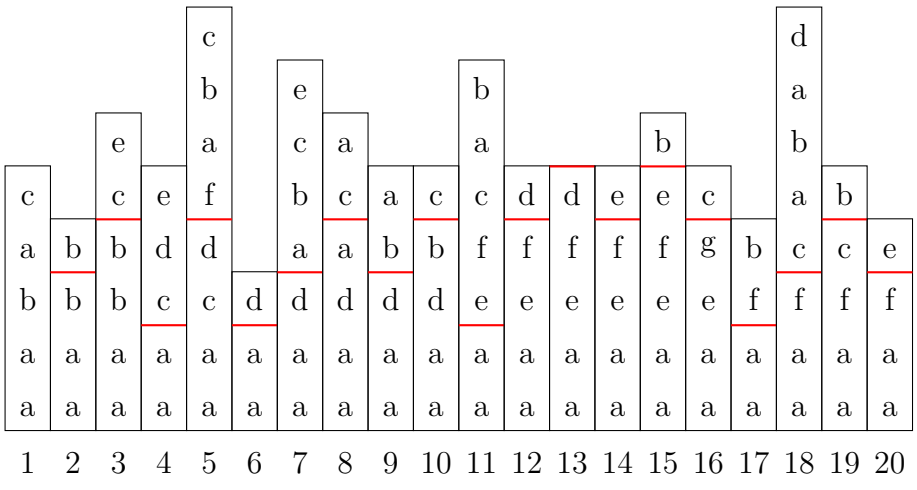


Figure 16: String sequence with LCP level (red line).

be found at the four positions 4, 6, 11 and 17, dividing the sequence into the five disjoint areas $[0, 4)$, $[4, 6)$, $[6, 11)$, $[11, 17)$ and $[17, 20]$. As described before, the minimum LCP in these areas is at least of height $l + 1 = 3$ and all strings in an area have a common character at index $l + 1 = 3$.

Depending on the input data and alphabet, splitting only at positions of global LCP minimum, might not yield enough independent merge jobs. However, the same approach can be applied to sub-areas of already splitted regions, since they can be considered to be independent sequences of their own. Due to the fact, that the independent sub-regions created in the first run, have a minimum LCP of at least $l + 1$, the minimum LCP in these areas will also be at least $l + 1$.

Combining these ideas, a splitting heuristic is developed, which creates merge jobs by scanning the LCP arrays of the K input sequences sequentially once. The algorithm starts with reading w characters from the first string of the K input sequences and selects the sequences with the lowest character block $\bar{c}$. The LCP array of the selected inputs is then scanned skipping all entries greater than w. Entries equal to w need to be checked for equal character blocks. When an entry smaller than w or an unequal character block is found, the scanning is stopped. This forward scan skips all strings with prefix $\bar{c}$ and an independent merge job can be started. The algorithm starts again with reading the w characters of the first strings of all remaining sequences.

However, simply applying the above process can result in very different amounts of created merge jobs. When used on input sets with large average common prefixes, only a few jobs may get created, whereas to many will be produced when used on sets with low average LCP, e.g. on random data. To be able to adapt to input characteristics, we use a heuristic adjusting w, the number of inspected characters. Before the heuristic starts, we calculate an estimated number of jobs to be produced by the splitting algorithm, depending on input length and number of available processing units. The heuristic starts with $w = 8$ (loading a complete 64-bit register of characters) and keeps track of the number of produced jobs in correlation to the number of already used strings of the input sequences, to adjust w accordingly. Whenever too many jobs are created, w gets decreased and vice versa. This prevents a flood of too small merge jobs but ensures creation of enough independent work packages.

4. Implementation Details

We implemented LCP-Mergesort and K-way LCP-Merge in C++, parallelized K-way LCP-merge and used it to implement a fully parallel LCP-Mergesort, as well as to improve performance of pS^5 on NUMA architectures. Our implementations are available from http://tbingmann.de/2013/parallel-string-sorting. Detailed experimental results and discussion can be found in Section 5. In this section focus is set on implementation and practical refinements to improve performance.

4.1. Tournament Tree and K-Way LCP-Merge

The LCP-aware K-way tournament tree described in Section 3.2 is a basic part of the further work. It is used to build an independently working parallel top level K-way LCP-Merge (Section 4.2), a fully parallel LCP-Mergesort (Section 4.3) and to optimize pS^5 for NUMA systems (Section 4.4). Therefore improving this basic component has major impact on all of these algorithms. Additionally, specific challenges of the different applications need to be considered.

As modern many-core architectures have a strong memory hierarchy with dramatically differing memory access times between each level, cache-efficiency is a key aspect to be considered. This becomes even more important on NUMA systems, where there is an additional level in this hierarchy as NUMA nodes have fast access to local memory, but only slow access to the remote NUMA node's memories.

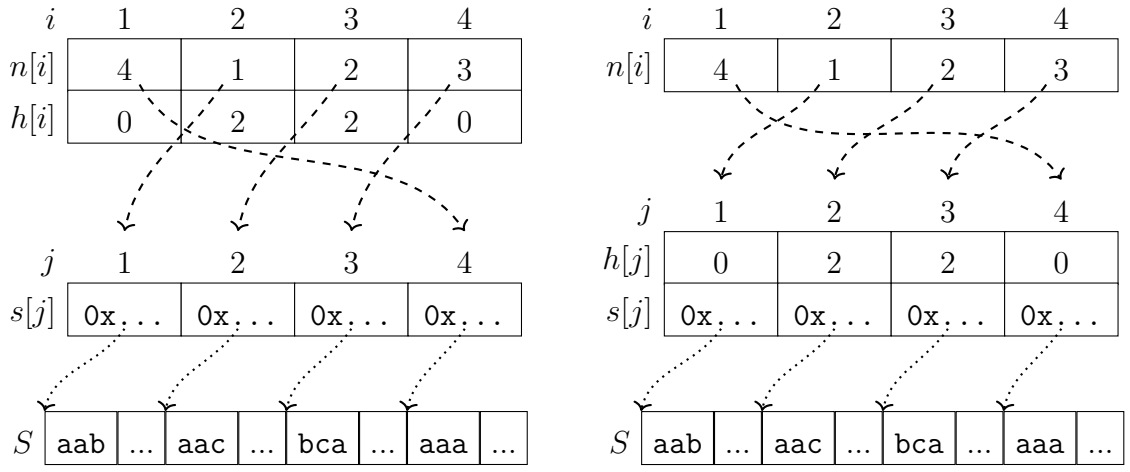
4.1.1. Ternary Comparison

The `LCP-Compare` operation introduced in Section 3.1.1 requires to select one of three cases by comparing two integer values. In order to do so, the algorithm needs to find out, which of the LCPs is smaller or if they are equal. A simple way to achieve this, is to execute two comparisons as shown in Algorithm 1. The first comparison checks if both LCPs are equal. Depending on the result, case 1 is executed or a second comparison finds the smaller LCP of both candidates, hence deciding between case 2 and case 3.

However, a more advanced solution uses only one comparison and detects the cases depending on CPU flags set during the internal compare operation. When executing a `CMP` assembly operation with parameters a and b, the following CPU flags are set: the `ZF` flag determines if the compared values are equal and the `SF` flag gives the ordering of the two parameters [Int14]. Evidently, these two flags contain all the information required to decide the three cases. Moreover, the assembly instruction sets contain special `jump` commands directly using those flags.

4.1.2. Memory Layout of LCP Tournament Tree

In Section 3.2 the LCP-aware tournament tree is described to store the index to the player that lost the match of node i inside the node as $n[i]$. The LCP value $h[i]$ of the player is stored in the node as shown in Figure 6 on page 24, resulting in the memory layout visualized in Figure 17a. The node's index i is used to index $n[i] = j$, pointing to the loser of the match, and the corresponding LCP $h[i]$, whereas index j indexes the pointers in s. Both, n and h are arrays of integers, s is an array of pointers to the start of the string sequence. In Figure 17, another array of pointers to the start of the string sequence's LCP array is omitted for better comprehensibility.



(a) LCP information stored in nodes. (b) Nodes only contain index of loser.

Figure 17: Different memory layouts of an LCP-aware tournament tree.

In contrast to that memory layout, one could also think of storing the current LCP value outside the tree, in the player's LCP value h'_i, whose memory is already allocated. Doing so reduces the memory footprint of the tournament tree by $K*64$ bit, since 64 bit are used for LCP values. But much more importantly, this reduces the number of *swap* operations required when a player wins a match and therefore has to be exchanged with its challenger. The design shown in Figure 17a requires to exchange both, the index $n[i]$ as well as the LCP value $h[i]$, whereas only the index swap would be needed, if the LCP value is not stored in the tree node.

Although the reduced number of swap operations can improve performance, practical analysis showed that the write operations to the player's possibly non-local LCP values, have great performance impact. Storing the current LCP value in the sequences, potentially having their memory located on another memory node, introduces great penalties, especially on NUMA systems.

As a result, the memory layout shown in Figure 17b is proposed. Here, the intermediate LCP values of the nodes are stored in a separate local array h in the tournament tree's data structure. In order to not store the LCP value in the tree's nodes, we index the array h with the player index j instead of the node index i. Therefore, when the player at node i changes its position in the tree, we only need to update $n[i]$ as the current LCP value, stored in $h[n[i]] = h[j]$, does not need to be moved. The minor calculation 'overhead' caused by the further indirection to access the LCP, has no impact, because memory access times dominate runtime. This approach allows us to combine the improvements achieved by reducing the number of swap operations and by storing the LCPs locally in low cache levels, which greatly improves performance.

4.1.3. Caching Distinguishing Characters

Further improvements can be achieved by exploiting the observation that it is possible to predict the first character to be compared by the character comparison loop of `LCP-Compare`. This character is the former distinguishing character, that means, the character at position $h + 1$, where $h = h_a = h_b$ is the common LCP value. By caching the distinguishing character, we again improve cache efficiency and reduce the number of accesses to remote memory nodes on NUMA systems. As the distinguishing character

Algorithm 6: LCP-Compare with Character Caching

```
Input: (a, s_a, h_a, c_a) and (b, s_b, h_b, c_b), with s_a, s_b two strings, h_a, h_b corresponding
       LCPs and c_a, c_b cached characters; assume ∃ string p with p ≤ s_i,
       h_i = lcp(p, s_i) and c_i = s_i[h_i + 1], ∀i ∈ {a, b}.
1 if h_a = h_b then                                          // Case 1: LCPs are equal
2     h' := h_a + 1
3     c'_a := c_a, c'_b := c_b                    // Assign cached characters to local variables
4     while c'_a ≠ 0 & c'_a = c'_b do                        // Execute character comparisons
5         c'_a := s_a[h'], c'_b := s_b[h'], h'++     // Increase LCP and load next characters
6     if c'_a ≤ c'_b then return (a, h_a, c_a, b, h', c'_b)          // Case 1.1: s_a ≤ s_b
7     else return (b, h_b, c_b, a, h', c'_a)                        // Case 1.2: s_a > s_b
8 else if h_a < h_b then return (b, h_b, c_b, a, h_a, c_a)          // Case 2: s_a > s_b
9 else return (a, h_a, c_a, b, h_b, c_b)                            // Case 3: s_a < s_b
Output: (w, h_w, c_w, l, h_l, c_l) where {w, l} = {a, b} with p ≤ s_w ≤ s_l, h_i = lcp(s, i) and
        c_i = s_i[h_i + 1], ∀i ∈ {w, l}
```

is always retrieved in the last step of the character comparison loop, it can directly be cached for the next time, the loop is called.

