Combining Digital Access and Parallel Partition for Quicksort and Quickselect

Leonor Frias* and Jordi Petit†
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
lfrias@lsi.upc.edu

Abstract

In this paper, we combine digital access to strings with parallel partition to enhance parallel quicksort and quickselect implementations. Previously, digital access had only been combined with sequential comparison-based algorithms and data structures. We present broadly our approach, not only algorithmically but from the design and implementation point of view. Finally, we give some experimental results that indicate its effectiveness in practice.

1. Introduction

The partitioning of an array is a basic building block of many key algorithms, as quicksort and quickselect. Partitioning an array with respect to a pivot x consists of rearranging its elements such that, for some position s, all elements at the left of s are smaller than x, and all other elements are greater or equal than x. It is well known that an array of n elements can be partitioned sequentially and in-place using exactly n comparisons and m swaps, where m is the number of greater elements than x whose original position is smaller than s.

In parallelizing quicksort and quickselect, a key problem is parallelizing the partition. Recently, in [6] a summary of existing parallel partitioning algorithms and a further modification to obtain the minimal number of comparisons (i.e. n) has been presented.

Note that partition is a comparison-based algorithm, i.e. the keys are taken as a whole (atomic) to make comparisons. This is simple, there is often no other way to compare keys, and in many cases leads to a good performance. However, for string keys this approach is rough and string keys are indeed common.

String keys can be seen as a sequence of digits or characters, whose alphabet size is typically relatively small. Efficient *ad hoc* string algorithms and data structures (i.e. tries and its variants) exploit the digital structure of string keys to avoid redundant character comparisons. However, their worst-case performance is tied to the string length. Instead, it is possible to specialize comparison-based data structures and algorithms, so that efficient string comparisons are made whilst keeping the rest of combinatorial properties (see [9, 7, 4]). The application of these techniques relies on the relative order in which elements are compared. Specifically, information on the common prefixes between one element and its predecessor/successor in the access path must be kept. In the case of binary search trees (BSTs) and sorting (with an underlying BST structure), these techniques are amenable to be implemented and result in competitive data structures in practice. In particular, in [3] a detailed experimental comparison between these and other string searching data is presented.

However, to the best of our knowledge, digital access techniques have not been combined with parallel comparison-based algorithms or data structures. In order to do so, some precautions must be taken. In this paper, we describe how to perform digital access on string keys for parallel partitioning based algorithms, namely parallel quicksort and quickselect. We focus on currently widely avaliable multi-core architectures. Besides, we show the effectiveness of our approach in practice on the top of basic parallel quicksort and quickselect implementations.

The paper is structured as follows. First, we give an overview of the two worlds we aim to combine, namely, parallel partitioning based algorithms and the techniques to enhance digital access on the top of comparison-based algorithms. Second, we broadly describe how this can be done from the algorithmic, design and implementation point of view. Third, we present some experimental results for parallel quicksort and quickselect built using our approach. Finally, we sum up and discuss some possible future work.

*Supported by grant number 2005FI 00856 of the *Agència de Gestió d'Ajuts Universitaris i de Recerca* with funds of the European Social Fund and by Spanish project ALINEX (ref. TIN2005-05446)

†Partially supported by FET proactive integrated project 15964 (AEOLUS) and by Spanish project FORMALISM (ref. TIN2007-66523)

2. Preliminaries

Parallelizing quicksort can be done in two (not exclusive) ways: parallelizing the partition and parallelizing the divide and conquer. In the case of quickselect, one can only parallelize the partition. In the following, we first give an overview of parallel partition for atomic keys. Then, we summarize how sequential partition can be enhanced with digital access.

