Algorithmica manuscript No.
(will be inserted by the editor)

Engineering Parallel String Sorting

Timo Bingmann · Andreas Eberle · Peter Sanders

Received: date / Accepted: date

Abstract We discuss how string sorting algorithms can be parallelized on modern multi-core shared memory machines. As a synthesis of the best sequential string sorting algorithms and successful parallel sorting algorithms for atomic objects, we first propose string sample sort. The algorithm makes effective use of the memory hierarchy, uses additional word level parallelism, and largely avoids branch mispredictions. Then we focus on NUMA architectures, and develop parallel multiway LCP-merge and -mergesort to reduce the number of random memory accesses to remote nodes. Additionally, we parallelize variants of multikey quicksort and radix sort that are also useful in certain situations. As base-case sorter for LCP-aware string sorting we describe sequential LCP-insertion sort which calculates the LCP array and accelerates its insertions using it. Comprehensive experiments on five current multi-core platforms are then reported and discussed. The experiments show that our parallel string sorting implementations scale very well on real-world inputs and modern machines.

1 Introduction

Sorting is perhaps the most studied algorithmic problem in computer science. While the most simple model for sorting assumes *atomic* keys, an important class of keys are strings or vectors to be sorted lexicographically. Here, it is important to exploit the structure of the keys to avoid costly repeated operations on the entire string. String sorting is for example needed in database index construction, some suffix sorting algorithms, or MapReduce tools. Although there is a correspondingly large volume of work on sequential string sorting, there is very little work on parallel string sorting. This is surprising since parallelism is now the only way to get performance out of Moore's law so that any performance critical algorithm needs to be parallelized. We therefore started to look for practical parallel string sorting algorithms for modern multi-core shared memory machines. Our focus is on

Timo Bingmann · Andreas Eberle · Peter Sanders
Karlsruhe Institute of Technology, Kaiserstraße 12, 76131 Karlsruhe, Germany
Tel.: +49 721 608-43985, Fax: +49 721 608-43088, E-mail: {bingmann,sanders}@kit.edu

large inputs which fit into RAM. This means that besides parallelization we have to take the memory hierarchy, layout, and processor features like the high cost of branch mispredictions, word parallelism, and super scalar processing into account. Looking beyond single-socket multi-core architectures, we also consider many-core machines with multiple sockets and non-uniform memory access (NUMA).

In Section 3 we give an overview of basic sequential string sorting algorithms, acceleration techniques and more related work. We then propose our first new string sorting algorithm, super scalar string sample sort (S^5), in Section 4. Thereafter, we turn our focus to NUMA architectures in Section 5, and develop parallel LCP-aware multiway merging as a top-level algorithm for combining presorted sequences. Broadly speaking, we propose both multiway distribution-based string sorting with S^5 and multiway merge-based string sorting with LCP-aware mergesort, and parallelize both approaches.

Section 6 describes parallelizations of caching multikey quicksort and radix sort, which are two more competitors. We then compare both parallel and sequential string sorting algorithms experimentally in Section 7.

For all our input instances, except random strings, parallel S^5 achieves higher speedups on modern single-socket multi-core machines than our own parallel multikey quicksort and radixsort implementations, which are already better than any previous ones. For our Intel multi-socket NUMA machine, parallel multiway LCP-merge with node-local parallel S^5 achieves higher speedups for large real-world inputs than all other implementations in our experiment, while on our AMD many-core NUMA machine, parallel caching multikey quicksort was slightly faster on many inputs.

Shorter versions of Section 4, 6 and 7 have appeared in our conference paper [7]. We would like to thank our students Florian Drews, Michael Hamann, Christian Käser, and Sascha Denis Knöpfle who implemented prototypes of our ideas.

2 Preliminaries

Our input is a set $\mathcal{S} = \{s_1, \ldots, s_n\}$ of n strings with total length N. A *string* s is a one-based array of $|s|$ characters from the *alphabet* $\Sigma = \{1, \ldots, \sigma\}$. We assume the canonical lexicographic ordering relation '$<$' on strings, and our goal is to sort $\mathcal{S}$ lexicographically. For the implementation and pseudo-code, we require that strings are zero-terminated, i.e. $s[|s|] = 0 \notin \Sigma$, but this convention can be replaced using other end-of-string indicators, like string length.

Let D denote the *distinguishing prefix size* of $\mathcal{S}$, i.e., the total number of characters that need to be inspected in order to establish the lexicographic ordering of $\mathcal{S}$. D is a natural lower bound for the execution time of sequential string sorting. If, moreover, sorting is based on character comparisons, we get a lower bound of $\Omega(D + n \log n)$.

Sets of strings are usually represented as arrays of pointers to the beginning of each string. Note that this indirection means that, in general, every access to a string incurs a cache fault even if we are scanning an array of strings. This is a major difference to atomic sorting algorithms where scanning is very cache efficient. Our target machine is a shared memory system supporting p hardware threads or processing elements (PEs), on $\Theta(p)$ cores.

2.1 Notation and Pseudo-code

The algorithms in this paper are written in a pseudo-code language, which mixes Pascal-like control flow with array manipulation and mathematical set notation. This enables powerful expressions like $A := [\,(i^2 \bmod 7, i) \mid i \in [0\!:\!5)\,]$, which sets A to be the array of pairs $[\,(0,0),(1,1),(4,2),(2,3),(2,4)\,]$. We write ordered sequences like arrays using square brackets $[\,\ldots\,]$, overload '+' to also concatenate arrays, and let $[1\!:\!n] := [\,1,\ldots,n\,]$ and $[1\!:\!n) := [\,1,\ldots,n-1\,]$ be ranges of integers. To make array operations more concise, we assume A_i and $A[i]$ both to be the i-th element in the array A. We do not allocate or declare arrays and variables beforehand, so $A_i := 1$ also implicitly defines an array A. The unary operators '++' and '−−' increment and decrement integer variables by one.

To avoid special cases, we use the following sentinels: 'ε' is the empty string, which is lexicographically smaller than any other string, '∞' is a character or string larger than any other character or string, and '$\perp$' is an undefined variable.

For two arrays s and t, let $\mathrm{lcp}(s,t)$ denote the length of the *longest common prefix* (LCP) of s and t. This function is symmetric, and for one-based arrays the LCP value denotes the last index where s and t match, while position $\mathrm{lcp}(s,t)+1$ differs in s and t, if it exists. In a sequence x let $\mathrm{lcp}_x(i)$ denote $\mathrm{lcp}(x_{i-1}, x_i)$. For a sorted sequence of strings $\mathcal{S} = [\,s_1,\ldots,s_n\,]$ the *associated LCP array* H is $[\,\perp, h_2,\ldots,h_n\,]$ with $h_i = \mathrm{lcp}_{\mathcal{S}}(i) = \mathrm{lcp}(s_{i-1}, s_i)$. For the empty string ε, let $\mathrm{lcp}(\varepsilon, s) = 0$ for any string s.

We will often need the sum over all items in an LCP array H (excluding the first), and denote this as $L(H) := \sum_{i=2}^{n} H_i$, or just L if H is clear from the context. The distinguishing prefix size D and L are related but not identical. While D includes all characters counted in L, additionally, D also accounts for the distinguishing characters, some string terminators and characters of the first string. In general, we have $L \le D \le 2L + n$.

3 Basic Sequential String Sorting Algorithms

We begin by giving an overview of most efficient sequential string sorting algorithms. Nearly all algorithms classify the original string set $\mathcal{S}$ into smaller sets with a distinct common prefix. The smaller sets are then sorted further recursively, until the sets contain only one item or another string sorter is called.

Multikey quicksort [6] is a simple but effective adaptation of quicksort to strings (called multi-key data). When all strings in $\mathcal{S}$ have a common prefix of length ℓ, the algorithm uses character $c = s[\ell+1]$ of a pivot string $s \in \mathcal{S}$ (e.g. a pseudo-median) as a *splitter* character. $\mathcal{S}$ is then partitioned into $\mathcal{S}_<$, $\mathcal{S}_=$, and $\mathcal{S}_>$ depending on comparisons of the $(\ell+1)$-th character with c. Recursion is done on all three subproblems. The key observation is that the strings in $\mathcal{S}_=$ have common prefix length $\ell+1$ which means that compared characters found to be equal with c will never be considered again. Insertion sort is used as a base case for constant size inputs. This leads to a total execution time of $\mathcal{O}(D + n \log n)$. Multikey quicksort works well in practice in particular for inputs which fit into the cache. Since a variant of multikey quicksort was the overall best sequential algorithm in our experiments, we develop a parallel version in Section 6.2.

Most significant digit (MSD) radix sort [21,5,23,20] with common prefix length ℓ looks at the $(\ell+1)$-th character producing σ subproblems which are then sorted recursively with common prefix $\ell+1$. This is a good algorithm for large inputs and small alphabets since it uses the maximum amount of information within a single character. For input sizes $o(\sigma)$ MSD radix sort is no longer efficient and one has to switch to a different algorithm for the base case. The running time is $\mathcal{O}(D)$ plus the time for solving the base cases. Using multikey quicksort for the base case yields an algorithm with running time $\mathcal{O}(D + n \log \sigma)$. A problem with large alphabets is that one will get many cache faults if the cache cannot support σ concurrent output streams (see [22] for details). We discuss parallel radix sorting in Section 6.1.

Burstsort dynamically builds a trie data structure for the input strings. In order to reduce the involved work and to become cache efficient, the trie is build lazily – only when the number of strings referenced in a particular subtree of the trie exceeds a threshold, this part is expanded. Once all strings are inserted, the relatively small sets of strings stored at the leaves of the trie are sorted recursively (for more details refer to [31,32,30] and the references therein).

LCP-Mergesort is an adaptation of mergesort to strings that saves and reuses the LCPs of consecutive strings in the sorted subproblems [24]. In Section 5, we develop a parallel multiway variant of LCP-merge, which is used to improve performance on NUMA machines. Our multiway LCP-merge is also interesting for merging of string sets stored in external memory.

Insertion sort [18] keeps an ordered array, into which unsorted items are inserted by linearly scanning for their correct position. If strings are considered atomic, then full string comparisons are done during the linear scan. This is particularly cache-efficient and the algorithm is commonly used as base case sorter. However, if one keeps additionally the associated LCP array, the number of character comparisons can be decreased, trading them for integer comparisons of LCPs. We needed a base case sorter that also calculates the LCP array and found no reference for LCP-aware insertion sort in the literature, so we describe the algorithm in Section 5.5.

3.1 Architecture Specific Enhancements

To increase the performance of basic sequential string sorting algorithms on real hardware, we have to take its architecture into consideration. In the following list we highlight some of most important optimization principles.

Memory access time varies greatly in modern systems. While the RAM model considers all memory accesses to take unit time, current architectures have multiple levels of cache, require additional memory access on TLB misses, and may have to request data from "remote" nodes on NUMA systems. While there are few hard guarantees, we can still expect recently used memory to be in cache and use these assumptions to design cache-efficient algorithms. Furthermore, on NUMA systems we can instruct the kernel on how to distribute memory by specifying allocation policies for memory segments

Caching of characters is very important for modern memory hierarchies as it reduces the number of cache misses due to random access on strings. When performing character lookups, a caching algorithm copies successive characters

of the string into a more convenient memory area. Subsequent sorting steps can then avoid random access, until the cache needs to be refilled. This technique has successfully been applied to radix sort [23], multikey quicksort [25], and in its extreme to burstsort [32]. However, caching comes at the cost of increased space requirements and memory accesses, hence a good trade-off must be found.

Super-Alphabets can be used to accelerate string sorting algorithms which originally look only at single characters. Instead, multiple characters are grouped as one and sorted together. However, most algorithms are very sensitive to large alphabets, thus the group size must be chosen carefully. This approach results in 16-bit MSD radix sort and fast sorters for DNA strings. If the grouping is done to fit many characters into a machine word for processing as a whole block using arithmetic instructions, then this is also called *word parallelism*.

Unrolling, fission and vectorization of loops are methods to exploit out-of-order execution and super scalar parallelism now standard in modern CPUs. The processor's instruction scheduler automatically analyses the machine code, detects data dependencies and can dispatch multiple parallel operations. However, only specific, simple data independencies can be detected and thus inner loops must be designed with care (e.g. for radix sort [20]). The performance increase by reorganizing loops is most difficult to predict.

3.2 More Related Work

There is a huge amount of work on parallel sorting of atomic objects so that we can only discuss the most relevant results. Cole's celebrated merge sort [10] runs on a CREW or EREW PRAM with n processors in $\mathcal{O}(\log n)$ time, but it is mostly of theoretical interest. For parallel algorithms we will use the CREW PRAM as analysis model in which p independent RAM processors can perform operations in parallel on a shared memory, as long as write operations are without conflict. Besides more simple versions of (multiway) mergesort [3,29], perhaps the most practical parallel sorting algorithms are parallelizations of radix sort (e.g. [34]) and quicksort [33] as well as sample sort [8].

There is some work on PRAM algorithms for string sorting (e.g. [14]). By combining pairs of adjacent characters into single characters, one obtains algorithms with work $\mathcal{O}(N \log N)$ and time $\mathcal{O}(\log N / \log\log N)$. Compared to the sequential algorithms this is suboptimal unless $D = \mathcal{O}(N) = \mathcal{O}(n)$ and with this approach it is unclear how to avoid work on characters outside distinguishing prefixes.

We found no publications on practical parallel string sorting, aside from our conference paper [7]. However, Takuya Akiba has implemented a parallel radix sort [2], Tommi Rantala's library [25] contains multiple parallel mergesorts and a parallel SIMD variant of multikey quicksort, and Nagaraja Shamsundar [28] also parallelized Waihong Ng's LCP-mergesort [24]. Of all these implementations, only the radix sort by Akiba scales reasonably well to many-core architectures. We discuss the scalability issues of these implementations in Section 7.3.

4 Super Scalar String Sample Sort (S^5)

Already in a sequential setting, theoretical considerations and experiments (see Section A) indicate that *the* best string sorting algorithm does not exist. Rather, it depends at least on n, D, σ, and the hardware. Therefore we decided to parallelize several algorithms taking care that components like data distribution, load balancing or base case sorter can be reused. Remarkably, most algorithms in Section 3 can be parallelized rather easily and we will discuss parallel versions in Sections 4–6. However, none of these parallelizations make use of the striking new feature of modern many-core systems: many multi-core processors with individual cache levels but relatively few and slow memory channels to shared RAM. Therefore we decided to design a new string sorting algorithm based on *sample sort* [13], which exploits these properties. Preliminary results on string sample sort have been reported in the bachelor thesis of Sascha Denis Knöpfle [17].

4.1 Traditional (Parallel) Atomic Sample Sort

Sample sort [13,8] is a generalization of quicksort working with $k-1$ pivots at the same time. For small inputs, sample sort uses some sequential base case sorter. Larger inputs are split into k *buckets* $b_1, \ldots, b_k$ by determining $k-1$ splitter keys $x_1 \leq \cdots \leq x_{k-1}$ and then classifying the input elements – element s goes to bucket b_i if $x_{i-1} < s \leq x_i$ (where x_0 and x_k are defined as sentinel elements – x_0 being smaller than all possible input elements and x_k being larger). Splitters can be determined by drawing a random sample of size $\alpha k - 1$ from the input, sorting it, and then taking every α-th element as a splitter. Integer parameter $\alpha \geq 1$ is the *oversampling* factor. The buckets are then sorted recursively and concatenated. "Traditional" parallel sample sort chooses $k = p$ and uses a sample big enough to assure that all buckets have approximately equal size. Sample sort is also attractive as a sequential algorithm since it is more cache efficient than quicksort and since it is particularly easy to avoid branch mispredictions (super scalar sample sort – S^4) [27]. In this case, k is chosen in such a way that classification and data distribution can be done in a cache efficient way.

4.2 String Sample Sort

In order to adapt the atomic sample sort from the previous section to strings, we have to devise an efficient classification algorithm. Most importantly, we want to avoid comparing whole strings needlessly, and thus focus on character comparisons. Also, in order to approach total work $\mathcal{O}(D+n \log n)$, we have to use the information gained during classification in the recursive calls. This can be done by observing that strings in buckets have a common prefix depending on the LCP of the two splitters:

$$\forall 1 \leq i \leq k : \forall s, t \in b_i : \mathrm{lcp}(s,t) \geq \mathrm{lcp}_x(i) \ . \tag{1}$$

Another issue is that we have to reconcile the parallelization and load balancing perspective from traditional parallel sample sort with the cache efficiency perspective of super scalar sample sort, where the splitters are designed to fit into cache. We do this by using dynamic load balancing which includes parallel execution of

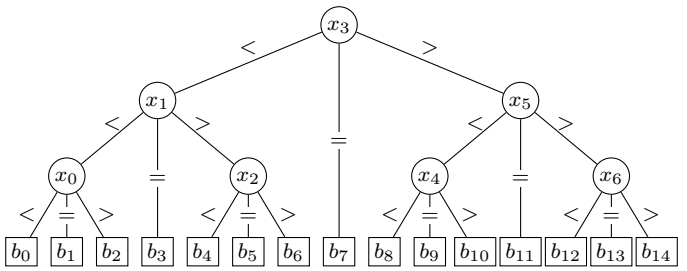

Fig. 1 Ternary classification tree for $v = 7$ splitters and $k = 15$ buckets.

recursive calls as in parallel quicksort. Dynamic load balancing is very important and probably unavoidable for parallel string sorting, because any algorithm must adapt to the input string set's characteristics.

4.3 Super Scalar String Sample Sort (S^5) – A Pragmatic Solution

We adapt the implicit binary search tree approach used in (atomic) super scalar sample sort (S^4) [27] to strings. Algorithm 1 shows pseudo-code of one variant of S^5 as a guideline through the following discussion, and Figure 1 illustrates the classification tree with buckets and splitters.

Rather than using whole strings as arbitrarily long splitters, or all characters of the alphabet as in radix sort, we design the splitter keys to consist of *as many characters as fit into a machine word*. In the following let w denote the number of characters fitting into one machine word (for 8-bit characters and 64-bit machine words we would have $w = 8$). We choose $v = 2^d - 1$ splitters $x_0, \ldots, x_{v-1}$ (for some integer d) from a sorted sample to construct a *perfect binary search tree*, which is used to classify a set of strings based on the next w characters at common prefix h. The main disadvantage of this approach is that we may have many input strings whose next w characters are identical. For these strings, the classification does not reveal much information. We make the best out of such inputs by explicitly defining *equality buckets* for strings whose next w characters exactly match x_i. For equality buckets, we can increase the common prefix length by w in the recursive calls, i.e., these characters will never be inspected again. In total, we have $k = 2v+1$ different buckets $b_0, \ldots, b_{2v}$ for a ternary search tree (see Figure 1).

Testing for equality can either be implemented by explicit equality tests at each node of the search tree (which saves time when most elements end up in a few large equality buckets) or by going down the search tree all the way to a bucket b_i (i even) doing only $\leq$-comparisons, followed by a single equality test with $x_{\frac{i}{2}}$, unless $i = 2v$. This last variant is shown in Algorithm 1, and the equality test is done in line 11.

Postponing the equality test allows us to completely unroll the loop descending the search tree (line 8), since there is no exit condition. We can then also unroll the loop over the elements (line 6), interleaving independent tree descent operations. The number of interleaved descents is limited in practice by the number of registers to hold local variables like i and c. As in [27], this is an important optimization

Algorithm 1: Sequential Super Scalar String Sample Sort – a single step

Input: $\mathcal{S} = \{s_1, \ldots, s_n\}$ a set of strings with common prefix h, $v = 2^d - 1$ a number of splitters, $v' := v + 1$, and $\alpha \geq 1$ an oversampling factor.

