Parallel String Sample Sort

Timo Bingmann, Peter Sanders

Karlsruhe Institute of Technology, Karlsruhe, Germany
{bingmann,sanders}@kit.edu

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 propose string sample sort. The algorithm makes effective use of the memory hierarchy, uses additional word level parallelism, and largely avoids branch mispredictions. Additionally, we parallelize variants of multikey quicksort and radix sort that are also useful in certain situations.

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 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 large inputs. This means that besides parallelization we have to take the memory hierarchy and the high cost of branch mispredictions into account.

In Section 2 we give an overview of string sorting algorithms, acceleration techniques and parallel atomic sample sort. We then propose our new string sorting algorithm super scalar string sample sort (S^5) in Section 3.1. The rest of Section 3 describes two competitors and we compare them experimentally in Section 4. For all instances except random strings, S^5 achieves higher speedups on modern multi-core machines than our own parallel multikey quicksort and radixsort implementations, which are already better than any previous ones.

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 is a zero-based array of $|s|$ characters from the alphabet $\Sigma = \{1, \ldots, \sigma\}$. For

the implementation, we require that strings are zero-terminated, i.e., $s[|s|-1] = 0 \notin \Sigma$. 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. Let $\mathrm{lcp}(s, t)$ denote the length of the *longest common prefix* (LCP) of s and t. In a sequence or array of strings x let $\mathrm{lcp}_x(i)$ denote $\mathrm{lcp}(x_{i-1}, x_i)$. Our target machine is a shared memory system supporting p hardware threads (PEs – processing elements) on $\Theta(p)$ cores.

2.1 Basic Sequential String Sorting Algorithms

Multikey quicksort [1] is a simple but effective adaptation of quicksort to strings. When all strings in $\mathcal{S}$ have a common prefix of length ℓ, the algorithm uses character $c = s[\ell]$ 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 ℓ-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.

MSD radix sort [2,3,4] with common prefix length ℓ looks at the ℓ-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 [5] for details).

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 [6,7,8] 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 [9].

2.2 Architecture Specific Enhancements

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 [3], multikey quicksort [10], and in its extreme to burstsort [7].

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, 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 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 [4]).

2.3 (Parallel) Atomic Sample Sort

There is a huge amount of work on parallel sorting so that we can only discuss the most relevant results. Besides (multiway)-mergesort, perhaps the most practical parallel sorting algorithms are parallelizations of radix sort (e.g. [11]) and quicksort [12] as well as *sample sort* [13]. Sample sort 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 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. Parameter α 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) [14]. In this case, k is chosen in such a way that classification and data distribution can be done in a cache efficient way.

2.4 More Related Work

There is some work on PRAM algorithms for string sorting (e.g. [15]). 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. However, Takuya Akiba has implemented a parallel radix sort [16], Tommi Rantala's library [10] contains multiple parallel mergesorts and a parallel SIMD variant of multikey quicksort, and Nagaraja Shamsundar [17] also parallelized Waihong Ng's LCP-mergesort [9]. Of all these implementations, only the radix sort by Akiba scales fairly well to many-core architectures. For the abstract, we exclude the other implementations and discuss their scalability issues in Appendix C.

3 Shared Memory Parallel String Sorting

Already in a sequential setting, theoretical considerations and experiments (see Appendix B.2) indicate that *the* best string 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 2.1 can be parallelized rather easily and we will discuss parallel versions in Sections 3.2–3.4. 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, which exploits these properties. Preliminary result on string sample sort have been reported in the bachelor thesis of Knöpfle [18].

3.1 String Sample Sort

In order to adapt the atomic sample sort from Section 2.3 to strings, we have to devise an efficient classification algorithm. 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

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

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. We do this by using dynamic load balancing which includes parallel execution of recursive calls as in parallel quicksort.

In Appendix A.1 we outline a variant of string sample sort that uses a trie data structure and a number of further tricks to enable good asymptotic performance. However, we view this approach as somewhat risky for a first reasonable implementation. Hence, in the following, we present a more pragmatic implementation.

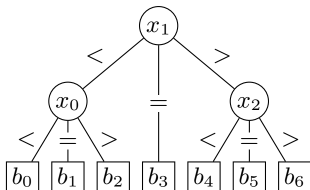

Fig. 1. Ternary search tree for $v = 3$ splitters.

Super Scalar String Sample Sort (S^5) – A Pragmatic Solution. We adapt the implicit binary search tree approach used in S^4 [14] to strings. Rather than using arbitrarily long splitters as in trie sample sort, 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 $x_0, \ldots, x_{v-1}$ 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 ℓ. The main disadvantage compared to trie sample sort 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 allows us to completely unroll the loop descending the search tree. We can then also unroll the loop over the elements, interleaving independent tree descents. Like in [14], this is an important optimization since it allows the instruction scheduler in a super scalar processor to parallelize the operations by drawing data dependencies apart. The strings in the "$< x_0$" and "$> x_{v-1}$" buckets b_0 and b_{2v} keep common prefix length ℓ. 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 yields the following asymptotic running time bound.

Lemma 1. *String sample sort with implicit binary trees and word parallelism can be implemented to run in time* $\mathcal{O}\big(\frac{D}{w} \log v + n \log n\big)$.

Implementation Details. 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. As

most commonly done in radix sort, we first classify strings, counting how many fall into each bucket, then calculate a prefix sum and redistribute the string pointers accordingly. To avoid traversing the tree twice, the bucket index of each string is stored in an oracle. Additionally, to make higher use of super scalar parallelism, we even separate the classification loop from the counting loop [4].

Like in S^4, the binary tree of splitters is stored in level-order as an array, allowing efficient traversal using $i := 2i + \{0, 1\}$, without branch mispredictions. We noticed that predicated instructions `CMOVA` were more efficient than flag arithmetic involving `SETA`.

To perform the equality check after traversal without extra indirections, the splitters are additionally stored in order. Another idea is to keep track of the last $\leq$-branch during traversal; this however was slower and requires an extra register. A third variant is to check for equality after each comparison, which requires only an additional `JE` instruction and no extra `CMP`. The branch misprediction cost is counter-balanced by skipping the rest of the tree. An interesting observation is that, when breaking the tree traversal at array index i, then the corresponding equality bucket b_j can be calculated from i using only bit operations (note that i is an index in level-order, while j is in-order). Thus in this third variant, 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, we use 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 is calculated without a loop using two instructions: `XOR` and `BSR` to count the number of leading zero bits in the result.

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, which is only 64 KiB for this v, but is the bucket counter array containing $2v + 1$ counters, each 8 bytes long. We did not look into methods to reduce this array's size, because the search tree must also be stored both in level-order and in in-order.

Parallelization of S^5. Parallel S^5 (pS^5) is composed of four sub-algorithms for differently sized subsets of strings. For string sets $\mathcal{S}$ 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 3.3) is called, which internally uses insertion sort when $|\mathcal{S}| < t_i$.

The fully parallel version of S^5 uses $p' = \lceil \frac{|\mathcal{S}|}{p} \rceil$ 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 the sequential version of S^5 we permute the string pointer array in-place by walking cycles of the permutation [2]. Compared to out-of-place redistribution into buckets, the in-place algorithm uses less input/output streams and requires no extra space. The more complex instruction set seems to have only little negative impact, as today, memory access is the main bottleneck. However, for fully parallel S^5, an in-place permutation cannot be done in this manner. We therefore resort to out-of-place redistribution, using an extra string pointer array of size n. The string pointers are not copied back immediately. Instead, the role of the extra array and original array are swapped for the recursion.

All work in parallel S^5 is dynamically load balanced via a central job queue. 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. 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 boolean flag is set to indicate that other threads should share their work. These then free the *bottom level* of their local recursion stack (containing the largest subproblems) and enqueue this level as separate, independent jobs. This method avoids costly atomic operations on the local stack, replacing it by a faster (not necessarily synchronized) boolean flag check. 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.

3.2 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 full 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.

3.3 Parallel Caching Multikey Quicksort

Our preliminary experiments with sequential string sorting algorithms (see Appendix B.2) showed a surprise winner: an enhanced variant of multikey quicksort by Tommi Rantala [10] 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. Key to the algorithm's good performance, is that the cached characters are reused for the recursive subproblems $\mathcal{S}_<$ and $\mathcal{S}_>$, which greatly reduces the number of string accesses to at most $\lceil \frac{D}{w} \rceil + n$ in total.

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 [12] to three-way partitioning. The input array is viewed as a sequence of blocks containing B string pointers together with their w cache characters. 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 bottom level when re-balancing is required. We empirically determined $B = 128\,\text{Ki}$ as a good block size.

3.4 Burstsort and LCP-Mergesort

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.

One would like to generalize LCP-mergesort to a parallel p-way LCP-aware merging algorithm. This looks promising in general but we leave this for future work since LCP-mergesort is not really the best sequential algorithm in our experiments.

4 Experimental Results

We implemented parallel S^5, multikey quicksort and radixsort in C++ and compare them with Akiba's radix sort [16]. We also integrated many sequential implementations into our test framework, and compiled all programs using gcc 4.6.3 with optimizations `-O3 -march=native`. In Appendix B.2 we discuss the performance of sequential string sorters. Our implementations and test framework are available from `http://tbingmann.de/2013/parallel-string-sorting`.

We used several platforms for experiments, and summarized their properties in Table 2 in the appendix. Results we report in the abstract stem from the largest machine, IntelE5, which has four 8-core Intel Xeon E5-4640 processors containing a total of 32 cores and supporting $p = 64$ hardware threads, and from a consumer-grade Intel i7 920 with four cores and $p = 8$ hardware threads. Turbo-mode was disabled on IntelE5. We selected the following datasets, all with 8-bit alphabets. More characteristics of these instances are shown in Table 1.

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 [6] are strings of length $[0, 20)$ over the ASCII alphabet $[33, 127)$, with both length and characters chosen uniform at random.

