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 comparisons of entire strings. 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 high cost of branch mispredictions and the memory hierarchy into account. For most multi-core systems, this hierarchy exhibits many processor-local caches but disproportionately few shared memory channels to RAM.

After introducing notation and previous approaches in Section 2, Section 3 explains our parallel string sorting algorithms, in particular super scalar string sample sort (S^5) but also multikey quicksort and radix sort. These algorithms are evaluated experimentally in Section 4.

We would like to thank our students Florian Drews, Michael Hamann, Christian Käser, and Sascha Denis Knöpfle who implemented prototypes of our ideas.

* This paper is a short version of the technical report [3].

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 $\text{lcp}(s, t)$ denote the length of the *longest common prefix* (LCP) of s and t. In a sequence or array of strings x let $\text{lcp}_x(i)$ denote $\text{lcp}(x_{i-1}, x_i)$. Our target machine is a shared memory system supporting p hardware threads (processing elements – PEs) on $\Theta(p)$ cores.

2.1 Basic Sequential String Sorting Algorithms

Multikey quicksort [2] 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 never need to 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 [8,10,7] 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 [9] 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 built 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 [16,17,15] 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 [11].

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 [10], multikey quicksort [12], and in its extreme to burstsort [17].

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. However, only specific, simple data in-dependencies can be detected and thus inner loops must be designed with care (e.g. for radix sort [7]).

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. [19]) and quicksort [18] as well as *sample sort* [4]. 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 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. 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) [13]. 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. [5]). 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 [1], Tommi Rantala's library [12] contains multiple parallel mergesorts and a parallel SIMD variant of multikey quicksort, and Nagaraja Shamsundar [14] also parallelized Waihong Ng's LCP-mergesort [11]. Of all these implementations, only the radix sort by Akiba scales fairly well to many-core architectures. For this paper, we exclude the other implementations and discuss their scalability issues in our technical report [3].

3 Shared Memory Parallel String Sorting

Already in a sequential setting, theoretical considerations and experiments [3] 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 [6].

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 into buckets b_i in the recursive calls. This can be done by observing that

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

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 our technical report [3] 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.

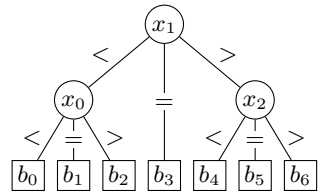

Fig. 1. Ternary search tree for $v = 3$ splitters.

Super Scalar String Sample Sort (S^5) – A Pragmatic Solution. We adapt the implicit binary search tree approach used in S^4 [13] to strings. Rather than using arbitrarily long splitters as in trie sample sort [3], 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 of this approach is that there may be 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 [13], 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 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 [7].

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. 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 $a'..b$.

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 thresholds t_i and t_m depend on hardware specifics, see Section 4 for empirically determined values.

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 [8]. Compared to out-of-place redistribution into buckets, the in-place algorithm uses fewer 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 [3] showed a surprise winner: an enhanced variant of multikey quicksort by Tommi Rantala [12] 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 [18] 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. 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 [1]. 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 our report [3] we discuss the performance of sequential string sorters. Our implementations and test framework are available from `http://tbingmann.de/2013/parallel-string-sorting`.

Experimental results we report in this paper stem from two platforms. The larger machine, IntelE5, has four 8-core Intel Xeon E5-4640 processors containing a total of 32 cores and supporting $p = 64$ hardware threads. The second platform is a consumer-grade Intel i7 920 with four cores and $p = 8$ hardware threads. Turbo-mode was disabled on IntelE5. Our technical report [3] contains further details of these machines and experimental results from three additional platforms. We selected the following datasets, all with 8-bit alphabets. More characteristics of these instances are shown in Table 1.

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

Random from [16] are strings of length $[0, 20)$ over the ASCII alphabet $[33, 127)$, with both lengths 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** [16] 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, 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 our report [3]. 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