1 $p_i := \texttt{chars}_h(s_{\texttt{random}(1,\ldots,n)}) \quad \forall\, i = 1, \ldots, v\alpha + \alpha - 1$ // *Read sample p of $\mathcal{S}$,*
2 $\texttt{sort}([\, p_1, \ldots, p_{v\alpha+\alpha-1}\,])$ // *sort it, and select*
3 $[\, x_1, x_2, \ldots, x_{v-1}, x_v\,] := [\, p_\alpha, p_{2\alpha}, p_{3\alpha}, \ldots, p_{v\alpha}\,]$ // *equidistant splitters.*
4 $[\, t_1, \ldots, t_v\,] := [\, x_{\frac{v'}{2}}, x_{\frac{v'}{4}}, x_{\frac{3v'}{4}}, x_{\frac{v'}{8}}, x_{\frac{3v'}{8}}, x_{\frac{5v'}{8}}, x_{\frac{7v'}{8}}, \ldots\,]$ // *Construct tree, save*
5 $[\, h'_0, \ldots, h'_v\,] := [\,0\,] + [\,\mathrm{lcp}(x_{i-1}, x_i) \mid i = 1, \ldots, v-1\,] + [\,0\,]$ // *LCPs of splitters.*
6 **for** $j := 1, \ldots, n$ **do** // *Process strings (interleavable loop).*
7 **local** $i := 1, \quad c := \texttt{chars}_h(s_j)$ // *Start at root, get w chars from s_j,*
8 **for** $1, \ldots, \log_2(v+1)$ **do** // *and traverse tree (unrollable loop)*
9 $i := 2i + (c \leq t_i)$ // *without branches using "$(c \leq t_i)$" $\in \{0, 1\}$.*
10 $i := i - (v+1)$, **local** $m := 2i$ // *Calculate matching non-equality bucket.*
11 **if** $x_i = c$ **then** $m{+}{+}$ // *Test for equality with next splitter.*
12 $o_j := m$ // *Save final bucket number for string s_j in an oracle.*
13 $b_i := 0 \quad \forall\, i = 0, \ldots, 2v$ // *Inclusive prefix sum*
14 **for** $i := 1, \ldots, n$ **do** (b_{o_i})++ // *over bucket sizes*
15 $[\, b_0, \ldots, b_{2v}, b_{2v+1}\,] := [\, \sum_{j \leq i} b_j \mid i = 0, \ldots, 2v\,] + [\,n\,]$ // *as fissioned loops.*
16 **for** $i := 1, \ldots, n$ **do** $s'_{(b_{o_i})--} := s_i$ // *Reorder strings into new subsets.*

Output: $\mathcal{S}'_i = \{s'_j \mid j = b_i, \ldots, b_{i+1} - 1 \text{ if } b_i < b_{i+1}\}$ for $i = 0, \ldots, 2v$ are string subsets with $\mathcal{S}'_i < \mathcal{S}'_{i+1}$. The subsets have common prefix $h + h'_{i/2}$ for i even, and common prefix $h + w$ for i odd.

since it allows the instruction scheduler in a super scalar processor to parallelize the operations by drawing data dependencies apart.

After reordering, the strings in the "$< x_0$" and "$> x_{v-1}$" buckets b_0 and b_{2v} keep common prefix length h. For other even buckets b_i the common prefix length is increased by $\mathrm{lcp}_x(\frac{i}{2})$.

An analysis similar to the one of multikey quicksort [6] lets us conjecture the following asymptotic time bound.

Conjecture 1 String sample sort with implicit binary trees, word parallelism and equality checking at each splitter node can be implemented to run in expected time $\mathcal{O}(\frac{D}{w} + n \log n)$.

We now argue the correctness of Conjecture 1, without giving a formal proof[1]. The classification schemes of multikey quicksort and string sample sort can both be seen as a tree. In this tree, edges are either associated with characters of a distinguishing prefix, or with string ranges determined by splitters.

In multikey quicksort each inner node z of the tree has three children: $<$, $=$, and $>$. We can associate each character comparison during partitioning at node z with the thereby determined edge. By selecting pivots randomly or using a sample median, the expected number of $<$ and $>$ edges in all paths from the root is $\mathcal{O}(\log n)$ [15,6], since this approach is identical to atomic quicksort. Thus the time spent over all comparisons accounted by these edges is expected $\mathcal{O}(n \log n)$.

[1] We are currently working on this for a final version of this paper

All comparisons associated with = edges correspond to characters from the distinguishing prefix, and are thus bounded by D. In total we have $\mathcal{O}(D + n \log n)$ work in the multikey quicksort tree.

We can view string sample sort as a multikey quicksort using multiple pivots in the classification tree, as seen in Figure 1. In string sample sort, an = edge matches w characters, of which at least one is from the distinguishing prefix D (but usually all are). If any of the w characters is not counted in D, then the = edge leads to a leaf, which does not require further sorting. There are at most n such comparisons leading to leaves, all other = edges match w characters. Thus we have at most $\frac{D}{w} + n$ comparisons leading to = edges. To prove our conjecture, we need to show that the expected number of $<$ and $>$ edges on all paths from the root is $\mathcal{O}(\log n)$. However, we are not aware of any analysis of sample sort showing this expected run time for a *fixed sample size*. Furthermore, in string sample sort we have to deal with the probability of multiple equal samples and need to resample strings repeatedly at higher depths, thus the known analysis of a single top-level sampling approach [8,35,13] do not apply. Nevertheless, due to repeated resampling, we can conjecture that the bucket sizes grow small very fast, just as they do in atomic sample sort. By using the additional LCP information gained at $<$ and $>$ edges from Equation 1 one could decrease the expected path length from the root further, though probably not asymptotically.

If the = edges are taken immediately, as done in the variant with explicit equality checking at each node, then we conjecture expected $\mathcal{O}(\frac{D}{w} + n \log n)$ time. However, if we choose to unroll descents of the tree, then the splitter at the root may match and the $\Theta(\log v)$ additional steps down the tree are superfluous. This happens when many strings are identical, and the corresponding splitters are high up in the tree. We thus have to attribute $\mathcal{O}((\frac{D}{w} + n) \log v)$ time to the = edges. Together with the expected cost of $<$ and $>$ edges, we conjecture in total an expected $\mathcal{O}((\frac{D}{w} + n) \log v + n \log n)$ bound. However, due to the unclear interaction of repeated sampling with fixed size and the probability of hitting the equality buckets, we leave a full proof of our conjecture to future work.

String sample sort is particularly easy to parallelize for large string sets and current multi-core architectures where $n \gg pv$, and we can state the following theorem.

Theorem 1 *A single step of super scalar string sample sort (Algorithm 1) can be implemented to run on a CREW PRAM with $p < \frac{n}{v}$ processors in $\mathcal{O}(\frac{n}{p} \log v + \log p)$ time and $\mathcal{O}(n \log v + pv)$ work.*

Proof Sorting the sample requires $\mathcal{O}(\frac{a \log a}{p} + \log p)$ time and $\mathcal{O}(a \log a)$ work [10,9], where $a := \alpha v + \alpha - 1 \ll n$ is the sample size. Selecting the sample, picking splitters, constructing the tree and saving LCP of splitters is all $\mathcal{O}(\frac{a}{p})$ time and $\mathcal{O}(a)$ work. Each processors gets $\frac{n}{p}$ strings and in worst case runs down all $\log v$ steps in the classification tree, which is $\mathcal{O}(\frac{n}{p} \log v)$ time and $\mathcal{O}(n \log v)$ work. Departing from lines 13–16, each processor keeps its own bucket array b_i of size $2v + 1$, initializes it in $\mathcal{O}(v)$ time, and classifies only those strings in its string set. Then, an interleaved global prefix sum over the $p(2v + 1)$ bucket counters yields the boundaries in which each processor can independently redistribute its strings. The prefix sum runs in $\mathcal{O}(\log pv)$ time and $\mathcal{O}(pv)$ work [19], while counting and redistribution runs in $\mathcal{O}(\frac{n}{p})$ time and $\mathcal{O}(n)$ work. Summing all time and work yields our result.

We only consider a single step here, and thus cannot use the distinguishing prefix D to bound the overall work.

4.4 Implementation Details

One goal of S^5 is to have a common classification data structure that fits into the cache of all cores. Using this data structure, all PEs can independently classify a subset of the strings into buckets in parallel. The process follows the classic distribution-based sorting steps: we first classify strings (lines 6–12), counting how many fall into each bucket (line 14), then calculate a prefix sum (line 15) and redistribute the string pointers accordingly (line 16). To avoid traversing the tree twice, the bucket index of each string is stored in an oracle (lines 12, 14, 16). Additionally, to make higher use of super scalar parallelism, we even separate the classification loop (line 6) from the counting loop (line 14), as done by [20].

Like in S^4, the binary tree of splitters is stored in level-order as an array t (line 4), allowing efficient traversal using $i := 2i+\{0,1\}$, without branch mispredictions in line 9. The pseudo-code "$(c \leq t_i)$", which yields 0 or 1, can be implemented using different machine instructions. One method is to use the instruction `SETA`, which sets a register to 0 or 1 depending on a preceding comparison. Alternatively, newer processors have predicated instruction like `CMOVA` to conditionally move one register to another, again depending on a preceding comparison's outcome. We noticed that `CMOVA` was slightly faster than flag arithmetic.

While traversing the classification tree, we compare w characters using one arithmetic comparison. However, we need to make sure that these comparisons have the desired outcome, e.g., that the most significant bits of the register hold the first character. For little-endian machines and 8-bit characters, which are used in all of our experiments, we need to *swap the byte order* when loading character from a string. In our implementation we do this using the `BSWAP` machine instruction. In the pseudo-code (Algorithm 1) this operation is symbolized by $\texttt{chars}_h(s_i)$, which fetches w characters from s_i at depth $h+1$, and swaps them appropriately.

For performing the equality check, already mentioned in the previous section, we want to discuss four different alternatives in more technical details here:

1. One can traverse the tree using only $\leq$-comparisons and perform the equality check afterwards, as shown in Algorithm 1. For this we keep the splitters x_i in an in-order array, in addition to the classification tree t, which contains them in level-order. Duplicating the splitters avoids additional work in line 11, where i is an in-order index. This variant, called S^5-Unroll, was our final choice as it was overall fastest.
2. The additional in-order array from the previous variant, however, can be removed. Instead, a rather simple calculation involving only bit operations can be used to transform the in-order index i back to level-order, and reuse the classification tree t. We tried this variant, but found no performance advantage over the first.
3. Another idea is to keep track of the last $\leq$-branch during tree traversal, this however was slower and requires an extra register for each of the interleaved descents.
4. The last variant is to check for equality after each comparison in line 9. This requires only an additional `JE` instruction and no extra `CMP` in the inner-most

loop. The branch misprediction cost of the `JE` is counter-balanced by skipping the rest of the tree. As i is a tree-order index when exiting the inner loop, we need to apply the inverse of the transformation mentioned in the second method to i to determine the correct equality bucket. Thus in this fourth variant, named S^5-Equal, no additional in-order splitter array is needed.

The sample is drawn pseudo-randomly with an oversampling factor $\alpha = 2$ to keep it in cache when sorting with STL's introsort and building the search tree. Instead of using the straight-forward equidistant method to draw splitters from the sample, as shown in Algorithm 1 (line 3), we developed a simple recursive scheme that tries to avoid using the same splitter multiple times: Select the middle sample m of a range $a..b$ (initially the whole sample) as the middle splitter $\bar{x}$. Find new boundaries b' and a' by scanning left and right from m *skipping* samples equal to $\bar{x}$. Recurse on $a..b'$ and $a'..b$. The splitter tree selected by this heuristic was never slower than equidistant selection, but slightly faster for inputs with many equal common prefixes. It is used in all our experiments.

The LCP of two consecutive splitters in line 5 can be calculated without a loop using just two machine instructions: `XOR` and `BSR` (to count the number of leading zero bits in the result of `XOR`). In our implementation, these calculation are done while selecting splitters. Similarly, we need to check if splitters contain end-of-string terminators, and skip the recursion in this case.

For current 64-bit machines with 256 KiB L2 cache, we use $v = 8191$. Note that the limiting data structure which must fit into L2 cache is not the splitter tree t, which is only 64 KiB for this v, but is the bucket counter array b containing $2v + 1$ counters, each 8 bytes long. We did not look into methods to reduce this array's size, because the search tree is stored both in level-order and in in-order, and thus we could not increase the tree size anyway.

4.5 Practical Parallelization of S^5

Parallel S^5 (pS^5) is composed of four sub-algorithms for differently sized subsets of strings. For a string subset $\mathcal{S}$ with $|\mathcal{S}| \geq \frac{n}{p}$, a *fully parallel version* of S^5 is run, for large sizes $\frac{n}{p} > |\mathcal{S}| \geq t_m$ a sequential version of S^5 is used, for sizes $t_m > |\mathcal{S}| \geq t_i$ the fastest sequential algorithm for medium-size inputs (caching multikey quicksort from Section 6.2) is called, which internally uses insertion sort when $|\mathcal{S}| < t_i$. We empirically determined $t_m = 1\,\text{Mi}$ and $t_i = 64$ as good thresholds to switch sub-algorithms.

The fully parallel version of S^5 uses $p' = \Theta(\frac{p}{n}|\mathcal{S}|)$ threads for a subset $\mathcal{S}$. It consists of four stages: selecting samples and generating a splitter tree, parallel classification and counting, global prefix sum, and redistribution into buckets. Selecting the sample and constructing the search tree are done sequentially, as these steps have negligible run time. Classification is done independently, dividing the string set evenly among the p' threads. The prefix sum is done sequentially once all threads finish counting.

In both the sequential and parallel versions of S^5 we permute the string pointer array using out-of-place redistribution into an extra array. In principle, we could do an in-place permutation in the sequential version by walking cycles of the permutation [21]. Compared to out-of-place copying, the in-place algorithm uses

fewer input/output streams and requires no extra space. However, we found that modern processors optimize the sequential reading and writing pattern of the out-of-place version better than the random access pattern of the in-place walking. Furthermore, for fully parallel S^5, an in-place permutation cannot be done in the same manner. We therefore always use *out-of-place redistribution*, with an extra string pointer array of size n. For recursive calls, the role of the extra array and original array are swapped, which saves superfluous copying work.

All work in parallel S^5 is dynamically load balanced via a central job queue. We use the lock-free queue implementation from Intel's Thread Building Blocks (TBB) and threads initiated by OpenMP to create a *light-weight thread pool*.

To make work balancing most efficient, we modified all sequential sub-algorithms of parallel S^5 to use an explicit recursion stack. The traditional way to implement dynamic load balancing would be to use work stealing among the sequentially working threads. This would require the operations on the local recursion stacks to be synchronized or atomic. However, for our application fast stack operations are crucial for performance as they are very frequent. We therefore choose a different method: *voluntary work sharing*. If the global job queue is empty and a thread is idle, then a global atomic counter is incremented to indicate that other threads should share their work. These then free the stack level with the *largest subproblems* from their local recursion stack and enqueue these as separate, independent jobs. This method avoids costly atomic operations on the local stacks, replacing it by a faster counter check, which itself *need not be synchronized* or atomic. The short wait of an idle thread for new work does not occur often, because the largest recursive subproblems are shared. Furthermore, the global job queue never gets large because most subproblems are kept on local stacks.

5 Parallel Multiway LCP-Mergesort

When designing pS^5 we considered L2 cache sizes, word parallelism, super scalar parallelism and other modern features. However, new architectures with large amounts of RAM are now commonly non-uniform memory access (NUMA) systems, and the RAM chips are distributed onto different memory banks, called *NUMA nodes*. In preliminary synthetic experiments, access to memory on "remote" nodes was 2–5 times slower than memory on the local socket, because the requests must pass over an additional interconnection bus. This latency and throughput disparity brings algorithms for external and distributed memory to mind, but the divide is much less pronounced and block sizes are smaller.

In light of this disparity, we propose to use *independent string sorters* on each NUMA node, and then *merge* the sorted results. During merging, the amount of information per transmission unit passed via the interconnect (64-byte cache lines) should be maximized. Thus, besides the sorted string pointers, we also want to use LCP information to skip over known common prefixes, and cache the distinguishing characters.

While merging sorted sequences of strings with associated LCP information is a very intuitive idea, remarkably, only one very recent paper by Ng and Kakehi [24] fully considers LCP-aware mergesort for strings. They describe *binary* LCP-mergesort and perform an average case analysis yielding estimates for the number of comparisons needed. For the NUMA scenario, however, we need a practical *parallel*

K-way LCP-merge, where K is the number of NUMA nodes. Furthermore, we also need to extend our existing string sorting algorithms to save the LCP array.

In the next section, we first review binary LCP-aware merging. On this foundation we then propose and analyze parallel K-way LCP-merging with tournament trees in Sections 5.2–5.4. For node-local LCP calculations, we extended pS^5 appropriately, and describe the necessary base case sorter, LCP-insertion sort, in Section 5.5. Further information on the results of this section are available in the bachelor thesis of Andreas Eberle [12].

Another way to perform a K-way LCP-merge is to use the LCP-aware string heap described by Kent, Lewenstein, and Sheinwald [16] as an on-demand string sorter for the smallest strings of the K sorted sequences. This solution yields the same asymptotic time bounds as our approach, but as with merging of atomic objects, tournament trees promise smaller constant factors and better practical runtime than heaps. Our extension of tournament trees and insertion sort to use LCP-aware comparisons can be seen as an application of the theoretical "block-box" framework by Amir et al. [4], which takes any comparison-driven data structure and augments it with LCP-awareness. However, this framework is complex and does not yield practical algorithms, as the authors themselves note.

5.1 Binary LCP-Compare and LCP-Mergesort

We reformulate the binary LCP-merge and -mergesort presented by Ng and Kakehi [24] here in a different way. Our exposition is somewhat more verbose than necessary, but this is intentional and prepares for a simpler description of K-way LCP-merge in the following section.

Consider the basic comparison of two strings s_a and s_b. If there is no additional LCP information, the strings must be compared character-wise until a mismatch is found. However, if we have additionally the LCP of s_a and s_b to another string p, namely $\text{lcp}(p, s_a)$ and $\text{lcp}(p, s_b)$, then we can first compare these LCP values. Since both reference p, we know that s_a and s_b share a common prefix $\min\{\text{lcp}(p, s_a), \text{lcp}(p, s_b)\}$ and that this common prefix is maximal (i.e. longest). Thus if $\text{lcp}(p, s_a) < \text{lcp}(p, s_b)$, then the two strings s_a and s_b differ at position $\ell := \text{lcp}(p, s_a) + 1$. If we now furthermore assume $p \leq s_a$, then we immediately see $p[\ell] = s_b[\ell] < s_a[\ell]$, from which follows $s_b < s_a$. The argument can be applied symmetrically if $\text{lcp}(p, s_b) < \text{lcp}(p, s_a)$.

There remains the case $\text{lcp}(p, s_a) = \text{lcp}(p, s_b)$. Here, the LCP information only reveals that both have a common prefix $\text{lcp}(p, s_a)$, and additional character comparisons starting at the common prefix are necessary to order the strings.

The pseudo-code in Algorithm 2 implements these three cases. In preparation for K-way LCP-merge, the function `LCP-Compare` additionally takes variables a and b, which are corresponding indexes and returns these instead of s_a or s_b. It also calculates more information than just the order of s_a and s_b, since future LCP-aware comparisons also require $\text{lcp}(s_a, s_b)$.

