Received: 15 December 2020 | Revised: 16 August 2021 | Accepted: 18 August 2021

DOI: 10.1002/cpe.6606

RESEARCH ARTICLE

WILEY

CPP11sort: A parallel quicksort based on C++ threading

Daniel Langr | **Klára Schovánková**

Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic

Correspondence
Daniel Langr, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 16000 Prague, Czech Republic.
Email: daniel.langr@fit.cvut.cz

Funding information
Czech Ministry of Education, Youth and Sports, Grant/Award Number: CZ.02.1.01/0.0/0.0/16_019/0000765

Summary

A new efficient implementation of the multithreaded quicksort algorithm called CPP11sort is presented. This implementation is built exclusively upon the threading primitives of the C++ programming language itself. The performance of CPP11sort is evaluated and compared with its mainstream competitors provided by GNU, Intel, and Microsoft. It is shown that out of the considered implementations, CPP11sort mostly yields the shortest sorting times and is the only one that is portable to any conforming C++ implementation without a need of external libraries or nonstandard compiler extensions. The experimental evaluation with various input data distributions resulted in parallel speedup between 16.1 and 44.2 on a 56-core server and between 6.8 and 14.5 on a 10-core workstation with enabled hyperthreading.

KEYWORDS

C++, in-place sorting, multithreading, parallel algorithm, parallel sorting, quicksort, shared memory

1 | INTRODUCTION

Sorting is one of the most fundamental building blocks of computer software in general. In performance-aware applications, utilizing multicore hardware for sorting is inevitable in shared-memory systems. Due to growing memory capacity installed in computers, single-threaded sorting of large data sets can take minutes or even tens of minutes.[1] This problem is especially related to shared-memory systems such as workstations, servers, and computational cluster nodes, which generally run high-performance software. Such software is commonly written in C++. When a C++ programmer encounters a multithreaded sorting problem, they, nowadays, do not have many options for its solution. The available ones are either immature or require external libraries and nonstandard compiler extensions; see Section 3 for more details. To our best effort, we have not found any multithreaded sorting algorithm implemented in portable C++ that would provide sorting performance and scalability comparable with existing mainstream alternatives. In this article, we present such a multithreaded sorting implementation dubbed *CPP11sort*. The name stems from the fact that its code is built upon the C++ threading primitives introduced by the C++11 standard.[2] This standard has been already supported by all mainstream C++ compiler and standard library vendors for many recent years,[3] which makes CPP11sort portable among the vast majority of today's development platforms.

Fast generic sorting algorithms—that is, algorithms not focused or limited on data with some specific characteristics—prevails in practice in two types. The first ones are *unstable* but *in-place*, which means that they do not preserve the input order of equal elements but their auxiliary space requirements are lower than $O(n)$. On the contrary, the second ones are *stable* but *not in-place*—they preserve the order of equal elements but require $O(n)$ auxiliary space. The most commonly used representatives of generic unstable in-place and stable not in-place algorithms are *quicksort*[4,5] and *mergesort*,[6] respectively. Both these algorithms form a basis of generic sorting solutions available in many programming languages and their libraries. Both also have been transformed into multithreaded forms, which are significantly more complex than their single-threaded counterparts.

In the presented work, we focused on the problem of generic unstable in-place sorting on shared-memory CPU-based systems* (this focus stemmed from our previous research activities where we frequently needed to sort data sets too large for out-of-place sorting[7-9]). The keystone of

*Note that this work is not related to distributed-memory sorting nor to sorting on non-CPU architectures such as GPUs.

CPP11sort is a new variant of multithreaded quicksort, which we introduce in detail in the text below. Next, we provide its comprehensive scalability study and experimental comparison with all its major competitors identified in Section 3.

2 | BACKGROUND

In this section, we provide a theoretical background which is necessary to understand the further presentation of our work. Namely, we discuss threading options for the C++ programming language, describe the original single-threaded quicksort algorithm, and introduce its multithreaded parallelization.

2.1 | Threading in C++

Before C++11, there was no threading provided by the C++ programming language itself. Developers must have used either lower-level threading APIs provided by a particular system, such as POSIX threads (pthreads) of Windows threads, or higher-level APIs such as OpenMP or Intel Thread Building Blocks (TBB). C++11 introduced an abstract memory model in the core language and some basic threading primitives in the C++ Standard Library, which allow creating threads and synchronizing memory accesses. It consists of a modern portable object-oriented threading API, which, however, lacks many higher-level features, such as parallelization of loops, thread pools, and task parallelism with task scheduling. The C++17 standard added the support for parallel "algorithms" defined in the C++ Standard Library.[10] However, even nowadays, most implementations has either not included their support at all, or provided them only in an experimental form of a wrapper around some external library used as a backend (see Section 3 for details).