Algorithm 6 shows the new `LCP-Compare` procedure with character caching. The input arguments have been extended to supply the already cached characters. Likewise the output got additional parameters returning the new cached characters. In line 3 the cached characters are assigned to local variables, as only one LCP and therefore only one character can change during an execution of `LCP-Compare`. In addition, the loop in line 4, as well as the conditional statement in line 6 have been adapted to use the current cached characters.

In order to reuse the cached characters in further merges, the LCP tournament tree has been extended to take string sequences annotated with an LCP array and an array of corresponding cached characters. Furthermore the algorithm creates an array of cached characters for the output sequence.

Character caching becomes especially valuable in top-level merges on NUMA architectures. In the top-level merge most times only one character needs to be inspected to decide, which of both strings is lexicographically smaller. In these cases, accessing the string can completely be replaced by only accessing the cached characters.

Figure 18 shows the scheme of the extended LCP-aware tournament tree. The nodes

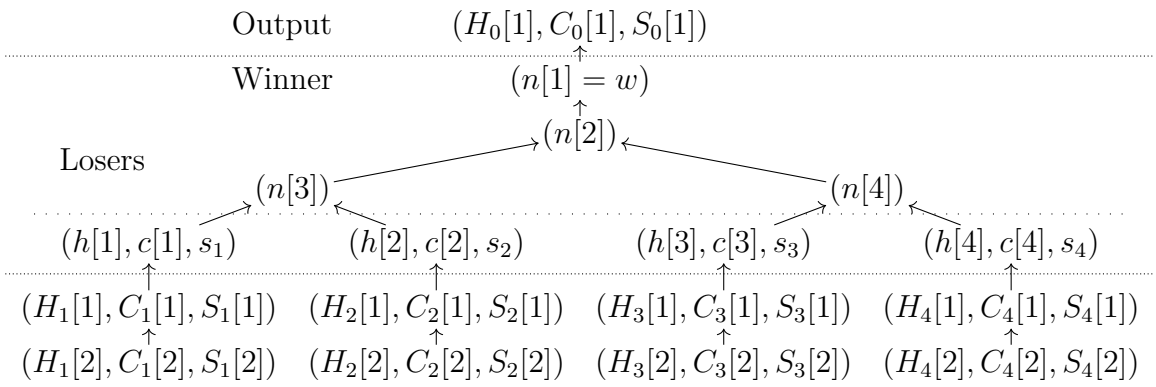


Figure 18: LCP-aware tournament tree with $K = 4$ plus LCP and character caching.

have been reduced to only contain $n[i]$, the index to the player that lost the game at that node. In contrast, input and output sequences have been extended to additionally contain an array C_i of cached characters. The players are extended to keep the current cached character as well as the current LCP. Although the players LCP and cached character are not stored in the nodes, they are part of the tournament tree's data structure, whereas the player's string is still only kept in its input sequence, as it is never changed.

4.2. Parallelization of K-Way LCP-Merge

In order to parallelize K-way LCP-Merge the merge problem will be split up into sub-tasks by either classical, binary or the LCP splitting algorithm described in Sections 3.3.1, 3.3.2 and 3.3.3. All three algorithms can easily be exchanged with each other or even a further one. Because the amount of work of a merge job depends on the number of strings and the length of the distinguishing prefixes, the required processing time can not be calculated beforehand. Therefore, dynamic load balancing is required to achieve good utilization of all processing units.

For easier combination of parallel K-way LCP-Merge with pS^5, we apply the same lightweight load balancing framework. It consists of a job queue, supplying associated threads with available tasks. To improve load balancing and reduce the number of splitting runs, classical and LCP splitting create more jobs than available threads at the start. In contrast, binary splitting creates only the needed number of jobs but is able to react more dynamically to idling threads. For all algorithms, working threads regularly check if another one is idling as the queue got empty. If such a situation is detected, a thread having enough work, starts splitting itself up into smaller sub-tasks and adds them to the queue. As trade-off between overhead of checking for idle threads and response time to idling threads, checking is only done about every 4000 outputted strings.

To prevent generation of too few and too small jobs resulting in a frequent need for splitting, only large jobs should be split. One way to find the biggest job in a distributed system, is to use an atomic variable storing the size of the largest one. All currently processed jobs regularly check if their remaining work is larger than the one in the counter. Only the biggest job decrements the counter when it finishes a part of its work, to adjust its remaining work size. If now an idle thread is detected, only the biggest job will split itself up.

However, this method requires an atomic variable, which is already expensive on multi-core systems, not to mention on NUMA architectures. Yet, the above method can be applied to a non-atomic variable with small adaptions. Since we do not require the biggest job to be split, but rather a fairly large one, the heuristic result achieved with this method works perfectly in practice.

When splitting the sequences of a merge job with either one of the splitting procedures, arbitrary numbers of non-empty sequences will occur in sub-tasks. For example, an initial merge job might have eight input sequences, whereas a subtask sometimes even consists of just one non-empty sequence. Clearly one could always apply a K-way LCP-Merge with K being the number of initial input sequences. However, merging overhead can be reduced by creating specialized merge jobs for different numbers of input sequences. Therefore a dedicated copy job is used whenever only one sequence remains. As the name predicts, it only copies the data of the input sequence to the output sequence. For two sequences, binary LCP-Mergesort is used, because it does not

require the overhead of the tournament tree. For every further number of sequences, a K-way LCP-Merge is used with K being the next larger power of two. To reduce splitting overhead, only K-way LCP-Merge jobs can be split up further, since copy and binary merge jobs tend to have smaller sizes.

We currently need parallel K-way LCP-Merge solely as top level merger to combine the separated work done by multiple instances of another sorting algorithm. Therefore we were able to optimize it by only outputting the sorted string sequences. The creation of the final LCP and cached character arrays are omitted because they are not needed after the merge. However, generating the LCP and cached character array would not require great modifications, since the contained algorithms already supply the needed data. Only one additional step would be needed at the end. During this step, the LCPs and cached characters of the connection points between different jobs need to be calculated separately. Due to time limitations, we leave this to future work.

4.3. Parallel K-Way LCP-Mergesort

With parallel K-way LCP-Merge described before, a parallel K-way LCP-Mergesort can be implemented. The work done by K-way LCP-Mergesort is divided into two steps as shown in Figure 19. In the first step, the unsorted input sequence is split into p equal-sized parts, with p being the number of available hardware threads. Each thread is then sorting one part of the input with sequential LCP-Mergesort. The second step is to apply the parallel K-way LCP-Merge with $K = p$ to combine the p sorted parts to one complete sorted sequence. Note, that the LCP-Mergesort used in step one can either be binary LCP-Mergesort or K-way LCP-Mergesort with an arbitrary K, as it is completely independent from the parallel K-way LCP-Merge applied in step two.

This approach requires top-level K-way LCP-Merge to merge a large number of sequences, clearly making the optimizations to the tournament tree important. Analysis showed that sequential K-way LCP-Mergesort performs best with $K = 64$ and becomes worse with higher numbers of input streams. This effect, explicable by cache behaviour, implies some limitations to the current approach, since it is not ideal to further increase the number of sequences.

During the first step, equal-sized parts are created, which is in fact a static load balancing strategy. As described in Section 4.2, this does not imply equal problem sizes

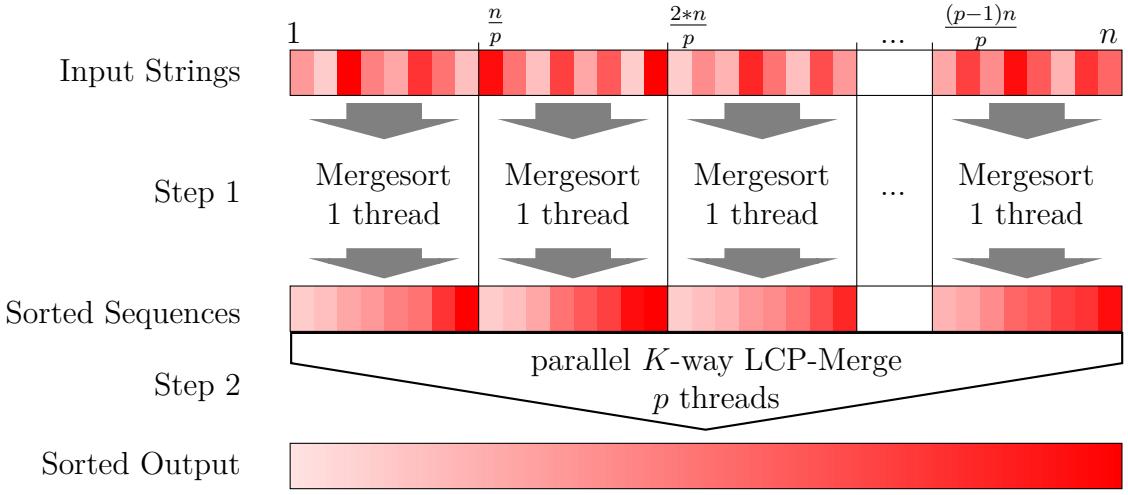


Figure 19: Scheme of Parallel K-way LCP-Mergesort.

and therefore some threads will probably finish sooner than others. Because the second step with parallel top-level K-way LCP-Merge can not start before all threads finished step one, some threads will have idling time.

4.4. NUMA Optimized pS^5

Parallel Super Scalar String Sample Sort (pS^5) [BS13] is a fast parallel sorting algorithm considering L2 cache sizes, word parallelism and super scalar parallelism. However, new architectures having large amounts of RAM are now mostly *non uniform memory access* (NUMA) systems. In these systems, RAM chips are separated onto different RAM banks called NUMA nodes. A processor now only has direct access to its *local* node, whereas access to *remote* nodes is achieved via an interconnection bus as shown in Figure 1 on page 15. Preliminary synthetic experiments showed memory access to remote NUMA nodes being 2-8 times slower than local memory access. These differences in latency and throughput, can be handled well by algorithms for external and distributed memory models.

To improve pS^5 on NUMA systems, a similar two-step approach is used like with parallel K-way LCP-Mergesort (Section 4.3). As visualized in Figure 20, the given input sequence is split up into m equal-sized parts, where m is the number of NUMA nodes. In step one, each part is sorted in parallel with pS^5 by $\frac{p}{m}$ threads. During data loading, it is possible to *segment* the data as equal-sized parts onto the different NUMA nodes. We now pin the threads of every pS^5 execution to the node, where its part of the input sequence is located. Therefore only local memory accesses are done by pS^5 preventing remote access penalties.

The second step is to merge the m sorted sequences and can be accomplished by applying a K-way merge with $K = m$. Hence, the top-level merge inherently requires memory accesses to remote NUMA nodes, those accesses should be minimized for maximizing performance. K-way LCP-Merge like described in Section 4.2 achieves that by exploiting known LCPs and caching of the distinguishing character. Moreover, by applying parallel K-way LCP-Merge we exploit parallelism.