In the cases $\text{lcp}(p, s_a) \neq \text{lcp}(p, s_b)$, the $\text{lcp}(s_a, s_b)$ is easily inferred since the character after the smaller LCP differs in s_a and s_b. From this follows $\text{lcp}(s_a, s_b) = \min\{\text{lcp}(p, s_a), \text{lcp}(p, s_b)\}$, as already stated above. For $\text{lcp}(p, s_a) = \text{lcp}(p, s_b)$ each additionally compared equal character is common to both s_a and s_b, and the

Algorithm 2: Binary LCP-Compare

1 **Function** `LCP-Compare`$((a, s_a, h_a), (b, s_b, h_b))$
  Input: (a, s_a, h_a) and (b, s_b, h_b) where s_a and s_b are two strings together with LCPs $h_a = \text{lcp}(p, s_a)$ and $h_b = \text{lcp}(p, s_b)$, and p is another string with $p \leq s_a$ and $p \leq s_b$.
2 **if** $h_a = h_b$ **then** *// case 1: LCPs are equal ⇒ compare more characters,*
3 $h' := h_a$ *// starting at $h' = h_a = h_b$.*
4 **while** $s_a[h'] \neq 0$ **and** $s_a[h'] = s_b[h']$ **do** *// Compare characters and*
5 h'++ *// increase total LCP.*
6 **if** $s_a[h'] \leq s_b[h']$ **then return** (a, h_a, b, h')
7 **else return** (b, h_b, a, h')
8 **else if** $h_a < h_b$ **then return** (b, h_b, a, h_a) *// case 2: $s_b[h_a{+}1] < s_a[h_a{+}1]$.*
9 **else return** (a, h_a, b, h_b) *// case 3: $s_a[h_b{+}1] < s_b[h_b{+}1]$.*
  Output: (x, h_x, y, h') with $s_x \leq s_y$, $\{x, y\} = \{a, b\}$, and $h' = \text{lcp}(s_a, s_b)$.

Algorithm 3: Binary LCP-Merge

Input: $\mathcal{S}_1$ and $\mathcal{S}_2$ two sorted sequences of strings with LCP arrays H_1 and H_2. Assume sentinels $\mathcal{S}_k[|\mathcal{S}_k| + 1] = \infty$ for $k = 1, 2$, and $\mathcal{S}_0[0] = \varepsilon$.
1 $i_1 := 1, \quad i_2 := 1, \quad j := 1$ *// Indexes for $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_0$.*
2 $h_1 := 0, \quad h_2 := 0$ *// Invariant: $h_k = \text{lcp}(\mathcal{S}_k[i_k], \mathcal{S}_0[j-1])$ for $k = 1, 2$.*
3 **while** $i_1 + i_2 \leq |\mathcal{S}_1| + |\mathcal{S}_2|$ **do**
4 $s_1 := \mathcal{S}_1[i_1], \quad s_2 := \mathcal{S}_2[i_2]$ *// Fetch strings s_1 and s_2,*
5 $(x, \bot, y, h') =$ `LCP-Compare`$((1, s_1, h_1), (2, s_2, h_2))$ *// compare them,*
6 $(\mathcal{S}_0[j], H_0[j]) := (s_x, h_x), \quad j$++ *// put smaller into output*
7 i_x++, $\quad (h_x, h_y) := (H_x[i_x], h')$ *// and advance to next.*
Output: $\mathcal{S}_0$ contains sorted $\mathcal{S}_1$ and $\mathcal{S}_2$, and $\mathcal{S}_0$ has the LCP array H_0

comparison loop in line 4 of Algorithm 2 breaks at the first mismatch or zero termination. Thus afterwards $h' = \text{lcp}(s_a, s_b)$, and can be returned as such.

Using `LCP-Compare` we can now build a binary LCP-aware merging method, which merges two sorted string sequences with associated LCP arrays. One only needs to take s_a and s_b, compare them using `LCP-Compare`, write the smaller of them, say s_a, to the output and fetch its successor s'_a from the sorted sequence. The written string s_a then plays the role of p in the discussion above, and the next two candidate strings s'_a and s_b can be compared, since $\text{lcp}(p, s_b) = \text{lcp}(s_a, s_b)$ is returned by `LCP-Compare` and $\text{lcp}(p, s'_a) = \text{lcp}(s_a, s'_a)$ is known from the corresponding LCP array. This procedure is detailed in Algorithm 3. For binary merging, we can ignore the h_x returned by `LCP-Compare`. Notice that using the indexes x and y, the LCP invariant can be restored using just one assignment in line 7.

Theorem 2 *Using Algorithm 3, one can implement a binary LCP-mergesort algorithm, which requires at most $L + n\lceil \log_2 n \rceil$ character comparisons and runs in $\mathcal{O}(D + n \log n)$ time.*

Proof We assume the divide step of binary LCP-mergesort to do straight-forward halving as in non-LCP mergesort [18], which is why we omitted its pseudo-code.

Likewise, the recursive division steps have at most depth $\lceil \log_2 n \rceil$ when reaching the base case. If we briefly ignore the character comparison loop in `LCP-Compare`, line 4, and regard it as a single comparison, then the standard divide-and-conquer recurrence $T(n) \leq T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n$ of non-LCP mergesort still holds. Regarding the character comparison loop, we can establish that each increment of h' ultimately increases the overall LCP sum by exactly one, since in all other statements LCPs are only moved, swapped or stored, but never decreased or discarded. Another way to see this is that the character comparison loop is the only place where characters are compared, thus to be able to establish the correctly sorted order, all distinguishing characters must be compared here.

We regard the three different comparison expressions in lines 4–6 as one ternary comparison, as the same values are checked again and zero-terminators can be handled using flag tests. To count the total number of comparisons, we can thus account for all *true*-outcomes of the while loop condition in `LCP-Compare` (line 4) using L, and all *false*-outcomes using $n\lceil \log_2 n \rceil$, since this is the highest number of times case 1 can occur in the mergesort recursion. This is an upper bound, and for most string sets, cases 2 and 3 reduce the number of comparisons in the second term. Since $L \leq D$, the time complexity $\mathcal{O}(D + n \log n)$ follows immediately. □

Ng and Kakehi [24] do not give an explicit worst case analysis. Their average case analysis shows, that the total number of character comparisons of binary LCP-mergesort is about $n(\mu_a - 1) + P_\omega n \log_2 n$, where μ_a is the average length of distinguishing prefixes and P_ω the probability of a "breakdown", which corresponds to case 1 in `LCP-Compare`. Taking $P_\omega = 1$ and $\mu_a = \frac{D}{n}$, their equation matches our worst-case result, except for the minor difference between D and L.

5.2 K-way LCP-Merge

To accelerate LCP-merge for NUMA systems, we extended the binary LCP-merge approach to K-way LCP-merge using tournament trees [18,26], since current NUMA systems have four or even eight nodes. We could not find any reference to K-way LCP-merge with tournament trees in the literature, even though the idea to store and reuse LCP information inside the tree is very intuitive. The algorithmic details, however, require precise elaboration. Compared to the LCP-aware string heap [16], our K-way LCP-aware tournament tree is only better by constant factors, but these are very important for practical implementations.

As commonly done in multiway mergesort, to perform K-way merging one regards selection of the next item as a tournament with K players (see Figure 2). Players compete against each other using binary comparisons, and these games are organized in a binary tree. Each node in the tree corresponds to one game, and we label the nodes of the tree with the "losers" of that particular game. The "winner" continues upward and plays further games, until the overall winner is determined. The winner is commonly placed on the top, in an additional node, and with this node, the tournament tree contains each player exactly once. Hence the tree has exactly K nodes, since we do not consider the input, output or players part of the tree. For sorting strings into ascending sequences, the "overall winner" of the tournament is the lexicographically smallest string.

The first winner is determined by playing an initial round on all K nodes from bottom up. This winner can then be sent to the output, and the next item from

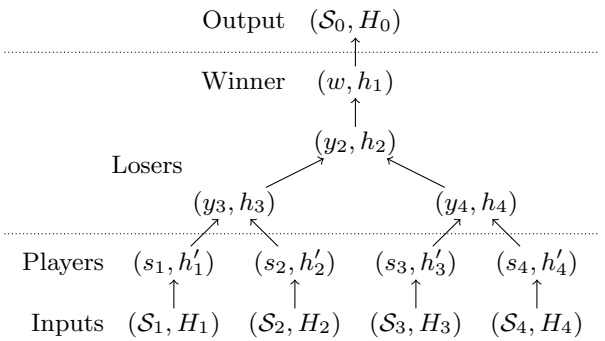

Fig. 2 LCP-aware tournament tree with $K = 4$ showing input and output streams, their front items as players, the winner node (w, h_1), and loser nodes (y_i, h_i), where y_i is the index of the losing player of the particular game and h_i is the LCP of s_{y_i} and the winner of the comparison at node i.

the corresponding input sequence takes its place. Thereafter, only $\log_2 K$ games must be replayed, since the previous winner only took part in those games along the path from the corresponding input to the root of the tournament tree. This can be repeated until all streams are empty. By using sentinels for empty inputs, special cases can be avoided, and we can assume K to be a power of two, filling up with empty inputs as needed. Thus the tournament tree can be assumed to be a perfect binary tree, and can be stored implicitly in an array. Navigating upward in the tree corresponds to division by two: $\lceil \frac{i}{2} \rceil$ is the parent of child i, unless $i = 1$ (note that we use a one-based array here). Thus finding the path from input leaf to root when replaying the game can be implemented very efficiently. Inside the tree nodes, we save the loser *input index* y_i, or winner index w (renamed from y_1), instead of storing the string s_i or a reference thereof.

We now discuss how to make the tournament tree LCP-aware. The binary comparisons between players are done using `LCP-Compare` (Algorithm 2), which may perform explicit character comparisons in case 1. Since we want to avoid comparing characters already found equal, we store alongside the loser input index y_i an LCP value h_i in the tree node. The LCP h_i represents the LCP of the stored losing string s_{y_i} with the particular game's winner string, which passes upward to play further comparisons. If we call the corresponding winner x_i, even though it's not explicitly stored, then $h_i = \mathrm{lcp}(s_{x_i}, s_{y_i})$.

After the initial overall winner w is determined, we have to check that all requirements of `LCP-Compare` are fulfilled when replaying the games on the path from input w to the root. The key argument is that the overall winner w was also the immediate winner of all individual games on the path, while the losers on this path are themselves winners of the subtree leading to the path but not *on* the path. These subtree winners are exactly all players against which w won along the way to being overall winner. The LCP outcome of this game remained unchanged at the loser position, since any winner prior to w cannot have originated below this game.

Overall, this results in that all games i on that path have $h_i = \mathrm{lcp}(s_w, s_{y_i})$. Thus after writing s_w to the output, and advancing to the next item (s'_w, h''_w) from the input $(\mathcal{S}_w, H_w)$, we have $p = s_w$ as the common, smaller predecessor string.

Algorithm 4: K-way LCP-Merge

Input: $\mathcal{S}_1, \ldots, \mathcal{S}_K$ sorted sequences of strings with LCP arrays $H_1, \ldots, H_K$ and common prefix $\overline{h}$. Assume sentinels $\mathcal{S}_k[|\mathcal{S}_k| + 1] = \infty$ for $k = 1, \ldots, K$, and K a power of two.

1 $i_k := 1 \; \forall k = 1, \ldots, K, \quad j := 1$ *// Initialize indexes for $\mathcal{S}_1, \ldots, \mathcal{S}_K$ and $\mathcal{S}_0$.*
2 **for** $k := 1, \ldots, K$ **do** *// Initialize loser tree, building*
3 $\quad s_k := \mathcal{S}_k[i_k]$ *// perfect subtrees left-to-right.*
4 $\quad (x, h') := (k, \overline{h}), \quad v := K + k$ *// Play from input node v, upward till the root*
5 $\quad$ **while** v *is even* **and** $v > 2$ **do** *// of a perfect odd-based subtree is reached.*
6 $\quad\quad v := \frac{v}{2}, \quad (x, h', y_v, h_v) :=$ `LCP-Compare`$((x, s_x, h'), (y_v, s_{y_v}, h_v))$
7 $\quad v := \lceil \frac{v}{2} \rceil, \; (y_v, h_v) := (x, h')$ *// Save winner node at top of odd-based subtree.*
8 $w := y_1$ *// Initial winner after all games (rename $y_1 \to w$).*
9 **while** $j \leq \sum_{k=1}^{K} |\mathcal{S}_k|$ **do** *// Loop until output is done.*
10 $\quad (\mathcal{S}_0[j], H_0[j]) := (s_w, h_1), \quad j{+}{+}$ *// Output winner string s_w with LCP h_1.*
11 $\quad i_w{+}{+}, \quad s_w := \mathcal{S}_w[i_w]$ *// Replace winner with next item from input.*
12 $\quad (x, h') := (w, H_w[i_w]), \quad v := K + w$ *// Play from input node v, all games*
13 $\quad$ **while** $v > 2$ **do** *// upward to root (unrollable loop).*
14 $\quad\quad v := \lceil \frac{v}{2} \rceil, \quad (x, h', y_v, h_v) :=$ `LCP-Compare`$((x, s_x, h'), (y_v, s_{y_v}, h_v))$
15 $\quad (w, h_1) := (x, h')$ *// Save next winner at top.*

Output: $\mathcal{S}_0$ contains sorted $\mathcal{S}_1, \ldots, \mathcal{S}_K$ and has the LCP array H_0

The previous discussion about the path to the overall winner w is also valid for the path to the individual winner x_i of any node i in the tree, since it is the winner of all games leading from input x_i to node i.

The function signature $(x, h_x, y, h_y) =$ `LCP-Compare`$((a, s_a, h_a), (b, s_b, h_b))$ was designed to be played on two nodes (a, h_a) and (b, h_b) of the LCP-aware tournament tree. When replaying a path, we can picture a node (a, h_a) moving "upward" along the edges. `LCP-Compare` is called with this moving node and the loser information $(b, h_b) := (y_i, h_i)$ saved in the encountered node i. After performing the comparisons, the returning values (x, h_x) are the winner node, which passes upwards, and (y, h_y) are the loser information, which is saved in the node i. Thus `LCP-Compare` effectively selects the winner of each game, and computes the loser information for future LCP-aware comparisons. Due to the recursive property discussed in the previous paragraph, the requirements of `LCP-Compare` remains valid along all paths, and `LCP-Compare` can switch between them.

This LCP-aware K-way merging procedure is shown in pseudo-code in Algorithm 4. We build the initial tournament tree incrementally from left to right, playing all games only on the right-most path of every odd-based perfect subtree. This right-most side contains only nodes with even index.

The pseudo-code works with one-based arrays, but our implementation uses a zero-based implicit tree, which reduces the number of operations slightly. The pseudo-code also contains a superfluous run of lines 11–15, which we keep for better exhibition, as it separates initialization from output iterations. This run can be removed by moving the output instruction (line 10) to the end of the second loop, and replacing the last run ($k = K$) of the first loop with a run of second loop by setting $w = K$ and $i_w = 0$.

The following theorem considers only a single execution of K-way LCP-merging, since this is what we need in our NUMA scenario:

Theorem 3 *Algorithm 4 requires at most $\Delta L + n \log_2 K + K$ character comparisons, where $n = |\mathcal{S}_0|$ is the total number of strings and $\Delta L = L(H_0) - \sum_{k=1}^{K} L(H_k)$ is the sum of increments to LCP array entries.*

Proof We focus on the character comparisons in the sub-function `LCP-Compare`, since Algorithm 4 itself does not contain any character comparisons. As in the proof of Theorem 2, we can account for all *true*-outcomes of the while loop condition in `LCP-Compare` (line 4) using ΔL, since it increments the overall LCP. We can bound the number of *false*-outcomes by bounding the number of calls to `LCP-Compare`, which occurs exactly K times when building the tournament tree, and then $\log_2 K$ times for each of the n output string (excluding the superfluous run in the pseudo-code). As before, this upper bound, $\Delta L + n \log_2 K + K$, is only attained in pathological cases, and for most inputs, cases 2 and 3 in `LCP-Compare` reduce the overall number of character comparisons. □

Theorem 4 *Using Algorithm 4 one can implement a K-way LCP-mergesort algorithm, which requires less than $L + n\lceil \log_K n \rceil \log_2 K + (n-1)\frac{K}{K-1}$ character comparisons and runs in $\mathcal{O}(D + n \log n + nK)$ time.*

Proof We assume the divide step of K-way LCP-mergesort to split into K subproblems of nearly even size. Using Theorem 3 yields the recurrence $T(n) = K \cdot T(\frac{n}{K}) + n \log_2 K + K$ with $T(1) = 0$, if we ignore the character comparisons loop. Assuming $n = K^d$ for some integer d, the recurrence can be solved elementary using induction, yielding $T(n) = n \log_K n \cdot \log_2 K + \frac{K(n-1)}{K-1}$. For $n \neq K^d$, the input cannot be split evenly into recursive subproblems. However, to keep this analysis simple, we use K-way mergesort even when $n < K$, and thus incur the cost of Theorem 3 also at the base level. So, we have $\lceil \log_K n \rceil$ levels of recursion. As in previous proofs, we account for all matching character comparisons with L, and all others with the highest number of occurrences of case 1 in `LCP-Compare` in the whole recursion, which is $T(n)$. Since $L \leq D$, the run time follows. □

In the proof we assume K-way LCP-merge even in the base level. In an implementation, one would chose a different merger when $n < K$. By selecting 2-way LCP-merge, the number of comparisons in the lowest recursion is reduced, and we can get a bound of $L + n \log_2 n + \mathcal{O}(nK)$.

5.3 Practical Parallelization of K-way LCP-Merge

We now discuss how to parallelize K-way LCP-merge when given K sorted input streams. The problem is that merging itself cannot be parallelized without significant overhead [10], as opposed to the classification and distribution in pS^5. Instead, we want to split the problem into disjoint areas of independent work, as done commonly in practical parallel multiway mergesort sorting algorithms and implementations [3,29].

In contrast to atomic merging, a perfect split with respect to the number of elements in the subproblems by no means guarantees good load balance for string

merging. Rather, the amount of work in each piece depends on the unknown values of the common prefixes. Therefore, dynamic load balancing is needed anyway and we can settle for a simple and fast routine for splitting the input into pieces that are small enough to allow good load balance. We now outline our current approach. Since access to string characters incurs costly cache faults, we want to use the information in the LCP array to help split the input streams. In principle, in the following heuristic we merge the top of the LCP interval trees [1] of the K input streams to find independent areas.

If we consider all occurrences of the global minimum in an LCP array, then these split the input stream into disjoint areas starting with the same distinct prefix. The only remaining challenge is to match equal prefixes from the K input streams, and for this matching we need to inspecting the first distinguishing characters of any string in the area. Matching areas can then be merged independently.

Depending on the input, considering only the global LCP minima may not yield enough independent work. However, we can apply the same splitting method again on matching sub-areas, within which all strings have a longer common prefix, and the global minimum of the sub-area is larger.

We put these ideas together in a splitting heuristic, which scans the K input LCP arrays sequentially once, and creates merge jobs while scanning. We start by reading w characters from the first string of all K input streams, and select those inputs with the smallest character block $\bar{c}$. In each of these selected inputs, we scan the LCP array forward, skipping over all entries $> w$, and checking entries $= w$ for equal character blocks, until either an entry $< w$ or a mismatching character block is found. This forward scan encompasses all strings with prefix $\bar{c}$, and an independent merge job can be started. The process is then repeated with the next strings on all K inputs.

We start the heuristic with $w = 8$ (loading a 64-bit register full of characters), but reduce w depending on how many jobs are started, as otherwise the heuristic may create too many splits, e.g. for random input strings. We therefore calculate an expected number of independent jobs, and adapt w depending on how much input is left and how many jobs were already created. This adaptive procedure keeps w high for inputs with high average common prefix and low otherwise.