2.1. Parallel partition for atomic keys

A summary of existing parallel partitioning algorithms was presented in [6]. These include a simple algorithm by Francis and Pannan [5] (STRIDED), a fetch-and-add algorithm by Tsigas and Zhang [13] and a variation of the former avaliable in the MCSTL library [10] (F&A). Though very different in nature, these algorithms can be divided into three main phases: a) A sequential setup of each processor's work, b) a parallel main phase in which most of the partitioning is done, and c) a cleanup phase, which is usually sequential. In [6] it was noted that all the above algorithms disregard part of the work done in the main parallel phase when cleaning up. In order to overcome this drawback, an alternative parallel cleanup phase that uses the whole comparison information of the parallel phase was proposed. With that new method, scalable parallel partitioning algorithms that compare each element exactly once with the pivot are obtained. Moreover, a generalization of STRIDED to blocks (BLOCKED) was proposed in order to improve cache performance. For most of the aforementioned algorithms, though, it was shown experimentally that avoiding those redundant key comparisons is not noticeable in performance. Indeed, for fetch-and-add based algorithms the number of extra comparisons is constant with respect to n and for the rest of algorithms that number is expected to be constant.

2.2. Sequential partition for digital keys

Combining efficient digital access with (sequential) quicksort was particularly tackled in [9].

The following properties on the comparisons made in quicksort are used. First, if the pivot is chosen otherwise than comparing elements, comparisons are made solely in partitioning. In particular, the partition compares each element exactly once against the pivot. Besides, the implicit structure defined by the recursive calls in quicksort corresponds to a BST, where each pivot choice (and partitioning) constitutes a node and base cases in the recursion correspond to leaves. This relationship is depicted in Figure 1 with an example: the array to sort is {`quicksort`, `string`, `partition`, `quickselect`, `parallel`, `tree`, `sequential`}, each dependent quicksort step is depicted in a row where light gray squares correspond to pivot choices and dark grey squares correspond to elements that are already in their final position (previously chosen pivots). Note that, in the case of quickselect (not explicitly tackled in [9]), the structure of the recursive calls corresponds to a path, which is a particular case of BST.

Then, we proceed as follows to enhance fast string comparisons. We keep for each string x the length of the maximum common prefix of x with its predecessor and successor in the implicit BST structure. We denote them respectively $p(x)$ and $s(x)$. Thus, linear auxiliary space linear in the number of elements n is needed (which is much smaller than the sum of all string lengths). In Figure 1, for each string x the maximum of $p(x)$ and $s(x)$ is shown in each step. Finally, the following changes must be done in partition. Let y be the string acting as pivot, and let x be an string in the array to be partitioned. Then, the comparison function of x against y must be specialized so that it uses $p(x)$, $p(y)$, $s(x)$ and $s(y)$ to avoid redundant character comparisons, and updates $p(x)$ and $s(x)$ accordingly ($p(y)$ and $s(y)$ remain unchanged). Besides, the swap function used in partition to swap two strings x_1 and x_2 must be specialized so that, in addition, $p(x_1)$ is swapped with $p(x_2)$, and $s(x_1)$ is swapped with $s(x_2)$.

Note that for every string x, $p(x)$ and $s(x)$ must be initialized to 0 at the beginning of quicksort and quickselect algorithms (i.e. initially, there are no common prefixes). Moreover, the comparisons results are related between partitioning steps but not inside one partitioning step. In particular, the first call to partition must compare all the strings from the beginning as in the non-specialized partitioning algorithm and in addition, must update $p(x)$ and $s(x)$. However, from the second call on, the computed prefixes $p(x)$ and $s(x)$ are used and updated accordingly to avoid redundant character comparisons. Thus, specializing partition is only beneficial if it is going to be used repetitively (as in quicksort and quickselect) because the prefix information gathered in previous iterations can be used.

3. Putting all together

In the previous section, we have seen that combining efficient digital access to strings with partitioning based algorithms is based in the following properties. On the one hand, partition compares each element exactly once against the pivot. That is guaranteed precisely by the parallel partitioning algorithms in [6] . On the other hand, the implicit structure defined by the recursive calls corresponds to a BST. Indeed, the parallel execution of recursive calls in the case of quicksort does not break the relative order of calls. Thus, we can enhance digital access to strings on the