Because K-way LCP-Merge requires not only the sorted sequences, but also the LCP and cached character arrays, pS^5 needs to be adapted. Since the LCPs and distinguishing characters are already retrieved internally, no significant performance penalty

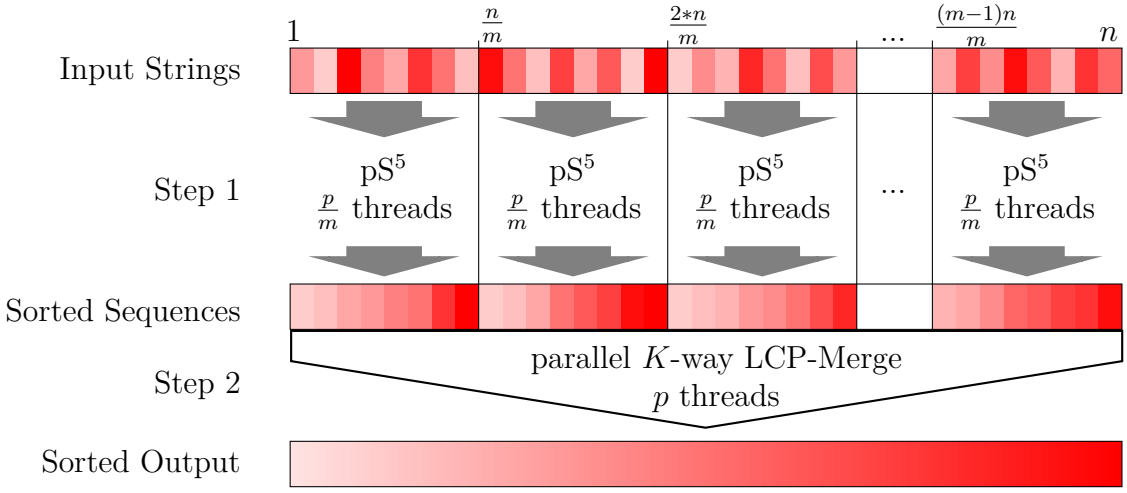


Figure 20: Scheme of NUMA optimized pS^5.

is introduced. Additionally, due to the fact that the LCPs can be stored in an already required array, only the array for cached characters increases the memory footprint.

In comparison to K-way LCP-Mergesort the number of sequences to be combined by the top level merge is much smaller, as only m instead of p streams need to be merged (our systems have $m \in \{4, 8\}$). Resulting from this, the special cases for merging just two sequences or copying only one sequence occur more often, which leads to faster merging performance. Moreover, since runtime of the job splitting and the merging done by the tournament tree, increases with K, there is an even bigger difference.

4.5. Further Improvements

Besides the optimizations presented before, some more came up during development. However, due to time constraints and scope limitations, not all of them were fully implemented. Therefore not all of the following proposals have been properly implemented and tested yet. We leave it to future work to optimize the implementations further.

4.5.1. Improved Binary Search

Binary search is an important algorithm used by classical and binary splitting. In order to divide a large merge job, splitter strings are searched in every input sequence, thus, separating work disjoint parts. Since the sequences are already sorted, binary search can be applied to find the splitting positions, as described in Section 3.3.1. Although binary splitting requires only a logarithmic number of string comparisons, those can still be very expensive for long strings. Moreover, the number of searches and therefore string comparisons increases linearly with the number of sequences to be merged and the number of jobs to be created. The combination of these aspects makes optimizing binary search an important task.

A way to improve performance of binary string search is to reuse LCPs calculated during the search [Ohl13]. The basic idea is that the minimum LCP of any string of an interval to the searched one, can be calculated from the LCPs of the strings at the borders of that area. Therefore, for a sorted sequence S, an interval $[a, b]$ with $1 \leq a \leq b \leq |S|$ and the searched string p, we have $\text{lcp}(S[i], p) \geq \min(\text{lcp}(S[a], p), \text{lcp}(S[b], p))$ for all $a \leq i \leq b$.

Algorithm 7: Improved Binary Search

`Input`: Sorted Sequence S and searched string p

```
1  l := 1, r := |S|
2  (h_l, f) := String-Compare(p, S[l], 0)        // Compare first string with searched one
3  if f ≤ 0 then return 1                        // If search string is smaller than all in S
4  (h_r, f) := String-Compare(p, S[r], 0)        // Compare last string with searched one
5  if f > 0 then return |S| + 1                  // If search string is larger than all in S
6  while r − l > 1 do                                                    // Run binary search
7      m := (r+l)/2, h' := min(h_l, h_r)         // Calculate middle position and known LCP
8      (h_m, f) := String-Compare(p, S[m], h')   // Compare strings, starting at h' + 1
9      if f ≤ 0 then (h_r, r) := (h_m, m)                    // Searched string is in left half
10     else (h_l, l) := (h_m, m)                             // Searched string is in right half
11 return r
```

`Output`: Index i, so that $S[j] < p$ for all $1 \leq j < i$.

Algorithm 8: String-Compare

Input: s_a, s_b and h, with $h \leq \text{lcp}(s_a, s_b)$

```
1 while s_a[h] ≠ 0 & s_a[h] = s_b[h] do        // Execute character comparisons
2     h++                                       // Increase LCP
3 return (h, s_a[h] − s_b[h])                   // Return difference of distinguishing characters
```

Output: (h, f), with $h = \text{lcp}(s_a, s_b)$ and $f(x) \begin{cases} < 0, & s_a < s_b \\ = 0, & s_a = s_b \\ > 0, & s_a > s_b \end{cases}$

This can directly be applied to binary string search. As usual, the search starts with $a = 1$ and $b = |S|$. There LCPs $h_a = lcp(S[a], p)$ and $h_b = lcp(S[b], p)$ can directly be calculated in a first check, if the search string is lexicographically smaller or larger than any string of the sequence. After that, checking the middle position $m = \frac{a+b}{2}$ can be done by starting at the characters at position $\min(h_a, h_b) + 1$. The new LCP calculated by this string comparison will then be assigned to the either h_a or h_b, depending on which half is to be inspected further. Algorithm 7 implements this strategy to create a faster binary search and is used in our implementations of classical and binary splitting.

4.5.2. K-Way LCP-Merge with Multi-Character Caching

In order to optimize cache and NUMA transfer-efficiency, character caching for K-way LCP-Merge was introduced in Section 4.1. By extending character caching to multi-character caching, the gain can be increased in exchange for linearly increased memory usage.

Instead of caching only a single character (one byte in size), a super-character, consisting of w single characters, is read, compared and cached for further usage. To extend `LCP-Compare` with character caching, shown in Algorithm 6, loading of characters in line 5 needs to be adapted to load w characters as one super-character. This means, the w characters starting from h' need to be loaded, with the first character stored in the most significant byte, the second in the second-most significant byte and so on. If a string does not have enough characters to fill the super-character, it is filled up with zeros instead. Additionally, the current LCP h' needs to be increased by w instead of just one. Doing so makes it possible to execute the equality check in the loop like before.

However, if the two super characters c'_a and c'_b are not equal, the LCP of them needs to be calculated. The LCP of two super characters x and y is given by $\text{lcp}_{super}(x, y) = min(high\_zero\_bytes(x \oplus y), w - low\_zero\_bytes(x))$. Whereas the first part, of the *min* clause counts the number of equal characters at the beginning of the super-character, the second part ensures that the LCP of two equal strings is not too long (this case might occur, when equal strings reach their ends).

Whenever the mentioned LCP of the first unequal super-characters is greater than zero, they can not directly be returned as new cached characters. Since we compare the cached super-characters, we need to ensure the correct contained characters are compared with each other. Therefore the first cached character always needs to be the distinguishing character, that means, the character at position $h + 1$ of the string. An easy way to accomplish this would be to just load the correct super-character when exiting `LCP-Compare`, if required. However, this increases the number of memory accesses

Algorithm 9: LCP-Compare Caching w Characters

Input: (a, s_a, h_a, c_a) and (b, s_b, h_b, c_b), with s_a, s_b two strings, h_a, h_b corresponding LCPs and c_a, c_b cached characters; assume $\exists$ string p with $p \leq s_i$, $h_i = \texttt{lcp}_{super}(s, s_i)$, $\forall i \in \{a, b\}$.

```
1  if h_a = h_b then                                          // Case 1: LCPs are equal
2      h' := h_a                                              // Variable with current LCP
3      mask0Bytes := max(lowZeroBytes(c_a), lowZeroBytes(c_b))  // Number of unused Bytes
4      mask := mask_w(mask0Bytes)   // Mask: #mask0Bytes low 0 Bytes, rest 0xFF
5      c'_a := c_a & mask, c'_b := c_b & mask                 // Mask cached with common mask
6      isEnd := false   // Due to masking, cached can not contain end of string byte
7      while ¬isEnd & c'_a = c'_b do                          // Execute super character comparisons
8          h' := h' + lcp_super(c'_a, c'_b)                   // Increase current LCP value
9          c'_a := loadCharacters_w(s_a, h')   // Load next super character from s_a at h'
10         c'_b := loadCharacters_w(s_b, h')   // Load next super character from s_b at h'
11         isEnd := (lowZeroBytes(c'_a) > 0)                  // Is at least one low byte 0?
12     Δlcp := lcp_super(c'_a, c'_b)                          // Calculate LCP of last super characters
13     h' := h' + Δlcp                                        // Increase LCP value accordingly
14     if c'_a ≤ c'_b then                                    // Case 1.1: s_a ≤ s_b
15         c'_b = c'_b ≪ (Δlcp * 8)   // Remove equal characters from super character
16         return (a, h_a, c_a, b, h', c'_b)
17     else                                                   // Case 1.2: s_a > s_b
18         c'_a = c'_a ≪ (Δlcp * 8)   // Remove equal characters from super character
19         return (b, h_b, c_b, a, h', c'_a)
20 else if h_a < h_b then return (b, h_b, c_b, a, h_a, c_a)   // Case 2: s_a > s_b
21 else return (a, h_a, c_a, b, h_b, c_b)                     // Case 3: s_a < s_b
```

Output: $(w, h_w, c_w, l, h_l, c_l)$ where $\{w, l\} = \{a, b\}$ with $p \leq s_w \leq s_l$, $h_i = \texttt{lcp}_{super}(p, i)$, $\forall i \in \{w, l\}$

to almost the same amount as without character caching.

Like shown in Algorithm 9, another way is to use bit-shifting and bit-masking to be able to use a cached super-character until its last contained character is used up. Although this requires more calculations to be executed, the number of memory accesses can be decreased significantly. In combination with the fact that bit operations are executed very fast, the given algorithm improves performance of K-Way LCP-Merge especially on NUMA architectures. Please note that the implementation of Algorithm 9 can still be optimized, but is kept simpler to ease comprehensibility.

Lines 3 to 5 of Algorithm 9 ensure the cached super-characters have equal length and can be compared at all. Due to the reuse of non-complete super-characters (see lines 15 and 18), an arbitrary number of lower bytes of cached super-characters may be zero. In order to compare the two cached super characters, their number of *low zero bytes* must be equal. Therefore the low zero bytes are counted and the maximum is selected to create a bit mask, used to shorten the longer super-character to the length of the shorter one.

Because this shortened implementation results in losing the information if a super-character contained an *end of string* character (also a zero byte), *isEnd* must be set to $false$ in line 6. In the character comparison loop, the current LCP h' is increased by the LCP of the super-characters (line 8), the new super-characters are loaded (lines 9

and 10) and it is checked, if string s_a reached its end (line 11).

The LCP of the last super-characters (Δlcp) is calculated in line 12. It is first added to the current LCP h' to calculate the complete length of the LCP. Afterwards, in lines 15 and 18, it is used to remove the first Δlcp equal characters ensuring the first character of the super-character to be the distinguishing character.

5. Experimental Results

We implemented NUMA-aware pS^5 and parallel versions of K-way LCP-Mergesort. Both can be run with the three described splitting algorithms for the top-level K-way LCP-Merge. Additionally, the original pS^5 implementation is included in the test set, as fastest parallel reference algorithm. The performance of the various algorithms is discussed in Section 5.4. The implementations, the test framework and most input sets can be downloaded from http://tbingmann.de/2013/parallel-string-sorting.