We use the same load balancing framework as with pS^5 (see Section 4.5). During merge jobs, we check if other threads are idle via the global unsynchronized counter variable. To reduce balancing overhead, we check only every 4 096 processed strings. If idle threads are detected, then a K-way merge job is split up into further independent jobs using the same splitting heuristic, except that a common prefix of all strings may be known, and is used to offset the character blocks of size w.

5.4 Implementation Details

Our experimental platforms have $m \in \{4, 8\}$ NUMA nodes, and we use parallel K-way LCP-merge only as a top-level merger on m input streams. Thus we assume the N inputs characters to be divided evenly onto the m memory nodes. On the individual NUMA memory nodes, we pin about $\frac{p}{m}$ threads and run pS^5 on the string subset. We do not rebalance the string sets or thread associations when pS^5 finishes early on one of the NUMA nodes, as the synchronization overhead was

more costly than the gain. For skewed inputs, however, this problem remains to be solved.

Since K-way LCP-merge requires the LCP arrays of the sorted sequences, we extended pS^5 to optionally save the LCP value while sorting. The string pointers and LCP arrays are kept separate, as opposed to interleaving them as "annotated" strings [24]. This was done, because pS^5 already requires an additional pointer array during out-of-place redistribution. The additional string array and the original string array are alternated between in recursive calls. When a subset is finally sorted, the correctly ordered pointers are copied back to the original array, if necessary. This allows us to place the LCP values in the additional array.

The additional work and space needed by pS^5 to save the LCP values is very small, we basically get LCPs for free. Most LCPs are calculated in the base case sorter of pS^5, and hence we describe LCP-aware insertion sort in the next section. All other LCPs are located at the boundaries of buckets separated by either multikey quicksort or string sample sort. We calculate these boundary LCPs after recursive levels are finished, and use the saved splitters or pivot elements whenever possible.

The splitting heuristic of parallel K-way LCP-merge creates jobs with varying K, and we created special implementations for the 1-way (plain copying) and 2-way (binary merging) cases, while all other K-way merges are performed using the LCP-aware tournament tree.

To make parallel K-way LCP-merge more cache- and NUMA transfer-efficient, we devised a *caching variant*. In `LCP-Compare` the first character needed for additional character comparisons during the merge can be predicted (if comparisons occur at all). This character is the distinguishing character between two strings, which we label $\hat{c}_i = s_i[h_i]$, where $h_i = \mathrm{lcp}_{\mathcal{S}}(i)$. Caching this character while sorting is easy, since it is loaded in a register when the final, distinguishing character comparison is made. We extended pS^5 to save $\hat{c}_i$ in an additional array and employ it in a modified variant of `LCP-Compare` to save random accesses across NUMA nodes. Using this caching variant all character comparisons accounted for in the $n \log_2 K + K$ term in Theorem 3 can be done using the cached $\hat{c}_i$, thus only ΔL random accesses to strings are needed for a K-way merge.

5.5 LCP-Insertion Sort

As mentioned in the preceding section, we extended pS^5 to save LCP values. Thus its base-case string sorter, insertion sort, also needed to be extended. Again, saving and reusing LCPs during insertion sort is a very intuitive idea, but we found no reference or analysis in the literature.

Assuming the array $\mathcal{S} = [\, s_1, \ldots, s_{j-1} \,]$ is already sorted and the associated LCP array H is known, then insertion sort locates a position i at which a new string $x = s_j$ can be inserted, while keeping the array sorted. After insertion, at most the two LCP values h_i and h_{i+1} may need to be updated. While scanning for the correct position i, customarily from the right, values of both $\mathcal{S}$ and H can already be shifted to allocate a free position.

Using the information in the preliminary LCP array, the scan for i can be accelerated by skipping over certain areas in which the LCP value attests a mismatch. The scan corresponds to walking down the LCP interval tree, testing only

Algorithm 5: LCP-InsertionSort

Input: $\mathcal{S} = \{s_1, \ldots, s_n\}$ is a set of strings with common prefix $\overline{h}$

```
1  for j = 1,...,n do                    // Insert x = s_j into sorted sequence [s_1,...,s_{j-1}].
2      i := j,  (x, h') := (s_j, h̄)      // Start candidate LCP h' with common prefix h̄.
3      while i > 1 do
4          if h_i < h' then break        // case 1: LCP decreases ⇒ insert after s_{i-1}.
5          else if h_i = h' then         // case 2: LCP equal ⇒ compare more characters.
6              p := h'                   // Save LCP of x and s_i.
7              while x[h'] ≠ 0 and x[h'] = s_{i-1}[h'] do   // Compare characters.
8                  h'++
9              if x[h'] ≥ s_{i-1}[h'] then   // If x is larger, insert x after s_{i-1},
10                 h_i := h',  h' := p       // set h_i, the LCP of s_{i-1} and x, but
11                 break                     // set h_{i+1} after loop in line 14.
12         (s_i, h_{i+1}) := (s_{i-1}, h_i)  // case 3: LCP is larger ⇒ no comparison needed.
13         i--
14     (s_i, h_{i+1}) := (x, h')         // Insert x at correct position, update h_{i+1} with LCP.
```

Output: $\mathcal{S} = [\, s_1, \ldots, s_n \,]$ is sorted and has the LCP array $[\, \bot, h_2, \ldots, h_n \,]$

one item of each child node, and descending down if it matches. In plainer words, areas of strings with a common prefix can be identified using the LCP array (as already mentioned in Section 5.3), and it suffices to *check once* if the candidate matches this common prefix. If not, then the whole area can be skipped.

In the pseudo-code of Algorithm 5, the common prefix of the candidate x is kept in h', and increased while characters match. When a mismatch occurs, the scan is continued to the left, and all strings can be skipped if the LCP reveals that the mismatch position remains unchanged (case 3). If the LCP goes below h', then a smaller strings precedes and therefore the insertion point i is found (case 1). At positions with equal LCP more characters need to be compared (case 2). In the pseudo-code these three cases are fused with a copy-loop moving items to right. Note that the pseudo-code sets $h_{n+1} := \overline{h}$ in the last iteration when $i = j = n$, which requires a sentinel array position or an out-of-bounds check in the last iteration.

Theorem 5 *LCP-aware insertion sort (Algorithm 5) requires at most* $L + \frac{n(n-1)}{2}$ *character comparisons and runs in* $\mathcal{O}(D + n^2)$ *time.*

Proof The only lines containing character comparisons in Algorithm 5 are lines 7 and 9. If the while loop condition is *true*, then h' is incremented. In the remaining algorithm the value of h' is only shifted around, never discarded or decreased. Thus we can count the number of comparisons yielding a while-loop repetition with L. The while loop is encountered at most $\frac{n(n-1)}{2}$ times, as this is the highest number of times the inner loop in lines 4–13 runs. We can regard the exiting comparison of line 7 and the following comparison in line 9 as one ternary comparison, as the same values are checked again. This ternary comparison occurs at most once each run of the inner loop, thus $\frac{n(n-1)}{2}$ times. With $L \leq D$, the run time follows

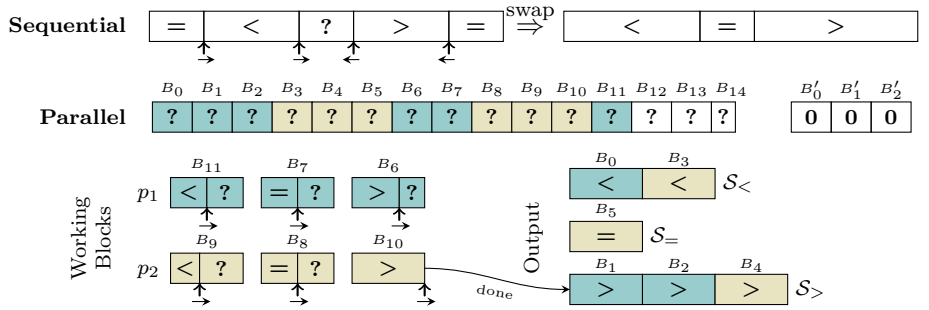

Fig. 3 Block schema of sequential and parallel multikey quicksort's ternary partitioning process

from the number of iterations of the for loop (line 1–14) and the while loop (lines 3–13). □

We close with the remark that non-LCP insertion sort requires $\mathcal{O}(nD)$ steps in the worst case, when all strings are equal except for the last character.

6 More Shared Memory Parallel String Sorting

6.1 Parallel Radix Sort

Radix sort is very similar to sample sort, except that classification is much faster and easier. Hence, we can use the same parallelization toolkit as with S^5. Again, we use three sub-algorithms for differently sized subproblems: fully parallel radix sort for the original string set and large subsets, a sequential radix sort for medium-sized subsets and insertion sort for base cases. Fully parallel radix sort consists of a counting phase, global prefix sum and a redistribution step. Like in S^5, the redistribution is done out-of-place by copying pointers into a shadow array.

We experimented with 8-bit and 16-bit radixes for the fully parallel step. Smaller recursive subproblems are processed independently by sequential radix sort with in-place permuting, and here we found 8-bit radixes to be faster than 16-bit sorting. Our parallel radix sort implementation uses the same work balancing method as parallel S^5, freeing the largest subproblems when other threads are idle.

6.2 Parallel Caching Multikey Quicksort

Our preliminary experiments with sequential string sorting algorithms (see Section A) showed a surprise winner: an enhanced variant of multikey quicksort by Tommi Rantala [25] often outperformed more complex algorithms.

This variant employs both caching of characters and uses a super-alphabet of $w = 8$ characters, exactly as many as fit into a machine word. The string pointer array is augmented with w cache bytes for each string, and a string subset is *partitioned by a whole machine word* as splitter. Thereafter, the cached characters

are reused for the recursive subproblems $\mathcal{S}_<$ and $\mathcal{S}_>$, and access to strings is needed only for sorting $\mathcal{S}_=$, unless the pivot contains a zero-terminator. In this section "caching" means *copying* of characters into another array, not necessarily into the processor's cache. Key to the algorithm's good performance is the following observation:

Theorem 6 *Caching multikey quicksort needs at most* $\lfloor \frac{D}{w} \rfloor + n$ *(random) accesses to string characters in total, where* w *is the number of characters cached per access.*

Proof Per string access w characters are loaded into the cache, and these w characters are never fetched again. We can thus account for all accesses to distinguishing characters using $\lfloor \frac{D}{w} \rfloor$, since the characters are fetched in blocks of size w. Beyond these, at most one access per string can occur, which accounts for fetching w characters of which not all are need for sorting.

In light of this variant's good performance, we designed a parallelized version. We use three sub-algorithms: *fully parallel caching multikey quicksort*, the original sequential caching variant (with explicit recursion stack) for medium and small subproblems, and insertion sort as base case. For the fully parallel sub-algorithm, we generalized a block-wise processing technique from (two-way) parallel atomic quicksort [33] to three-way partitioning.

The input array is viewed as a sequence of blocks containing B string pointers together with their w cache characters (see Figure 3). Each thread holds exactly three blocks and performs ternary partitioning by a globally selected pivot. When all items in a block are classified as $<$, $=$ or $>$, then the block is added to the corresponding output set $\mathcal{S}_<$, $\mathcal{S}_=$, or $\mathcal{S}_>$. This continues as long as unpartitioned blocks are available. If no more input blocks are available, an extra empty memory block is allocated and a second phase starts. The second partitioning phase ends with fully classified blocks, which might be only partially filled. Per fully parallel partitioning step there can be at most $3p$ partially filled blocks. The output sets $\mathcal{S}_<$, $\mathcal{S}_=$, and $\mathcal{S}_>$ are processed recursively with threads divided as evenly among them as possible. The cached characters are updated only for the $\mathcal{S}_=$ set.

In our implementation we use atomic compare-and-swap operations for block-wise processing of the initial string pointer array and Intel TBB's lock-free queue for sets of blocks, both as output sets and input sets for recursive steps. When a partition reaches the threshold for sequential processing, then a continuous array of string pointers plus cache characters is allocated and the block set is copied into it. On this continuous array, the usual ternary partitioning scheme of multikey quicksort is applied sequentially. Like in the other parallelized algorithms, we use dynamic load balancing and free the largest level when re-balancing is required. We empirically determined $B = 128\,\text{Ki}$ as a good block size.

6.3 Burstsort

Burstsort is one of the fastest string sorting algorithms and cache-efficient for many inputs, but it looks difficult to parallelize it. Keeping a common burst trie would require prohibitively many synchronized operations, while building independent burst tries on each PE would lead to the question how to merge multiple tries of different structure. This problem of merging tries is related to parallel K-way LCP-merge, and future work may find a way to combine these approaches.

7 Experimental Results

We implemented parallel versions of S^5, K-way LCP-merge, multikey quicksort and radix sort in C++ and compare them with the few parallel string sorting implementations we could find online in Section 7.3. We also integrated many sequential implementations into our test framework, and discuss their performance in Section A. Our implementations, the test framework and most input sets are available from `http://panthema.net/2013/parallel-string-sorting`.

7.1 Experimental Setup

We tested our implementations and those by other authors on five different platforms. All platforms run Linux and their main properties are listed in Table 1. We compiled all programs using gcc 4.6.3 with optimizations `-O3 -march=native`. The five platforms were chosen to encompass a wide variety of multi-core systems, which exhibit different characteristics in their memory system and also cover today's most popular hardware. By experimenting on a large number of systems (and inputs), we demonstrate how robust our implementations and algorithm designs are.

The test framework sets up a separate environment for each run. To isolate heap fragmentation, it was very important to fork() a child process for each run. The string data is loaded before the fork(), allocating exactly the matching amount of RAM, and shared read-only with the child processes. No precaution to lock the program's memory into RAM was taken (as opposed to previous experiments reported in [7]). Turbo-mode was disabled on IntelE5.

Before an algorithm is called, the string pointer array is generated inside the child process by scanning the string data for zero characters, thus flushing caches and TLB entries. Time measurement is done with clock_gettime() and encompasses only the sorting algorithm. Because many algorithms have a deep recursion stack for our large inputs, we increased the stack size limit to 64 MiB. For non-NUMA experiments, we took no special precautions of pinning threads to specific cores or nodes, and used the default Linux task scheduling system as is. Memory for NUMA-unaware algorithms was interleaved across all nodes by setting the default allocation policy.

For our experiments with NUMA-aware string sorting, the characters array is segmented equally onto the NUMA memory banks before running an algorithm. The algorithm then pins its threads to the appropriate node, enabling node-local memory access. Additional allocations are also taken preferably from the local memory node.

The output of each string sorting algorithm was verified by first checking that the resulting pointer list is a permutation of the input set, and then checking that strings are in non-descending order. The input was shared read-only with the algorithm's process and thus cannot have been modified.

Methodologically we have to discuss, whether measuring only the algorithm's run time is a good decision. The issue is that deallocation and defragmentation in both heap allocators and kernel page tables is done lazily. This was most notable when running two algorithms consecutively. The fork() process isolation excludes both variables from the experimental results, however, for use in a real program

Table 1 Hard- and software characteristics of experimental platforms

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
AMD16	AMD Opteron 8350	2.0	4 × 4	16 × 64	16 × 512	4 × 2	64
Inteli7	Intel Core i7 920	2.67	1 × 4 × 2	4 × 32	4 × 256	1 × 8	12
IntelX5	Intel Xeon X5355	2.67	2 × 4 × 1	8 × 32	4 × 4096		16

Name	Codename	Memory Channels	NUMA Nodes	Interconnect	Linux/Kernel Version
IntelE5	Sandy Bridge	4 × DDR3-1600	4	2 × 8.0 GT/s QPI	Ubuntu 12.04/3.2.0
AMD48	Magny-Cours	4 × DDR3-667	8	4 × 3.2 GHz HT	Ubuntu 12.04/3.2.0
AMD16	Barcelona	2 × DDR2-533	4	3 × 1.0 GHz HT	Ubuntu 10.04/2.6.32
Inteli7	Bloomfield	3 × DDR3-800		1 × 4.8 GT/s QPI	openSUSE 11.3/2.6.34
IntelX5	Clovertown	2 × DDR2-667		1 × 1.3 GHz FSB	Ubuntu 12.04/3.2.0

context these costs cannot be ignored. We currently do not know how to invoke the lazy cleanup procedures to regenerate a pristine memory environment. These issues must be discussed in greater detail in future work for sound results with big data in RAM. We briefly considered HugePages, but these did not yield a performance boost. This is probably due to random accesses being the main time cost of string sorting, while the number of TLB entries is not a bottleneck.

7.2 Inputs

We selected the following datasets, all with 8-bit characters. Most important characteristics of these instances are shown in Table 2.

URLs contains all URLs found on a set of web pages which were crawled breadth-first from the author's institute website. They include the protocol name.

Random (from [31]) are strings of length $[0:20)$ over the ASCII alphabet $[33:127)$, with both length and characters chosen uniformly random.

GOV2 is a TREC test collection consisting of 25 million HTML pages, PDF and other documents retrieved from websites under the .gov domain. We consider the whole corpus for line-based string sorting, concatenated by document id.

Wikipedia is an XML dump of the most recent version of all pages in the English Wikipedia, which was obtained from `http://dumps.wikimedia.org/`; our dump is dated `enwiki-20120601`. Since the XML data is not line-based, we perform *suffix sorting* on this input.

We also include the three largest inputs Ranjan **Sinha** [31] tested burstsort on: a set of **URLs** excluding the protocol name, a sequence of genomic strings of length 9 over a **DNA** alphabet, and a list of non-duplicate English words called **NoDup**. The "largest" among these is NoDup with only 382 MiB, which is why we consider these inputs more as reference datasets than as our target.

The inputs were chosen to represent both real-world datasets, and to exhibit extreme results when sorting. Random has a very low average LCP, while URLs have a high average LCP. GOV2 is a general text file with all possible ASCII characters, and Sinha's DNA has a small alphabet size. By taking suffixes of Wikipedia

Table 2 Characteristics of the selected input instances.

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 URLs	10 M	304 Mi	97.5 %	29.4	114	31.9
Sinha DNA	31.6 M	302 Mi	100 %	9.0	4	10.0
Sinha NoDup	31.6 M	382 Mi	73.4 %	7.7	62	12.7

we have a very large sorting problem instance, which needs little memory for characters.

Our inputs are very large, one infinite, and most of our platforms did not have enough RAM to process them. For each platform, we determined a large prefix $[0:n)$, which can be processed with the available RAM and time, and leave sorting of the remainder to future work.

7.3 Performance of Parallel Algorithms

In this section we report on our experiments on the platforms shown in Table 1, which contains a wide variety of multi-core machines of different age. The results plotted in Figures 4–8 show the speed up of each parallel algorithm over the best sequential one, for increasing thread count. Tables 7–13 show absolute running times of our experiments, with the fastest algorithm's time highlighted in bold text.

Overall, our parallel string sorting implementations yield high speedups, which are generally much higher than those of all previously existing parallel string sorters. Each individual parallel algorithm's speedup depends highly on hardware characteristics like processor speed, RAM and cache performance[2], the interconnection between sockets, and the input's characteristics. In general, the speedup of string sorting for high thread counts is bounded by memory bandwidth, not processing power. On both non-NUMA platforms (Figures 7, 8), our implementations of pS^5 are the string sorting algorithm with highest speedups, except for Random and Sinha's NoDup inputs. On NUMA many-core platforms, the picture is more complex and results mostly depend on how well the inner loops and memory transfers are optimized on each particular system.