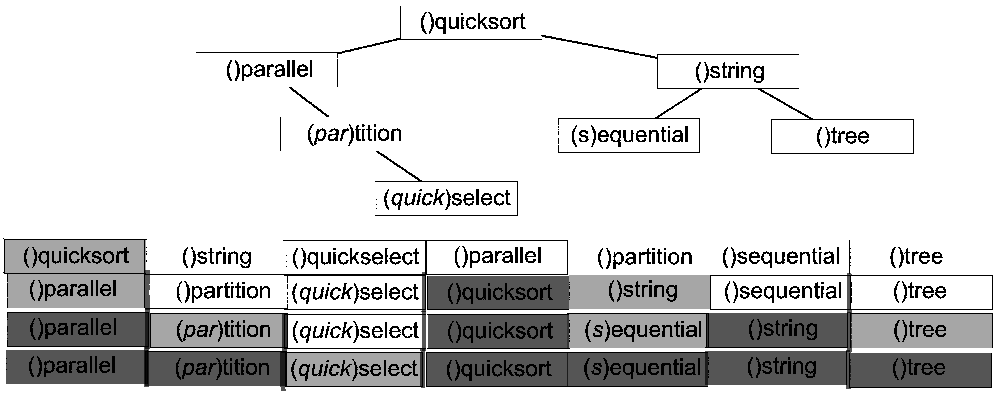

Figure 1. Relationship between quicksort and BSTs

aforementioned parallel partitioning algorithms to build efficient parallel quicksort and quickselect implementations.

In the following we describe how to do so in more detail. We start from an outer view, related to the interface, then comment on the main design and implementation issues, and finally make some additional observations.

3.1. Outer changes

The parallel partitioning algorithms in [6] are provided according to the specification of the `partition` algorithm of the Standard Template Library (STL) of the C++ programming language [8]. The input parameters of `partition` and their based algorithm, `sort` and `nth_element`, are a sequence and a comparator. Sequences are defined in the STL by a begin and end iterator (iterators are high level pointers). In particular, we consider random access sequences, i.e. sequences that can be operated like an array. Besides, the interface of `sort` in the Java API is similar to the STL `sort`.

Unfortunately, in order to enhance (parallel or sequential) partition with digital access is not enough with providing an specialized comparator but a swap function (see Section 2.2). Thus, we must provide a full replacement for string partition. In particular, we need a replacement for the sequential and the parallel version. We call them respectively `string_partition` and `parallel_string_partition`.

Note that `string_partition` is not only needed when the input data is too small but to be called by `parallel_string_partition`. In particular, the implementation of some of the parallel partitioning algorithms in [6] call sequential partition on virtual sequences made by wrapping random access iterators into new ones (in the following, wrapper random access iterators), such that, a fixed number of blocks or elements can be jumped transparently.

3.2. Design and implementation

We want to stick to the purpose of providing a not intrusive specialization (minimal additional code) of `string_partition` and `parallel_string_partition`. We propose the following. First of all, we store the information on the common prefixes separately from the data itself, in particular in some array A. The rank of the elements is used to access both the elements themselves and A, in particular in comparison and swap functions. We have resorted to define swap and comparison functions on random access iterators. In the case of wrapper random access iterators, we need to obtain the original random access iterator. Instead of modifying their definition, we provide a functor class `getRandAccIt` for each of them.

In addition, we encapsulate the specialized comparison and swap functions, together with the pivot and the rest of attributes in a class called `DigitalComparison`. In this way, only the following further two modifications are needed. First, (`parallel_`)`string_partition` have an instance of the class as an input paremeter. Second, the calls to swap and comparison functions need to be updated according to the new interface. Moreover, this approach provides a flexible framework to be able to provide several implementations of `DigitalComparison`. In particular, it can be implemented trivially taking keys as a whole.

Furthermore, the pivot consists both on the string key and its prefix information. We can either store an iterator that gives access to them, or we can store both pieces of information (i.e. a reference to the string key and a reference or copy to its prefix information). The second option gives more flexibility and should be faster in practice.

Finally, given that several instances of `DigitalComparison` may exist at a certain point of a parallel quicksort execution, we resorted that `DigitalComparison` stores only a reference to A. A must be created once at the beginning of parallel quicksort and

quickselect and can be initialized in parallel.

3.3. Additional precautions

Many implementations of sequential and parallel quicksort and quickselect exist. Crucial aspects in their performance are the choice of the pivot and how repeated elements are handled (see [2], [1]). However, in order to enhance digital access, these decisions must keep the prefixes information consistent. In the following, we discuss how this can be done.