5.1. Experimental Setup

Our implementations have been tested on a IntelE5 platform having four NUMA nodes and an AMD48 platform with eight NUMA nodes. The exact properties of the hardware are listed in Table 1. Both systems are running Ubuntu 12.04 LTS with kernel version 3.2.0 and all programs have been compiled using gcc 4.6.3 with optimizations **-03 -march=native**.

Name	Processor	Clock [GHz]	Sockets × Cores × HT	Cache: L1 [KiB]	L2 [KiB]	L3 [MiB]	RAM [GiB]
IntelE5	Intel Xeon E5-4640	2.4	4 × 8 × 2	32 × 32	32 × 256	4 × 20	512
AMD48	AMD Opteron 6168	1.9	4 × 12	48 × 64	48 × 512	8 × 6	256

Name	Codename	Memory Channels	NUMA Nodes	Interconnect
IntelE5	Sandy Bridge	4 × DDR3-1600	4	2 × 8.0 GT/s QP
AMD48	Magny-Cours	4 × DDR3-667	8	4 × 3.2 GHz HT

Table 1: Hardware characteristics of experimental platforms, see [BES14].

In order to separate different runs, the test framework forks each execution as a child process. Especially the influences caused by heap fragmentation and lazy deallocation, made this step important. The input data is loaded before forking the actual sort process and allocates exactly the specified amount of RAM. It is shared with the child process as read-only dataset. In contrast, the string pointer array is generated in the forked process by linearly scanning the input data for *end of string* characters.

Time measurement is done via clock_gettime() and only includes execution of the sorting algorithm. Since some algorithms have deep recursion depths, stack size has been increased to 64 MiB. When executing NUMA-aware algorithms, the input sequence has been split up into equal-sized parts, with each of them located on one NUMA memory bank. Sorting threads are then pinned to the NUMA node holding the memory they are processing, which enables node-local memory accesses. Further allocations are also done on node-local memory. Due to the distribution of used memory onto all NUMA nodes, no executions with less threads than NUMA nodes are considered for these algorithms.

In contrast, for executing non NUMA-aware algorithms, memory allocation was interleaved across all memory banks by using the default allocation method. Threads are not pinned to specific nodes. Instead, the default Linux task scheduling system is used.

For verifying the outputted list of string pointers, generated by a sorting algorithm, a first check ensures that the output is a permutation of the input. Afterwards, a

validation of the sort order is achieved by checking that strings are in non-descending order.

However, because only the direct algorithm execution times are measured, it arises the question, if this is a valid decision. The main concern is that memory deallocation and defragmentation is done lazily in heap allocators and kernel page tables, most notable when running two algorithms consecutively. Running the sorting algorithms in isolated forked processes effectively prevents that. Yet, for real applications, these aspects also need to be considered in future work.

Table 2 lists the analysed algorithms with their name used in the following plots, as well as a short description of them.

Name	Description
pS^5-Unroll	Original parallel super scalar string sample sort implementation with interleaved loop over strings, unrolled tree traversal, caching multikey quicksort and insertion sort as base sorter as introduced by Timo Bingmann [BS13].
pS^5-Unroll + BS-Merge pS^5-Unroll + CS-Merge pS^5-Unroll + LS-Merge	Modified pS^5-Unroll implementation, outputting LCP array and cached characters, made NUMA aware as described in Section 4.4, using either *binary*, *classical* or *LCP* splitting (BS, CS or LS).
pLcpMergesort + BS-Merge pLcpMergesort + CS-Merge pLcpMergesort + LS-Merge	Parallel LCP-Mergesort, as described in Section 4.3, using either *binary*, *classical* or *LCP* splitting (BS, CS or LS).

Table 2: Name and Description of tested parallel string sorting algorithms.

5.2. Input Datasets

For our tests, we selected the following datasets, all having 8-bit characters. Their most important characteristics can be found in Table 3.

URLs contains all URLs found on crawled web pages via breadth-first search starting from our institute's website. The protocol name is included.

Random from [SZ04] is a set of strings of length $[0, 20)$. The characters are taken from a subset of the ASCII alphabet in $[33, 127)$. Length and characters are both chosen uniformly at random.

GOV2 is a TREC test collection containing 25 million HTML pages, MS Word and PDF documents retrieved from websites, having a .gov top-level domain. For our string sorting, we consider the whole content for line-based string sorting, concatenated by document id.

Wikipedia is an XML dump obtained from `http://dumps.wikimedia.org` on the 2012-06-01. Since the XML data is not line-based, suffix-sorting is performed on this input.

Sinha DNA is a test set used by Ranjan Sinha [SZ04] to test burst sort. It contains genomic strings of a length of nine characters from the DNA alphabet. Although its size of 302 MiB is rather small in comparison to our other test sets, we include it, due to its extremely small alphabet of just four characters.

The chosen inputs represent real-world datasets, but also generate extreme results when sorted. Whereas Random has a very small average LCP, both URLs and Sinha's DNA have large ones. On the other hand, GOV2 is a test set with general text containing all ASCII characters. In contrast, Sinha's DNA has a very small alphabet. By suffix-sorting the Wikipedia test set, we get a very large sorting instance needing only little memory for characters.

As our large input sets do not fit into the main memory of all our machines, we only sort a large prefix of the input containing the strings $[1, n]$. This allows us to sort parts of the input sequences matching the available RAM and time.

Name	n	N	$\frac{D}{N}$ (D)	$\frac{L}{n}$	$\lvert\Sigma\rvert$	avg. $\lvert s\rvert$
URLs	1.11 G	70.7 Gi	93.5 %	62.0	84	68.4
Random	∞	∞	–	–	94	10.5
GOV2	11.3 G	425 Gi	84.7 %	32.0	255	40.3
Wikipedia	83.3 G	$\frac{1}{2}n(n+1)$	(79.56 T)	954.7	213	$\frac{1}{2}(n+1)$
Sinha DNA	31.6 M	302 Mi	100 %	9.0	4	10.0

Table 3: Characteristics of the selected input instances, see [BES14].

5.3. Performance of Splitting Methods

In Section 3.3, parallel K-way LCP-Merge has been introduced, which can be used with the *classical splitting*, *binary splitting* or *LCP splitting* algorithms. In this section, we report on our experiments, regarding the three splitting methods, to evaluate differences and advantages. The measurements were executed on the IntelE5 and AMD48 platforms, shown in Table 1 on page 45.

Since this section focuses on comparing the three splitting algorithms, parallel LCP-Mergesort (described in Section 4.3) is used as base sorter, leaving all parts, but the splitting algorithm, the same between executions. All graphs in Figures 21, 22, 23 and 24 visualize the median of five executions of parallel LCP-Mergesort with the respective splitting algorithm. In Figures 21 and 22 Sinha's complete DNA test set has been sorted, one time with the IntelE5 and the other time with the AMD48 platform. In addition, Figures 23 and 24 display the results of sorting 20 GiB of the URLs on both test systems.

These two test sets have been chosen due to their greatly differing characteristics. In contrast to Sinha's DNA test set, whose strings are all ten characters long and of a small alphabet, the URLs test set has much longer strings, longer average LCPs, a usual alphabet and is more extensive. Especially the input size is an important factor, since parallel top-level LCP-Merge is to be used for making pS^5 NUMA-aware. However, NUMA awareness is only important for larger input sets, requiring NUMA systems at all.

In all four figures, the values of the graphs are displayed over the number of threads available for sorting, allowing to evaluate the scaling qualities of the different splitting algorithms. Whereas graph a) of Figures 21, 22, 23 and 24 shows the overall runtime, the plot b) displays only the runtime of the top-level merge, which itself contains the total time consumed by the splitting algorithm, visualized in graph c). Plot d) shows the number of merge jobs created by the splitting algorithm and graph e) draws the time required to create a job, which is calculated as the total splitting time over the number of created jobs.

5.3.1. Splitting Analysis on Sorting 302 MiB Sinha DNA

The effects of the special properties of Sinha's DNA test set, can be seen in Figures 21 and 22. Here, almost no dynamic load balancing is required. This is caused by the fact that D, the sum of the distinguishing prefixes, contains all characters and each string has the exact same length of ten characters. Moreover, the small alphabet contains only four characters. However, because of the high costs of splitting runs with classical splitting and LCP splitting, more jobs than available processing units are created. This strategy is very important for other test sets, requiring dynamic load balancing and is the cause for the roughly linearly increasing number of created jobs, as observed in the fourth graph. In contrast, binary splitting is able to directly adapt to the little need for additional merge jobs, resulting in less time required by the splitting algorithm, a faster top-level merge and the difference in the overall sorting runtime.

Likewise, the sorting runtime difference between classical splitting and LCP splitting

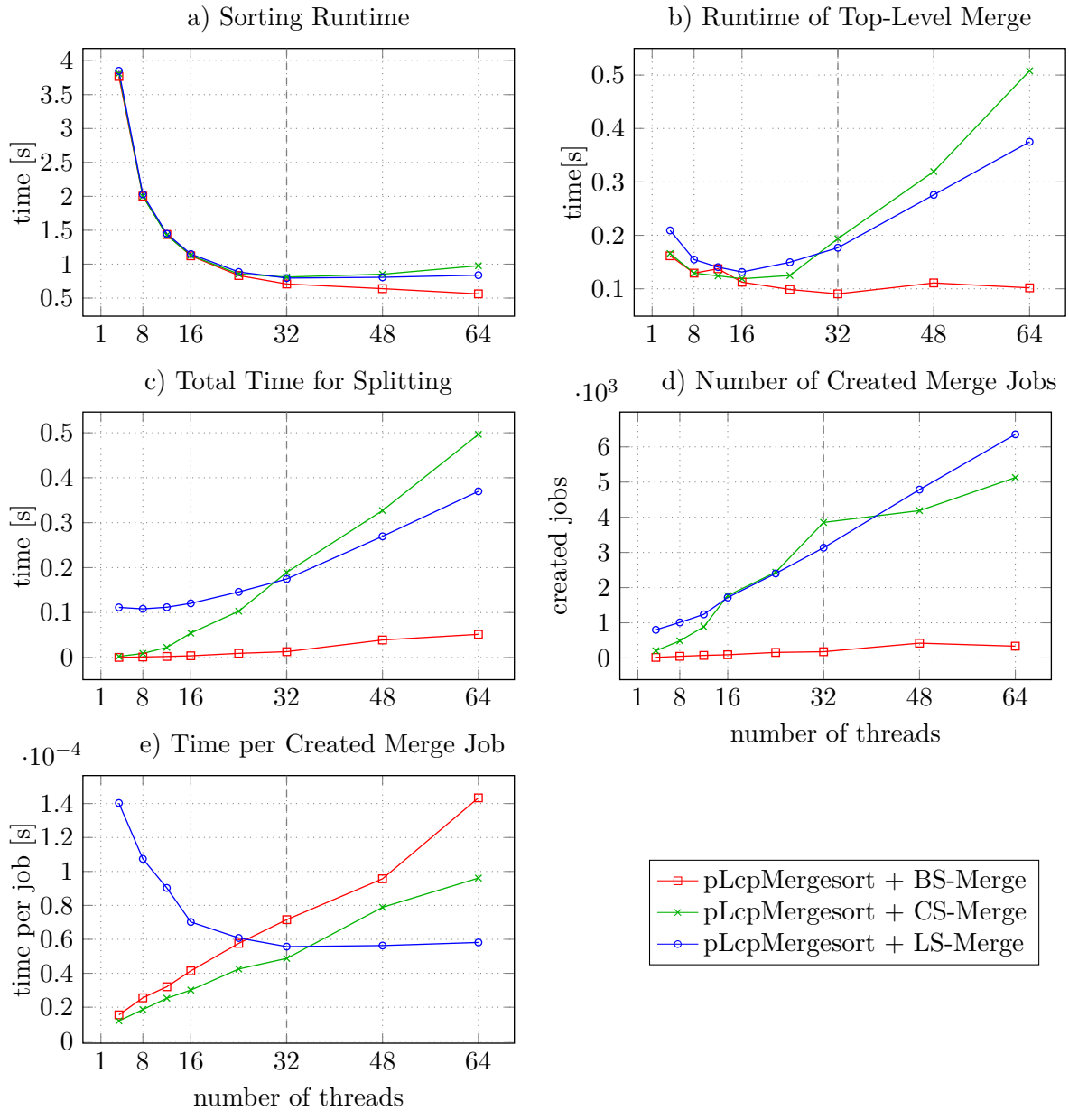


Figure 21: Analysis of splitting algorithms on IntelE5 sorting 302 MiB Sinha DNA.