2.2 | Quicksort

Let $(a_1, \ldots, a_n)$ denote the input sequence of the elements to be sorted. Without a loss of generality, we will say that a_i is:

- *less than* a_j if a_i should be placed before a_j in the sorted sequence,
- *greater than* a_j if a_i should be placed after a_j in the sorted sequence,
- *equal to* a_j if their order in the sorted sequence does not matter.

Quicksort is a recursive algorithm. Initially, it takes the input sequence and *partitions* it, in-place, into two subsequences with respect to a selected element called the *pivot*. In the left subsequence, all the elements are less than the pivot, while in the right subsequence, all the elements are greater than or equal to the pivot. Then, the algorithm is recursively applied to both subsequences until either an empty or a single-element subsequence is reached, which is sorted intrinsically.

In practice, such a basic form of quicksort would not perform well. The first problem is that it has $O(n \log n)$ complexity in an *average case*, but $O(n^2)$ in the *worst case*.[11] To avoid worst-case scenarios, implementations typically set a limit for the recursion depth and if it is exceeded, they switch from quicksort to some other $O(n \log n)$ algorithm such as *heapsort*. Second, the recursive calls impose some runtime overhead, which is negligible if a subsequence length is large, but becomes significant as it gets smaller. At some threshold, it pays off to switch from quicksort to some $O(n^2)$ sorting algorithm, usually to *insertion sort*. Finally, recursive processing of both subsequences after each partitioning would generate unnecessary many function calls. Therefore, only the smaller subsequence in usually processed recursively and the larger subsequence is processed in a loop. This optimization technique not only reduces sorting times, but also reduces the overall auxiliary space requirements, namely memory requirements for stack frames.

The overall performance of quicksort may be also highly affected by the method of pivot selection. An ideal pivot is the median of the partitioned sequence; however, its finding is too costly in practice. Instead, several variants are typically used, such as the selection of an element on a fixed position, selection of an element on a random position, or selection of the median of some subset of fixed-/randomly positioned elements (for instance, the so-called *median-of-three*). Generally, the more complex is the median selection, the more balanced is the partitioning, but also the more runtime overhead is imposed into its resolution.

2.3 | Parallel quicksort

After each partitioning, the recursive processing of both subsequences forms independent tasks. Basic parallelization of quicksort is therefore relatively straightforward—we just need to map these tasks to different threads. There are two main approaches on how to accomplish that:

1. For each task, a new thread is created. A drawback of this approach is that it suffers from thread-creation and context-switching overhead and can also result in reaching a limit of the number of threads on a given system.
2. A *thread pool* with a fixed number of threads is used and the tasks are scheduled between them; this alternative is usually more efficient.

The drawback of the second approach is that it needs to be supported by the used threading implementation. For instance, OpenMP and Intel TBB provide thread pools with task scheduling, while pthreads and C++ do not.

Additionally, efficient parallel quicksort implementations switch to sequential sorting—in the context of the actual thread—once the recursion depth or subsequence length reaches some threshold. This reduces the overhead of creating new threads or creating new tasks and their scheduling.

2.4 | Parallel partitioning

The above-described task-based quicksort parallelization can be relatively easily achieved. However, it does not scale well since, at higher recursion levels, there are not enough tasks to utilize all CPU cores. Namely, the top-level partitioning is performed by a single thread only, which makes all the cores except a single one idle. The only possible resolution of this issue is to perform partitioning in parallel.

Parallel partitioning is a nontrivial problem. Efficient algorithms process data in blocks, which increases cache utilization and avoids false sharing. Typically, the threads fetch a block from both the beginning and the end of the partitioned subsequence and swap their elements such that, in the end, all the elements in the left block are less than the pivot, or all the elements in the right block are greater than or equal to the pivot (such a block is called being *neutralized*). This process continues until fetching from the left and from the right meets. Then, there are at most as many not yet neutralized blocks as many threads participate in partitioning. These remaining blocks are processed such that they are, for example, swapped in the middle of the whole sequence. This effectively makes a new shorter subsequence, which can be partitioned either again in parallel or by a single thread only. Finally, elements that do not fit into any block (due to the general indivisibility of the sequence length and the block size) are processed.

With parallel partitioning, an additional load-balancing problem emerges. At the top level of the quicksort recursion, the situation is simple—the partitioning of the input sequence is performed by all the threads. However, then these threads need to be split between the tasks of the partitioning of both created subsequences. Different strategies may be applied here, such as splitting the thread evenly, or proportionally to the lengths of the subsequences. Finally, when the number of threads mapped to an actual task drops to one, the corresponding subsequence is usually processed by a sequential quicksort.