Pivot choice. A good pivot choice is crucial for guaranteeing the quasilinear performance of quicksort. A good choice is one that results in partitioning the input in two pieces of asymptotically equal size. In order to guarantee this property with high probability, typical approaches are choosing randomly the pivot (*random sampling*), using the median of 3 (or more elements) or combining both.

However, computing the median implies making comparisons and so, causing that some pair of elements are compared twice. In principle, that would not allow enhancing digital access. However, we can compare the elements as long as the prefix information is not corrupted. One possibility is providing an additional comparison function that does use prefix information but does not update it. Even better, given that we are not that interested in the exact result, we can rely merely on prefix information to make a decision (to the detriment of the quality of the partitioning). We call the later technique *approximate comparison*.

Finally, we must avoid comparing the pivot against itself. A typical approach is placing the element chosen as a pivot in a fixed end of the input.

Handling repeated elements. Repeated elements are also a source of biased size partitionings if these are not dealt with carefully. Dealing with repetitions turns almost obligatory with strings because most real data contain repetitions. The most effective methods are always swapping equal elements and the 3-way partition in [1].

In any case, both methods guarantee that no pair of elements are compared twice, so they can be combined with digital access smoothly.

4. Experimental analysis

In the following we present some performance results on combining digital access to strings and parallel partitioning based algorithms. Specifically, we analyse parallel quicksort and quickselect performance.

The tests have been run on a machine with 4 GB of main memory and two sockets, each one with an Intel Xeon quadcore processor at 1.66 GHz with a shared L2 cache of 4 MB shared among two cores. Thus, there are 8 cores in total. We have used the GCC 4.2.0 compiler with `-O3` optimization.

All tests have been repeated 20 times; figures show averages.

This section is organized as follows. First, we present our implementation. Then, we describe the tested datasets. It follows an overview of sequential results. Finally, we discuss parallel results.

4.1. Tested implementation

We have implemented (`parallel_`)`partition_string` generically with respect of the specific string type. In particular, our implementation can tackle `char*` and `std::string`, which add a level of indirection to be able to offer more advanced features. In this sense, applying our approach on the top of `std::string` has less relative overhead in space and time. However, we present our results only for `char*` because most existing implementations for string algorithms are keyed for `char*` (actually, in most cases are C implementations). Among these are the implementations in [3] on combining digital-access with search trees and burstsort [11], a cache-efficient (burst)-trie based sorting algorithm.

With respect to the sequential partitioning algorithm, we have implemented 3-way partition. With respect to the parallel partitioning algorithms, we have considered F&A and BLOCKED. Besides, in order to choose the pivot we use median of three with approximate and exact comparison.

Finally, the parallel partitioning algorithms in [6] are defined using OpenMP. We have also used OpenMP to define basic parallel implementation for quickselect and quicksort on the top of them.

Whilst parallelizing quickselect relies merely on parallelizing the partition, parallelizing quicksort can be done additionally by parallelizing the independent work by divide and conquer. In particular, we have implemented simple static distribution of the work. Anyway, any further enhancement that does not imply comparisons, such as load-balancing techniques (see [10, 12]), are perfectly compatible. It is out of the scope of this paper analysing how quicksort can be parallelized the best.

The implementation is available at: `http://www.lsi.upc.edu/~lfrias/research/parstr/parstr.zip`.

4.2. Datasets

We consider several string datasets that depend on string parameters (such as string length and alphabet size) and on the distribution of the string instances. We have run experiments with synthetic and real datasets. In all cases, once

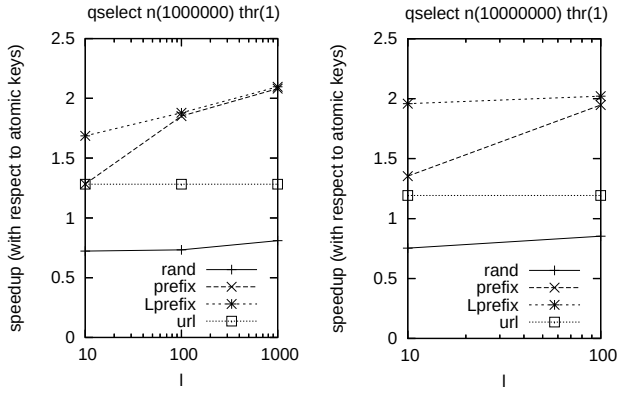

Figure 2. Sequential quickselect with digital access (left: $n = 10^6$; right: $n = 10^7$)