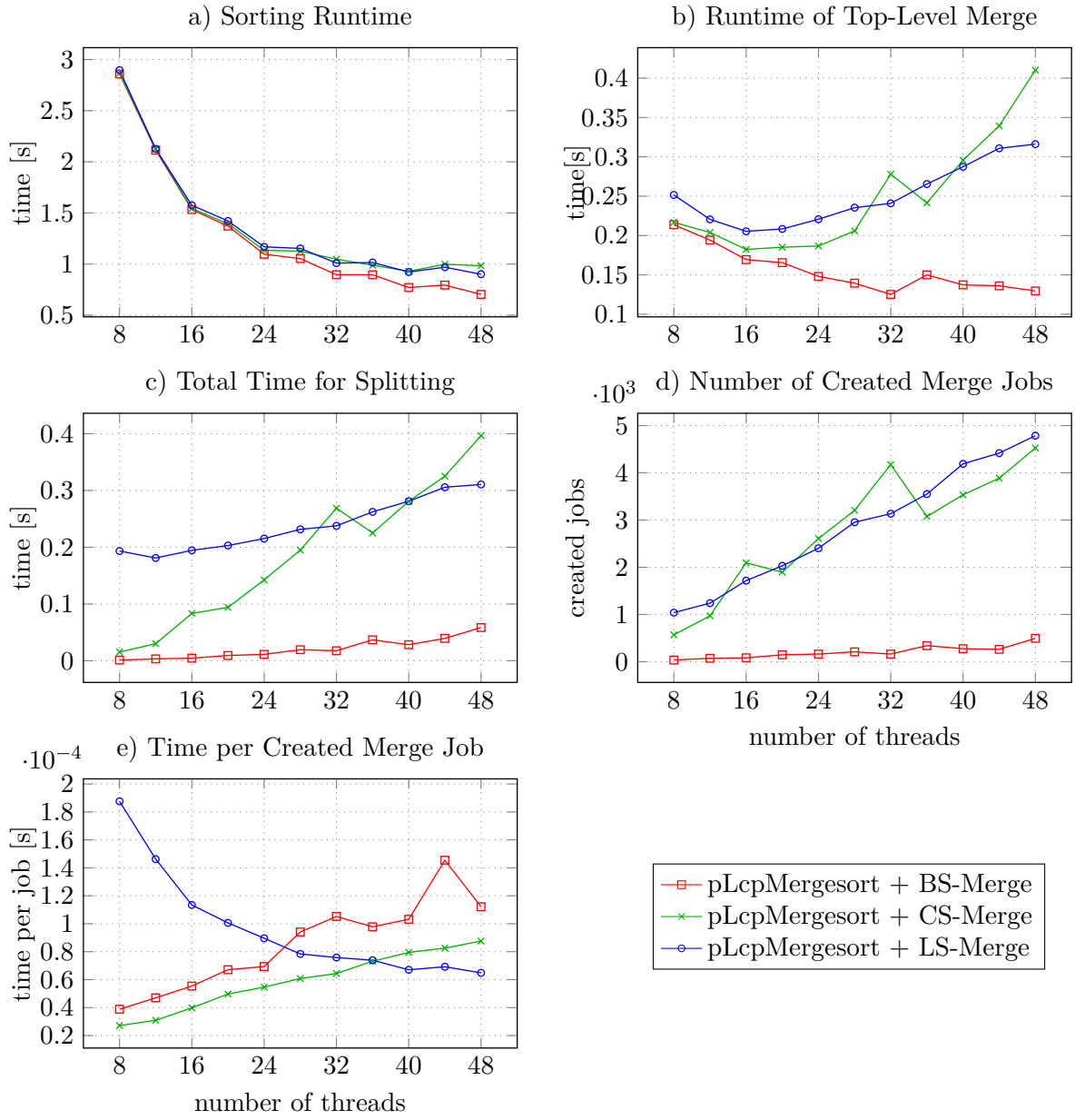


Figure 22: Analysis of splitting algorithms on AMD48 sorting 302 MiB Sinha DNA.

is also based on the difference of the splitting runtime seen in Figure 21c. Due to the small size of Sinha's DNA test set, linearly scanning the LCP array, like it is done by LCP splitting, is more efficient than doing a binary search for splitters.

Moreover, the runtime of LCP splitting does not grow much with an increased number of created merge jobs and available threads. Like shown in the graph e), the time per job of LCP splitting decreases with a growing number of threads, whereas the durations rise for classical and binary splitting. This can be explained by the fact, that LCP splitting always has to scan the exact same number of LCP values, independent of the number of threads and therefore the number of input sequences of the top-level merge. More precisely, LCP splitting has high constant costs, and experiences only little increases depending on the number of created jobs or available input sequences.

In contrast, binary and classical splitting need to use binary search to find every splitter in an increased number of sequences. Although these sequences are shorter and the overall length remains the same, the runtime increases, since the binary search only takes

logarithmic time depending on the length of the input. Due to $p \cdot \log(n/p) > \log(n)$, with p being the number of processing elements and n the overall number of strings, p searches on sequences of length $\frac{n}{p}$ are more expensive than one search on a sequence of length n.

To sum up the observations of Figures 21 and 22, binary splitting has an advantage, because it does not need to work with fixed oversampling factors, but adjusts fully dynamically to the input set's requirements. LCP splitting works well with small input sizes and can easily create a large number of jobs in test sets with low average LCP. In contrast, classical splitting shows an increasing runtime with a growing number of jobs to be created. Although its costs for creating jobs are much smaller compared to the ones of binary splitting, its fixed oversampling factor causes the creation of too many jobs, resulting in an increased runtime.

Another important observation is that the runtime behaviour of the three splitting algorithm's is very similar on both, the IntelE5 and the AMD48 platforms. Even though the two platforms have highly differing specifications and memory performances between local and remote NUMA memories, the splitting performance is not effected significantly.

5.3.2. Splitting Analysis on Sorting 20 GiB URLs

In contrast to Sinha's DNA, the URLs test set is much larger, has an alphabet of 84 characters and an average LCP length of 62. With the input being more skewed, dynamic load balancing is much more important and therefore more independent merge jobs will need to be created.

The graph a) of Figures 23 and 24 shows that classical and binary splitting outperform LCP splitting. On the IntelE5 platform, there is a gap of about 5 seconds in the overall sorting time, which is even larger on the AMD48 system. Moreover, the distance between the algorithms remains nearly constant with increasing number of threads. Like plot b) shows, the gap is primarily caused by the difference of the top-level merge runtimes, which themselves are mainly determined by the runtimes of the splitting algorithms. Thus, the runtime differences can be explained by the the high fixed costs of LCP splitting, caused by linearly scanning the LCP arrays with a combined length of 325 million entries. This induces high constant costs that are not changing with increasing number of threads and sequences, seen in graph c).

In contrast, binary and classical splitting have almost equal overall sorting performance. Because the input set requires real dynamic load balancing, binary splitting needs to create more jobs. This is why the difference in the number of created jobs between both methods is much smaller now. However, the fixed oversampling rate of classical splitting still yields about twice as much merge jobs as required. But since the costs of the actual merging part are much larger now, the small difference in splitting time, seen in the graph c), shows no effect to the resulting overall sorting time.

Again, both evaluated platforms show about the same behaviour, regarding the splitting algorithms. Thus suggesting, mostly the input set's characteristics determine the splitting algorithm's performance. Moreover, the difference between classical, binary and LCP splitting between these two very differing test sets reinforces the need to select an appropriate splitting method for the considered test set.

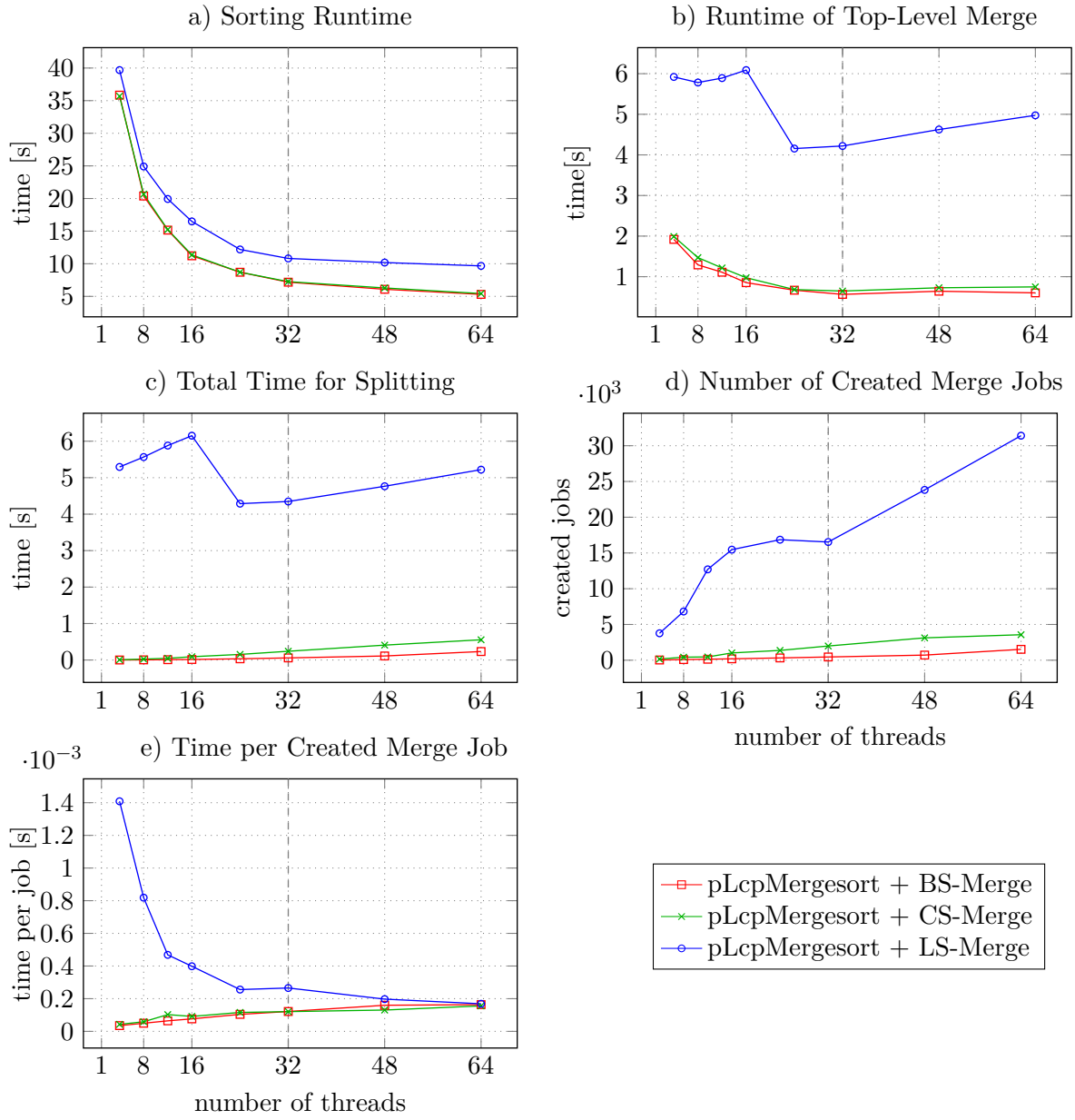


Figure 23: Analysis of splitting algorithms on IntelE5 sorting 20 GiB URLs.