3 | RELATED WORK

There have been published several papers about parallel quicksort algorithms. Tsigas and Zhang[12] proposed a parallel quicksort with an efficient parallel partitioning strategy that has been widely used in practice and frequently mentioned in the literature. Alternative solutions and their comparison were presented by Pasetto and Akhriev,[13,14] who also proposed a new approach to parallel partitioning, however, not in the form of an algorithm.[14] Another parallel quicksort was proposed by Mahafzah,[15] but their paper lacked experimental comparison with other solutions. Moreover, they presented results only for small data (up to 80 MB of memory footprint) and a small number of threads (up to 8). Süs and Leopold[16] compared several parallel implementations of quicksort based on OpenMP and POSIX threads (Pthreads).

Many implementations of parallel quicksort algorithms may be found in public open-source repositories. However, there are only a few that employ parallel partitioning internally. The others will not be further considered in this text due to their low sorting performance and scalability. For illustration, the Parallel File Quicksort library[17] provides a multithreaded quicksort implementation with sequential partitioning, and, in our auxiliary experiments, it was around seven times slower than CPP11sort on a 20-core system.

The efficient parallel quicksort implementations with parallel partitioning we have found are the following ones:

1. *Parallel mode of libstdc++*—Libstdc++ is an implementation of the C++ Standard Library provided by GNU. It contains an experimental parallel version of the C++ algorithms referred to as the "parallel mode,"[18] which originally stemmed from the (now outdated) multicore standard template library (MCSTL).[19] It uses OpenMP as a threading mechanism and provides parallel quicksort in two variants. The *unbalanced* quicksort tries to partition a sequence into subsequences of the same lengths by selecting a pivot from 100 elements (by default). Then, it maps the threads to both emerging subtasks evenly. The *balanced* quicksort selects the pivot as a median of three elements only, but then maps the threads to the emerging tasks proportionally to the lengths of their subsequences.
2. *Intel TBB* contains a parallel quicksort provided by a function template `tbb::parallel_sort`. It is based on the custom thread pooling and task scheduling that the library implements.
3. *Microsoft STL* provides a parallel sorting implementation conforming with the C++17 Standard. However, it is tightly bound to Microsoft's development tools, which hinders its portability to any other development environment.

4. *AQsort* is our previous parallel quicksort implementation build upon OpenMP.[1] Its main feature is that—on the contrary to the other implementations—it is capable of working with a user-provided function for swapping elements. This slightly reduces optimization options, but, effectively, allows sorting multiple datasets (such as arrays) at once.

None of these implementations provides portable parallel sorting built upon C++ threading.

There also exist some additional alternatives. GNU has included the C++17 parallel algorithms into libstdc++, however, without their C++-based implementation. Instead, it only wraps the functionality provided by the Intel TBB backend. The same holds for the Intel Parallel STL (PSTL), where, again, the Intel TBB backend is employed. Moreover, the Intel Parallel STL wraps the not-in-place parallel sorting implementation from Intel TBB, which can be discovered by inspecting the PSTL source code or by measuring the memory consumption of any sorting test program. Another parallel quicksort implementation was provided by CilkPlus—the extension of C and C++ languages to support data and task parallelism. Its parallel quicksort was experimentally evaluated in our previous work.[1] It was not portable and has been deprecated since 2018.

Some other related parallel sorting algorithms of possible interest, which are beyond the domain of generic parallel in-place sorting on shared-memory CPU architectures, are as follows: parallel radix sort for graphic processing units (GPUs) by Xiao et al.,[20] array sort—an adaptive merge sort-based sorting with multithreaded version by Huang et al.,[21] parallel quicksort algorithm on optical transpose interconnection systems hyper hexa-cell (OTIS-HHC) optoelectronic architecture by Al-Adwan et al.,[22] and bitonic sort on a chained-cubic tree (CCT) interconnection network by Al-Haj Baddar and Mahafzas.[23]

4 | CPP11SORT

Our goal was to implement an efficient, fast, and scalable parallel quicksort with parallel partitioning capabilities based exclusively on the threading primitives provided by the C++11 Standard. The solution we have achieved and called CPP11sort was initially developed by Schovánková and presented in her thesis.[24] The corresponding source code was then ported to GitLab.[25] Since this source code consists of templates to provide generic sorting (sorting of data of any type), it is available in the form of header files. The main header file to be included by a CPP11sort user is `cpp11sort.h`. The algorithm can be controlled by several parameters that are introduced in Table 1 and further explained in the text below.