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

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** [6] 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 test framework sets up a separate run environment for each test run. The program's memory is locked into RAM, and to isolate heap fragmentation,

Table 1. Characteristics of the selected input instances.

Name	n	N	$\frac{D}{N}$ (D)	σ	avg. $\lvert s \rvert$
URLs	1.11 G	70.7 Gi	93.5 %	84	68.4
Random	∞	∞	–	94	10.5
GOV2	11.3 G	425 Gi	84.7 %	255	40.3
Wikipedia	83.3 G	$\frac{1}{2}n(n+1)$	(79.56 T)	213	$\frac{1}{2}(n+1)$
Sinha URLs	10 M	304 Mi	97.5 %	114	31.9
Sinha DNA	31.6 M	302 Mi	100 %	4	10.0
Sinha NoDup	31.6 M	382 Mi	73.4 %	62	12.7

it was very important to fork() a child process for each run. We use the largest prefix $[0, 2^d)$ of our inputs which can be processed with the available RAM. We determined $t_m = 64\,\text{Ki}$ and $t_i = 64$ as good thresholds to switch sub-algorithms.

Figure 2 shows a selection of the detailed parallel measurements from Appendix C. For large instances we show results on IntelE5 (median of 1–3 repetitions) and for small instances on Inteli7 (of ten repetitions). The plots show the speedup of our implementations and Akiba's radix sort over the best sequential algorithm from Appendix B.2. We included pS^5-Unroll, which interleaves three unrolled descents of the search tree, pS^5-Equal, which unrolls a single descent testing equality at each node, our parallel multikey quicksort (pMKQS), and radix sort with 8-bit and 16-bit fully parallel steps. On all platforms, our parallel implementations yield good speedups, limited by memory bandwidth, not processing power. On IntelE5 for all four test instances, pMKQS is fastest for a small number of threads. But for higher numbers, pS^5 becomes more efficient than pMKQS, because it utilizes memory bandwidth better. On all instances, except Random, pS^5 yields the highest speedup for both the number of physical cores and hardware threads. On Random, our 16-bit parallel radix sort achieves a slightly higher speedup. Akiba's radix sort 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. On Inteli7, pS^5 is consistently faster than pMKQS for Sinha's smaller datasets, achieving speedups of 3.8–4.5, which is higher than the three memory channels on this platform. On IntelE5, the highest speedup of 19.2 is gained with pS^5 for suffix sorting Wikipedia, again higher than the 4×4 memory channels. For all test instances, except URLs, the fully parallel sub-algorithm of pS^5 was run only 1–4 times, thus most of the speedup is gained in the sequential S^5 steps. The pS^5-Equal variant handles URL instances better, as many equal matches occur here. However, for all other inputs, interleaving tree descents fares better. Overall, pS^5-Unroll is currently the best parallel string sorting implementation on these platforms.

5 Conclusions and Future Work

We have demonstrated that string sorting can be parallelized successfully on modern multi-core shared memory machines. In particular, our new string sample sort algorithm combines favorable features of some of the best sequential

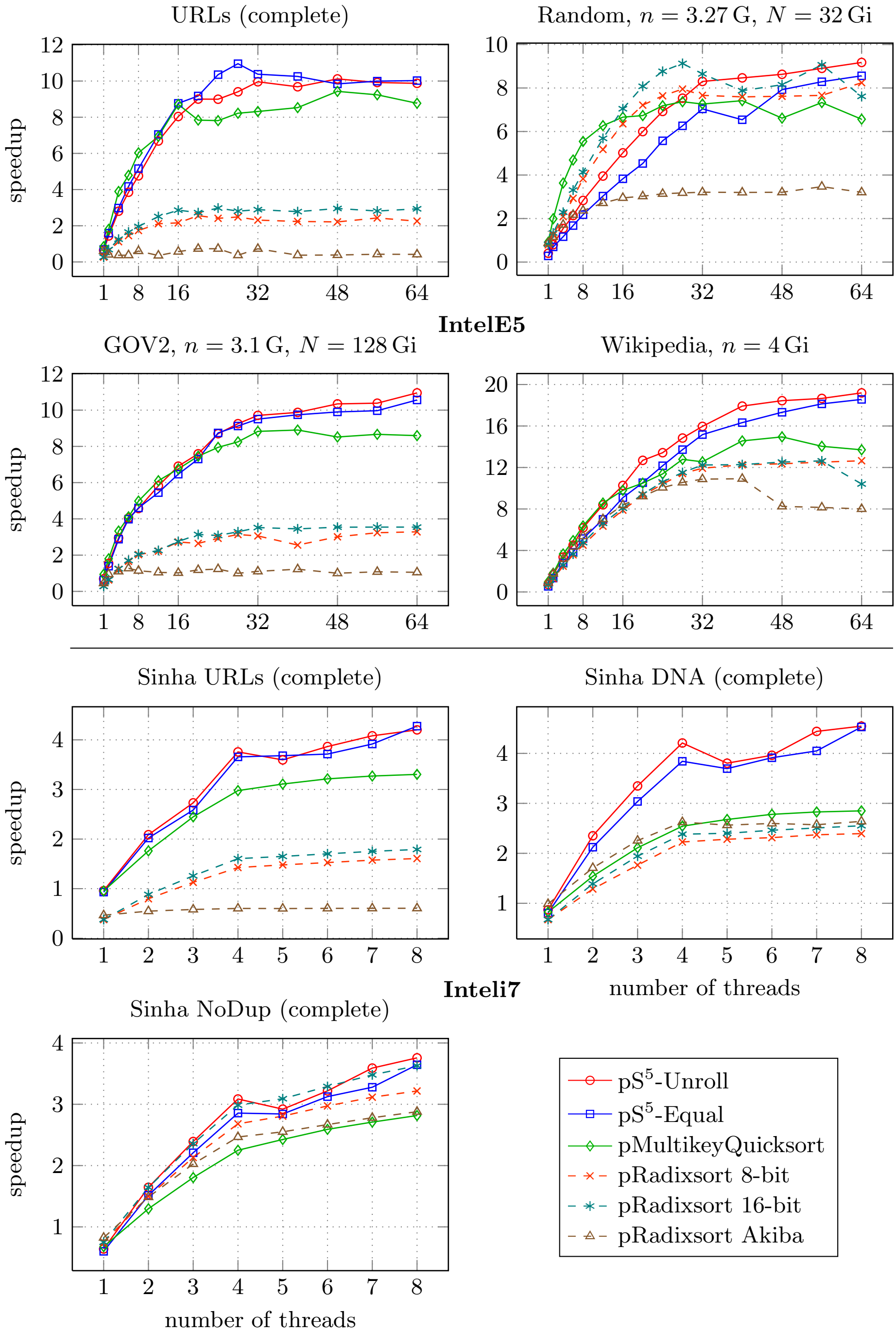

Fig. 2. Speedup of parallel algorithm implementations on IntelE5 (top four plots) and Inteli7 (bottom three plots)

algorithms – robust multiway divide-and-conquer from burstsort, efficient data distribution from radix sort, asymptotic guarantees similar to multikey quicksort, and word parallelism from cached multikey quicksort.

Implementing some of the refinements discussed in Appendix A.2 are likely to yield further improvements for string sample sort. To improve scalability on large machines, we may also have to look at NUMA (non uniform memory access) effects more explicitly. Developing a parallel multiway LCP-aware mergesort might then become interesting.

References

1. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In ACM, ed.: 8th Symposium on Discrete Algorithms. (1997) 360–369
2. McIlroy, P.M., Bostic, K., McIlroy, M.D.: Engineering radix sort. Computing Systems **6**(1) (1993) 5–27
3. Ng, W., Kakehi, K.: Cache efficient radix sort for string sorting. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **E90-A**(2) (2007) 457–466
4. Kärkkäinen, J., Rantala, T.: Engineering radix sort for strings. In: String Processing and Information Retrieval. Number 5280 in LNCS. Springer (2009) 3–14
5. Mehlhorn, K., Sanders, P.: Scanning multiple sequences via cache memory. Algorithmica **35**(1) (2003) 75–93
6. Sinha, R., Zobel, J.: Cache-conscious sorting of large sets of strings with dynamic tries. J. Exp. Algorithmics **9** (December 2004) 1.5:1–31
7. Sinha, R., Zobel, J., Ring, D.: Cache-efficient string sorting using copying. J. Exp. Algorithmics **11** (February 2007) 1.2:1–32
8. Sinha, R., Wirth, A.: Engineering Burstsort: Toward fast in-place string sorting. J. Exp. Algorithmics **15** (March 2010) 2.5:1–24
9. Ng, W., Kakehi, K.: Merging string sequences by longest common prefixes. IPSJ Digital Courier **4** (2008) 69–78
10. Rantala, T.: Library of string sorting algorithms in C++. `http://github.com/rantala/string-sorting` (2007) Git repository accessed November 2012.
11. Wassenberg, J., Sanders, P.: Engineering a multi-core radix sort. In: Euro-Par 2011 Parallel Processing. Number 6853 in LNCS. Springer (2011) 160–169
12. Tsigas, P., Zhang, Y.: A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In: PDP, IEEE Computer Society (2003) 372–381
13. 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. (1991) 3–16
14. Sanders, P., Winkel, S.: Super scalar sample sort. In: 12th European Symposium on Algorithms. Volume 3221 of LNCS., Springer (2004) 784–796
15. Hagerup, T.: Optimal parallel string algorithms: sorting, merging and computing the minimum. In: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing. STOC '94, New York, NY, USA, ACM (1994) 382–391
16. Akiba, T.: Parallel string radix sort in C++. `http://github.com/iwiwi/parallel-string-radix-sort` (2011) Git repository accessed November 2012.