The parallel experiments cover all algorithms we describe in this paper: pS^5-Unroll is a variant of pS^5 from Section 4, which interleaves three unrolled descents of the classification tree, while pS^5-Equal unrolls only a single descent, but tests equality at each splitter node. In the NUMA-aware variant called "pS^5-Unroll + pLCP-Merge" we first run pS^5-Unroll independently on each NUMA node for separate parts of the input, and then merge the presorted parts using our parallel K-way LCP-merge algorithm (Section 5). From the additional parallel algorithms in Section 6, we draw our parallel multikey quicksort (pMKQS) implementations, and radix sorts with 8-bit and 16-bit fully parallel steps. Furthermore, we included

[2] See http://panthema.net/2013/pmbw/ for parallel memory bandwidth experiments

the parallel radix sort implemented by Takuya Akiba [2] in the experiments on all platforms.

For the tests on Inteli7 and IntelX5, we added three more parallel implementations: pMKQS-SIMD is a multikey quicksort implementation from Rantala's library, which uses SIMD instructions to perform vectorized classification against a single pivot. We improved the code to use OpenMP tasks for recursive sorting steps. The second implementation is a parallel 2-way LCP-mergesort also by Rantala, which we also augmented with OpenMP tasks. However, only recursive merges are run in parallel, the largest merge is performed sequentially. The implementation uses insertion sort for $|\mathcal{S}| < 32$, all other sorting is done via merging. N. Shamsundar's parallel LCP-mergesort is the third additional implementation, but it also uses only 2-way merges. As seen in Figures 7–8, only Akiba's radix sort scales fairly well, which is why we omitted the other three algorithms on the platforms with more than eight cores.

Inteli7 (Figure 7, Table 10–11) is a consumer-grade, single socket machine with fast RAM and cache hierarchy. *IntelX5* (Figure 8, Table 12–13) is our oldest architecture, and shows the slowest absolute speedups. Both are not NUMA architectures, which is why we did not run our NUMA-aware algorithm on them. They are more classic architectures, and exhibit most of the effects we targeted in our algorithms to gain good speedups. Our pS^5 variants are fastest on all inputs, except very random ones (Random and NoDup), where radix sorts are slightly faster on Inteli7. Remarkably, on IntelX5 radix sort was not faster. We suspect that newer processors can optimize the inner loops of radix sort (counting, prefix sums and data redistributions with few input/output streams [20]) better than older ones. Our pMKQS also shows good overall speedups, but is never particularly fast. This is due to the high memory bandwidth caching multikey quicksort requires, as it reads and rereads an array to just partition by one pivot.

For all test instances, except URLs, the fully parallel sub-algorithm of pS^5 was run only 1–4 times. Thereafter, the input was split up into enough subsets, and most of the additional speedup is gained by load-balancing the sequential sub-algorithms well. The pS^5-Equal variant handles URL instances better, as many equal matches occur here. However, for all other inputs, pS^5-Unroll with interleaved tree descents fares better, even though it has higher theoretical running time.

Comparing our radix sorts with Akiba's we already see the implementation's main problems: it does not parallelize recursive sorting steps (only the top-level is parallelized) and only performs simple load balancing. This can be seen most pronounced on URLs and GOV2. All three additional implementations, pMKQS-SIMD, pMergesort-2way by Rantala, and the same by Shamsundar do not show any good speedup, partly because they are already pretty slow sequentially, and partly because they are not fully parallelized.

On the Inteli7 machine, which has four real cores and four Hyper-Threading cores, pS^5 achieves speedups ≥ 3.2 on all inputs, except Random where it gains only 2.5. This is remarkable, as the machine has only three memory channels, and a single core can fully utilize two of them. Thus in pS^5 a lot of computation work is parallelized. On IntelX5, which has eight real cores, pS^5 achieves speedups ≥ 3 on all inputs. We attribute this to the early dual-socket architecture, on which many other parallel implementations also do not scale well.

Table 3 Description of parallel string sorting algorithms in experiment

Name	Description and Author
pS^5-Unroll	Our parallel super scalar string sample sort (see Section 4) with unrolled and interleaved tree descents.
pS^5-Equal	Our parallel super scalar string sample sort (see Section 4) with equality checking at each splitter node.
pMKQS	Our parallel multikey quicksort (see Section 6.2) with caching of $w = 8$ characters.
pS^5+LCP-M	Our parallel multiway LCP-merge with pS^5 on each NUMA node.
pRS-8bit	Our parallel radix sort (see Section 6.1) with 8-bit alphabet
pRS-16bit	Our parallel radix sort (see Section 6.1) with 16-bit alphabet at the fully parallel levels, and 8-bit alphabets for sequentially processed subproblems.
pRS/Akiba	Takuya Akiba's [2] radix sort.
p2w-MS/R	Parallel 2-way LCP-mergesort from Tommi Rantala's library [25].
pMKQS-SIMD/R	Parallel multikey quicksort with SIMD operations from Tommi Rantala's library [25].
pLCP-2w-MS/S	Parallel 2-way LCP-mergesort by Nagaraja Shamsundar [28], which is based on Waihong Ng's LCP-mergesort [24].

IntelE5 (Figure 4, Table 7) is our newest machine with 32 real cores across four sockets with one NUMA node each. It contains one of Intel's most recent many-core processors. *AMD48* (Figure 5, Table 8) is a somewhat older AMD many-core machine with high core count, but relatively slow RAM and a slower interconnect. Compared to the previous results on Inteli7, we notice that parallel multikey quicksort (pMKQS) is very fast, and achieves slightly higher speedups than pS^5 on most inputs on IntelE5 and significantly higher ones on AMD48. This effect is clearly due to pS^5 ignoring the NUMA architecture and thus incurring a relatively large number of expensive inter-node random string accesses. We analyzed the number of string access of pMKQS in Theorem 6, after which the characters are saved and accessed in a scanning pattern. This scanning apparently works well on the NUMA machines, as it is very cache-efficient, can be easily predicted by the processor's memory prefetcher, and a costly inter-node transfered cache line contains saved characters of eight strings.

These expected results were the reason to focus on NUMA-aware string sorting algorithms, and to develop parallel K-way LCP-merge for top-level merging of presorted sequences. In our experiments we ran "pS^5-Unroll + pLCP-Merge" only when there is at least one thread per NUMA node. We tried to rebalance threads to other NUMA nodes once work on a node is done, but this did not work well, since the additional inter-node synchronization was too costly. We thus have to leave the question of how to balance sorting work on NUMA systems for highly skewed inputs open to future research. Plain LCP-merge also contains costly inter-node random string accesses in case 1 of `LCP-Compare`. As predicted in Section 5.4, we saw a huge speed improvement due to *caching* of just the distinguishing character $\hat{c}$, and don't consider the non-caching variant in our results.

On IntelE5, with four NUMA nodes, pS^5-Unroll + pLCP-Merge reaches the highest speedups on URLs, GOV2 inputs and Wikipedia suffixes. On the AMD48 machine with eight NUMA nodes, random access is even more costly and the inter-node connections are easily congested, which is why pMKQS fairs better against our NUMA-aware sorting. In future (possibly the next revision of this journal paper), experiments with caching more than just one character may lead to larger

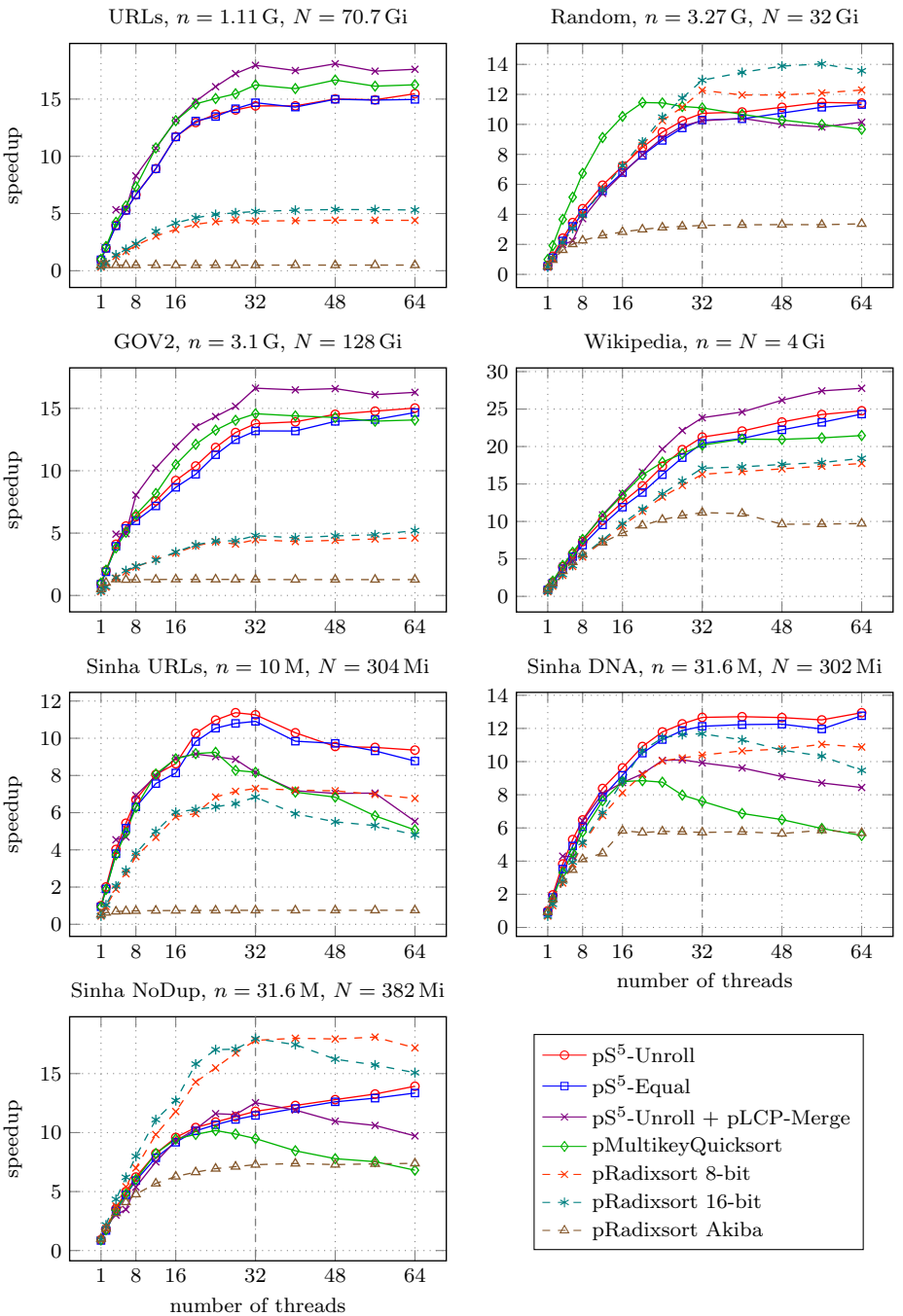

Fig. 4 Speedup of parallel algorithm implementations on IntelE5, median of 1–3 runs

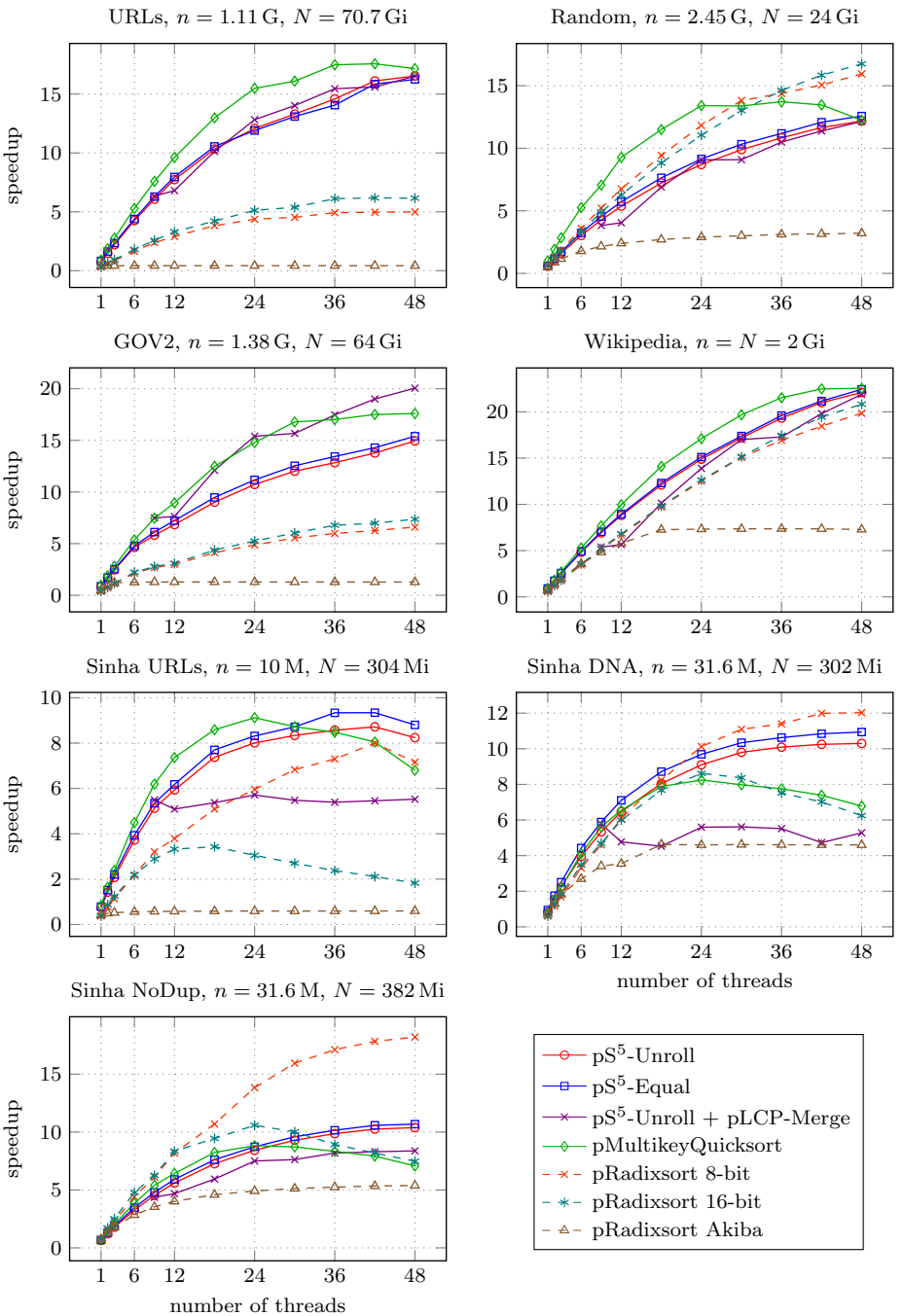

Fig. 5 Speedup of parallel algorithm implementations on AMD48, median of 1–3 runs

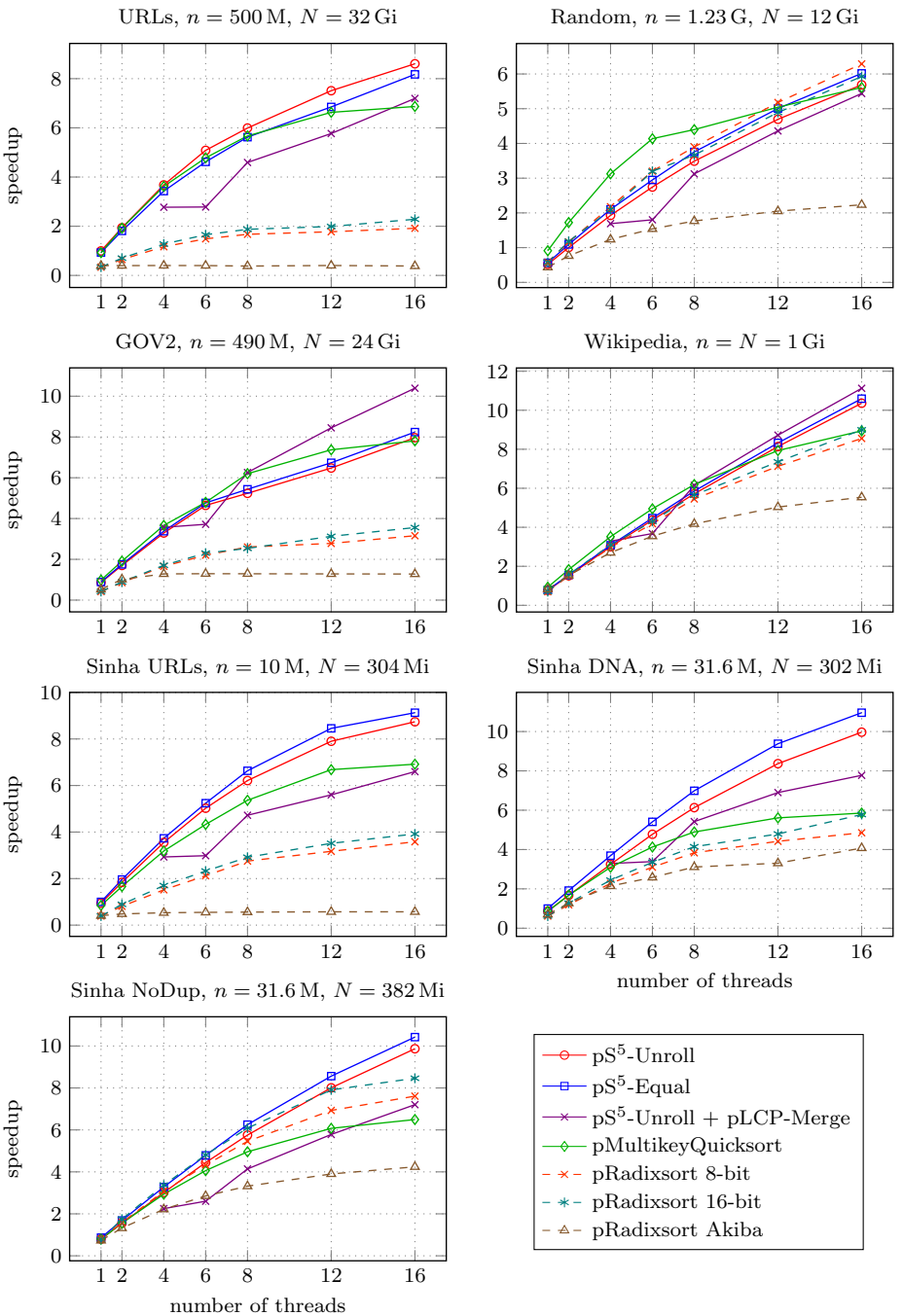

Fig. 6 Speedup of parallel algorithm implementations on AMD16, median of 1–3 runs