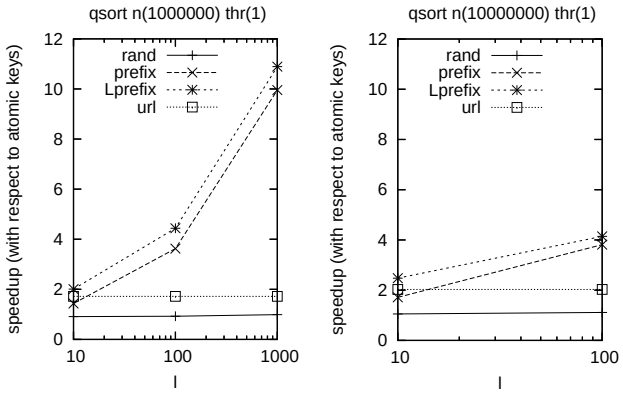

Figure 3. Sequential quicksort with digital access (left: $n = 10^6$; right: $n = 10^7$)

.

the data is generated, it is shuffled, so that any accidental memory locality in generating the data is broken.

With respect to synthetic datasets we have evaluated the following:

- string length (denoted by l): 10, 100, 1000
- alphabet: binary, A-Z
- distribution: uniformly random (denoted by `rand`), common prefix of uniformly random length (denoted by `prefix`), long prefix of fixed size (denoted by `lprefix`).
  In the case of `prefix` and `lprefix` the strings are made of two parts: a prefix of length l_p containing only one distinct character, and a suffix of length $l - l_p$ randomly generated. In the case of `prefix`, l_p is chosen randomly, in the case of `lprefix`, l_p is the same for all words.

With respect to real datasets we have considered the ones in burstsort [11]. From them, we focus on an url dataset (denoted by `url`) because the average length of the strings is the highest. Recall that enhancing digital access on the top of comparison-based algorithms is specially useful for arbitrarily long strings and prefixes.

Finally, we have considered input sizes of 1 and 10 million elements. In the latter case, the longest string length considered is 100 because otherwise the dataset does not fit in main memory.

4.3. Sequential performance

Figures 2 and 3 show respectively some performance results on sequential quickselect and quicksort with enhanced digital access. The results are shown as speedups with respect to the regular sequential quicksort and quickselect implementations in which keys are considered atomic. We consider several combinations of string length and number of elements. Specifically, we show results for words in the A-Z alphabet generated following `rand`, `prefix` and `lprefix` distributions. Moreover, we show the results compared to the `url` dataset (the assigned string length is arbitrary, just for plotting the results together). Finally, the pivot is selected approximately (not significative differences where shown in performance with respect to exact comparison).

In the case of quickselect (Figure 2), enhancing digital access for strings harms if the common prefixes are scarce (i.e. random strings). Instead, in the case of quicksort (Figure 3), applying this technique is always beneficial. Recall from Section 2.2 that the first partition always does more work than atomic partition and it is as we go deeper into recursion, that the gathered information on common prefixes becomes really useful. In the case of quickselect, only part of work done in one partitioning is used later, and so, the overhead of the first recursive calls is more notorious.

Moreover, logically, as longer the common prefixes, the performance with enhanced digital access is better and better than without. Also the results for binary alphabets are generally better because it is much more likely to generate common prefixes when doing random choices.

4.4. Parallel performance

In the following, we describe performance results for parallel quickselect (Figures 4 to 9) and parallel quicksort (Figures 10 and 11). The results are shown both for atomic and digital keys. Besides, we analyse the effect in parallel performance of several combinations of string length and number of elements. We focus in parallel quickselect results because the only source of parallelism is partitioning.