5.4. Performance of Parallel Algorithms

In this section, focus is put on comparing an unmodified pS^5-Unroll [BES14] implementation with our newly presented parallel K-way LCP-Mergesort (see Section 4.3) and our NUMA optimized pS^5 (see Section 4.4).

The graphs plotted in Figures 25 and 26 show the speedup of the algorithms over the best sequential execution of pS^5-Unroll. Whereas Figure 25 shows the results for all test sets listed in Table 3 on the IntelE5 machine, Figure 26 visualizes the results for the AMD48 system. The hard- and software specifications of both platforms are listed in Section 5.1.

In the first graph of Figures 25 and 26, the speedups, when sorting 20 GiB of the URLs test set, are shown. It is clearly visible that NUMA-aware pS^5, as well as parallel LCP-Mergesort with classical and binary splitting outperform the original pS^5 implementation by almost a factor of two. Although NUMA-pS^5 performs better than

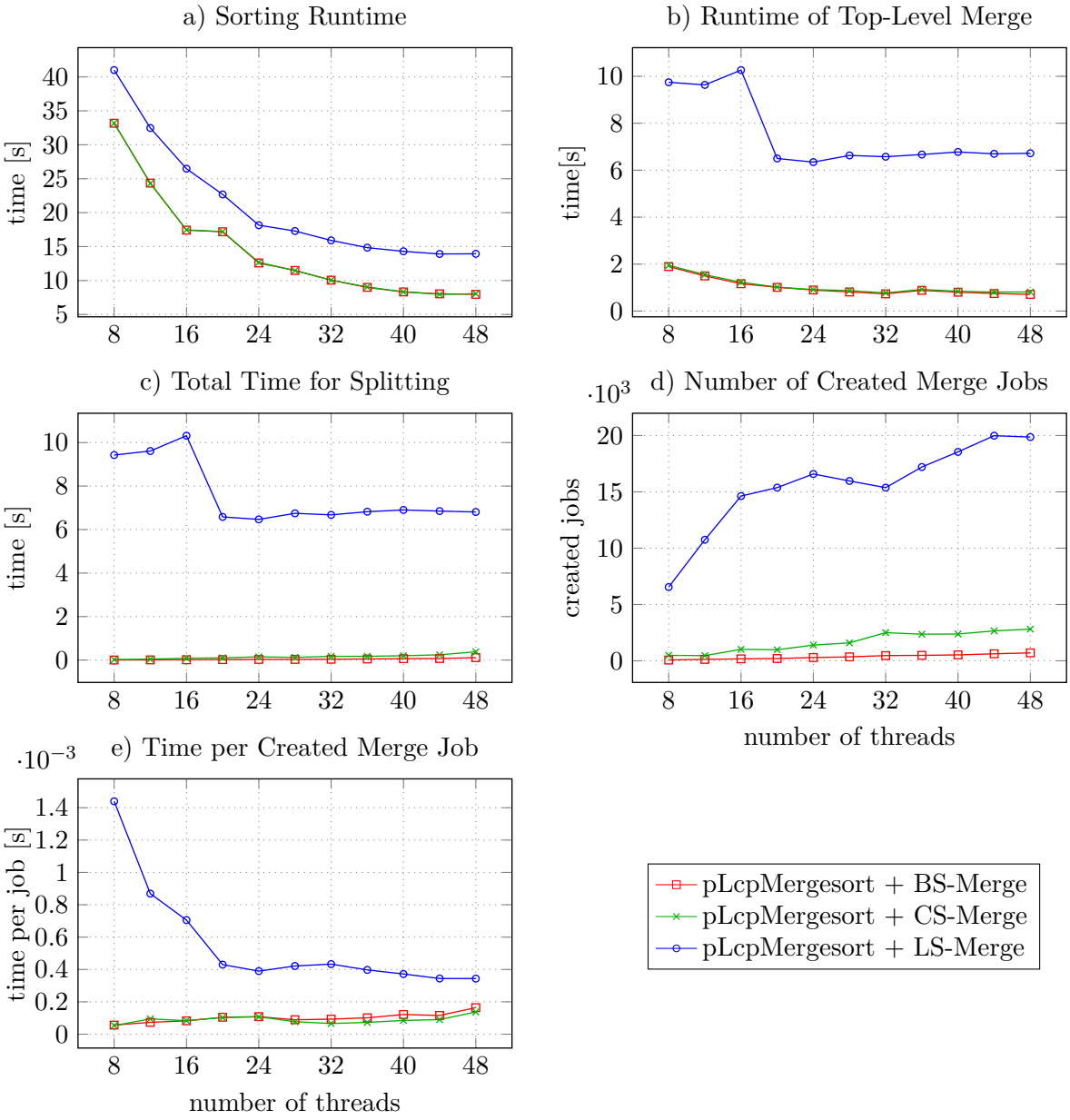


Figure 24: Analysis of splitting algorithms on AMD48 sorting 20 GiB URLs.

parallel LCP-Mergesort with less than 64 threads, the latter one is able to catch up with an increased number of threads. The observations of Section 5.3.2 become visible again, since classical and binary splitting perform equally well and LCP splitting hits great penalties for scanning the large LCP array. Especially the last effect can be seen with most of the longer input sizes.

For the Random test set, shown in the second graph, we get a partly different situation. Again, NUMA-pS^5 performs very well with speedups of 25 compared to 17 achieved by the original implementation. However, parallel LCP-Mergesort performs much worse than the pS^5 implementations. This is mainly caused by the low average LCP of a random test set, making it impossible for LCP-Mergesort to effectively exploit the LCPs. Classical and binary splitting perform quite similar again.

When looking at the third plot, kind of a mix of the Random and URLs result can be observed. Whereas NUMA-pS^5 dominates again with significantly higher speedups, parallel LCP-Mergesort is just slightly better than the original pS^5 implementation.

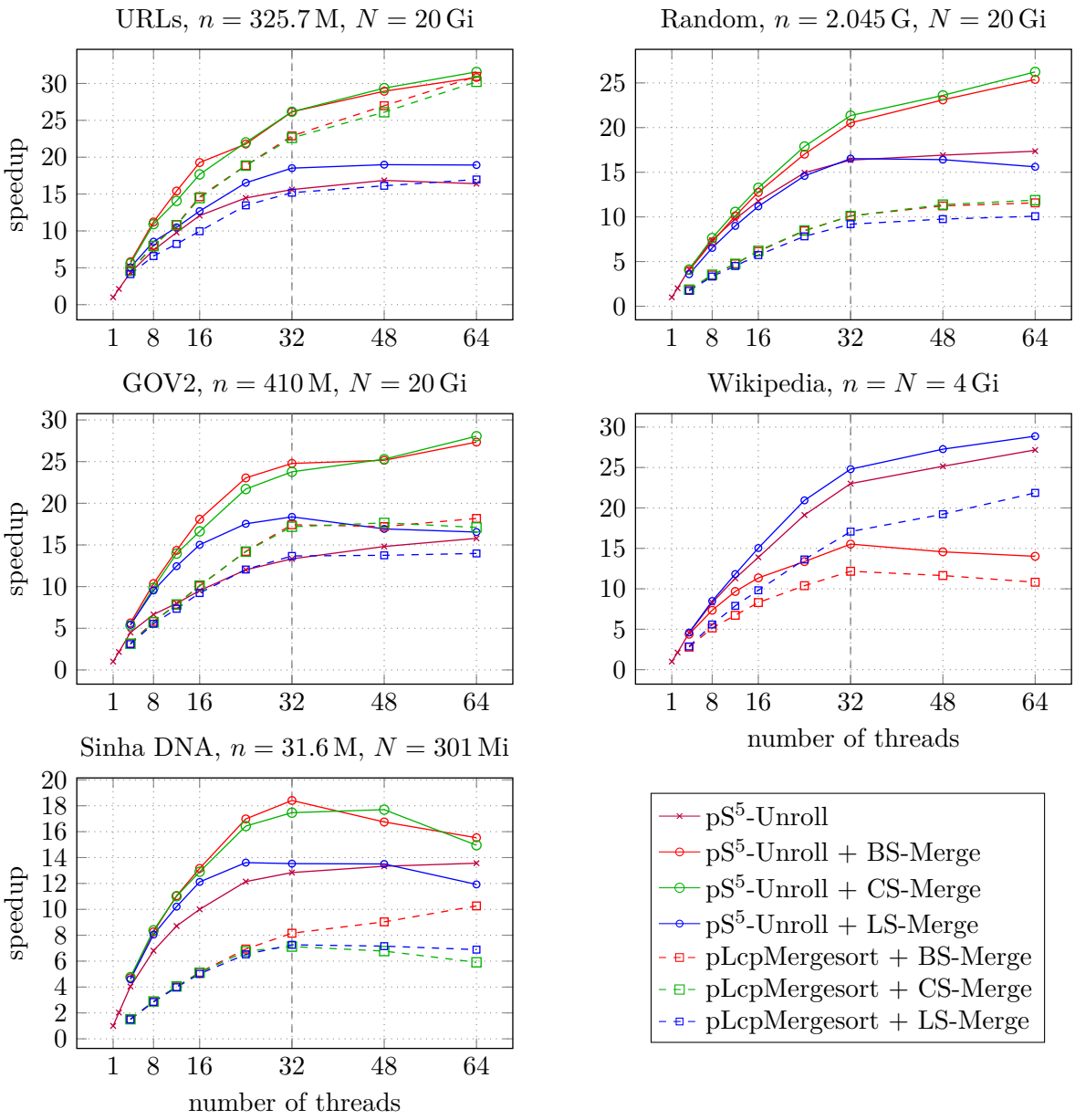


Figure 25: Speedup of parallel algorithm implementations on IntelE5.

Once more, this can be explained by the average LCP of the input set, since GOV2 has an average LCP of 32, which is about half as long as the one of URLs. Although, this is enough for parallel LCP-Mergesort to outperform the original pS^5, NUMA optimized pS^5 performs way better.

The Wikipedia test set, is very different from the others, because we do suffix sorting for a set of strings with an average length of about 955 characters. As explained before, LCP splitting experiences great penalties for linearly scanning very long LCP arrays. However, although the suffix sorted Wikipedia test set has 4 Gi strings, LCP splitting performs way better than binary splitting. In fact, classical splitting performs so bad, that it can not be shown in this plot (for 64 threads, classical splitting required about 50 times longer than LCP splitting). The bad performance of classical and binary splitting is mainly caused by the long average string length of the input set. The binary search for splitters used by these splitting methods, requires to execute many string compare operations, which are seriously slow for strings of such lengths.