CPP11sort implements the parallel partitioning algorithm proposed by Tsigas and Zhang,[12] which has also been adopted by the MSCTL and libstdc++. Our parallel partitioning implementation forms a standalone code module that may be used also for other purposes than the sorting itself. The algorithm is based on the fetching of partitioned sequence blocks as described in Section 2.4. The size of the block is denoted by B and may be set according to Table 1. Its default value stemmed from a series of auxiliary experiments with various kinds of sorted data. A CPP11sort user should be aware that higher values of B may result in less synchronization but also worse load balancing between threads. On the contrary, lower values of B then may yield better load balancing but at a price of more synchronization overhead.

Since C++ does not provide any thread pooling and task scheduling mechanisms, our solution is based on a custom thread pool with a thread-safe work-sharing queue. The created tasks are inserted into this queue and then fetched by idle threads. The number of threads is fixed and there are two thread pools. The first one is used for the recursive quicksort itself, while the second one is used for parallel partitioning. Due to different aspects of both these problems, it would be inefficient and also highly complex to implement them with a single thread pool only. The synchronization of threads is based on atomic variables, mutexes, and condition variables. Especially, the latter prevents busy waiting in situations where it would unnecessarily block a CPU core.

The source code was designed to be highly modular. As already mentioned, our parallel partitioning implementation can be used independently of the CPP11sort sorting code. Moreover, for experimental purposes, we have also implemented a second parallel quicksort version based on the spawning of new threads for all created tasks. Both the thread-pool and thread-spawn versions share the common code base, which may be even used for the development of additional custom load balancing strategies on top of it. (The abovementioned thread-spawn solution is not provided by the CPP11sort API and will not be further considered in the text below. Even though it yielded similar sorting times as the thread-pool version in our experiments, its drawback are significantly larger memory requirements due to allocated stacks for individual threads.)

TABLE 1 Parameters that control the CPP11sort algorithm

Parameter	Symbol	Preprocessor symbol	Default value
Partitioning block size	B	`CPP11SORT_PARTITION_BLOCK_SIZE`	6000
Sequential threshold	S	`CPP11SORT_SEQUENTIAL_THRESHOLD`	50,000
Pivot samples	P	`CPP11SORT_PIVOT_SAMPLES`	30

Note: To take effect, the preprocessor symbols needs to be defined prior to the inclusion of the `cpp11sort.h` header file into the source code.

CPP11sort switches to sequential sorting once the subsequence length drops below the limit denoted by *S*; see Table 1 for details. The remaining algorithm parameter *B* denotes the number of samples for the selection of the pivot, which is then evaluated as their median.

4.1 | Pseudocode

The pseudocode of CPP11sort is presented in Algorithm 1. Its execution starts with a procedure `Start`, which first initializes two thread pools—*pool_qs* for recursive quicksort tasks and *pool_pp* for parallel partitioning tasks—and then puts the initial quick sort task into *pool_qs*. This task is represented by the `QuickSortTask` procedure call where the entire input sequence *a* and the required total number of threads *T* are passed as arguments. Initialization of thread pools is performed by the `ThreadPoolInit` procedure. It creates a desired number of threads, which then start to wait for tasks being inserted into the thread pool queue. Once there is a task in this queue, some waiting thread is resumed, it fetches this task from the queue, and executes it. Procedure `QuickSortTask` recursively resolves the parallel quicksort itself. Until there is enough data for parallel processing, it first creates tasks for parallel partitioning of its input sequence by using the required number of threads. These tasks are represented by `PartitioningTask` calls and are put into the queue of *pool_pp*. Once partitioning is accomplished, both resulting left and right subsequences need to be processed. For the right one, a new task is created—with the corresponding `QuickSortTask` call—and enqueued into *pool_qs*. The left subsequence is then recursively processed in the current task. The thread counts for processing of both subsequences are split proportionally to their lengths. Finally, when there is not enough data to be processed in parallel, the sequential C++ in-place sorting function `std::sort` is called and the recursion is terminated.

4.2 | API

The application programming interface (API) of CPP11sort consists of a single header file `cpp11sort.h`. Inside, there are four sorting function templates called `sort` defined in the `cpp11sort` namespace. The most generic sorting function has the following signature:

```
template <typename RAIter, typename Comp>
void sort(RAIter first, RAIter last, Comp comp, unsigned int num\_threads);
```

The parameters `first` and `last` define the input sequence by the iterators to its first and past-the-last elements, respectively. These iterators must meet the requirements of the *random-access iterators* defined by the C++ Standard. The `comp` parameter represents a function object that compares two elements and returns `true` when the first element should be placed before the second one in the sorted sequence. It must meet the *strict-week-ordering* requirements defined by the C++ Standard. For instance, in the case of sorting with respect to the ascending order, `comp` represents the *less-than* relationship. Finally, the `num_threads` parameter allows user to control the number of threads used for sorting in parallel.