17. Shamsundar, N.: A fast, stable implementation of mergesort for sorting text files. http://code.google.com/p/lcp-merge-string-sort (May 2009) Source downloaded November 2012.
18. Knöpfle, S.D.: String samplesort. Bachelor Thesis, Karlsruhe Institute of Technology, in German (November 2012)
19. Ferragina, P., Grossi, R.: The string B-tree: a new data structure for string search in external memory and its applications. Journal of the ACM **46**(2) (1999) 236–280
20. Mehlhorn, K., Näher, S.: Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. Information Processing Letters **35**(4) (1990) 183–189
21. 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, New Orleans (2004)

A More on String Sample Sort

A.1 Trie Sample Sort – A Theoretical Solution.

We would like to design a classification algorithm here that fulfills two conflicting goals. On the one hand, we would like to classify the input strings into the k buckets defined by the splitters exactly, regardless how long the substrings are that need to be inspected. The classifier from S^5 falls short of this goal since it looks only at a fixed number of input characters which may only allow a very rough classification if the splitters have long common prefixes. On the other hand, we do not want to repeatedly inspect long common substrings in recursive calls. In the following, we outline such an algorithm. Since the parallelization is analogous to S^5, we focus on the amount of work needed by the algorithm.

Our starting point is to use a Patricia trie as a classification data structure, adapting a technique for String B-Trees [19]: When classifying s using *blind search*, it suffices to check the first character for each trie edge traversed. When the search has reached a leaf, we can compare s with the splitter associated with the leaf. The first mismatch tells us in which bucket s has to be placed. When the mismatched character of s is smaller than the label character, then the output bucket is associated with the splitter in the leftmost leaf of the current sub-trie. If the character is larger, then the output bucket is associated with the splitter *following* the rightmost leaf of the current sub-trie. Pointers to these buckets can be precomputed and stored for each trie edge. The advantage of this approach over an ordinary trie is that only a single access to a splitter key is needed and all the other information needed fits into a cache of size $\mathcal{O}(k)$.

However, we have the problem that the tries might be very deep (up to $\Omega(k)$) in the worst case. This will destroy the desired time bounds if the classification frequently runs all the way down the trie and later finds a mismatch high up in the trie (in a character ignored by the blind search). Going beyond [19], we therefore augment the trie edges with a hash signature of the entire substring associated with this edge – not just its first character. We can then stop traversing a trie as soon as there is a mismatch in the hash signature. We will still sometimes miss the first opportunity to stop traversing the trie, but the expected cost for finding the first mismatch is proportional to the actual common prefix length.

We also have to explain how to handle the case when there is a mismatch for the first character in a trie-edge label. In this case, we have to locate the next character of s among the first characters of all edge labels. Using hashing and van Emde Boas trees, this can be done in expected time $\mathcal{O}(\log\log\sigma)$ [20] using space proportional to the degree of the current trie-node.

When a string s is located into bucket i, we find $\mathrm{lcp}(s, x_i)$ equal characters. In light of the analysis of multikey quicksort, this is unfortunate since only $\mathrm{lcp}_x(i)$ of those will not be considered again. We address this problem by changing the splitter keys. Rather than using the complete input strings, we only use their distinguishing prefixes. This does not get rid of all cases where characters of s will have to be reinspected later. However, this now only happens when bucket

$i+1$ has common prefix length at least $\mathrm{lcp}(s, x_i)$. In an amortized analysis, and assuming that buckets have about equal size, we can therefore charge the comparisons of characters $s[\mathrm{lcp}_x(i)..\mathrm{lcp}(s, x_i)]$ to characters in bucket $i+1$. We did not yet do a rigorous analysis taking the probabilistic nature of bucket sizes into account. However, we conjecture that an expected time bound of

$$\mathcal{O}(D + n \log_k n \cdot \log\log\sigma)$$

can be shown for trie sample sort. Choosing a small fixed k we can then obtain an algorithm that is reasonably cache efficient. Choosing $k = \sqrt{n}$ could reduce the number of recursion levels to $\mathcal{O}(\log\log n)$ leading to expected work $\mathcal{O}(D + n \log\log n \cdot \log\log\sigma)$.

A.2 Practical Refinements

Multipass data distribution: There are two constraints for the maximum sensible value for k: 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 ($k = \mathcal{O}(M)$ versus $k = \mathcal{O}(M/B)$). 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 k 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 the pragmatic solution, 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 $\log k$ key comparisons.

Jump tables: In the pragmatic solution, the a most significant bits of a key are often already sufficient to define a path in the search 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 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 [21] 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[1] 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.

B More Experimental Results

B.1 Experimental Setup

We tested our implementations and those by other authors on five different test platforms. All platforms run Linux and their main properties are listed in Table 2.

As described in Section 4, we isolate runs of different algorithms using following methods: before each run, the program fork()s into a child process. The string data is loaded before the fork(), allocating exactly the matching amount of RAM, and shared read-only with the child processes. The program's memory is locked into RAM using mlockall(). Before the algorithm is called, the string pointer array is generated inside the child process by scanning the string data for NUL 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. We took no special precautions of pinning threads to specific cores, and used the regular Linux task scheduling system as is.

The output of each string sorting algorithm was verified by first checking that the output pointer list is a permutation of the input set, and then checking that strings are in non-descending order.

Methodologically we have to discuss, whether measuring only the algorithms 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 was done to exclude both variables from the experimental results, however, for use in a real program context these costs cannot not 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 also did not look into HugePages (yet), which may or may not yield a performance boost.

[1] 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.

Table 2. 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	$4 \times 8 \times 2$	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	$1 \times 4 \times 2$	4×32	4×256	1×8	12
IntelX5	Intel Xeon X5355	2.67	$2 \times 4 \times 1$	8×32	4×4096		16

Name	Codename	Memory Channels	Interconnect	Linux/Kernel Version
IntelE5	Sandy Bridge	4 × DDR3-1600	2 × 8.0 GT/s QPI	Ubuntu 12.04/3.2.0-38
AMD48	Magny-Cours	4 × DDR3-667	4 × 3.2 GHz HT	Ubuntu 12.04/3.2.0-38
AMD16	Barcelona	2 × DDR2-533	3 × 1.0 GHz HT	Ubuntu 10.04/2.6.32-45
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 12.04/3.2.0-38

B.2 Performance of Sequential Algorithms

We collected many sequential string sorting algorithms in our test framework.

The algorithm library by Tommi Rantala [10] 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 twelve implementations listed in Table 3 to represent both the fastest ones in a preliminary test and each of the basic algorithms from Section 2.1. The twelve algorithms were run on all our five test platforms on small portions of the test instances described in Section 4. Tables 4 and 5 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 3. Description of selected sequential algorithms

Name	Description and Author
mkqs	Original multikey quicksort by Bentley and Sedgewick [1].
mkqs_cache8	Modified multikey quicksort with caching of eight characters by Tommi Rantala [10], slightly improved.
radix8_CI	8-bit in-place radix sort by Tommi Rantala [4].
radix16_CI	Adaptive 16-/8-bit in-place radix sort by Tommi Rantala [4].
radixR_CE7	Adaptive 16-/8-bit out-of-place radix sort by Tommi Rantala [4], version CE7 (preallocated swap array, unrolling, sorted-check).
CRadix	Cache efficient radix sort by Waihong Ng [3], unmodified.
LCPMergesort	LCP-mergesort by Waihong Ng [9], 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 [6], from [10].
fbC-burstsort	Copy-Burstsort with "free bursts" by Ranjan Sinha [7], 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 [7], 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.

We also measured the peak memory usage of the sequential implementations using a heap and stack profiling tool[2] for the selected sequential test instances. The bottom of Table 4 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

[2] http://tbingmann.de/2013/malloc_count/, by one of the authors.

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 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.

C More Results on Parallel String Sorting

In this section we report on experiments run on all platforms shown in Table 2, which contains a wide variety of multi-core machines of different age. The results in Figures 3–7 show that each parallel algorithm's speedup depends highly on hardware characteristics like processor speed, RAM and cache performance[3], and the interconnection between sockets. However, except for Random input, our implementations of pS^5 is the string sorting algorithm with highest speedup across all platforms.

We included two further parallel algorithm implementations in the tests on Inteli7 and IntelX5: 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 also uses only 2-way merges, and is omitted from the graphs because Rantala's version is consistently faster.

IntelE5 (Figure 3) is the newest machine, and most results have already been discussed in Section 4. The lower three plots show that on this platform, parallel sorting becomes less efficient for small inputs (around 300 MiB). Compared to the following results, we also notice that parallel multikey quicksort is relatively fast; apparently the Sandy Bridge processor architecture optimizes sequential memory processing very well.

AMD48 (Figure 4) is a many-core machine with high core count, but relatively slow RAM and a slower interconnect. Qualitatively, the results look similar to the IntelE5 platform, albeit with reduced speedups.

AMD16 (Figure 5) is an older many-core architecture with the slowest RAM speed and interconnect in our experiment. However, on this machine random access and processing power (in cache) seems to be most "balanced" for S^5.

Inteli7 (Figure 6) is a consumer-grade, single socket machine with fast RAM and cache hierarchy. All sorting algorithms profit from faster random access,

[3] See http://tbingmann.de/2013/pmbw/ for parallel memory bandwidth experiments

but the gain is highest for radix sorts. Both of the additional implementation pMKQS-SIMD and pMergesort-2way do not show any good speedup, probably because they are already pretty slow in sequential.

IntelX5 (Figure 7) is the oldest architecture, and shows the slowest absolute speedups. Nevertheless, the S^5 variants yield the best gains, even for Random input on this platform.

We included the absolute running times of all our speedup experiments in Tables 6–12 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.