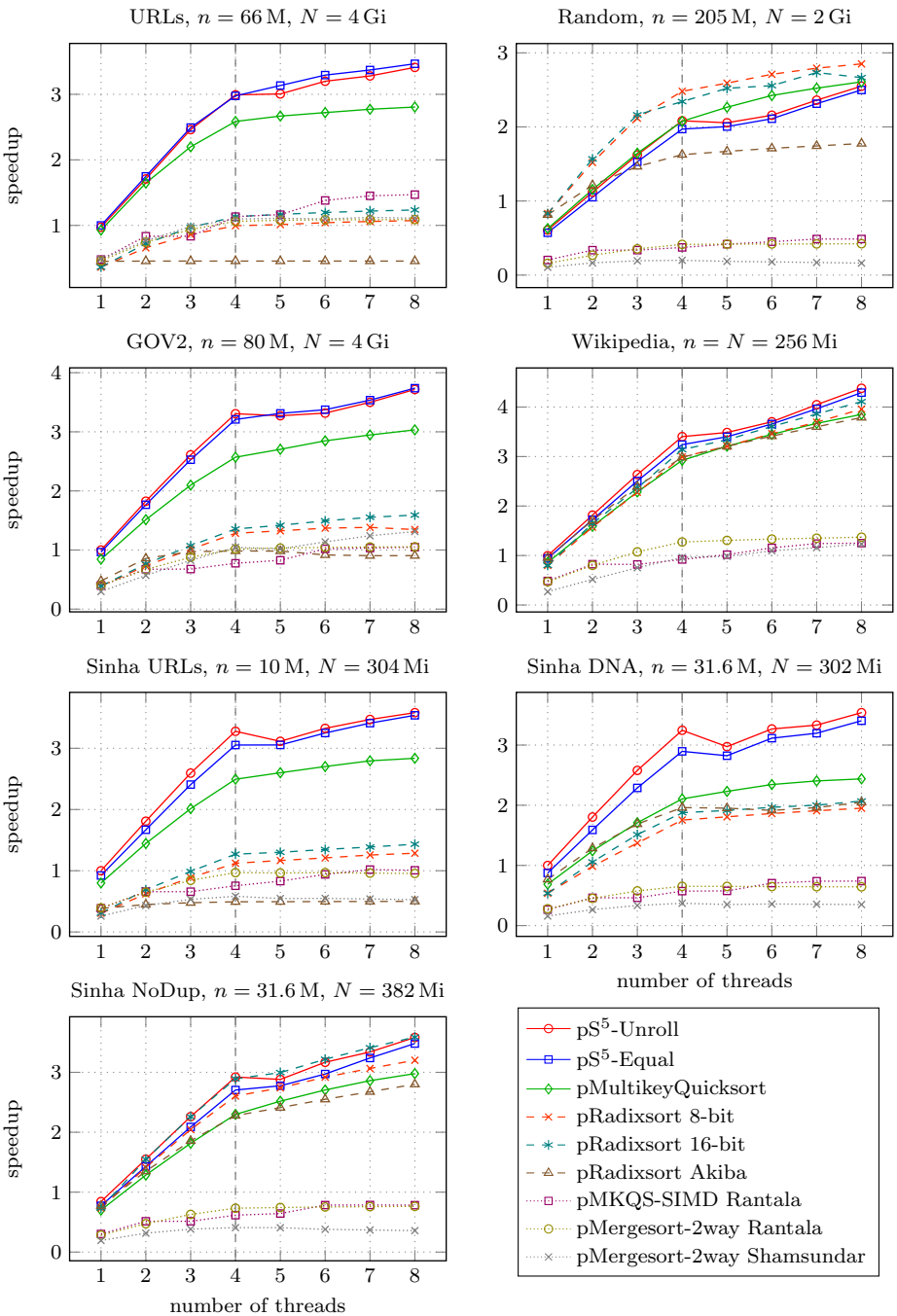

Fig. 7 Speedup of parallel algorithm implementations on Inteli7, median of fifteen runs

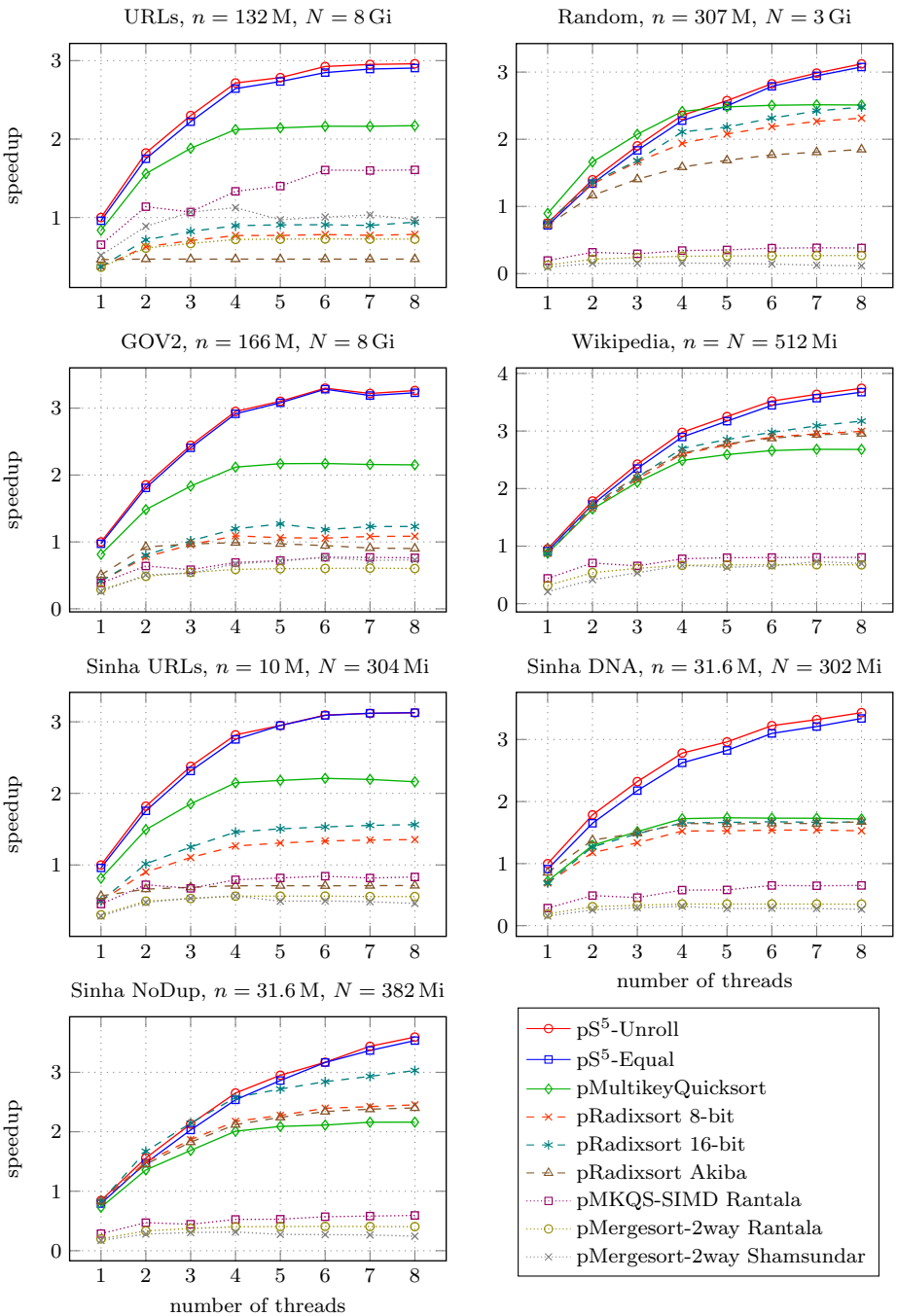

Fig. 8 Speedup of parallel algorithm implementations on IntelX5, median of fifteen runs

speedups on these NUMA systems. Remarkably, radix sort is still very fast on both NUMA machines for random inputs.

The lower three plots in Figure 4 and 5 show that on these large many-core platforms, parallel sorting becomes less efficient for small inputs (around 300 MiB). This is expected due to the high cost of synchronization, but our parallel algorithms still fare well.

AMD16 (Figure 6, Table 9) is an earlier NUMA architecture with four NUMA nodes, and the slowest RAM speed and interconnect in our experiment. However, on this machine random access, memory bandwidth and processing power (in cache) seems to be more balanced for pS^5 than on the newer NUMA machines.

We included the absolute running times of all our speedup experiments in Tables 7–13 for reference and to show that our parallel implementations scale well both for very large instances on many-core platforms and also for small inputs on machines with fewer cores.

8 Conclusions and Future Work

We have demonstrated that string sorting can be parallelized successfully on modern multi-core shared memory and NUMA machines. In particular, our new string sample sort algorithm combines favorable features of some of the best sequential algorithms – robust multiway divide-and-conquer from burstsort, efficient data distribution from radix sort, asymptotic guarantees similar to multikey quicksort, and word parallelism from caching multikey quicksort. For NUMA machines we developed parallel K-way LCP-merge to further decrease costly inter-node random access.

Both algorithms are practical for many applications, and our implementations are available as templates for further customization. For general use, our pS^5 implementation is recommended, as it works reliably well on all platforms.

We want to highlight that using our pS^5 (which can save LCPs) and K-way LCP-merge implementations it is straight-forward to construct a fast parallel external memory string sorter for short strings ($\leq B$) using shared memory parallelism. The sorting throughput of our string sorters is probably higher than the available I/O bandwidth.

Implementing some of the refinements discussed in the next section are likely to yield further improvements for string sample sort and K-way LCP-merge.

As most important vectors of future work, we see the splitting heuristic of LCP-Merging, and how to rebalance work for skewed inputs on NUMA machines.

8.1 Practical Refinements

Memory conservation: For use of our algorithms in applications like database systems or MapReduce libraries, it is paramount to give hard guarantees on the amount of memory required by the implementations. Our experiments show clearly, that caching of characters accelerates string sorting, but this speed comes at the cost of memory space. A future challenge is thus to sort fast, but with limited memory. In this respect, pS^5 is a very promising candidate, as it can be

restricted to use only the classification tree and a recursion stack, if little additional memory is available. But if more memory is available, then caching, saving oracle values and out-of-place redistribution can be enabled adaptively.

Multipass data distribution: There are two constraints for the maximum sensible value for the number of splitters v: The cache size needed for the classification data structure and the resources needed for data distribution. Already in the plain external memory model these two constraints differ ($v = \mathcal{O}(M)$ versus $v = \mathcal{O}(M/B)$). In practice, things are even more complicated since multiple cache levels, cache replacement policy, TLBs, etc. play a role. Anyway, we can increase the value of v to the value required for classification by doing the data distribution in multiple passes (usually two). Note that this fits very well with our approach to compute oracles even for single pass data distribution. This approach can be viewed as LSD radix sort using the oracles as keys. Initial experiments indicate that this could indeed lead to some performance improvements.

Alphabet compression: When we know that only $\sigma' < \sigma$ different values from Σ appear in the input, we can compress characters into $\lceil \log \sigma' \rceil$ bits. For S^5, this allows us to pack more characters into a single machine word. For example, for DNA input, we might pack 32 characters into a single 64 bit machine word. Note that this compression can be done on the fly without changing the input/output format and the compression overhead is amortized over $\log v$ key comparisons.

Jump tables: In S^5, the a most significant bits of a key are often already sufficient to define a path in the classification tree of length up to a. We can exploit this by precomputing a jump table of size 2^a storing a pointer to the end of this path. During element classification, a lookup in this jump table can replace the traversal of the path. This might reduce the gap to radix sort for easy instances.

Using tries in practice: The success of burstsort indicates that traversing tries can be made efficient. Thus, we might also be able to use a tuned trie based implementation of S^5 in practice. One ingredient to such an implementation could be the word parallelism used in the pragmatic solution – we define the trie over an enlarged alphabet. This reduces the number of required hash table accesses by a factor of w. The tuned van Emde Boas trees from [11] suggest that this data structure might work in practice.

Adaptivity: By inspecting the sample, we can adaptively tune the algorithm. For example, when noticing that already a lot of information[3] can be gained from a few most significant bits in the sample keys, the algorithm might decide to switch to radix sort. On the other hand, when even the w most significant characters do not give a lot of information, then a trie based implementation can be used. Again, this trie can be adapted to the input, for example, using hash tables for low degree trie nodes and arrays for high degree nodes.

Acknowledgements We would like the thank the anonymous reviewer for extraordinarily thorough checking of our algorithms and proofs, and for kind suggestions on how to improve the paper.

[3] The entropy $\frac{1}{n}\sum_i \log \frac{n}{|b_i|}$ can be used to define the amount of information gained by a set of splitters. The bucket sizes b_i can be estimated using their size within the sample.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms (JDA) **2**(1), 53–86 (2004)
2. Akiba, T.: Parallel string radix sort in C++. http://github.com/iwiwi/parallel-string-radix-sort (2011). Git repository accessed November 2012
3. Akl, S.G., Santoro, N.: Optimal parallel merging and sorting without memory conflicts. IEEE Transactions on Computers **100**(11), 1367–1369 (1987)
4. Amir, A., Franceschini, G., Grossi, R., Kopelowitz, T., Lewenstein, M., Lewenstein, N.: Managing unbounded-length keys in comparison-driven data structures with applications to online indexing. SIAM Journal on Computing **43**(4), 1396–1416 (2014)
5. Andersson, A., Nilsson, S.: Implementing radixsort. Journal of Experimental Algorithmics (JEA) **3**, 7 (1998)
6. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: ACM (ed.) 8th Symposium on Discrete Algorithms (SODA), pp. 360–369 (1997)
7. Bingmann, T., Sanders, P.: Parallel string sample sort. In: 21th European Symposium on Algorithms (ESA), no. 8125 in LNCS. Springer-Verlag (2013)
8. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M.: A comparison of sorting algorithms for the connection machine CM-2. In: 3rd Symposium on Parallel Algorithms and Architectures (SPAA), pp. 3–16. ACM (1991)
9. Brent, R.P.: The parallel evaluation of general arithmetic expressions. Journal of the ACM (JACM) **21**(2), 201–206 (1974)
10. Cole, R.: Parallel merge sort. SIAM Journal on Computing **17**(4), 770–785 (1988)
11. Dementiev, R., Kettner, L., Mehnert, J., Sanders, P.: Engineering a sorted list data structure for 32 bit keys. In: 6th Workshop on Algorithm Engineering & Experiments (ALENEX), pp. 142–151. SIAM (2004)
12. Eberle, A.: Parallel multiway LCP-mergesort (2014). Bachelor Thesis, Karlsruhe Institute of Technology, to appear
13. Frazer, W.D., McKellar, A.C.: Samplesort: A sampling approach to minimal storage tree sorting. Journal of the ACM (JACM) **17**(3), 496–507 (1970)
14. Hagerup, T.: Optimal parallel string algorithms: sorting, merging and computing the minimum. In: 16th ACM Symposium on Theory of Computing (STOC), pp. 382–391 (1994)
15. Hoare, C.A.R.: Quicksort. The Computer Journal **5**(1), 10–16 (1962)
16. Kent, C., Lewenstein, M., Sheinwald, D.: On demand string sorting over unbounded alphabets. Theoretical Computer Science **426**, 66–74 (2012)
17. Knöpfle, S.D.: String samplesort (2012). Bachelor Thesis, Karlsruhe Institute of Technology, in German
18. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting And Searching, 2 edn. Addison Wesley Longman Publishing Co., Inc. (1998)
19. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Transactions on Computers **100**(8), 786–793 (1973)
20. Kärkkäinen, J., Rantala, T.: Engineering radix sort for strings. In: 16th International Conference on String Processing and Information Retrieval (SPIRE), no. 5280 in LNCS, pp. 3–14. Springer-Verlag (2009)
21. McIlroy, P.M., Bostic, K., McIlroy, M.D.: Engineering radix sort. Computing Systems **6**(1), 5–27 (1993)
22. Mehlhorn, K., Sanders, P.: Scanning multiple sequences via cache memory. Algorithmica **35**(1), 75–93 (2003)
23. Ng, W., Kakehi, K.: Cache efficient radix sort for string sorting. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **E90-A**(2), 457–466 (2007)
24. Ng, W., Kakehi, K.: Merging string sequences by longest common prefixes. IPSJ Digital Courier **4**, 69–78 (2008)
25. Rantala, T.: Library of string sorting algorithms in C++. http://github.com/rantala/string-sorting (2007). Git repository accessed November 2012
26. Sanders, P.: Fast priority queues for cached memory. Journal of Experimental Algorithmics (JEA) **5**, 7 (2000)
27. Sanders, P., Winkel, S.: Super scalar sample sort. In: 12th European Symposium on Algorithms (ESA), *LNCS*, vol. 3221, pp. 784–796. Springer-Verlag (2004)
28. Shamsundar, N.: A fast, stable implementation of mergesort for sorting text files. http://code.google.com/p/lcp-merge-string-sort (2009). Source downloaded November 2012

29. Singler, J., Sanders, P., Putze, F.: MCSTL: The multi-core standard template library. In: Euro-Par 2007 Parallel Processing, no. 4641 in LNCS, pp. 682–694. Springer-Verlag (2007)
30. Sinha, R., Wirth, A.: Engineering Burstsort: Toward fast in-place string sorting. Journal of Experimental Algorithmics (JEA) **15**, 2.5:1–24 (2010)
31. Sinha, R., Zobel, J.: Cache-conscious sorting of large sets of strings with dynamic tries. Journal of Experimental Algorithmics (JEA) **9**, 1.5:1–31 (2004)
32. Sinha, R., Zobel, J., Ring, D.: Cache-efficient string sorting using copying. Journal of Experimental Algorithmics (JEA) **11**, 1.2:1–32 (2007)
33. Tsigas, P., Zhang, Y.: A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In: 11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP), pp. 372–381. IEEE Computer Society (2003)
34. Wassenberg, J., Sanders, P.: Engineering a multi-core radix sort. In: Euro-Par 2011 Parallel Processing, no. 6853 in LNCS, pp. 160–169. Springer-Verlag (2011)
35. Yang, M.C.K., Huang, J.S., Chow, Y.C.: Optimal parallel sorting scheme by order statistics. SIAM Journal on Computing **16**(6), 990–1003 (1987)

A Performance of Sequential Algorithms

We collected many sequential string sorting algorithms in our test framework. We believe it to contain virtually every string sorting implementation publicly available.

The algorithm library by Tommi Rantala [25] contains 37 versions of radix sort (in-place, out-of-place, and one-pass with various dynamic memory allocation schemes), 26 variants of multikey quicksort (with caching, block-based, different dynamic memory allocation and SIMD instructions), 10 different funnelsorts, 38 implementations of burstsort (again with different dynamic memory managements), and 29 mergesorts (with losertree and LCP caching variants). In total these are 140 original implementation variants, all of high quality.

The other main source of string sorting implementations are the publications of Ranjan Sinha. We included the original burstsort implementations (one with dynamically growing arrays and one with linked lists), and 9 versions of copy-burstsort. The original copy-burstsort code was written for 32-bit machines, and we modified it to work with 64-bit pointers.

We also incorporated the implementations of CRadix sort and LCP-Mergesort by Waihong Ng, and the original multikey quicksort code by Bentley and Sedgewick.

Of the 203 different sequential string sorting variants, we selected the thirteen implementations listed in Table 4 to represent both the fastest ones in a preliminary test and each of the basic algorithms from Section 3. The thirteen algorithms were run on all our five test platforms on small portions of the test instances described in Section 7. Tables 5 and 6 show the results, with the fastest algorithm's time highlighted with bold text.

Cells in the tables without value indicate a program error, out-of-memory exceptions or extremely long runtime. This was always the case for the copy-burstsort variants on the GOV2 and Wikipedia inputs, because they perform excessive caching of characters. On Inteli7, some implementations required more memory than the available 12 GiB to sort the 4 GiB prefixes of Random and URLs.

Table 4 Description of selected sequential string sorting algorithms

Name	Description and Author
std::sort	`gcc`'s standard atomic introsort with full string comparisons.
mkqs	Original multikey quicksort by Bentley and Sedgewick [6].
mkqs_cache8	Modified multikey quicksort with caching of eight characters by Tommi Rantala [25], slightly improved.
radix8_CI	8-bit in-place radix sort by Tommi Rantala [20].
radix16_CI	Adaptive 16-/8-bit in-place radix sort by Tommi Rantala [20].
radixR_CE7	Adaptive 16-/8-bit out-of-place radix sort by Tommi Rantala [20], version CE7 (preallocated swap array, unrolling, sorted-check).
CRadix	Cache efficient radix sort by Waihong Ng [23], unmodified.
LCPMergesort	LCP-mergesort by Waihong Ng [24], unmodified.
Seq-S^5-Unroll	Sequential super scalar string sample sort with interleaved loop over strings, unrolled tree traversal and radix sort as base sorter.
Seq-S^5-Equal	Sequential super scalar string sample sort with equality check, unrolled tree traversal and radix sort as base sorter.
burstsortA	Burstsort using dynamic arrays by Ranjan Sinha [31], from [25].
fbC-burstsort	Copy-Burstsort with "free bursts" by Ranjan Sinha [32], heavily repaired and modified to work with 64-bit pointers.
sCPL-burstsort	Copy-Burstsort with sampling, pointers and only limited copying depth by Ranjan Sinha [32], also heavily repaired.