The additional three `sort` function template definitions just make some of these parameters having default values. The first variant:

```
template <typename RAIter, typename Comp>
void sort(RAIter first, RAIter last, Comp comp);
```

lets the number of threads to be derived automatically by the C++ implementation. Namely, it, internally, sets the number of threads to the result of the `std::thread::hardware_concurrency()` function call. The second variant:

```
template <typename RAIter>
void sort(RAIter first, RAIter last, unsigned int num_threads);
```

uses the *less-than* relationship for sorting in the form of an instance of the `std::less` class template, where the value type of `RAIter` is passed as a template argument. Finally, the third variant:

```
template <typename RAIter>
void sort(RAIter first, RAIter last);
```

combines the default settings for both the number of threads and the comparator function object.

Algorithm 1. CPP11sort algorithm pseudocode

Input: a: input data sequence to be sorted
Input: T: number of threads to be used
Input: S: sequential threshold
Input: B: partitioning block size
Input: P: number of pivot samples
Data: *pool_qs*: thread pool for parallel execution of quicksort tasks
Data: *pool_pp*: thread pool for parallel execution of partitioning tasks
`Start();`
Procedure `Start()`
  `ThreadPoolInit(`*pool_qs*`);`
  `ThreadPoolInit(`*pool_pp*`);`
  add initial task `QuickSortTask(`a, T`)` into *pool_qs*;
end
Procedure `ThreadPoolInit(`*pool*`)`
  create T new threads in *pool*;
  for *all created threads in parallel* **do**
    while *algorithm has not completed* **do**
      wait on the *pool* queue for a new task;
      try to fetch a task from the *pool* queue;
      if task has been fetched, execute it;
    end
  end
end
Procedure `QuickSortTask(`a', T'`)`
  if *size of* $a' < S$ **then**
    finish sorting by sequential `std::sort`;
    return
  end
  add T' tasks `PartitioningTask(`a'`)` into *pool_pp* to partition a' in parallel using T' threads;
  wait until the partitioning is finished;
  split a' into left and right subsequence a'_l and a'_r with respect to the resulting pivot position;
  split T' into T'_l and T'_r proportionally to lengths of a'_l and a'_r;
  add task `QuickSortTask(`a'_l, T'_l`)` into *pool_qs*;
  `QuickSortTask(`a'_r, T'_r`)`;
end
Procedure `PartitioningTask(`a'`)`
  perform parallel partitioning by all tasks added into *pool_pp* for partitioning of a' while using B for block size and P for number of pivot selection samples;
end

5 | EXPERIMENTAL EVALUATION

We have conducted an extensive experimental study to evaluate the performance and scalability of CPP11sort and to compare it with its existing competitors with respect to parallel in-place sorting. The measurements were performed on a dual-socket Intel Xeon 28-core CPUs server (56 cores in total) and a 10-core Intel Core i9 CPU workstation.

5.1 | Hardware and software setup

The CPU model on the server was Intel Xeon Platinum 8280 with the Cascade Lake microarchitecture. For development, we used GCC in version 8.1 with the system-installed versions of libstdc++ and Intel TBB. All the tested sorting implementations on this system allowed us to control the

number of threads used for parallel sorting and partitioning. Next to CPP11sort, we tested there the implementations provided by the libstdc++ parallel mode and Intel TBB. We did not evaluate our previously developed AQsort in this study since—due to its ability to work with a custom swap function—it is generally slightly slower than the implementations that does not support this feature. We also did not include the Intel PSTL C++17 solution since it just wraps the out-of-place sorting solution from Intel TBB.

On the workstation, the CPU model was Intel Core i9-10900X while the hyperthreading was enabled. The total number of virtual cores was therefore 20. For development, we used the Microsoft C++ compiler from the Microsoft Visual Studio 2019. On this system, we compared CPP11sort with the Microsoft's parallel version of `std::sort` algorithm mentioned in Section 3.

5.2 | Data sets

For evaluation of generic sorting algorithms, one needs to use data sets with different initial distribution/order of elements with respect to the order applied for their sorting. In our case, we used integer numbers (of type `int`) with the following initial distributions of the values of elements of input sequences:

- *random*—random numbers from the whole domain of the `int` data type;
- *repeated*—random numbers with only a few distinct values, namely, values from the interval [0,100);
- *globally partially sorted*—numbers split into chunks of 10,000 elements where each chunk had larger numbers than all the chunks at lower indexes, while the numbers within each chunk were random,
- *locally partially sorted*—random numbers split into chunks of 10,000 elements where the elements in each chunk were sorted,
- *sorted*—numbers ordered correspondingly with the required sorted order,
- *inversely sorted* numbers ordered inversely with respect to the required sorted order.