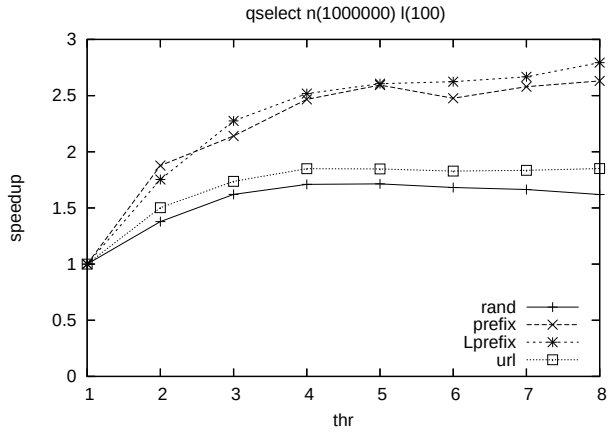

Figure 4. Parallel quickselect with atomic keys ($n = 10^6$, $l = 100$)

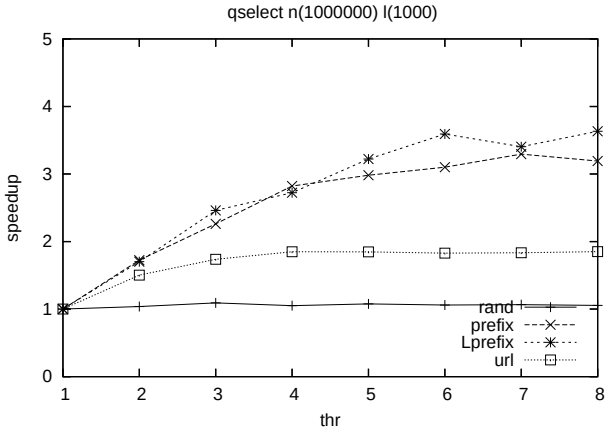

Figure 6. Parallel quickselect with atomic keys ($n = 10^6$, $l = 1000$)

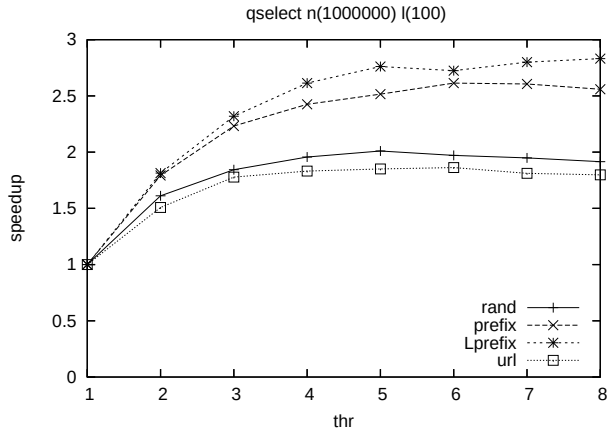

Figure 5. Parallel quickselect with digital access ($n = 10^6$, $l = 100$)

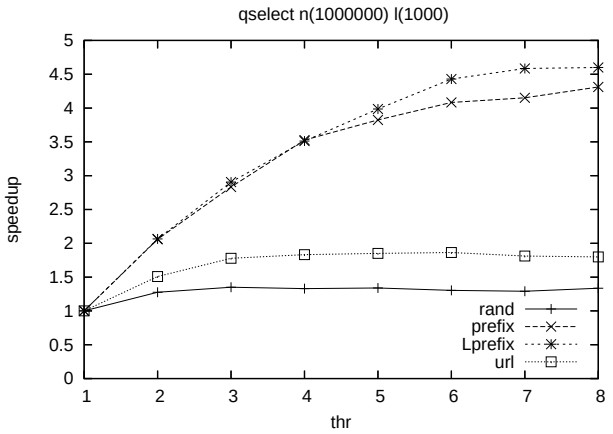

Figure 7. Parallel quickselect with digital access ($n = 10^6$, $l = 1000$)

In all cases, the results are shown as speedups with respect to the respective sequential implementation (i.e. the parallel performance for atomic keys is compared against the sequential performance for atomic keys and the parallel performance for digital keys is compared against the sequential performance for digital keys). The words are generated in the A-Z alphabet following `rand`, `prefix` and `lprefix` distributions. Moreover, we show the results compared to the `url` dataset (the assigned string length is arbitrary, just for plotting the results together). Finally, the results are for implementations build on the top of F&A partitioning algorithms (the speedups obtained for BLOCKED are similar, but generally worse).