Table 4. Run time of sequential algorithms on IntelE5 and AMD48 in seconds, and peak memory usage of algorithms on IntelE5

	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
$\frac{D}{N}$ (D)	92.6 %	43.0 %	69.7 %	(13.6 G)	97.5 %	100 %	73.4 %
	IntelE5						
mkqs	36.8	212	34.7	128	5.64	11.0	10.9
mkqs_cache8	**16.6**	67.8	**17.1**	79.2	**2.03**	4.64	6.05
radix8_CI	48.1	56.8	40.0	91.7	6.09	6.75	6.19
radix16_CI	36.9	71.4	40.0	88.1	5.42	5.26	5.84
radixR_CE7	37.4	**51.6**	34.7	**73.5**	4.80	4.64	**4.95**
Seq-S^5-Unroll	31.9	120	33.0	105	4.86	7.01	7.68
Seq-S^5-Equal	31.4	187	34.9	121	4.98	7.57	8.20
CRadix	55.5	57.7	41.4	111	6.77	10.1	8.55
LCPMergesort	25.4	266	37.6	165	4.94	14.3	16.8
burstsortA	29.6	127	30.7	125	5.64	8.49	8.53
fbC_burstsort	72.0	72.9			11.2	**3.88**	15.3
sCPL_burstsort	46.8	119			10.9	13.9	24.0
	AMD48						
mkqs	98.1	395	88.8	226	11.0	20.8	19.7
mkqs_cache8	**34.8**	95.9	**33.0**	**114**	**3.47**	7.05	8.76
radix8_CI	92.7	99.4	71.9	135	9.79	10.9	9.51
radix16_CI	73.8	135	61.9	134	8.77	9.17	9.45
radixR_CE7	85.2	98.6	66.5	120	8.27	8.31	**7.66**
Seq-S^5-Unroll	54.9	203	54.5	163	7.66	11.1	11.7
Seq-S^5-Equal	60.9	228	58.5	177	8.06	11.5	12.2
CRadix	99.3	**85.0**	76.0	147	8.02	12.4	11.1
LCPMergesort	47.1	452	73.3	232	7.33	20.7	24.5
burstsortA	47.1	190	56.2	205	8.52	13.3	13.3
fbC_burstsort	98.3	113			17.4	**5.85**	21.8
sCPL_burstsort	74.4	203			19.9	24.7	37.0
	Memory usage of sequential algorithms (on IntelE5) in MiB, excluding input and string pointer array						
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	77.6	256	9.55	30.2	30.2
radix16_CI	126	781	155	513	20.1	61.3	61.3
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			1 577	1 697	6 108

Table 5. Run time of sequential algorithms on AMD16, Inteli7, and IntelX5 in seconds

	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
$\frac{D}{N}$ (D)	92.6 %	43.0 %	69.7 %	(13.6 G)	97.5 %	100 %	73.4 %
	AMD16						
mkqs	121	561	96.3	272	14.5	26.7	25.0
mkqs_cache8	**43.1**	**110**	**37.5**	**135**	**4.74**	9.03	10.5
radix8_CI	101	156	76.7	148	11.2	13.0	11.1
radix16_CI	81.6	200	67.1	146	10.2	11.4	10.8
radixR_CE7	89.8	155	73.8	136	10.1	10.6	**9.32**
Seq-S^5-Unroll	62.3	288	57.1	195	8.50	12.3	12.5
Seq-S^5-Equal	67.5	313	61.0	211	9.00	12.4	12.9
CRadix	120	122	82.1	181	11.0	18.1	14.0
LCPMergesort	61.7	681	92.3	292	11.6	32.4	33.1
burstsortA	49.7	289	62.4	252	9.88	17.0	16.8
fbC_burstsort	124	171			25.1	**6.22**	29.7
sCPL_burstsort	83.7	307			29.0	41.0	55.2
	Inteli7						
mkqs	33.8	185	30.8	111	5.03	9.33	9.59
mkqs_cache8	**16.2**		**16.0**	73.6	**1.95**	4.32	5.65
radix8_CI	40.3	50.1	33.6	76.0	5.14	5.58	5.28
radix16_CI	32.0	69.2	29.3	74.6	4.59	4.72	5.16
radixR_CE7	32.8	**46.3**	30.2	**62.4**	4.04	3.83	**4.03**
Seq-S^5-Unroll	27.2	113	27.8	91.7	4.14	6.09	6.83
Seq-S^5-Equal	26.9	133	28.7	101	4.26	6.38	7.12
CRadix	46.6	145	35.7	91.4	5.59	8.19	6.87
LCPMergesort	23.4		33.3	142	4.37	12.4	14.6
burstsortA	23.1		25.2	106	4.62	6.79	7.17
fbC_burstsort					9.77	**3.23**	13.0
sCPL_burstsort					9.84	12.0	20.0
	IntelX5						
mkqs	78.0	323	55.1	151	7.53	13.9	14.0
mkqs_cache8	31.1	**84.2**	**25.9**	97.0	**3.50**	6.73	7.75
radix8_CI	70.7	103	49.2	93.6	5.95	7.38	7.00
radix16_CI	54.3	113	41.6	89.6	5.22	6.38	6.66
radixR_CE7	60.2	107	44.8	**86.0**	5.35	6.21	**6.19**
Seq-S^5-Unroll	38.4	162	34.6	111	4.38	7.87	7.96
Seq-S^5-Equal	38.5	191	35.9	125	4.64	8.30	8.75
CRadix	80.2	92.3	57.7	145	8.86	13.0	11.1
LCPMergesort	37.5	459	56.9	238	7.99	22.6	24.7
burstsortA	**29.0**	215	34.1	155	5.88	9.97	10.8
fbC_burstsort		87.2			17.3	**4.84**	21.3
sCPL_burstsort	50.7	205			20.3	26.5	37.8