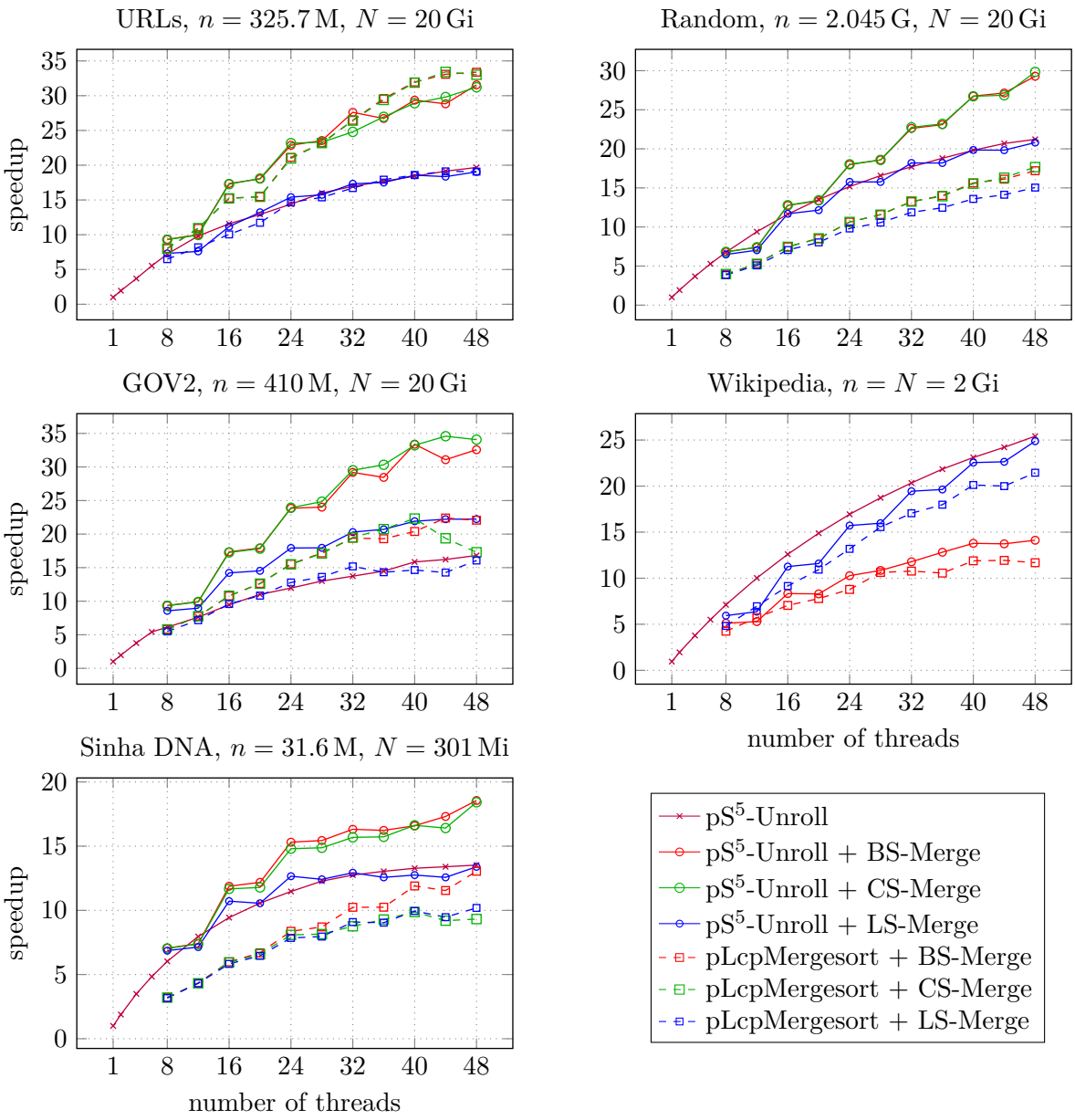


Figure 26: Speedup of parallel algorithm implementations on AMD48.

Moreover, since suffix sorting is done, the binary search needs to compare a lot of almost equal strings, requiring to compare almost all characters of those strings.

Furthermore, the Wikipedia test set is the only one showing a real difference in the algorithm's performances between the IntelE5 and the AMD48 platform. Whereas NUMA-pS^5 is able to slightly outperform original pS^5 on IntelE5, the latter one is slightly better on the AMD48 system. However, as all the splitting methods have troubles with either the large number of strings, like LCP splitting, or the great length of the strings, like classical and binary splitting, parallel top-level LCP-Merge spends large amounts of time on splitting, instead of actual merging.

The speedup plots of Sinha's DNA test set are very similar to the ones of the Random test set, with NUMA-pS^5 outperforming the original pS^5, which itself is better than parallel LCP-Mergesort. Again, the short average length of the LCPs is handled better by pS^5 than parallel LCP-Mergesort. Regarding the performance of different splitting methods, we see LCP splitting to be much better for parallel LCP-Mergesort than for

NUMA-pS^5. This is caused by the fact that the top-level merge for NUMA-pS^5 only has to merge 4 sorted sequences. In contrast, for parallel LCP-Mergesort, the number of sequences to be merged is equal to the number of available threads. As an increased number of sequences increases the costs for classical and binary splitting, LCP splitting becomes competitive for parallel LCP-Mergesort.

All graphs have in common that the growth of the speedup is reduced for increasing number of threads. This is caused by the limited memory bandwidth, restraining performance for high numbers of threads. Therefore not the processing power, but memory bandwidth is the restricting aspect to our algorithms. Moreover, as our test platforms are NUMA architectures, the graphs also show their dramatic influence to sorting performance. This can be seen from two aspects. The first one is the large impact our NUMA improvements have to pS^5. Although pure LCP-Mergesort is worse than original pS^5 in some test sets, NUMA-aware pS^5, utilizing the parallel LCP-Merge, is much faster than the original one in exactly those tests. The second aspect is visible in the speedup plots of the AMD48 platform, shown in Figure 26. As this system has 8 NUMA nodes, the performance of the illustrated algorithms is only slightly improved when adding four more threads to a thread count dividable by eight. For example, the speedup of 'pS^5-Unroll + BS-Merge' is not improved much when using 20 instead of 16 threads, which leads to the stairs in the graphs of NUMA-aware algorithms. In contrast, 'pS^5-Unroll', run with interleaved memory allocation, has a continuous speedup curve.

The absolute runtimes of all our speedup experiments are shown in Appendix A.

6. Conclusions

In this bachelor thesis, Ng's binary LCP-Mergesort [NK08] has been extended to introduce a K-way LCP tournament tree, multiway LCP-Mergesort, as well as a parallel top-level multiway LCP-Merge and therewith a parallel LCP-Mergesort. Moreover, the parallel top-level LCP-Merge has been used to optimize Timo Bingmann's pS^5 [BS13] for NUMA architectures.

Our experiments with various test sets, each emphasizing different important aspects, demonstrate that parallel LCP-Merge can be utilized for easy parallelization of most sequential string sorters. To be able to apply the same parallelization scheme as with parallel LCP-Mergesort, the sequential string sorter only needs to provide the LCP array. However, most times, this information is already calculated during sorting and thus, no big changes or extra calculations are required to gain great speedups with this method. Furthermore, as shown by our NUMA optimized pS^5 implementation, parallel algorithms can also be made NUMA-aware with little effort but huge wins.

As parallelization of K-way LCP-Merge requires not only a fast merging implementation, provided by our K-way LCP tournament tree, but particularly a good method to split the work into disjoint tasks, three splitting methods have been considered. Besides the classical splitting algorithm, binary splitting and LCP splitting have been introduced. Experimental analysis of the different splitting procedures showed that splitting performance highly depends on the characteristics of the input data. Whereas classical splitting works fine for most of our test sets, binary splitting inherently parallelizes the splitting work and is able to adapt directly to the input's characteristics. Although LCP splitting has high costs depending on the input size, it has its benefits when string comparisons become very expensive, due to very long strings.

We further want to highlight that the principle of our NUMA-aware pS^5 implementation can straight forwardly be extended for external string sorting with short strings ($\leq B$). As already observable in our experiments, memory throughput is the limiting aspect. Here, the combination of the LCP saving pS^5 and parallel multiway LCP-Merge can efficiently save memory accesses and therefore bandwidth.

Implementing further refinements, including but not limited to the ones discussed in the next section, will probably gain even more performance improvements. However, because of our algorithms already requiring additional space for storing LCP information, some optimizations, like character caching, may not be applicable to real-world applications like databases. Here, future challenges arise to reduce the memory footprint, while maintaining or even gaining additional performance.

6.1. Future Work

Although our algorithms already show great performance and especially NUMA-aware pS^5 achieves great wins in comparison to the original version, further improvements might be possible. In the following, potentially beneficial suggestions are presented and left for future work.

Adapting Job Sizes: As seen in the splitting analysis in Section 5.3, the performance of the splitting methods highly depends on characteristics of the input set. One advantage, binary splitting has over classical and LCP splitting, is that it inherently causes the merge to work on large jobs at the beginning and smaller jobs at the end. This reduces the number of required splittings clearly. Hence preventing unnecessary splitting overhead, while ensuring good dynamic load balancing. Applying this idea to classical and

LCP splitting is not trivial, since the size of the created merge jobs can not be adapted easily during a splitting run. However, it is important to note that a job's number of strings gives only a loose estimation of the real work to be done by the merge operation.

Caching More Characters: Due to the improvements achieved with caching the distinguishing character, caching more characters, like proposed in Section 4.5.2, is likely to further accelerate the top-level merge. However, this comes at the expense of an increased memory footprint, as well as a greatly increased need for adaptions in the base sorter, creating the sorted sequences. Often, the underlying algorithm normally would not have to access the additionally cached characters, which can lead to increased runtimes. Hence, evaluating the trade-off will be of importance.

Improved Load Balancing: Parallel K-way LCP-Mergesort can probably be improved by making it fully dynamically load balanced. However, this requires a K-way LCP-Merge outputting not just the combined strings, but also the LCP and cached character arrays, achievable as described in Section 4.2. This allows to apply the job queue framework of pS^5, with merge sort jobs regularly checking for idle threads and splitting up their work as needed. Exactly like with K-way LCP-Mergesort, NUMA-pS^5 applies a partly static load balancing scheme, since the unsorted input sequence is split into m parts up front. To optimize load balancing, the implementation of an extended NUMA-pS^5 prototype has been started. As soon as a thread group finishes the execution of its pS^5 instance, its threads start assisting another thread group until all groups are finished. Although, first experiments showed performance wins with highly scattered inputs, loses were experienced in more common use cases. Reducing synchronization overhead and improving decision making on which instances to be assisted first, is left for future work.