Over all run instances and platforms, multikey quicksort with caching of eight characters was fastest on 18 pairs, winning the most tests. It was fastest on all platforms for both URL list and GOV2 prefixes, except URL on IntelX5, and on all large instances on AMD48 and AMD16. However, for the NoDup input, short strings with large alphabet, the highly tuned radix sort radixR_CE7 consistently outperformed mkqs_cache8 on all platforms by a small margin. The copy-burstsort variant fbC_burstsort was most efficient on all platforms for DNA, which are short strings with small alphabet. For Random strings and Wikipedia suffixes, mkqs_cache8 or radixR_CE7 was fastest, depending on the platforms memory bandwidth and sequential processing speed. Our own *sequential* implementations of S^5 were never the fastest, but they consistently fall in the middle field, without any outliers. This is expected, since S^5 is mainly designed to be used as an efficient top-level parallel algorithm, and to be conservative with memory bandwidth, since this is the limiting factor for data-intensive multi-core applications.

We also measured the peak memory usage of the sequential implementations using a heap and stack profiling tool[4] for the selected sequential test instances. The bottom of Table 5 shows the results in MiB, excluding the string data array and the string pointer array (we only have 64-bit systems, so pointers are eight bytes). We must note that the profiler considers *allocated virtual memory*, which may not be identical to the amount of physical memory actually used. From the table we plainly see, that the more *caching* an implementation does, the higher its peak memory allocation. However, the memory usage of fbC_burstsort is extreme, even if one considers that the implementation can deallocate and recreate the string data from the burst trie. The lower memory usage of fbC_burstsort for Random is due to the high percentage of characters stored implicitly in the trie structure. The sCPL_burstsort and burstsortA variants bring the memory requirement down somewhat, but they are still high. Some radixsort variants and, most notable, mkqs_cache8 are also not particularly memory conservative, again due to caching. Our sequential S^5 implementation fares well in this comparison because it does no caching and permutes the string pointers in-place (Note that radixsort is used for small string subsets in sequential S^5. This is due to the development history: we finished sequential S^5 before focusing on caching multikey quicksort). For sorting with little extra memory, plain multikey quicksort is still a good choice.

[4] http://panthema.net/2013/malloc_count/, by one of the authors.

Table 5 Run time of sequential algorithms on IntelE5 and AMD48 in seconds, and peak memory usage of algorithms on IntelE5. See Table 4 for a short description of each.

	Our Datasets				Sinha's		
	URLs	Random	GOV2	Wikipedia	URLs	DNA	NoDup
n	66 M	409 M	80.2 M	256 Mi	10 M	31.5 M	31.6 M
N	4 Gi	4 Gi	4 Gi	32 Pi	304 Mi	302 Mi	382 Mi
D/N (D)	92.7 %	43.0 %	69.7 %	(13.6 G)	97.5 %	100 %	73.4 %
L/n	57.9	3.3	34.1	33.0	29.4	9.0	7.7
	IntelE5						
std::sort	122	422	153	287	11.6	26.9	25.4
mkqs	37.1	228	56.7	129	5.67	11.0	10.9
mkqs_cache8	**16.6**	67.1	**25.7**	79.5	**2.03**	4.62	6.02
radix8_CI	48.4	64.3	54.6	90.1	6.12	6.79	6.29
radixR_CE7	37.5	**58.6**	44.6	**72.4**	4.77	4.66	**4.73**
Seq-S^5-Unroll	32.4	142	39.9	103	4.90	7.05	7.78
Seq-S^5-Equal	32.8	169	45.6	120	5.11	7.68	8.38
CRadix	54.1	65.1	61.6	113	6.82	10.2	8.63
LCPMergesort	25.8	316	53.9	167	5.00	14.6	17.0
burstsortA	29.5	131	43.3	120	5.64	8.48	8.46
fbC_burstsort	60.9	69.8			11.0	**3.81**	15.4
sCPL_burstsort	45.0	122			11.1	14.3	24.8
	AMD48						
std::sort	243	1 071	197	494	21.7	55.7	44.2
mkqs	90.7	511	78.2	226	11.0	20.8	19.8
mkqs_cache8	**37.9**	**96.9**	**31.7**	**114**	**3.44**	7.08	8.75
radix8_CI	83.1	127	71.2	138	9.72	10.9	9.46
radixR_CE7	73.6	125	63.6	120	8.34	8.27	**7.70**
Seq-S^5-Unroll	59.4	283	55.6	167	7.93	11.2	11.7
Seq-S^5-Equal	56.5	292	57.7	180	8.06	11.5	12.2
CRadix	98.2	115	68.8	147	8.11	12.6	11.2
LCPMergesort	48.2	597	68.8	232	7.35	20.7	24.4
burstsortA	46.0	214	53.0	193	8.49	13.3	13.1
fbC_burstsort	85.8	115			17.7	**5.92**	22.1
sCPL_burstsort	73.1	266			20.1	24.6	37.9
	Memory usage of sequential algorithms (on IntelE5) in MiB, excluding input and string pointer array						
std::sort	0.002	0.002	0.002	0.003	0.002	0.002	0.002
mkqs	0.134	0.003	1.66	0.141	0.015	0.003	0.004
mkqs_cache8	1 002	6 242	1 225	4 096	153	483	483
radix8_CI	62.7	390	77.2	256	9.55	30.2	30.2
radixR_CE7	669	3 902	786	2 567	111	303	303
Seq-S^5-Unroll	129	781	155	513	20.3	60.8	60.9
Seq-S^5-Equal	131	781	156	513	20.8	60.8	61.0
CRadix	752	4 681	919	3 072	114	362	362
LCPMergesort	1 002	6 242	1 225	4 096	153	483	483
burstsortA	1 466	7 384	1 437	5 809	200	531	792
fbC_burstsort	31 962	6 200			2 875	436	4 182
sCPL_burstsort	9 971	7 262			1 578	1 697	5 830

Table 6 Run time of sequential algorithms on AMD16, Inteli7, and IntelX5 in seconds. See Table 4 for a short description of each.

	Our Datasets				Sinha's		
	URLs	Random	GOV2	Wikipedia	URLs	DNA	NoDup
n	66 M	409 M	80.2 M	256 Mi	10 M	31.5 M	31.6 M
N	4 Gi	4 Gi	4 Gi	32 Pi	304 Mi	302 Mi	382 Mi
D/N (D)	92.6 %	43.0 %	69.7 %	(13.6 G)	97.5 %	100 %	73.4 %
L/n	57.9	3.3	34.1	33.0	29.4	9.0	7.7
	AMD16						
std::sort	274	1 088	237		28.1	73.5	56.8
mkqs	138	586	99.7	284	15.9	29.6	26.9
mkqs_cache8	**45.4**	**114**	**40.2**	142	**4.77**	8.99	10.7
radix8_CI	112	158	84.5	171	11.7	13.6	11.5
radixR_CE7	91.4	156	75.3	**135**	10.5	10.8	**9.46**
Seq-S^5-Unroll	70.6	326	68.4	235	9.57	14.5	14.8
Seq-S^5-Equal	72.9	315	67.9	227	9.41	12.8	13.5
CRadix	132	128	91.8	201	11.7	19.0	14.7
LCPMergesort	56.8	631	86.6	285	9.22	25.5	30.9
burstsortA	52.4	284	64.4	252	10.1	17.2	17.0
fbC_burstsort	109	123			25.5	**6.34**	29.8
sCPL_burstsort	79.8	288			28.8	38.9	54.9
	Inteli7						
std::sort	94.4	360	75.6	233	9.41	23.3	21.2
mkqs	33.2	187	30.9	112	5.00	9.43	9.62
mkqs_cache8	**14.7**		**14.5**	66.1	**1.81**	3.91	5.10
radix8_CI	38.3	49.6	32.4	73.0	4.95	5.35	5.05
radixR_CE7	30.7	**46.6**	28.8	**59.7**	3.85	3.71	**3.84**
Seq-S^5-Unroll	25.3	108	26.2	86.5	3.94	5.75	6.44
Seq-S^5-Equal	25.9	130	27.6	97.4	4.11	6.14	6.82
CRadix	43.2		33.4	84.6	5.27	7.87	6.49
LCPMergesort	22.7		32.3	139	4.33	12.1	14.2
burstsortA	22.5		24.9	102	4.52	6.67	6.91
fbC_burstsort					9.31	**3.12**	12.7
sCPL_burstsort	35.3				9.82	13.0	20.0
	IntelX5						
std::sort	140	731	137	401	17.4	48.9	38.6
mkqs	74.2	333	56.6	148	7.18	13.9	14.0
mkqs_cache8	30.1	**80.1**	**25.5**	95.5	**3.35**	6.48	7.51
radix8_CI	69.0	109	49.7	88.9	5.84	7.47	7.06
radixR_CE7	55.9	110	44.6	**83.5**	4.92	6.12	**5.95**
Seq-S^5-Unroll	35.9	170	34.4	108	4.17	5.62	6.74
Seq-S^5-Equal	38.7	198	36.5	121	4.65	6.09	7.19
CRadix	77.7	94.3	57.7	141	8.26	13.3	11.0
LCPMergesort	36.2	454	55.1	208	6.61	19.3	22.8
burstsortA	**27.0**	215	34.9	153	4.70	9.17	10.1
fbC_burstsort		86.8			16.4	**4.31**	21.1
sCPL_burstsort	46.5	212			19.6	26.8	38.5

Table 7 Absolute run time of parallel and best sequential algorithms on IntelE5 in seconds, median of 1–3 runs. See Table 3 for a short description of each.

PEs	1	2	4	8	12	16	24	32	48	64
	URLs (complete), $n = 1.11\,\mathrm{G}$, $N = 70.7\,\mathrm{Gi}$, $\frac{D}{N} = 93.5\,\%$									
mkqs_cache8	**467**									
pS^5-Unroll	633	310	156	92.8	69.2	52.7	45.1	42.8	41.1	39.9
pS^5-Equal	646	316	157	93.0	69.3	52.8	45.8	42.0	41.2	41.2
pS^5+LCP-M			**116**	**74.4**	57.7	**47.0**	**38.4**	**34.4**	**34.1**	**35.1**
pMKQS	617	**292**	146	84.2	**57.5**	47.1	41.0	38.0	37.0	38.0
pRS-8bit	1 959	975	493	280	204	171	144	142	140	140
pRS-16bit	1 960	883	444	260	179	148	125	119	115	116
pRS/Akiba	1 293	1 258	1 255	1 249	1 256	1 255	1 249	1 259	1 255	1 249
	Random, $n = 3.27\,\mathrm{G}$, $N = 32\,\mathrm{Gi}$, $\frac{D}{N} = 44.9\,\%$									
mkqs_cache8	**609**									
pS^5-Unroll	1 209	601	301	166	122	101	76.7	67.7	65.3	63.7
pS^5-Equal	1 322	657	326	178	131	107	81.4	70.7	67.6	64.2
pS^5+LCP-M			367	196	135	108	80.0	71.0	72.7	71.7
pMKQS	732	**379**	**198**	**108**	**79.7**	**69.1**	**63.6**	65.5	70.7	75.2
pRS-8bit	1 530	706	343	183	127	100	70.9	59.3	60.8	59.2
pRS-16bit	1 530	657	343	185	129	100	69.4	**56.2**	**52.4**	**53.6**
pRS/Akiba	1 355	751	447	321	280	257	232	223	219	216
	GOV2, $n = 3.1\,\mathrm{G}$, $N = 128\,\mathrm{Gi}$, $\frac{D}{N} = 82.7\,\%$									
mkqs_cache8	**1 079**									
pS^5-Unroll	1 399	673	326	212	176	145	113	97.1	92.1	89.0
pS^5-Equal	1 476	705	339	224	186	154	119	101	95.8	91.2
pS^5+LCP-M			**272**	**166**	**131**	**112**	**93.3**	**80.5**	**80.7**	**82.1**
pMKQS	1 347	**661**	350	207	164	127	101	91.8	93.7	95.1
pRS-8bit	4 244	1 992	964	585	462	394	311	299	302	291
pRS-16bit	4 252	1 912	928	571	471	384	306	279	280	257
pRS/Akiba	2 645	1 306	1 028	1 055	1 048	1 034	1 037	1 045	1 051	1 052
	Wikipedia, $n = N = 4\,\mathrm{Gi}$, $D = 249\,\mathrm{G}$									
mkqs_cache8	**2 502**									
pS^5-Unroll	2 728	1 341	648	350	252	203	147	120	110	103
pS^5-Equal	2 986	1 435	694	374	268	215	157	125	115	105
pS^5+LCP-M			635	338	**235**	**186**	**130**	**107**	**97.6**	**92.0**
pMKQS	2 554	**1 259**	**620**	**336**	238	189	143	127	122	119
pRS-8bit	4 064	1 879	909	486	349	271	192	157	150	144
pRS-16bit	4 068	1 805	875	469	340	262	187	149	145	139
pRS/Akiba	2 862	1 450	754	453	355	302	249	229	265	263
	Sinha NoDup (complete), $n = 31.6\,\mathrm{M}$, $N = 382\,\mathrm{Mi}$, $\frac{D}{N} = 73.4\,\%$									
radixR_CE7	**6.00**									
pS^5-Unroll	8.27	4.17	2.13	1.22	0.921	0.790	0.695	0.642	0.592	0.544
pS^5-Equal	8.84	4.46	2.26	1.28	0.963	0.825	0.710	0.661	0.601	0.567
pS^5+LCP-M			2.52	1.41	1.01	0.815	0.653	0.604	0.691	0.779
pMKQS	8.44	4.30	2.21	1.25	0.920	0.801	0.744	0.798	0.973	1.11
pRS-8bit	8.35	4.06	2.01	1.08	0.770	0.643	0.489	0.425	**0.422**	**0.441**
pRS-16bit	8.35	**3.49**	**1.74**	**0.949**	**0.682**	**0.595**	**0.445**	**0.422**	0.467	0.502
pRS/Akiba	7.58	4.09	2.38	1.59	1.33	1.21	1.09	1.04	1.04	1.02

Table 8 Absolute run time of parallel and best sequential algorithms on AMD48 in seconds, median of 1–3 runs. See Table 3 for a short description of each.

PEs	1	2	3	6	9	12	18	24	36	42	48
	URLs (complete), $n = 1.11\,\mathrm{G}$, $N = 70.7\,\mathrm{Gi}$, $\frac{D}{N} = 93.5\,\%$										
mkqs_cache8	**773**										
pS^5-Unroll	1 030	521	352	181	127	99.9	74.9	64.0	53.0	48.0	46.8
pS^5-Equal	931	477	331	176	123	97.1	73.3	65.0	55.0	48.8	47.6
pS^5+LCP-M					122	114	76.3	60.3	50.0	49.6	46.8
pMKQS	844	**415**	**280**	**146**	**102**	**80.4**	**59.6**	**49.9**	**44.2**	**44.0**	**45.1**
pRS-8bit	2 552	1 306	897	468	325	266	202	177	157	156	155
pRS-16bit	2 550	1 210	823	428	299	234	183	151	126	125	126
pRS/Akiba	1 861	1 840	1 832	1 830	1 821	1 819	1 823	1 822	1 822	1 827	1 821
	Random, $n = 2.45\,\mathrm{G}$, $N = 24\,\mathrm{Gi}$, $\frac{D}{N} = 44.5\,\%$										
mkqs_cache8	**879**										
pS^5-Unroll	1 315	683	466	248	176	140	104	86.3	69.3	64.4	61.7
pS^5-Equal	1 225	634	433	232	165	131	98.3	82.1	67.1	62.1	59.7
pS^5+LCP-M					196	185	109	82.7	71.5	66.0	61.9
pMKQS	751	**392**	**264**	**143**	**106**	**81.0**	**65.3**	**56.0**	54.7	55.8	61.3
pRS-8bit	1 182	594	404	209	144	111	79.6	63.5	52.2	49.9	47.1
pRS-16bit	1 188	615	423	224	154	120	85.1	68.0	**51.3**	**47.4**	**44.8**
pRS/Akiba	1 525	861	643	421	348	312	277	259	241	238	234
	GOV2, $n = 1.38\,\mathrm{G}$, $N = 64\,\mathrm{Gi}$, $\frac{D}{N} = 77.0\,\%$										
mkqs_cache8	**750**										
pS^5-Unroll	881	449	305	162	129	110	83.2	69.9	58.4	54.4	50.2
pS^5-Equal	854	436	296	156	123	104	79.2	67.1	55.8	52.5	48.7
pS^5+LCP-M					**100**	98.2	62.0	**48.7**	**42.9**	**39.5**	**37.4**
pMKQS	785	**397**	**268**	**140**	101	**83.8**	**60.1**	50.7	44.1	42.8	42.6
pRS-8bit	2 071	1 015	676	351	280	251	182	154	125	120	114
pRS-16bit	2 061	980	652	338	269	242	171	143	111	107	102
pRS/Akiba	1 547	794	617	594	584	582	581	581	584	585	585
	Wikipedia, $n = N = 2\,\mathrm{Gi}$, $D = 116\,\mathrm{G}$										
mkqs_cache8	**1 442**										
pS^5-Unroll	1 634	838	569	299	208	164	119	96.9	74.6	68.7	65.3
pS^5-Equal	1 592	827	561	293	205	161	117	95.4	73.6	68.1	64.3
pS^5+LCP-M					267	255	142	104	83.4	72.8	65.9
pMKQS	1 556	**790**	**534**	**273**	**188**	**145**	**102**	**84.3**	**67.0**	**64.1**	**64.0**
pRS-8bit	2 547	1 216	825	417	281	215	148	115	85.2	78.1	72.6
pRS-16bit	2 547	1 168	795	405	276	211	147	114	82.6	74.2	69.2
pRS/Akiba	1 966	1 030	717	403	299	249	198	197	196	196	198
	Sinha NoDup (complete), $n = 31.6\,\mathrm{M}$, $N = 382\,\mathrm{Mi}$, $\frac{D}{N} = 73.4\,\%$										
radixR_CE7	**8.24**										
pS^5-Unroll	12.8	6.73	4.63	2.52	1.82	1.47	1.13	0.978	0.836	0.804	0.794
pS^5-Equal	12.0	6.30	4.34	2.37	1.72	1.39	1.08	0.944	0.812	0.779	0.772
pS^5+LCP-M					1.89	1.76	1.39	1.10	1.01	0.993	0.984
pMKQS	11.2	5.81	4.02	2.14	1.53	1.28	1.00	0.935	0.989	1.04	1.16
pRS-8bit	10.8	5.33	3.61	1.87	1.35	1.01	**0.771**	**0.596**	**0.482**	**0.462**	**0.453**
pRS-16bit	10.8	**4.82**	**3.27**	**1.72**	**1.32**	**0.988**	0.872	0.779	0.924	1.01	1.10
pRS/Akiba	11.5	6.33	4.62	2.90	2.33	2.05	1.80	1.68	1.57	1.54	1.53

Table 9 Absolute run time of parallel and best sequential algorithms on AMD16 in seconds, median of 1–3 runs. See Table 3 for a short description of each.