5.3 | Results

The main results of our experiments are scalability plots where the sorting time is measured as a function of the number of threads while the length of the input sequence is constant. These plots for all the input data distributions are shown in Figure 1. Therein, *libstdc++ PM QS* and *libstdc++ PM BQS* refer to the unbalanced and balanced parallel quicksort provided by the parallel mode of libstdc++; see Section 3 for details. *Intel TBB* refers to the parallel quicksort provided by the Intel TBB library, and *seq.* `std::sort` refers to the sequential `std::sort` function template provided by libstdc++ being part of the GCC development toolchain. It is obvious that CPP11sort provided comparable and mostly even shorter sorting times. The only exception was the sorted distribution where the runtime for Intel TBB was so short that it even did not fit into the scale of the y-axis of the corresponding plot. The reason is that Intel TBB first prechecks whether the input data are not already sorted, and, if they are, the sorting algorithm is not executed. The drawback of this approach is that it adds some runtime overhead for cases where this condition does not hold (we dare to say that sorting is mostly applied for data that are not already sorted in practice).

Next, we measured the sorting time as a function of growing input sequence length n with the utilization of all the hardware cores by setting the number of threads to 56. The results, again for all the initial data distributions, are shown in Figure 2. We may notice that sorting times are mostly almost proportional to n, which manifests high parallel sorting efficiency of all the implementations. CPP11sort slightly outperformed all its alternatives in this experiment as well, except for the presorted distribution of the input data, where Intel TBB benefited from its initial prechecking described above. In both presented experiments, we can also observe that all the implementations significantly reduced the sorting time when compared with the sequential sorting. For example, for $n = 10^9$ and 56 threads, the speedup of CPP11sort against `std::sort` was 44.2, 24.2, 38.2, 40.0, 24.8, and 16.1 with respect to the order of the defined data distributions.

On the workstation, we compared CPP11sort with `std::sort` provided by the Microsoft STL in both single-threaded and multithreaded variants. The C++ Standard Library from C++17 does not allow controlling the number of threads. Instead, it used 20 threads in our case, which was the number of hardware threads/virtual cores due to the enabled hyperthreading. Therefore, we used 20 threads as well for the CPP11sort measurements. The results are presented in Table 2. CPP11sort outperformed the parallel version of Microsoft's `std::sort` for all the data distributions. Its speedup with respect to the sequential sorting was between 6.08 and 14.54. In three cases, it was higher than the number of hardware cores, which implies that CPP11sort is able to benefit from hyperthreading.

In cases where CPP11sort outperformed its competitors, we have tried to determine the causes of its superiority. However, to our best effort, we succeeded only with the libstdc++ parallel mode implementation. The problem with such an analysis is that the Intel TBB and Microsoft STL implementations are not described in detail in the literature in the form of a presentation of the corresponding algorithms and the explanation of