A. Absolute Runtimes of Parallel Algorithms

PEs	1	2	4	8	12	16	24	32	48	64
	URLs (complete), $n = 325.7\,\mathrm{M}$, $N = 20\,\mathrm{Gi}$									
pS^5-Unroll	**164**	**76.8**	37.7	22.2	16.8	13.6	11.3	10.5	9.74	10.0
pS^5-Unroll + BS-Merge			**28.2**	**14.6**	**10.7**	**8.52**	7.54	6.28	5.67	5.33
pS^5-Unroll + CS-Merge			29.4	15.0	11.7	9.29	**7.46**	**6.28**	**5.59**	**5.20**
pS^5-Unroll + LS-Merge			32.5	19.2	15.7	12.9	9.93	8.87	8.65	8.67
pLCPMergesort + BS-Merge			35.8	20.4	15.2	11.2	8.70	7.17	6.09	5.30
pLCPMergesort + CS-Merge			35.7	20.7	15.2	11.4	8.70	7.26	6.29	5.43
pLCPMergesort + LS-Merge			39.7	24.9	19.9	16.5	12.2	10.8	10.2	9.67
	Random, $n = 2.045\,\mathrm{G}$, $N = 20\,\mathrm{Gi}$									
pS^5-Unroll	**649**	**322**	**156**	87.9	66.5	55.1	43.4	39.7	38.4	37.4
pS^5-Unroll + BS-Merge			162	90.2	64.2	50.9	38.1	31.6	28.1	25.6
pS^5-Unroll + CS-Merge			158	**85.0**	**61.5**	**49.0**	**36.3**	**30.4**	**27.5**	**24.7**
pS^5-Unroll + LS-Merge			181	99.4	72.2	58.1	44.5	39.3	39.6	41.6
pLCPMergesort + BS-Merge			350	186	137	106	76.9	64.2	57.8	56.1
pLCPMergesort + CS-Merge			349	184	137	105	76.6	64.3	57.1	54.7
pLCPMergesort + LS-Merge			375	195	144	113	83.0	70.7	66.5	64.4
	GOV2, $n = 410\,\mathrm{M}$, $N = 20\,\mathrm{Gi}$									
pS^5-Unroll	**154**	**71.6**	34.4	23.2	19.3	16.2	12.8	11.6	10.4	9.77
pS^5-Unroll + BS-Merge			**27.3**	**14.9**	**10.7**	**8.53**	**6.69**	**6.22**	6.13	5.64
pS^5-Unroll + CS-Merge			28.8	15.6	11.0	9.27	7.10	6.49	**6.10**	**5.50**
pS^5-Unroll + LS-Merge			28.5	16.2	12.4	10.3	8.79	8.39	9.11	9.32
pLCPMergesort + BS-Merge			48.4	26.5	19.6	15.3	10.8	8.85	8.96	8.48
pLCPMergesort + CS-Merge			49.0	26.9	19.6	15.2	10.9	8.97	8.74	9.01
pLCPMergesort + LS-Merge			49.9	27.9	21.0	16.7	12.8	11.3	11.2	11.0
	Wikipedia, $n = N = 4\,\mathrm{Gi}$, $D = 249\,\mathrm{G}$									
pS^5-Unroll	**2641**	**1244**	581	318	234	190	138	115	105	97.2
pS^5-Unroll + BS-Merge			602	359	273	233	198	170	181	188
pS^5-Unroll + CS-Merge										
pS^5-Unroll + LS-Merge			**575**	**311**	**223**	**176**	**126**	**106**	**96.8**	**91.5**
pLCPMergesort + BS-Merge			947	512	392	318	254	217	227	244
pLCPMergesort + CS-Merge										
pLCPMergesort + LS-Merge			919	473	334	269	194	155	137	121
	Sinha DNA (complete), $n = 31.6\,\mathrm{M}$, $N = 302\,\mathrm{Mi}$									
pS^5-Unroll	**5.77**	**2.84**	1.43	0.85	0.66	0.58	0.48	0.45	0.43	0.43
pS^5-Unroll + BS-Merge			1.22	0.70	**0.52**	**0.44**	**0.34**	**0.31**	0.34	**0.37**
pS^5-Unroll + CS-Merge			**1.21**	**0.69**	0.52	0.45	0.35	0.33	**0.33**	0.39
pS^5-Unroll + LS-Merge			1.26	0.72	0.57	0.48	0.42	0.43	0.43	0.48
pLCPMergesort + BS-Merge			3.77	2.00	1.44	1.12	0.83	0.71	0.64	0.56
pLCPMergesort + CS-Merge			3.81	2.00	1.43	1.13	0.85	0.81	0.85	0.98
pLCPMergesort + LS-Merge			3.85	2.03	1.45	1.15	0.88	0.79	0.81	0.84

Table 4: Absolute runtime of parallel algorithms on IntelE5 in seconds, median of 1–5 runs. See Table 2 for a short description of each.

PEs	1	4	8	12	16	24	28	32	40	48
	URLs (complete), $n = 325.7\,\mathrm{M}$, $N = 20\,\mathrm{Gi}$									
pS^5-Unroll	**267**	**72.1**	36.9	27.1	23.0	18.5	16.6	15.7	14.4	13.5
pS^5-Unroll + BS-Merge			**28.5**	26.6	**15.3**	11.6	**11.3**	**9.63**	9.06	8.43
pS^5-Unroll + CS-Merge			28.6	26.7	15.4	**11.5**	11.4	10.7	9.20	8.52
pS^5-Unroll + LS-Merge			36.4	35.0	23.8	17.3	16.9	15.3	14.3	14.0
pLCPMergesort + BS-Merge			33.2	24.4	17.4	12.6	11.5	10.1	**8.32**	**7.97**
pLCPMergesort + CS-Merge			33.2	**24.3**	17.5	12.7	11.5	10.1	8.34	8.06
pLCPMergesort + LS-Merge			41.0	32.5	26.5	18.2	17.3	15.9	14.3	13.9
	Random, $n = 2.045\,\mathrm{G}$, $N = 20\,\mathrm{Gi}$									
pS^5-Unroll	**1075**	**292**	159	**114**	92.3	70.8	64.8	60.7	54.2	50.6
pS^5-Unroll + BS-Merge			158	145	84.2	59.7	**57.8**	47.5	**40.2**	36.7
pS^5-Unroll + CS-Merge			**158**	145	**84.0**	**59.6**	57.8	**47.2**	40.2	**36.0**
pS^5-Unroll + LS-Merge			166	153	91.9	68.1	68.2	59.1	54.1	51.7
pLCPMergesort + BS-Merge			273	203	144	101	92.7	81.2	68.7	62.5
pLCPMergesort + CS-Merge			270	203	144	101	92.9	81.1	69.1	60.8
pLCPMergesort + LS-Merge			278	211	152	110	102	90.5	79.0	71.4
	GOV2, $n = 410\,\mathrm{M}$, $N = 20\,\mathrm{Gi}$									
pS^5-Unroll	**241**	**64.1**	39.4	31.7	25.5	20.2	18.6	17.6	15.2	14.3
pS^5-Unroll + BS-Merge			**25.8**	**24.3**	**13.9**	10.1	10.0	8.26	**7.22**	7.40
pS^5-Unroll + CS-Merge			25.9	24.4	14.0	**10.1**	**9.71**	**8.17**	7.26	**7.08**
pS^5-Unroll + LS-Merge			28.1	26.9	17.0	13.5	13.4	11.9	11.0	10.8
pLCPMergesort + BS-Merge			41.5	31.1	22.4	15.5	14.0	12.4	11.8	10.9
pLCPMergesort + CS-Merge			41.6	31.1	22.3	15.6	14.1	12.4	10.8	13.9
pLCPMergesort + LS-Merge			43.8	33.6	25.2	18.9	17.7	15.9	16.5	15.0
	Wikipedia, $n = N = 4\,\mathrm{Gi}$, $D = 249\,\mathrm{G}$									
pS^5-Unroll	**1729**	**433**	**230**	**163**	**130**	**96.5**	**87.3**	**80.4**	**70.8**	**64.3**
pS^5-Unroll + BS-Merge			320	309	196	159	151	139	119	116
pS^5-Unroll + CS-Merge										
pS^5-Unroll + LS-Merge			275	257	145	104	102	84.1	72.5	65.7
pLCPMergesort + BS-Merge			385	287	232	186	154	152	138	140
pLCPMergesort + CS-Merge										
pLCPMergesort + LS-Merge			339	235	179	124	105	95.9	81.3	76.2
	Sinha DNA (complete), $n = 31.6\,\mathrm{M}$, $N = 302\,\mathrm{Mi}$									
pS^5-Unroll	**9.18**	**2.63**	1.52	**1.15**	0.97	0.80	0.75	0.72	0.69	0.68
pS^5-Unroll + BS-Merge			**1.30**	1.24	**0.77**	**0.60**	**0.59**	**0.56**	0.55	**0.49**
pS^5-Unroll + CS-Merge			1.30	1.24	0.79	0.62	0.62	0.59	**0.55**	0.50
pS^5-Unroll + LS-Merge			1.34	1.29	0.86	0.73	0.74	0.71	0.72	0.69
pLCPMergesort + BS-Merge			2.86	2.11	1.53	1.10	1.05	0.90	0.77	0.70
pLCPMergesort + CS-Merge			2.87	2.13	1.54	1.13	1.13	1.05	0.93	0.98
pLCPMergesort + LS-Merge			2.90	2.13	1.58	1.17	1.15	1.01	0.92	0.90

Table 5: Absolute runtime of parallel algorithms on AMD48 in seconds, median of 1–5 runs. See Table 2 for a short description of each.

References

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. "Replacing Suffix Trees with Enhanced Suffix Arrays". In: *J. of Discrete Algorithms* 2.1 (Mar. 2004), pp. 53–86. ISSN: 1570-8667. DOI: 10.1016/S1570-8667(03)00065-0.

[AS87] S. G. Akl and N. Santoro. "Optimal parallel merging and sorting without memory conflicts". In: *IEEE Transactions on Computers* 36 (11 1987), pp. 1367–1369.

[BES14] Timo Bingmann, Andreas Eberle, and Peter Sanders. "Engineering Parallel String Sorting". In: *CoRR* abs/1403.2056 (2014).

[BS13] Timo Bingmann and Peter Sanders. "Super Scalar String Sample Sort". In: (8125 2013), pp. 53–86.

[Col88] Richard Cole. "Parallel merge sort". In: *SIAM Journal on Computing* 17 (4 1988), pp. 770–785.

[Int14] *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C*. Intel Corporation. 2014. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting And Searching*. Addison Wesley Longman Publishing Co., Inc., 1998, pp. 251–262.

[Knö12] Sascha Denis Knöpfle. "String samplesort". Bachelor Thesis (in German). Germany: Karlsruhe Institute of Technology, Nov. 2012.

[NK08] Waihong Ng and Katsuhiko Kakehi. "Merging String Sequences by Longest Common Prefixes". In: *IPSJ Digital Courier* 4 (2008), pp. 69–78.

[Ohl13] Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013, pp. 1–604. ISBN: 978-3000413162.

[PJV91] Balakrishna R. Iyer Gary R. Ricard Peter J. Varman Scott D. Scheufler. "Merging multiple lists on hierarchical-memory multiprocessors". In: *Journal of Parallel and Distributed Computing* (Special issue on shared-memory multiprocessors 1991), pp. 171–177.

[Sha09] Nagaraja Shamsundar. "A fast, stable implementation of mergesort for sorting text files." In: (2009).

[SSP07] Johannes Singler, Peter Sanders, and Felix Putze. "MCSTL: the multi-core standard template library". In: *Euro-Par 2007 Parallel Processing* (4641 2007), pp. 682–694.

[SZ04] Ranjan Sinha and Justin Zobel. "Cache-conscious Sorting of Large Sets of Strings with Dynamic Tries". In: *J. Exp. Algorithmics* 9 (Dec. 2004). ISSN: 1084-6654. DOI: 10.1145/1005813.1041517.