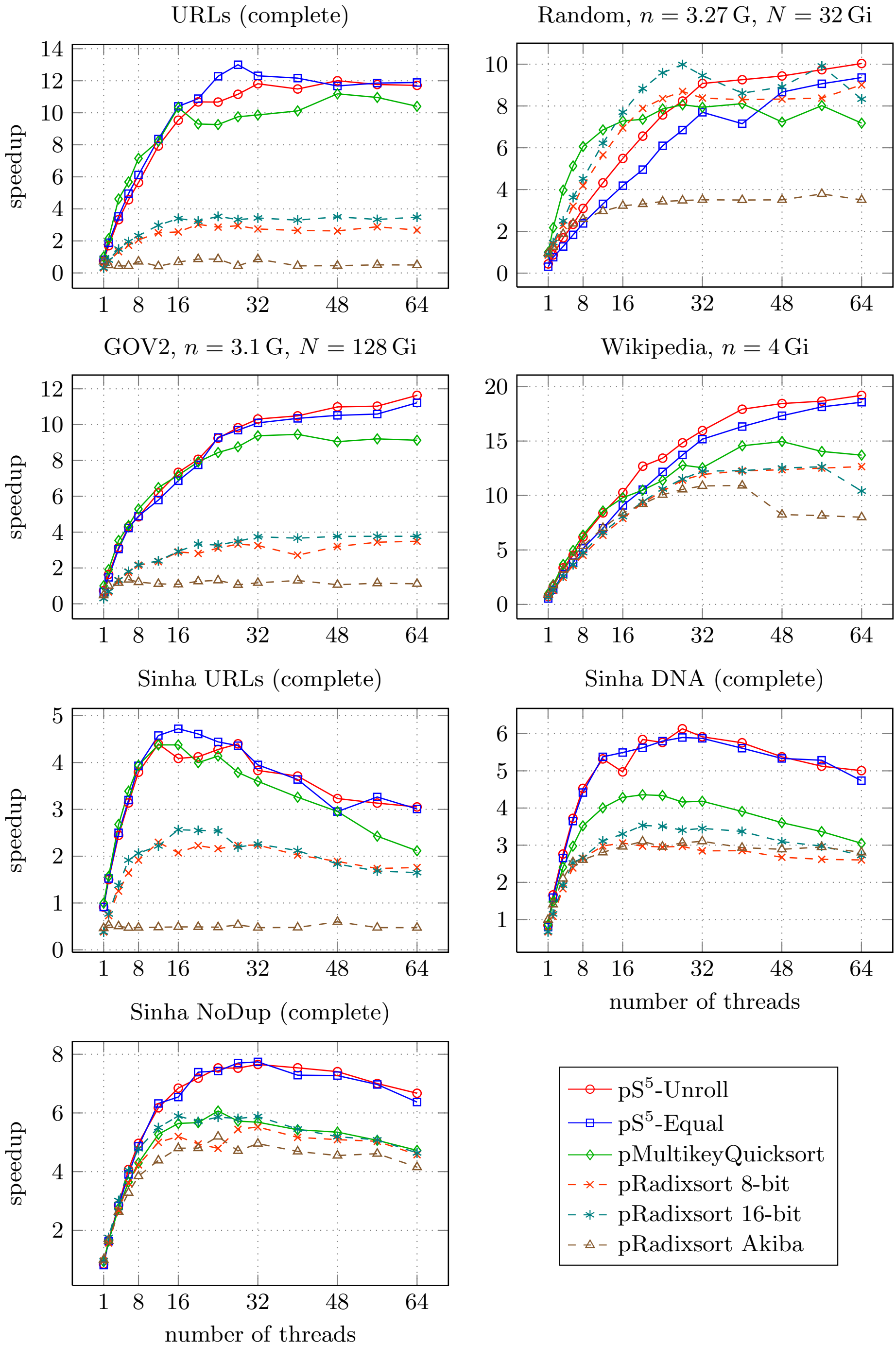

Fig. 3. Speedup of parallel algorithm implementations on IntelE5

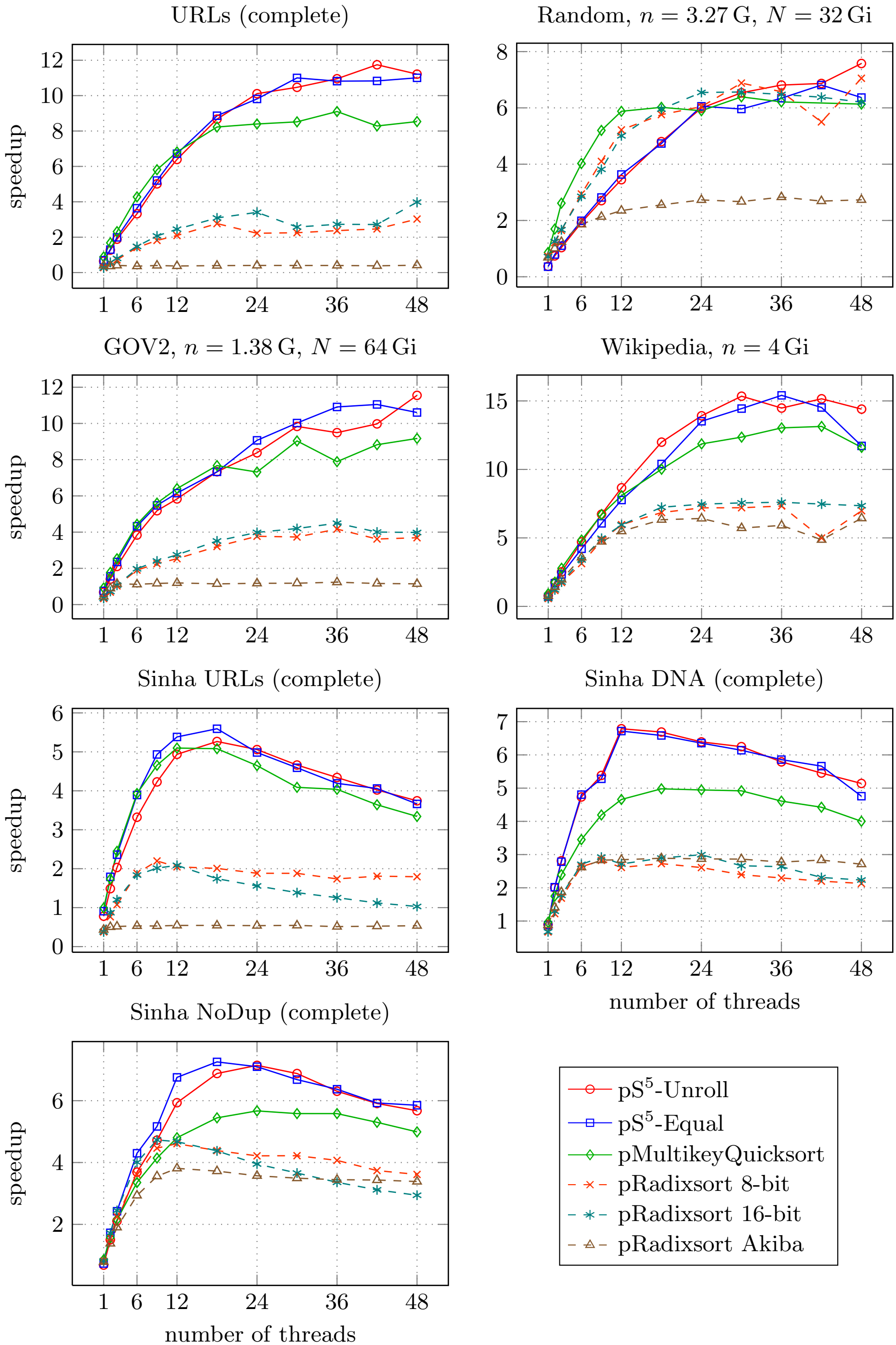

Fig. 4. Speedup of parallel algorithm implementations on AMD48

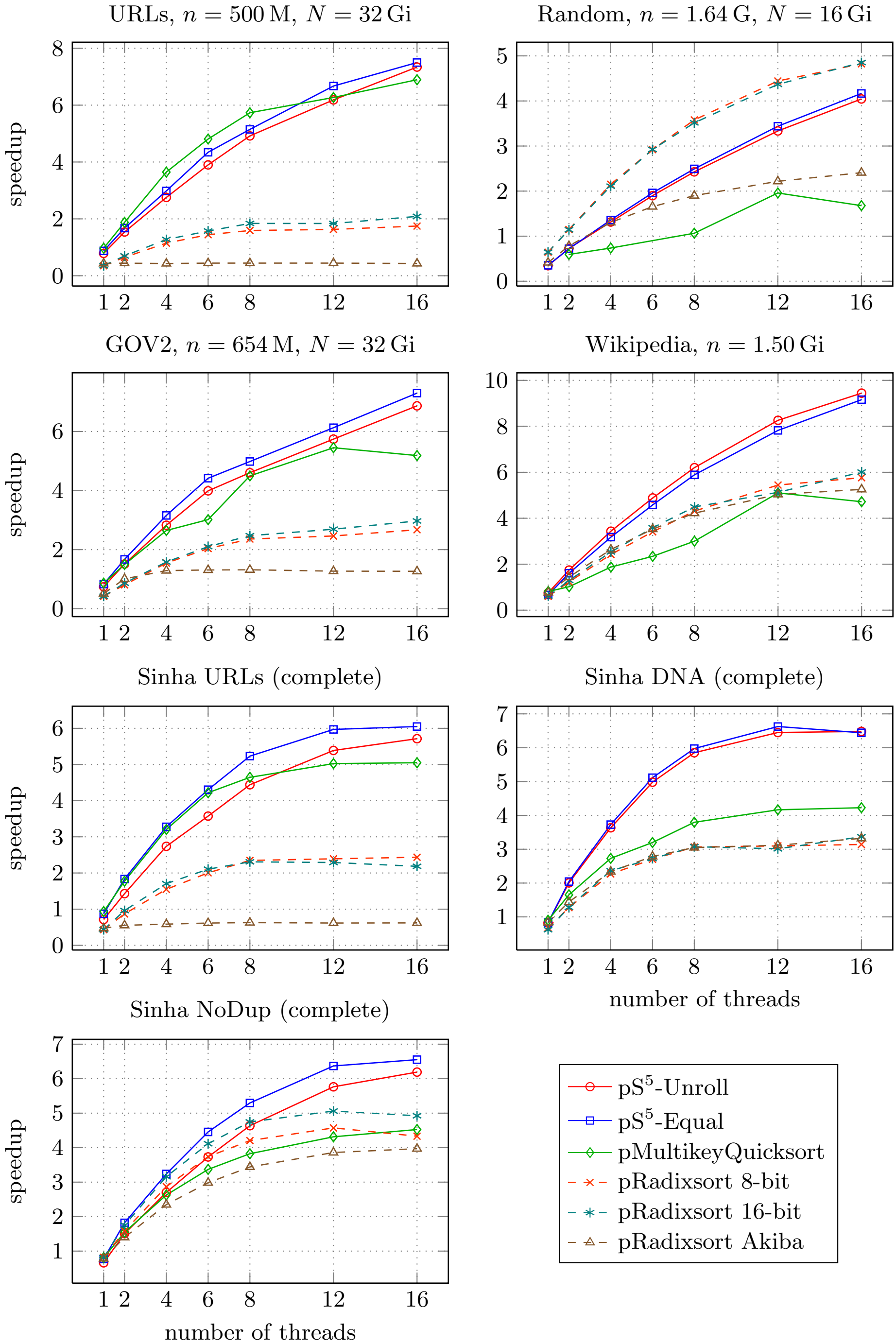

Fig. 5. Speedup of parallel algorithm implementations on AMD16

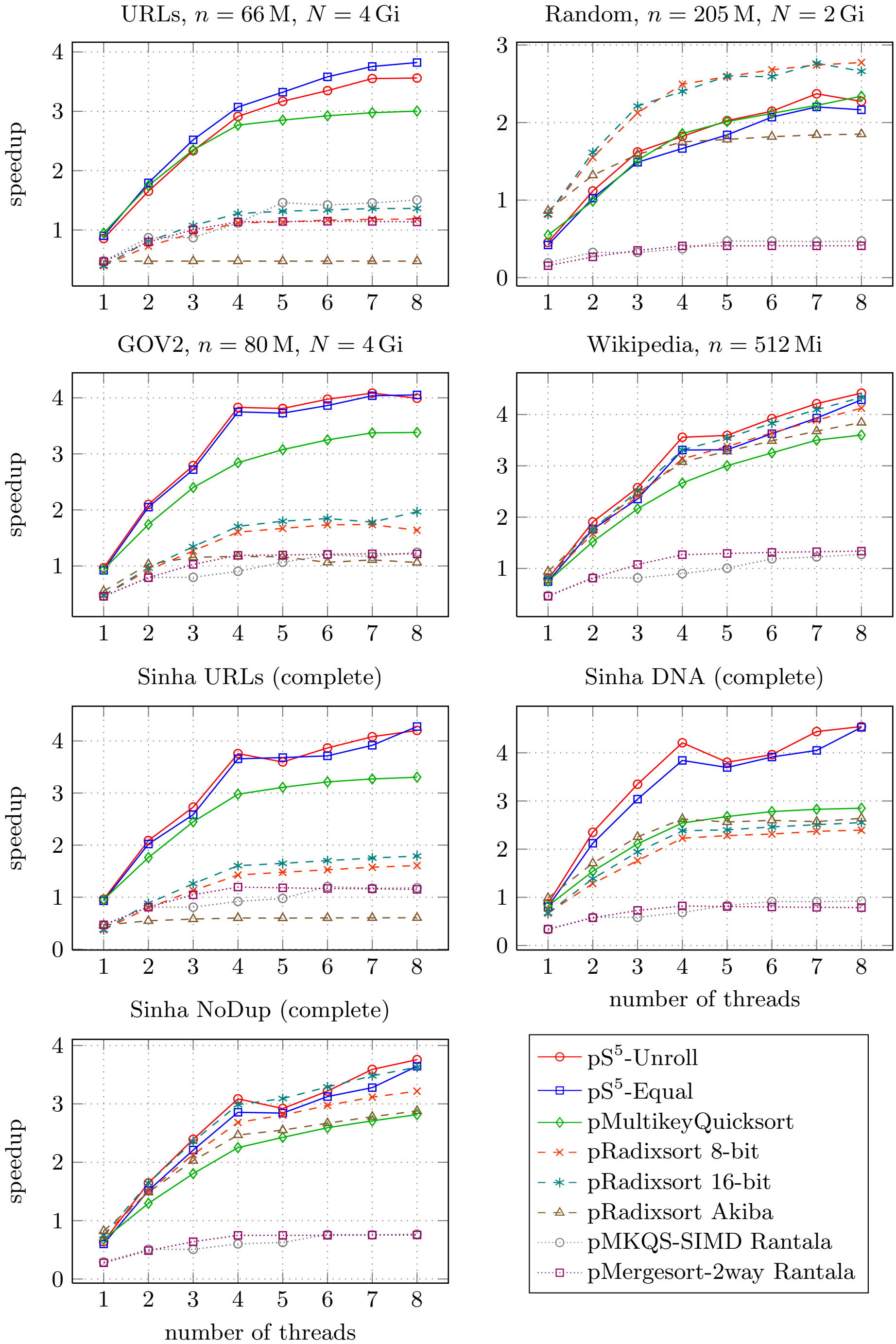

Fig. 6. Speedup of parallel algorithm implementations on Inteli7

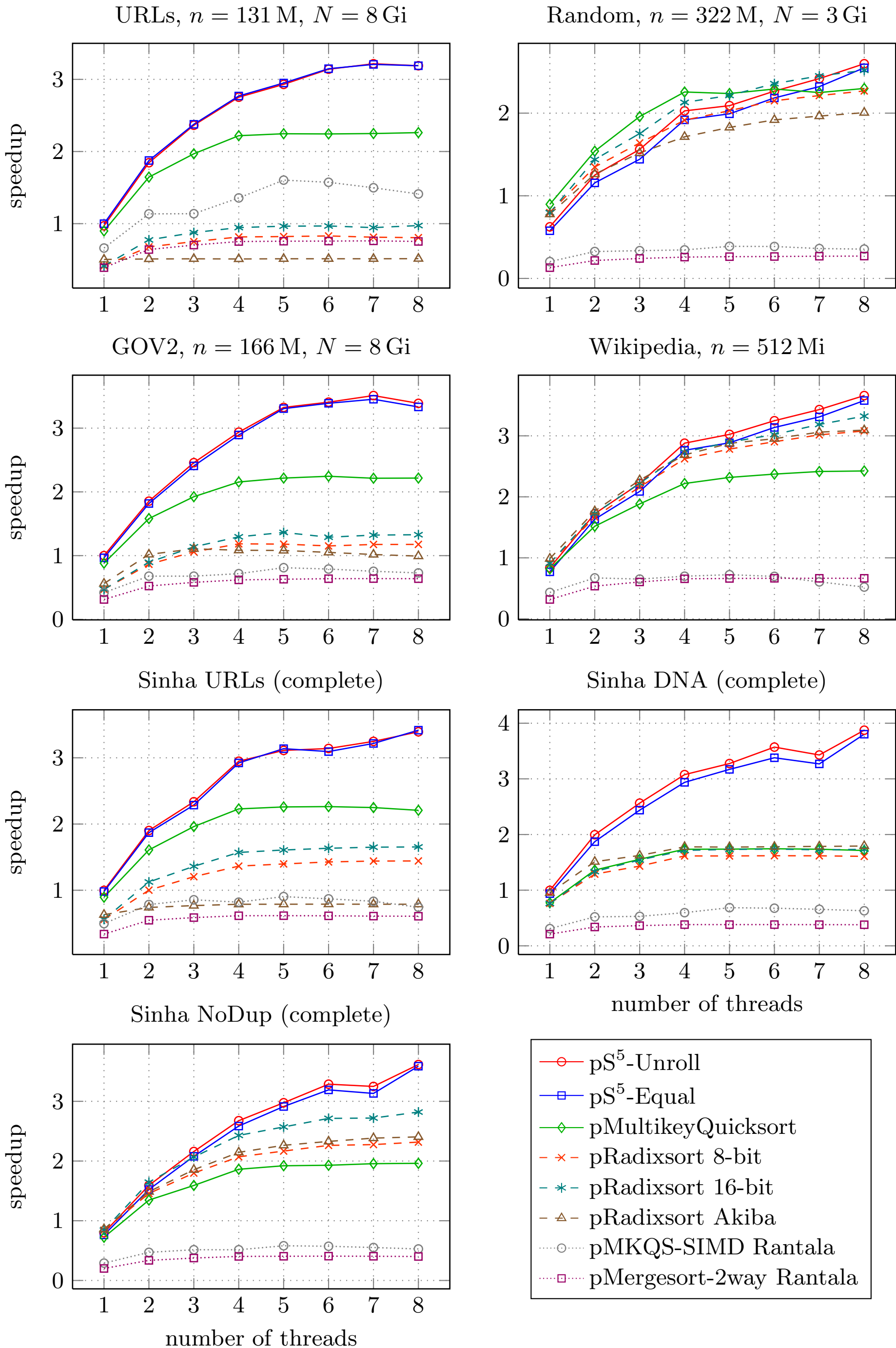

Fig. 7. Speedup of parallel algorithm implementations on IntelX5

Table 6. Absolute run time of parallel and best sequential algorithms on IntelE5 in seconds, median of 1–3 runs

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	537									
pS^5-Unroll	839	375	192	113	80.4	66.8	59.7	54.0	53.1	54.4
pS^5-Equal	774	338	180	104	76.3	61.3	51.9	51.8	54.6	53.6
pMKQS	637	298	138	89.1	77.4	61.7	68.8	64.6	57.0	61.3
pRS-8bit	2 017	881	480	309	254	249	223	232	243	238
pRS-16bit	2 031	787	428	274	214	188	180	186	182	183
pRS-Akiba	889	1 322	1 450	894	1 515	945	737	739	1 403	1 283
	Random, $n = 3.27\,\mathrm{G}$, $N = 32\,\mathrm{Gi}$, $\frac{D}{N} = 44.9\,\%$									
mkqs_cache8	755									
pS^5-Unroll	1 868	948	489	267	191	150	109	91.0	87.5	82.3
pS^5-Equal	2 669	1 082	648	347	249	197	136	107	95.3	88.2
pMKQS	826	379	208	136	120	113	105	104	114	115
pRS-8bit	957	611	362	197	146	119	98.9	98.6	99.1	91.6
pRS-16bit	926	565	330	183	133	107	86.1	87.3	92.7	99.2
pRS-Akiba	861	653	440	319	278	256	241	235	235	236