For a fixed set of string parameters, the greatest speedups are obtained when enhancing digital access (see Figures 4, 6, 8, 10 considering atomic keys against respectively Figures 5, 7, 9 and 11 considering digital keys). Therefore, combining digital access with parallel partitioning based algorithms not only does not hurt scalability but generally improves it.

Between our parallel quicksort and quickselect implementations, quickselect achieves higher parallelism. However, recall that our parallel quicksort distributes statically the divide and conquer work, which definitely is not an optimal strategy. Nonetheless, the absolute speedups for quickselect are worst than those obtained in [6] for the same machine but for integers. First of all, the tested number of elements is smaller (because otherwise the data does not fit in main memory) and thus, parallelism is not so effective. Indeed, we can see comparing performance results for quickselect (Figures 4 and 8, or 5 and 9), that for a fixed string length, the speedups are better as larger the num-

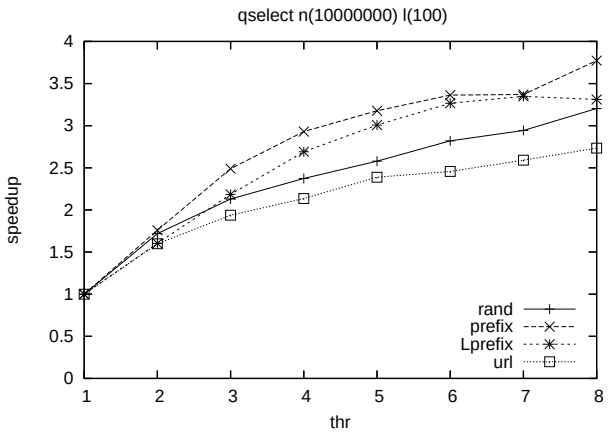

Figure 8. Parallel quickselect with atomic keys ($n = 10^7$, $l = 100$)

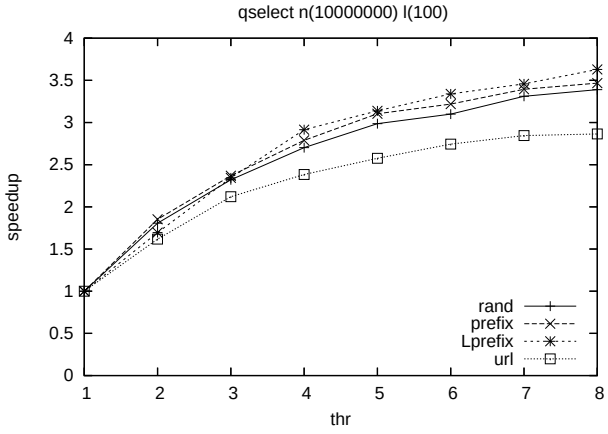

Figure 9. Parallel quickselect with digital access ($n = 10^7$, $l = 100$)

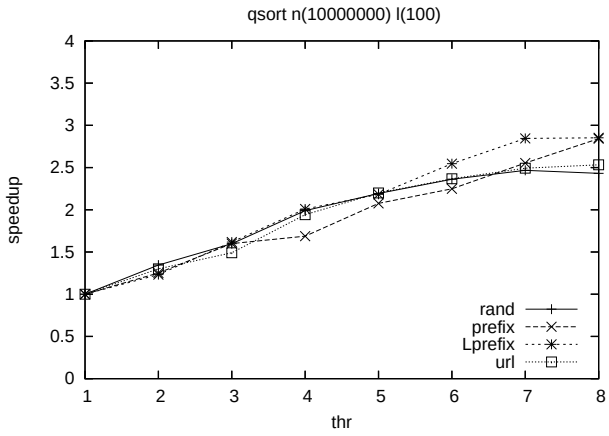

Figure 10. Parallel quicksort with atomic keys ($n = 10^7$, $l = 100$)