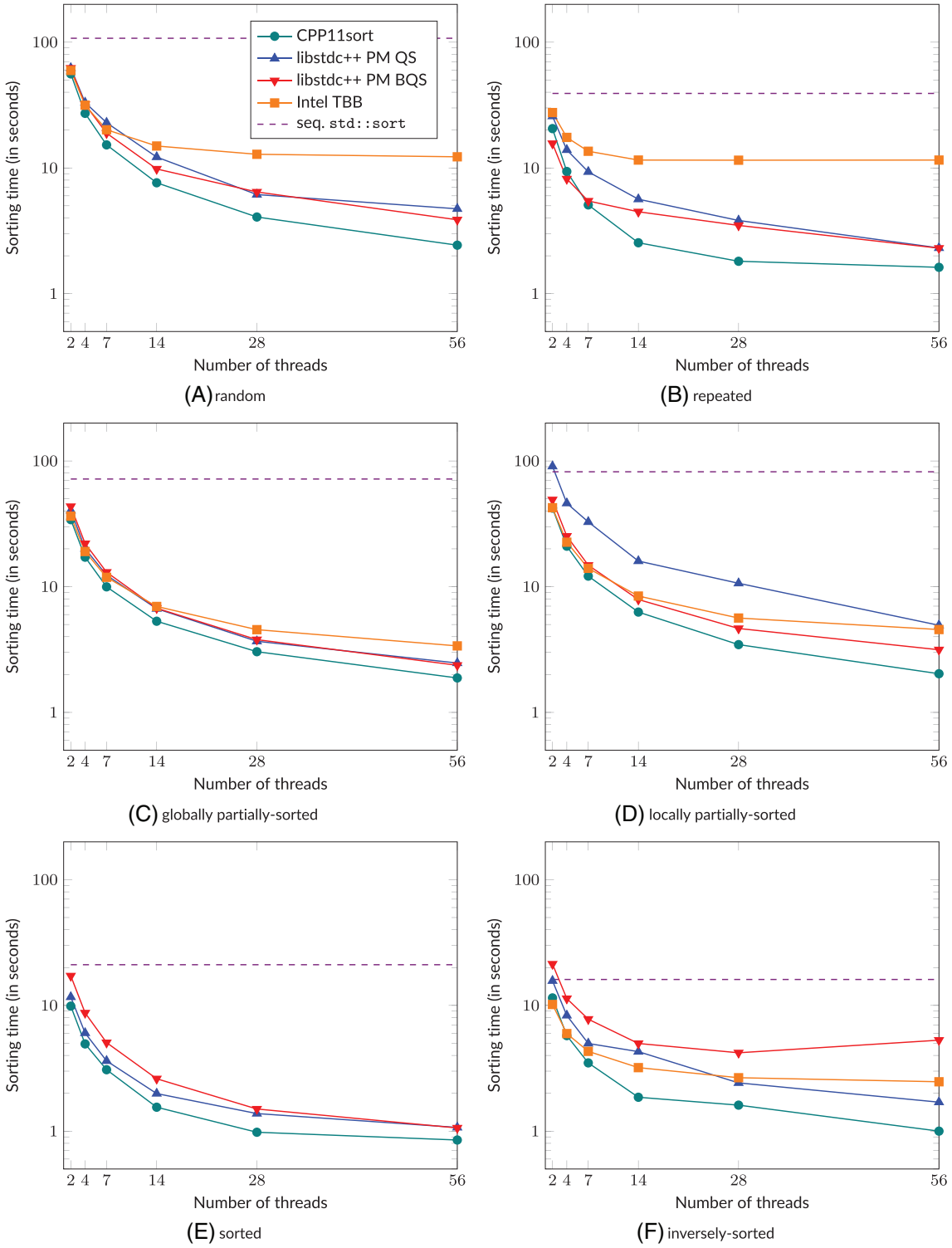

FIGURE 1 Sorting time as a function of the number of threads measured on dual 28-core Xeon server for $n = 10^9$

their implementations. Even the libstdc++ parallel mode implementation is not described to a detailed extent. However, its source code is relatively understandable, especially for ones who are familiar with OpenMP.

The parallel quicksort implementations from the libstdc++ parallel mode are based on nested parallelism. Initially, a desired number of threads is created in a top-level OpenMP parallel region. These threads are then used for the parallelization of the recursive quicksort calls. However, each time a parallel partitioning task is encountered, another parallel region is created, which involves the initiation of separate new threads. Creation of threads has some runtime overhead to which we attribute the superiority of CPP11sort. Recall that CPP11sort does not use nested parallelism.