	GOV2, $n = 3.1\,\mathrm{G}$, $N = 128\,\mathrm{Gi}$, $\frac{D}{N} = 82.7\,\%$									
mkqs_cache8	1 164									
pS^5-Unroll	1 686	753	395	255	200	168	134	120	113	106
pS^5-Equal	1 854	837	404	253	214	180	133	122	118	110
pMKQS	1 236	647	350	234	191	172	146	132	137	135
pRS-8bit	3 493	1 853	959	574	525	427	398	380	387	354
pRS-16bit	4 193	1 807	928	569	515	423	377	331	329	328
pRS-Akiba	2 573	1 223	1 055	1 020	1 111	1 139	945	1 054	1 161	1 106
	Wikipedia, $n = 4\,\mathrm{G}$, $D = 249\,\mathrm{G}$									
mkqs_cache8	2 467									
pS^5-Unroll	3 178	1 399	688	373	275	225	172	145	125	120
pS^5-Equal	4 272	1 727	826	448	330	254	190	152	133	124
pMKQS	2 373	1 277	630	363	269	236	203	184	154	168
pRS-8bit	3 071	1 801	930	512	364	292	222	193	187	182
pRS-16bit	3 553	1 743	883	485	349	286	218	189	184	222
pRS-Akiba	2 671	1 305	727	429	329	275	229	212	280	288
	Sinha NoDup (complete), $n = 31.6\,\mathrm{M}$, $N = 382\,\mathrm{Mi}$, $\frac{D}{N} = 73.4\,\%$									
radixR_CE7	6.35									
pS^5-Unroll	7.40	3.76	2.18	1.26	1.01	0.913	0.830	0.817	0.844	0.937
pS^5-Equal	7.67	3.84	2.21	1.29	0.989	0.956	0.842	0.808	0.860	0.981
pMKQS	6.72	3.81	2.28	1.46	1.19	1.11	1.03	1.10	1.17	1.32
pRS-8bit	6.53	3.99	2.34	1.48	1.25	1.20	1.30	1.13	1.23	1.37
pRS-16bit	6.53	3.56	2.07	1.31	1.14	1.06	1.07	1.06	1.20	1.35
pRS-Akiba	6.26	3.82	2.38	1.63	1.43	1.30	1.20	1.26	1.37	1.51

Table 7. Absolute run time of parallel and best sequential algorithms on AMD48 in seconds, median of 1–3 runs