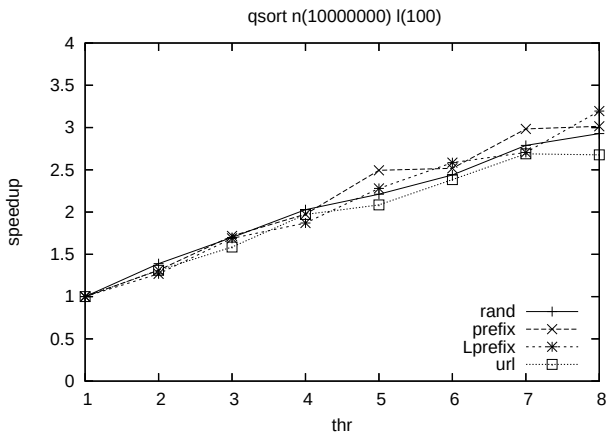

Figure 11. Parallel quicksort with digital access ($n = 10^7$, $l = 100$)

ber of elements. That is also the case for quicksort. On the other hand, partitioning strings is more memory intensive than partitioning integers because to access the actual string data a pointer must be followed (which might be far in memory). Indeed, the best speedups, both for atomic and digital keys, are obtained for the longest prefixes because the cost of arithmetic operations (comparing characters and using integer prefix information respectively) is more balanced with the cost of memory accesses. That is particularly true for quickselect (see Figures 6 and 7).

5. Conclusions

In this paper, we have shown how to combine digital access to strings with parallel partitioning based algorithms, namely, quicksort and quickselect. To the best of our knowledge, this enhancement had not been applied before on the top of parallel comparison-based algorithms or data structures. Actually, this approach is independent of parallelism itself as long as the relative order between comparisons is kept. The main issue in the case of parallel quicksort and quickselect is that partition must compare each element at most once. But this can be done relying on the modified partitioning algorithms proposed in [6].

From an engineering perspective, we have described how the C++ design and the implementation of these parallel partitioning algorithms can be modified to provide an specialization for strings. Our proposal aims being the less intrusive as possible. Besides, we have discussed on additional precautions that must be taken, for instance, when electing the pivot.

Finally, we have presented some experimental results

that indicate the practicability of this approach. Indeed, the scalability of parallel quicksort and quickselect is generally better when combined with digital access. However, the absolute speedup results are far to be optimal. The next step is applying our parallel partitioning algorithms for strings on the top of highly-tunned and tested implementations of parallel quicksort and quickselect. We expect that doing so will neither damage the scalability of the original algorithms. In particular, it could be interesting to integrate this work into some existing parallel implementation of the STL.

As further future work, digital access could be combined with other parallel comparison-based algorithms and data structures. That is interesting from a practical perspective because many more parallelizations of comparison-based algorithms exist than of *ad hoc* algorithms.

References

[1] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Softw. Pract. Exper.*, 23(11):1249–1265, 1993.

[2] T. H. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, Cambridge, 2nd edition, 2001.

[3] P. Crescenzi, R. Grossi, and G. F. Italiano. Search data structures for skewed strings. In *Experimental and Efficient Algorithms, Second International Workshop, WEA 2003, Ascona, Switzerland, May 26-28, 2003, Proceedings*, volume 2647 of *Lecture Notes in Computer Science*, pages 81–96. Springer, 2003.

[4] G. Franceschini and R. Grossi. A general technique for managing strings in comparison-driven data structures. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, 2004.

[5] R. S. Francis and L. J. H. Pannan. A parallel partition for enhanced parallel quicksort. *Parallel Computing*, 18(5):543–550, 1992.

[6] L. Frias and J. Petit. Parallel partition revisited. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008*, volume 5038 of *Lecture Notes in Computer Science*, pages 142–153. Springer, 2008.

[7] R. Grossi and G. F. Italiano. Efficient techniques for maintaining multidimensional keys in linked data structures. In *ICALP '99: Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, pages 372–381, London, UK, 1999. Springer-Verlag.

[8] International Standard ISO/IEC 14882. *Programming languages — C++*. American National Standard Institute, 1st edition, 1998.

[9] S. Roura. Digital access to comparison-based tree data structures and algorithms. *J. Algorithms*, 40(1):1–23, 2001.

[10] J. Singler, P. Sanders, and F. Putze. The Multi-Core Standard Template Library. In *Euro-Par 2007: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694, Rennes, France. Springer Verlag.

[11] R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics*, 9:1.5, 2004.

[12] D. Traoré, J.-L. Roch, N. Maillard, T. Gautier, and J. Bernard. Deque-free work-optimal parallel STL algorithms. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 887–897, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In *11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 372–381, 2003.