PEs	1	2	4	6	8	12	16
	URLs, $n = 500\,\mathrm{M}$, $N = 32\,\mathrm{Gi}$, $\frac{D}{N} = 95.4\,\%$						
mkqs_cache8	**422**						
pS^5-Unroll	424	**218**	**115**	**83.0**	**70.5**	**56.2**	**49.1**
pS^5-Equal	455	233	123	91.4	75.1	61.6	51.6
pS^5+LCP-M			152	152	91.9	73.2	58.7
pMKQS	456	220	117	88.3	74.4	63.7	61.5
pRS-8bit	1 256	652	362	284	253	238	220
pRS-16bit	1 253	601	331	255	225	212	185
pRS/Akiba	1 063	1 060	1 049	1 057	1 110	1 048	1 105
	Random, $n = 1.23\,\mathrm{G}$, $N = 12\,\mathrm{Gi}$, $\frac{D}{N} = 43.7\,\%$						
mkqs_cache8	**350**						
pS^5-Unroll	675	349	182	128	100	74.6	61.6
pS^5-Equal	621	321	167	119	93.2	70.0	58.2
pS^5+LCP-M			207	194	112	80.3	64.4
pMKQS	384	**203**	**112**	**84.6**	**79.6**	69.4	62.4
pRS-8bit	605	302	161	109	89.8	**67.6**	**55.7**
pRS-16bit	605	297	167	110	95.5	71.6	59.1
pRS/Akiba	805	459	283	227	198	171	157
	GOV2, $n = 490\,\mathrm{M}$, $N = 24\,\mathrm{Gi}$, $\frac{D}{N} = 72.4\,\%$						
mkqs_cache8	**291**						
pS^5-Unroll	336	171	88.3	62.6	55.5	44.9	36.5
pS^5-Equal	326	166	86.0	61.0	53.5	43.2	35.3
pS^5+LCP-M			81.4	78.2	**46.5**	**34.4**	**28.0**
pMKQS	296	**152**	**79.4**	**60.6**	46.9	39.4	37.2
pRS-8bit	692	338	176	132	112	105	92.1
pRS-16bit	691	327	170	126	115	92.9	81.7
pRS/Akiba	552	285	226	225	225	226	227
	Wikipedia, $n = N = 1\,\mathrm{Gi}$, $D = 40\,\mathrm{G}$						
mkqs_cache8	**642**						
pS^5-Unroll	840	424	214	147	112	78.9	62.0
pS^5-Equal	819	414	209	144	110	77.1	60.6
pS^5+LCP-M			195	175	104	**73.6**	**57.7**
pMKQS	693	**351**	**183**	**130**	**104**	80.9	71.8
pRS-8bit	920	425	216	153	118	90.1	75.1
pRS-16bit	917	408	207	149	114	87.3	71.4
pRS/Akiba	782	419	238	181	154	128	116
	Sinha NoDup (complete), $n = 31.6\,\mathrm{M}$, $N = 382\,\mathrm{Mi}$, $\frac{D}{N} = 73.4\,\%$						
radixR_CE7	**9.50**						
pS^5-Unroll	11.9	6.13	3.15	2.14	1.65	1.19	0.963
pS^5-Equal	11.0	5.65	2.90	1.99	**1.52**	**1.11**	**0.912**
pS^5+LCP-M			4.22	3.65	2.30	1.64	1.32
pMKQS	11.9	6.11	3.23	2.34	1.92	1.56	1.46
pRS-8bit	12.1	5.97	3.09	2.20	1.74	1.37	1.25
pRS-16bit	12.1	**5.44**	**2.82**	**1.98**	1.56	1.20	1.12
pRS/Akiba	13.0	7.19	4.28	3.33	2.86	2.44	2.24

Table 10 Absolute run time of parallel and best sequential algorithms on Inteli7 in seconds, median of fifteen runs, larger test instances. See Table 3 for a short description of each.

PEs	1	2	3	4	5	6	7	8
	URLs, $n = 65.7\,\text{M}$, $N = 4\,\text{Gi}$, $\frac{D}{N} = 92.7\,\%$							
mkqs_cache8	14.8							
pS^5-Unroll	14.9	8.46	5.87	**4.83**	4.81	4.52	4.41	4.24
pS^5-Equal	**14.5**	**8.27**	**5.80**	4.86	**4.61**	**4.39**	**4.29**	**4.17**
pMKQS	15.6	8.78	6.57	5.59	5.42	5.31	5.21	5.15
pRS-8bit	39.8	22.0	16.8	14.5	14.3	13.9	13.6	13.5
pRS-16bit	39.8	20.0	14.9	12.7	12.4	12.1	11.9	11.7
pRS/Akiba	32.0	31.7	31.7	31.7	31.7	31.7	31.7	31.7
p2w-MS/R	30.1	17.3	17.3	12.7	12.4	10.5	9.96	9.83
pMKQS-SIMD/R	31.4	19.4	15.4	13.6	13.4	13.3	13.3	13.3
pLCP-2w-MS/S	29.7	18.9	14.7	13.2	13.0	13.2	12.9	13.0
	Random, $n = 205\,\text{M}$, $N = 2\,\text{Gi}$, $\frac{D}{N} = 42.1\,\%$							
radixR_CE7	**19.6**							
pS^5-Unroll	32.2	17.5	12.1	9.39	9.51	9.07	8.29	7.67
pS^5-Equal	34.2	18.6	12.8	9.93	9.76	9.27	8.46	7.83
pMKQS	31.2	16.9	11.9	9.42	8.64	8.07	7.75	7.50
pRS-8bit	23.6	12.9	9.24	**7.89**	**7.55**	**7.22**	**7.00**	**6.86**
pRS-16bit	23.6	**12.5**	**9.04**	8.35	7.76	7.65	7.16	7.34
pRS/Akiba	24.0	16.1	13.3	12.0	11.7	11.4	11.2	11.0
p2w-MS/R	95.9	58.1	58.1	52.4	46.9	43.5	40.2	40.2
pMKQS-SIMD/R	123	72.9	55.3	47.2	46.8	46.6	46.5	46.2
pLCP-2w-MS/S	185	119	101	97.9	105	110	115	121
	GOV2, $n = 80\,\text{M}$, $N = 4\,\text{Gi}$, $\frac{D}{N} = 69.8\,\%$							
mkqs_cache8	14.6							
pS^5-Unroll	**13.1**	**7.15**	**5.00**	**3.95**	3.99	3.93	3.73	3.51
pS^5-Equal	13.5	7.39	5.16	4.06	**3.94**	**3.87**	**3.69**	**3.49**
pMKQS	15.5	8.62	6.22	5.08	4.82	4.58	4.43	4.30
pRS-8bit	33.5	17.8	12.9	10.2	9.85	9.52	9.43	9.69
pRS-16bit	33.4	17.1	12.2	9.62	9.20	8.73	8.40	8.20
pRS/Akiba	27.4	15.2	13.4	13.3	13.3	14.2	14.4	14.4
p2w-MS/R	32.6	19.3	19.3	16.8	15.8	12.8	12.6	12.4
pMKQS-SIMD/R	32.5	19.5	14.9	12.8	12.6	12.5	12.4	12.4
pLCP-2w-MS/S	43.9	22.9	15.8	12.5	12.8	11.5	10.5	9.94
	Wikipedia, $n = N = 256\,\text{Mi}$, $D = 13.8\,\text{G}$							
radixR_CE7	**59.6**							
pS^5-Unroll	59.9	**32.8**	**22.6**	**17.5**	**17.1**	**16.1**	**14.7**	**13.6**
pS^5-Equal	63.3	34.6	23.8	18.4	17.5	16.3	15.0	13.9
pMKQS	69.3	37.5	26.1	20.4	18.6	17.3	16.2	15.5
pRS-8bit	74.5	37.9	26.2	19.9	18.6	17.2	16.1	15.1
pRS-16bit	74.5	36.2	25.2	19.0	17.9	16.5	15.4	14.5
pRS/Akiba	62.4	34.8	24.9	20.0	18.6	17.5	16.6	15.7
p2w-MS/R	123	72.2	72.7	64.8	58.6	51.6	48.0	47.7
pMKQS-SIMD/R	127	74.4	55.5	46.8	45.7	44.7	44.2	43.5
pLCP-2w-MS/S	221	115	79.2	62.1	60.6	54.4	51.5	48.4

Table 11 Absolute run time of parallel and best sequential algorithms on Inteli7 in seconds, median of fifteen runs, smaller test instances. See Table 3 for a short description of each.

PEs	1	2	3	4	5	6	7	8
	Sinha URLs (complete), $n = 10\,\mathrm{M}$, $N = 304\,\mathrm{Mi}$, $\frac{D}{N} = 97.5\,\%$							
mkqs_cache8	1.81							
pS^5-Unroll	**1.54**	**0.853**	**0.595**	**0.471**	**0.495**	**0.464**	**0.445**	**0.431**
pS^5-Equal	1.67	0.924	0.641	0.505	0.505	0.475	0.452	0.436
pMKQS	1.93	1.07	0.766	0.618	0.593	0.571	0.552	0.544
pRS-8bit	5.05	2.48	1.74	1.37	1.32	1.28	1.23	1.20
pRS-16bit	5.06	2.23	1.56	1.21	1.19	1.14	1.11	1.08
pRS/Akiba	4.02	3.44	3.23	3.13	3.12	3.10	3.10	3.08
p2w-MS/R	3.98	2.35	2.34	2.05	1.86	1.64	1.51	1.53
pMKQS-SIMD/R	3.93	2.37	1.83	1.59	1.60	1.59	1.61	1.61
pLCP-2w-MS/S	5.93	3.57	2.89	2.65	2.83	2.81	2.86	2.93
	Sinha DNA (complete), $n = 31.6\,\mathrm{M}$, $N = 302\,\mathrm{Mi}$, $\frac{D}{N} = 100\,\%$							
radixR_CE6	3.69							
pS^5-Unroll	**2.94**	**1.63**	**1.14**	**0.906**	**0.989**	**0.900**	**0.883**	**0.831**
pS^5-Equal	3.37	1.85	1.29	1.02	1.04	0.944	0.920	0.864
pMKQS	4.28	2.37	1.72	1.40	1.32	1.25	1.22	1.21
pRS-8bit	5.47	2.99	2.14	1.68	1.63	1.58	1.54	1.51
pRS-16bit	5.47	2.78	1.94	1.57	1.54	1.50	1.47	1.42
pRS/Akiba	3.83	2.29	1.74	1.50	1.51	1.54	1.50	1.43
p2w-MS/R	11.1	6.36	6.36	5.13	5.10	4.17	3.98	3.97
pMKQS-SIMD/R	10.5	6.50	5.10	4.51	4.51	4.53	4.55	4.56
pLCP-2w-MS/S	18.5	11.1	8.79	7.91	8.39	8.18	8.29	8.40
	Sinha NoDup (complete), $n = 31.6\,\mathrm{M}$, $N = 382\,\mathrm{Mi}$, $\frac{D}{N} = 73.4\,\%$							
radixR_CE7	**3.83**							
pS^5-Unroll	4.54	**2.47**	**1.69**	**1.31**	1.33	1.21	1.15	1.07
pS^5-Equal	4.96	2.68	1.83	1.42	1.38	1.29	1.18	1.10
pMKQS	5.47	2.98	2.11	1.67	1.52	1.42	1.34	1.29
pRS-8bit	5.22	2.71	1.87	1.47	1.40	1.31	1.25	1.20
pRS-16bit	5.22	2.48	1.70	1.32	**1.28**	**1.19**	**1.12**	**1.07**
pRS/Akiba	4.97	2.84	2.06	1.68	1.59	1.50	1.43	1.37
p2w-MS/R	12.7	7.46	7.45	6.22	5.96	4.89	4.88	4.89
pMKQS-SIMD/R	13.6	8.08	6.10	5.23	5.15	5.09	5.05	5.02
pLCP-2w-MS/S	19.8	12.2	10.0	9.39	9.43	10.1	10.4	10.7

Table 12 Absolute run time of parallel and best sequential algorithms on IntelX5 in seconds, median of fifteen runs, larger test instances. See Table 3 for a short description of each.

PEs	1	2	3	4	5	6	7	8
	URLs, $n = 132\,\mathrm{M}$, $N = 8\,\mathrm{Gi}$, $\frac{D}{N} = 92.6\,\%$							
mkqs_cache8	64.2							
pS^5-Unroll	**56.2**	**30.8**	**24.4**	**20.7**	**20.2**	**19.2**	**19.0**	**19.0**
pS^5-Equal	58.6	32.1	25.3	21.2	20.5	19.7	19.4	19.3
pMKQS	67.1	36.0	29.8	26.5	26.2	26.0	26.0	25.9
pRS-8bit	150	89.5	79.6	72.9	72.8	71.7	72.6	71.6
pRS-16bit	150	78.5	68.3	62.7	61.9	61.8	62.4	59.7
pRS/Akiba	121	119	120	119	120	119	120	119
p2w-MS/R	85.9	49.3	52.5	42.1	40.1	35.0	35.1	34.9
pMKQS-SIMD/R	153	92.0	83.9	77.9	77.5	77.2	77.4	77.5
pLCP-2w-MS/S	108	63.3	52.5	49.9	57.8	55.8	54.4	57.7
	Random, $n = 307\,\mathrm{M}$, $N = 3\,\mathrm{Gi}$, $\frac{D}{N} = 42.8\,\%$							
mkqs_cache8	**58.9**							
pS^5-Unroll	78.5	42.2	31.0	25.0	**22.9**	**20.9**	**19.7**	**18.9**
pS^5-Equal	82.1	44.0	32.1	25.9	23.6	21.1	20.0	19.2
pMKQS	65.8	**35.5**	**28.4**	**24.4**	23.7	23.5	23.4	23.5
pRS-8bit	77.5	44.1	35.4	30.4	28.4	26.9	26.0	25.5
pRS-16bit	77.5	43.3	35.0	27.9	27.0	25.5	24.3	23.8
pRS/Akiba	81.7	50.6	41.9	37.2	34.9	33.3	32.6	31.9
p2w-MS/R	303	186	200	172	168	156	154	155
pMKQS-SIMD/R	467	277	246	229	225	222	220	219
pLCP-2w-MS/S	625	388	390	377	386	412	472	507
	GOV2, $n = 166\,\mathrm{M}$, $N = 8\,\mathrm{Gi}$, $\frac{D}{N} = 70.6\,\%$							
mkqs_cache8	55.3							
pS^5-Unroll	**47.8**	**25.8**	**19.5**	**16.2**	**15.4**	**14.5**	**14.8**	**14.6**
pS^5-Equal	49.1	26.4	19.9	16.4	15.5	14.6	15.0	14.8
pMKQS	58.7	32.2	26.0	22.6	22.0	22.0	22.1	22.2
pRS-8bit	115	61.7	49.8	43.8	44.9	45.2	44.2	44.0
pRS-16bit	115	59.5	47.0	40.0	37.7	40.4	38.8	38.8
pRS/Akiba	94.2	51.6	49.3	48.3	49.2	50.6	52.6	53.0
p2w-MS/R	124	74.7	81.7	69.0	66.2	62.0	62.1	62.7
pMKQS-SIMD/R	165	98.4	87.8	81.1	79.6	79.2	78.6	79.3
pLCP-2w-MS/S	185	93.9	90.1	71.2	67.0	62.4	66.0	65.3
	Wikipedia, $n = N = 512\,\mathrm{Mi}$, $D = 21.5\,\mathrm{G}$							
radixR_CE7	**185**							
pS^5-Unroll	194	**104**	**76.2**	**62.0**	**56.8**	**52.5**	**50.8**	**49.4**
pS^5-Equal	202	107	78.7	63.8	58.2	53.6	51.7	50.3
pMKQS	211	112	87.5	74.2	71.3	69.4	68.9	68.9
pRS-8bit	212	111	85.4	71.0	67.0	63.8	62.6	61.7
pRS-16bit	212	109	84.2	68.5	64.8	62.1	59.8	58.2
pRS/Akiba	192	107	83.4	70.6	66.5	64.3	62.9	62.5
p2w-MS/R	420	261	281	237	231	230	230	230
pMKQS-SIMD/R	583	343	301	277	274	273	273	274
pLCP-2w-MS/S	882	443	345	277	292	282	252	264

Table 13 Absolute run time of parallel and best sequential algorithms on IntelX5 in seconds, median of fifteen runs, smaller test instances. See Table 3 for a short description of each.

PEs	1	2	3	4	5	6	7	8
	Sinha URLs (complete), $n = 10\,\mathrm{M}$, $N = 304\,\mathrm{Mi}$, $\frac{D}{N} = 97\,5\,\%$							
mkqs_cache8	3.35							
pS^5-Unroll	**2.91**	**1.60**	**1.22**	**1.03**	**0.985**	**0.939**	0.932	0.929
pS^5-Equal	3.03	1.65	1.26	1.05	0.986	0.940	**0.931**	**0.929**
pMKQS	3.57	1.95	1.57	1.35	1.33	1.31	1.32	1.34
pRS-8bit	5.85	3.22	2.63	2.30	2.22	2.18	2.16	2.14
pRS-16bit	5.86	2.86	2.32	1.99	1.93	1.90	1.87	1.86
pRS/Akiba	5.12	4.38	4.22	4.10	4.09	4.10	4.09	4.08
p2w-MS/R	6.40	4.02	4.31	3.67	3.55	3.45	3.55	3.49
pMKQS-SIMD/R	9.56	5.87	5.47	5.15	5.15	5.14	5.20	5.20
pLCP-2w-MS/S	10.2	6.10	5.44	5.12	5.89	5.84	5.98	6.27
	Sinha DNA (complete), $n = 31.6\,\mathrm{M}$, $N = 302\,\mathrm{Mi}$, $\frac{D}{N} = 100\,\%$							
radixR_CE7	6.11							
pS^5-Unroll	**5.14**	**2.87**	**2.21**	**1.85**	**1.73**	**1.59**	**1.55**	**1.50**
pS^5-Equal	5.63	3.11	2.36	1.96	1.82	1.66	1.60	1.54
pMKQS	7.14	3.97	3.39	2.98	2.95	2.96	2.96	2.98
pRS-8bit	7.45	4.36	3.85	3.37	3.36	3.33	3.33	3.36
pRS-16bit	7.45	4.05	3.46	3.10	3.09	3.07	3.08	3.07
pRS/Akiba	6.00	3.72	3.44	3.12	3.13	3.11	3.12	3.06
p2w-MS/R	18.1	10.6	11.3	8.96	8.91	7.92	7.98	7.92
pMKQS-SIMD/R	27.1	16.8	15.6	14.7	14.7	14.7	14.8	14.8
pLCP-2w-MS/S	33.1	20.2	17.8	16.5	18.6	18.4	18.6	19.4
	Sinha NoDup (complete), $n = 31.6\,\mathrm{M}$, $N = 382\,\mathrm{Mi}$, $\frac{D}{N} = 73.4\,\%$							
radixR_CE7	**5.96**							
pS^5-Unroll	7.06	3.83	2.80	**2.25**	**2.02**	**1.88**	**1.74**	**1.66**
pS^5-Equal	7.51	4.05	2.94	2.35	2.08	1.88	1.77	1.69
pMKQS	8.17	4.39	3.54	2.97	2.85	2.82	2.76	2.76
pRS-8bit	7.22	3.99	3.20	2.74	2.62	2.49	2.46	2.43
pRS-16bit	7.22	**3.57**	**2.78**	2.32	2.19	2.10	2.03	1.97
pRS/Akiba	7.10	4.10	3.27	2.81	2.66	2.55	2.51	2.48
p2w-MS/R	20.8	12.6	13.4	11.3	11.3	10.4	10.2	10.1
pMKQS-SIMD/R	30.0	17.9	15.9	14.8	14.7	14.6	14.6	14.7
pLCP-2w-MS/S	34.5	21.3	19.5	18.9	21.7	22.0	22.5	24.5