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	693										
pS^5-Unroll	1 077	546	368	209	138	108	80.0	68.6	63.3	59.0	61.8
pS^5-Equal	994	534	349	191	133	103	78.2	70.7	64.1	64.0	63.0
pMKQS	786	413	300	162	119	102	84.3	82.6	76.2	83.7	81.2
pRS-8bit	2 466	1 356	960	490	381	331	250	312	292	281	229
pRS-16bit	2 630	1 200	867	469	335	283	225	204	254	255	174
pRS-Akiba	1 742	1 928	1 699	1 912	1 738	1 863	1 763	1 717	1 728	1 817	1 663
	Random, $n = 3.27\,\mathrm{G}$, $N = 32\,\mathrm{Gi}$, $\frac{D}{N} = 44.9\,\%$										
mkqs_cache8	894										
pS^5-Unroll	2 402	1 204	863	465	331	259	186	149	131	130	118
pS^5-Equal	2 492	1 132	810	449	317	246	189	148	141	131	140
pMKQS	1 036	527	342	222	172	152	149	152	144		146
pRS-8bit	1 236	737	543	304	218	171	155	148	136	163	127
pRS-16bit	1 249	714	535	313	234	179	150	137	138	140	144
pRS-Akiba	1 307	886	722	481	419	380	350	327	316	333	327
	GOV2, $n = 1.82\,\mathrm{G}$, $N = 64\,\mathrm{Gi}$, $\frac{D}{N} = 77.0\,\%$										
mkqs_cache8	681										
pS^5-Unroll	996	486	323	177	131	117	92.8	81.2	71.7	68.2	58.9
pS^5-Equal	901	437	289	157	124	111	92.9	75.1	62.4	61.6	64.2
pMKQS	745	383	269	154	122	106	88.7	93.0	86.3	77.1	74.2
pRS-8bit	2 035	989	660	357	301	269	212	181	163	188	184
pRS-16bit	1 991	939	637	345	283	248	194	171	152	170	171
pRS-Akiba	1 515	760	592	609	583	569	595	580	551	584	594
	Wikipedia, $n = 4\,\mathrm{G}$, $D = 249\,\mathrm{G}$										
mkqs_cache8	2 895										
pS^5-Unroll	3 740	1 660	1 149	615	430	334	241	208	200	191	201
pS^5-Equal	4 158	1 714	1 242	688	478	373	278	214	188	199	247
pMKQS	3 122	1 638	1 040	597	432	358	289	244	222	220	249
pRS-8bit	5 036	2 486	1 661	926	603	485	422	402	395	580	414
pRS-16bit	5 032	2 396	1 629	852	582	487	400	388	381	388	394
pRS-Akiba	3 806	2 046	1 474	809	611	527	457	451	490	595	450
	Sinha NoDup (complete), $n = 31.6\,\mathrm{M}$, $N = 382\,\mathrm{Mi}$, $\frac{D}{N} = 73.4\,\%$										
radixR_CE6	8.06										
pS^5-Unroll	11.8	5.37	3.82	2.18	1.71	1.36	1.17	1.13	1.28	1.36	1.42
pS^5-Equal	11.0	4.67	3.32	1.87	1.56	1.19	1.11	1.14	1.26	1.36	1.38
pMKQS	9.42	5.12	3.75	2.40	1.94	1.68	1.48	1.42	1.44	1.52	1.61
pRS-8bit	9.95	5.15	3.59	2.23	1.80	1.75	1.84	1.91	1.98	2.15	2.23
pRS-16bit	9.88	4.68	3.29	2.01	1.70	1.72	1.84	2.04	2.40	2.59	2.74
pRS-Akiba	10.2	5.83	4.23	2.74	2.27	2.11	2.16	2.26	2.34	2.35	2.38

Table 8. Absolute run time of parallel and best sequential algorithms on AMD16 in seconds, median of 1–3 runs

PEs	1	2	4	6	8	12	16
	URLs, $n = 500\,\mathrm{M}$, $N = 32\,\mathrm{Gi}$, $\frac{D}{N} = 95.4\,\%$						
mkqs_cache8	427						
pS^5-Unroll	544	278	155	110	86.9	69.1	58.2
pS^5-Equal	489	255	143	98.4	83.0	64.0	57.0
pMKQS	441	228	117	88.8	74.5	68.1	62.0
pRS-8bit	1 210	659	372	296	269	262	245
pRS-16bit	1 210	612	335	271	232	233	205
pRS-Akiba	987	967	995	963	962	963	993
	Random, $n = 1.64\,\mathrm{G}$, $N = 16\,\mathrm{Gi}$, $\frac{D}{N} = 44.0\,\%$						
mkqs_cache8	486						
pS^5-Unroll	1 411	666	370	256	200	146	120
pS^5-Equal	1 376	675	358	248	195	141	117
pMKQS		818	660		457	248	290
pRS-8bit	754	424	226	166	136	109	101
pRS-16bit	754	426	231	166	138	111	100
pRS-Akiba	1 176	622	373	294	256	219	202
	GOV2, $n = 654\,\mathrm{M}$, $N = 32\,\mathrm{Gi}$, $\frac{D}{N} = 73.3\,\%$						
mkqs_cache8	390						
pS^5-Unroll	526	258	138	97.7	84.7	67.9	56.8
pS^5-Equal	469	234	124	88.3	78.3	63.7	53.4
pMKQS	451	260	147	129	86.7	71.6	75.2
pRS-8bit	927	482	253	191	165	158	146
pRS-16bit	924	451	247	186	158	145	131
pRS-Akiba	761	386	302	297	296	307	308
	Wikipedia, $n = 1.50\,\mathrm{Gi}$, $D = 90\,\mathrm{G}$						
mkqs_cache8	1 214						
pS^5-Unroll	1 605	696	353	249	196	147	129
pS^5-Equal	1 834	755	382	265	206	155	132
pMKQS	1 495	1 190	647	518	404	238	257
pRS-8bit	2 011	985	501	356	281	223	211
pRS-16bit	1 975	938	479	338	270	237	202
pRS-Akiba	1 580	842	460	344	288	241	231
	Sinha NoDup (complete), $n = 31.6\,\mathrm{M}$, $N = 382\,\mathrm{Mi}$, $\frac{D}{N} = 73.4\,\%$						
radixR_CE7	9.51						
pS^5-Unroll	14.4	6.31	3.52	2.55	2.05	1.65	1.54
pS^5-Equal	12.3	5.23	2.94	2.13	1.80	1.49	1.45
pMKQS	11.7	6.15	3.62	2.82	2.49	2.20	2.10
pRS-8bit	11.6	6.01	3.31	2.54	2.26	2.08	2.19
pRS-16bit	11.7	5.43	3.01	2.31	2.00	1.88	1.93
pRS-Akiba	12.1	6.80	4.05	3.19	2.76	2.46	2.40

Table 9. Absolute run time of parallel and best sequential algorithms on Inteli7 in seconds, median of ten runs, larger test instances

PEs	1	2	3	4	5	6	7	8
	URLs, $n = 65.7\,\mathrm{M}$, $N = 4\,\mathrm{Gi}$, $\frac{D}{N} = 92.7\,\%$							
mkqs_cache8	16.2							
pS^5-Unroll	18.9	9.81	6.95	5.57	5.12	4.84	4.57	4.56
pS^5-Equal	18.0	9.04	6.44	5.28	4.88	4.53	4.32	4.24
pMKQS	17.1	9.28	6.90	5.86	5.69	5.55	5.45	5.40
pRS-8bit	40.7	22.3	17.1	14.5	14.3	13.9	13.8	13.7
pRS-16bit	40.7	20.0	15.1	12.7	12.3	12.1	11.9	11.9
pRS-Akiba	34.3	34.1	34.1	34.0	34.1	34.1	34.1	34.1
pMergesort	33.9	18.6	18.6	14.5	11.1	11.4	11.1	10.8
pMKQS-SIMD	34.5	20.3	16.1	14.3	14.2	14.1	14.2	14.3
	Random, $n = 205\,\mathrm{M}$, $N = 2\,\mathrm{Gi}$, $\frac{D}{N} = 42.1\,\%$							
radixR_CE7	20.0							
pS^5-Unroll	44.1	17.9	12.3	11.0	9.88	9.32	8.44	8.80
pS^5-Equal	47.5	19.5	13.4	12.0	10.9	9.67	9.09	9.24
pMKQS	36.4	20.3	13.2	10.8	9.94	9.44	9.00	8.55
pRS-8bit	24.5	12.9	9.41	8.02	7.72	7.46	7.29	7.21
pRS-16bit	24.5	12.4	9.04	8.33	7.70	7.72	7.23	7.52
pRS-Akiba	23.2	15.1	12.6	11.4	11.2	11.0	10.9	10.8
pMergesort	104	61.6	61.5	53.7	42.5	42.5	43.0	42.4
pMKQS-SIMD	130	74.8	57.0	48.9	48.9	48.7	48.8	48.7
	GOV2, $n = 80\,\mathrm{M}$, $N = 4\,\mathrm{Gi}$, $\frac{D}{N} = 69.8\,\%$							
mkqs_cache8	16.0							
pS^5-Unroll	16.5	7.62	5.73	4.18	4.20	4.02	3.92	4.01
pS^5-Equal	17.3	7.80	5.88	4.27	4.29	4.14	3.96	3.95
pMKQS	17.0	9.19	6.67	5.63	5.20	4.92	4.74	4.73
pRS-8bit	33.7	17.3	12.6	9.98	9.58	9.22	9.20	9.78
pRS-16bit	33.7	16.6	11.9	9.37	8.89	8.66	8.98	8.13
pRS-Akiba	29.1	15.5	13.8	13.7	13.8	15.0	14.4	15.1
pMergesort	34.8	20.1	20.1	17.7	15.0	13.4	13.7	12.8
pMKQS-SIMD	34.9	20.2	15.6	13.4	13.4	13.2	13.1	13.2
	Wikipedia, $n = 2\,\mathrm{G}$, $D = 13.8\,\mathrm{G}$							
radixR_CE7	62.7							
pS^5-Unroll	78.6	32.9	24.4	17.6	17.5	16.0	14.9	14.2
pS^5-Equal	84.3	35.6	26.6	19.0	18.9	17.3	16.0	14.6
pMKQS	83.4	41.3	29.0	23.5	20.9	19.3	17.9	17.4
pRS-8bit	76.7	37.4	26.0	20.0	18.6	17.2	16.1	15.2
pRS-16bit	76.7	35.7	25.0	18.9	17.7	16.4	15.3	14.5
pRS-Akiba	66.6	35.7	25.5	20.4	19.1	18.0	17.1	16.3
pMergesort	133	76.6	77.0	69.7	62.4	52.8	51.2	49.2
pMKQS-SIMD	137	77.2	58.2	49.5	48.5	47.8	47.3	46.9