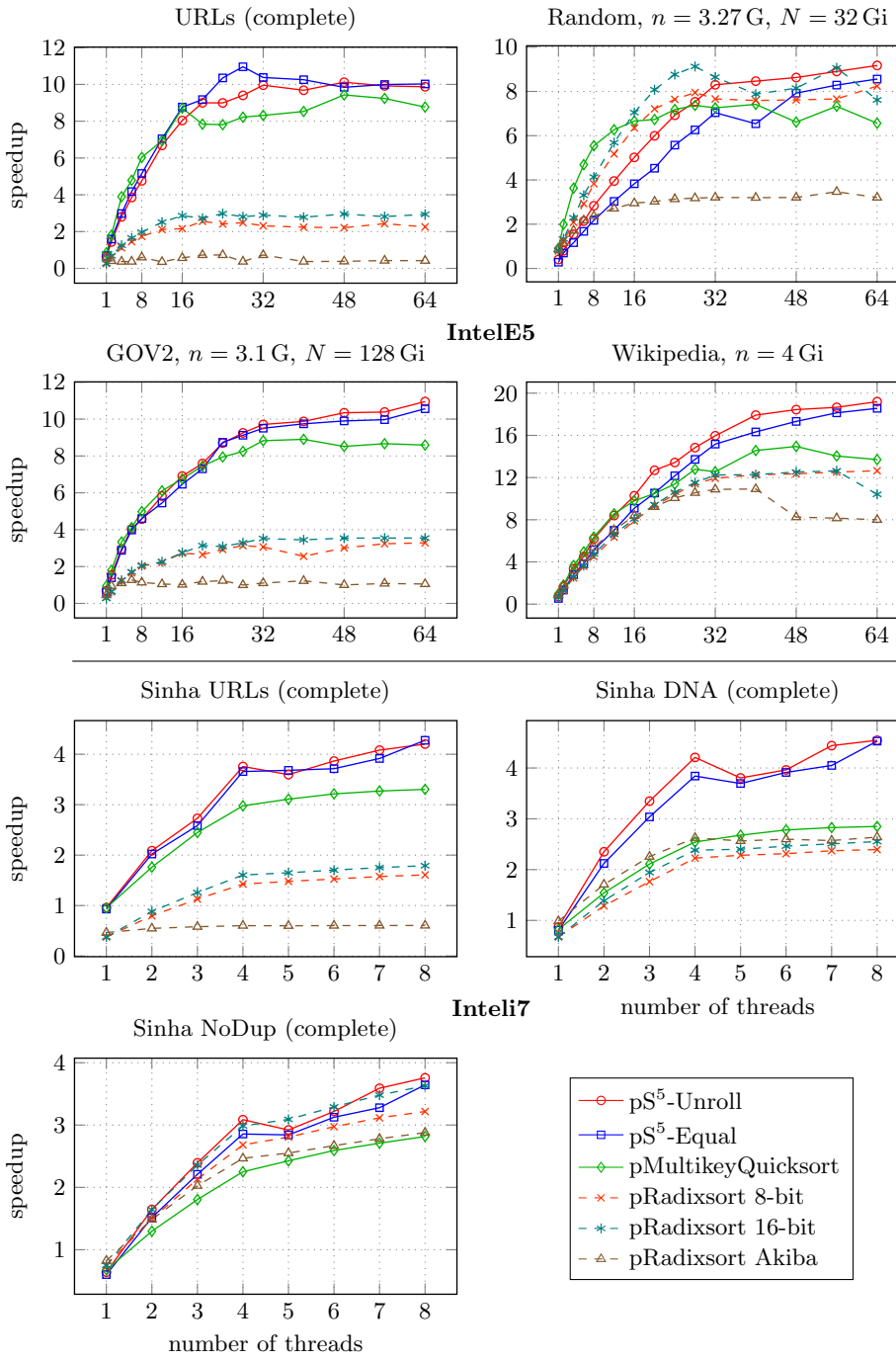

Fig. 2. Speedup of parallel algorithm implementations on IntelE5 (top four plots) and Inteli7 (bottom three plots)

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

speedup of our implementations and Akiba's radix sort over the best sequential algorithm [3]. 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 small numbers 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 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 our report [3] are likely to yield further improvements for pS^5. 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. Akiba, T.: Parallel string radix sort in C++. http://github.com/iwiwi/parallel-string-radix-sort (2011), git repository accessed November 2012
2. Bentley, J.L., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: ACM (ed.) 8th Symposium on Discrete Algorithms. pp. 360–369 (1997)
3. Bingmann, T., Sanders, P.: Parallel string sample sort. Tech. rep. (May 2013), see ArXiv e-print arXiv:1305.1157
4. 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. pp. 3–16 (1991)
5. 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. pp. 382–391. STOC '94, ACM, New York, NY, USA (1994)
6. Knöpfle, S.D.: String samplesort (November 2012), bachelor Thesis, Karlsruhe Institute of Technology, in German
7. Kärkkäinen, J., Rantala, T.: Engineering radix sort for strings. In: String Processing and Information Retrieval, pp. 3–14. No. 5280 in LNCS, Springer (2009)
8. McIlroy, P.M., Bostic, K., McIlroy, M.D.: Engineering radix sort. Computing Systems 6(1), 5–27 (1993)
9. Mehlhorn, K., Sanders, P.: Scanning multiple sequences via cache memory. Algorithmica 35(1), 75–93 (2003)
10. 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)
11. Ng, W., Kakehi, K.: Merging string sequences by longest common prefixes. IPSJ Digital Courier 4, 69–78 (2008)
12. Rantala, T.: Library of string sorting algorithms in C++. http://github.com/rantala/string-sorting (2007), git repository accessed November 2012
13. Sanders, P., Winkel, S.: Super scalar sample sort. In: 12th European Symposium on Algorithms. LNCS, vol. 3221, pp. 784–796. Springer (2004)
14. 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
15. Sinha, R., Wirth, A.: Engineering Burstsort: Toward fast in-place string sorting. J. Exp. Algorithmics 15, 2.5:1–24 (Mar 2010)
16. Sinha, R., Zobel, J.: Cache-conscious sorting of large sets of strings with dynamic tries. J. Exp. Algorithmics 9, 1.5:1–31 (Dec 2004)
17. Sinha, R., Zobel, J., Ring, D.: Cache-efficient string sorting using copying. J. Exp. Algorithmics 11, 1.2:1–32 (Feb 2007)
18. Tsigas, P., Zhang, Y.: A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In: PDP. pp. 372–381. IEEE Computer Society (2003)
19. Wassenberg, J., Sanders, P.: Engineering a multi-core radix sort. In: Euro-Par 2011 Parallel Processing, pp. 160–169. No. 6853 in LNCS, Springer (2011)