Table 10. Absolute run time of parallel and best sequential algorithms on Inteli7 in seconds, median of ten runs, smaller test instances

PEs	1	2	3	4	5	6	7	8
	Sinha URLs (complete), $n = 10\,\text{M}$, $N = 304\,\text{Mi}$, $\frac{D}{N} = 97\,5\,\%$							
mkqs_cache8	1.96							
pS^5-Unroll	2.04	0.936	0.716	0.520	0.544	0.506	0.479	0.465
pS^5-Equal	2.10	0.967	0.756	0.535	0.531	0.527	0.499	0.457
pMKQS	2.05	1.11	0.799	0.657	0.629	0.608	0.598	0.592
pRS-8bit	5.20	2.46	1.73	1.37	1.32	1.28	1.24	1.22
pRS-16bit	5.19	2.22	1.55	1.22	1.19	1.15	1.12	1.09
pRS-Akiba	4.21	3.58	3.37	3.25	3.27	3.25	3.24	3.23
pMergesort	4.22	2.41	2.42	2.13	2.00	1.62	1.67	1.66
pMKQS-SIMD	4.16	2.41	1.87	1.64	1.66	1.67	1.68	1.70
	Sinha DNA (complete), $n = 31.6\,\text{M}$, $N = 302\,\text{Mi}$, $\frac{D}{N} = 100\,\%$							
radixR_CE6	3.84							
pS^5-Unroll	4.41	1.63	1.15	0.912	1.01	0.968	0.864	0.844
pS^5-Equal	4.76	1.81	1.26	0.999	1.04	0.981	0.947	0.847
pMKQS	4.67	2.49	1.82	1.51	1.43	1.38	1.36	1.35
pRS-8bit	5.70	2.98	2.18	1.72	1.68	1.66	1.62	1.60
pRS-16bit	5.70	2.76	1.98	1.61	1.60	1.56	1.53	1.50
pRS-Akiba	3.89	2.25	1.70	1.46	1.50	1.48	1.49	1.45
pMergesort	11.7	6.54	6.57	5.60	4.62	4.20	4.22	4.16
pMKQS-SIMD	11.3	6.67	5.27	4.67	4.75	4.79	4.84	4.88
	Sinha NoDup (complete), $n = 31.6\,\text{M}$, $N = 382\,\text{Mi}$, $\frac{D}{N} = 73.4\,\%$							
radixR_CE6	4.06							
pS^5-Unroll	6.35	2.47	1.70	1.31	1.39	1.26	1.13	1.08
pS^5-Equal	6.75	2.68	1.84	1.42	1.43	1.30	1.24	1.11
pMKQS	5.99	3.13	2.25	1.80	1.67	1.57	1.50	1.44
pRS-8bit	5.41	2.71	1.90	1.51	1.45	1.37	1.30	1.26
pRS-16bit	5.41	2.47	1.72	1.36	1.31	1.23	1.17	1.12
pRS-Akiba	4.92	2.73	2.00	1.65	1.59	1.52	1.46	1.41
pMergesort	14.0	7.98	7.99	6.75	6.49	5.32	5.37	5.28
pMKQS-SIMD	14.6	8.34	6.35	5.43	5.43	5.42	5.39	5.38

Table 11. Absolute run time of parallel and best sequential algorithms on IntelX5 in seconds, median of ten runs, larger test instances

PEs	1	2	3	4	5	6	7	8
	URLs, $n = 65.7\,\mathrm{M}$, $N = 4\,\mathrm{Gi}$, $\frac{D}{N} = 92.7\,\%$							
mkqs_cache8	64.1							
pS^5-Unroll	62.3	32.8	25.6	22.0	20.6	19.2	18.8	19.0
pS^5-Equal	60.4	32.2	25.4	21.8	20.5	19.2	18.8	19.0
pMKQS	67.3	36.7	30.7	27.2	26.9	26.9	26.9	26.7
pRS-8bit	146	89.1	80.5	74.1	73.6	72.8	74.4	75.0
pRS-16bit	146	78.0	69.0	63.9	62.6	62.4	63.9	62.1
pRS-Akiba	120	118	118	118	118	118	118	117
pMergesort	91.3	53.2	53.1	44.6	37.7	38.4	40.4	42.8
pMKQS-SIMD	155	94.3	85.9	80.3	79.9	79.7	79.3	80.2
	Random, $n = 307\,\mathrm{M}$, $N = 3\,\mathrm{Gi}$, $\frac{D}{N} = 42.8\,\%$							
mkqs_cache8	61.2							
pS^5-Unroll	98.0	48.9	39.2	30.2	29.3	27.0	25.3	23.6
pS^5-Equal	106	52.9	42.4	31.9	30.7	28.0	26.4	24.0
pMKQS	68.2	39.7	31.2	27.1	27.3	26.7	27.2	26.6
pRS-8bit	76.8	45.4	37.3	32.0	30.1	28.5	27.7	27.0
pRS-16bit	76.9	42.5	34.9	28.7	27.6	26.0	25.0	24.3
pRS-Akiba	78.7	48.5	40.3	35.7	33.5	31.9	31.1	30.5
pMergesort	301	188	182	177	157	158	169	172
pMKQS-SIMD	469	282	253	237	232	231	228	226
	GOV2, $n = 166\,\mathrm{M}$, $N = 8\,\mathrm{Gi}$, $\frac{D}{N} = 70.6\,\%$							
mkqs_cache8	55.7							
pS^5-Unroll	52.4	28.2	21.3	17.8	15.8	15.4	14.9	15.5
pS^5-Equal	54.2	28.8	21.8	18.1	15.8	15.4	15.2	15.7
pMKQS	59.4	33.1	27.2	24.3	23.6	23.3	23.7	23.6
pRS-8bit	111	60.6	49.5	44.2	44.4	45.5	44.6	44.5
pRS-16bit	111	58.4	46.0	40.4	38.4	40.6	39.6	39.4
pRS-Akiba	93.2	51.4	47.3	48.3	48.5	49.8	51.5	52.8
pMergesort	125	77.2	77.2	73.0	64.6	66.3	69.4	71.8
pMKQS-SIMD	168	100	90.2	84.6	83.2	82.1	81.9	82.0
	Wikipedia, $n = 2\,\mathrm{G}$, $D = 13.8\,\mathrm{G}$							
	83.7							
pS^5-Unroll	99.9	48.5	37.7	29.1	27.7	25.8	24.4	22.9
pS^5-Equal	109	51.2	40.1	30.3	29.0	26.7	25.3	23.4
pMKQS	102	55.2	44.4	37.7	36.1	35.3	34.6	34.5
pRS-8bit	91.6	50.1	38.7	31.9	30.1	28.8	27.7	27.2
pRS-16bit	91.8	48.5	37.7	30.6	28.9	27.7	26.3	25.2
pRS-Akiba	84.7	47.4	36.8	31.1	29.1	28.3	27.3	27.0
pMergesort	193	125	128	120	116	120	138	161
pMKQS-SIMD	263	156	138	127	126	126	126	126

Table 12. Absolute run time of parallel and best sequential algorithms on IntelX5 in seconds, median of ten runs, smaller test instances

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	3.34							
pS^5-Unroll	3.22	1.69	1.38	1.09	1.03	1.02	0.989	0.947
pS^5-Equal	3.28	1.72	1.41	1.10	1.02	1.04	0.999	0.940
pMKQS	3.59	2.00	1.64	1.44	1.42	1.42	1.43	1.46
pRS-8bit	5.75	3.21	2.67	2.36	2.30	2.25	2.23	2.23
pRS-16bit	5.74	2.86	2.36	2.05	2.00	1.97	1.95	1.94
pRS-Akiba	5.09	4.34	4.18	4.08	4.07	4.07	4.07	4.07
pMergesort	6.48	4.10	3.76	3.92	3.55	3.69	3.86	4.32
pMKQS-SIMD	9.52	5.88	5.47	5.23	5.22	5.24	5.27	5.29
	Sinha DNA (complete), $n = 31.6\,\mathrm{M}$, $N = 302\,\mathrm{Mi}$, $\frac{D}{N} = 100\,\%$							
radixR_CE7	6.08							
pS^5-Unroll	5.71	2.85	2.22	1.85	1.74	1.59	1.66	1.47
pS^5-Equal	6.07	3.04	2.34	1.94	1.80	1.69	1.74	1.50
pMKQS	7.36	4.19	3.66	3.28	3.28	3.26	3.28	3.32
pRS-8bit	7.46	4.42	3.97	3.53	3.53	3.52	3.52	3.54
pRS-16bit	7.47	4.27	3.69	3.32	3.29	3.28	3.30	3.28
pRS-Akiba	5.98	3.77	3.51	3.20	3.21	3.20	3.19	3.18
pMergesort	18.4	10.9	10.7	9.56	8.30	8.42	8.68	9.04
pMKQS-SIMD	27.0	16.8	15.7	14.9	14.9	14.9	15.0	15.0
	Sinha NoDup (complete), $n = 31.6\,\mathrm{M}$, $N = 382\,\mathrm{Mi}$, $\frac{D}{N} = 73.4\,\%$							
radixR_CE7	6.06							
pS^5-Unroll	7.58	3.80	2.80	2.26	2.03	1.84	1.86	1.67
pS^5-Equal	7.94	3.98	2.91	2.34	2.08	1.90	1.93	1.69
pMKQS	8.37	4.50	3.81	3.25	3.15	3.14	3.10	3.09
pRS-8bit	7.19	4.16	3.37	2.92	2.79	2.68	2.66	2.61
pRS-16bit	7.19	3.68	2.93	2.49	2.36	2.23	2.23	2.15
pRS-Akiba	7.03	4.07	3.27	2.82	2.68	2.60	2.54	2.52
pMergesort	20.8	12.9	11.8	11.8	10.4	10.6	11.0	11.5
pMKQS-SIMD	30.1	18.0	16.1	15.0	14.9	14.9	14.9	15.1