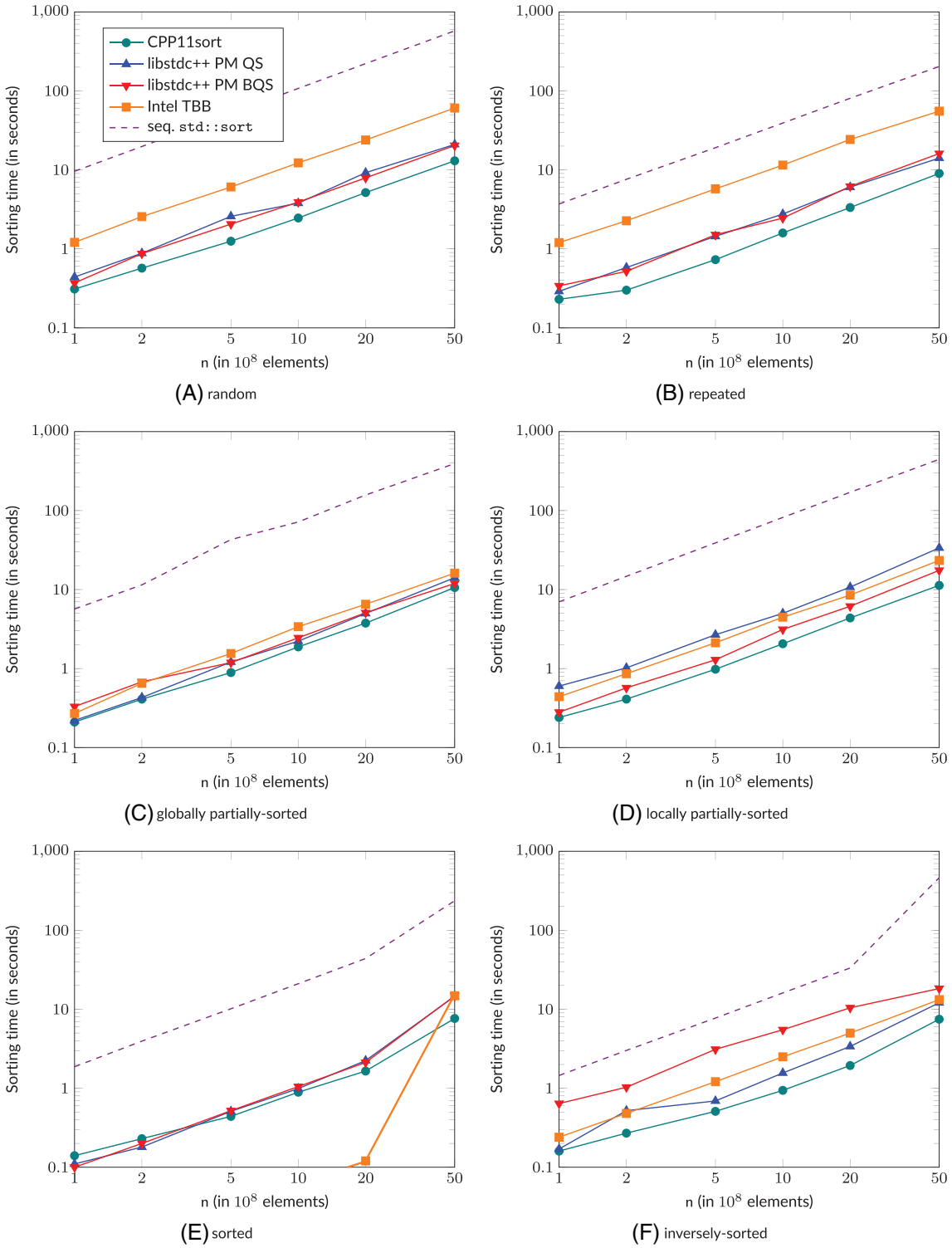

FIGURE 2 Sorting time as a function of the number of elements n measured on dual 28-core Xeon server for $T = 56$

TABLE 2 Sorting time measured on a 10-core Intel Core i9 workstation with enable hyperthreading for $n = 10^9$

Implementation	Random	Repeated	Glob. part. sorted	Loc. part. sorted	Sorted	Inv.-sorted
CPP11sort	7.00	3.50	5.35	5.98	1.82	1.98
par. `std::sort`	12.71	10.24	5.98	7.62	2.39	2.78
seq. `std::sort`	101.78	21.29	64.14	76.17	13.96	16.02

Instead, it uses two distinct thread pools which are initiated at the beginning of the algorithm run and no more threads are "nestedly" created for parallel partitioning tasks.

6 | CONCLUSIONS

The contribution of the presented work is a new implementation of the parallel multithreaded quicksort dubbed CPP11sort. It has three major advantages—it is fully portable to any C++ implementation conforming with the C++11 or newer standard, it does not require any additional third-party libraries or nonstandard language/compiler extensions, and it provides comparable or even better sorting performance than all the major current implementations, which are provided by GNU, Intel, and Microsoft. To our best effort, we have not found any other parallel in-place sorting alternative that would have all these characteristics at once.

The speedup achieved in the conducted experiments with respect to sequential sorting was between 16.1 and 44.2 on the dual-CPU Intel Xeon server with 56 cores in total, and between 6.8 and 14.5 on the Intel Core i9 workstation with 10 cores and enabled hyperthreading. Such speedups may considerably reduce sorting times for performance-aware applications that work with large data sets. The publicly available CPP11sort thus provides a generic in-place unstable parallel multithreaded sorting solution with the ability to control the number of used threads.

ACKNOWLEDGMENT

This research was supported by the Czech Ministry of Education, Youth and Sports under Grant Number CZ.02.1.01/0.0/0.0/16_019/0000765.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in the GitLab repository at https://gitlab.com/daniel.langr/cpp11sort, project ID 20395104.

ORCID

Daniel Langr https://orcid.org/0000-0001-9760-7068

REFERENCES

1. Langr D, Tvrdík P, Šimeček I. AQsort: scalable multi-array in-place sorting with OpenMP. *Scalable Comput Pract Exper*. 2016;21(3):369-391. https://doi.org/10.12694/scpe.v17i4.1207
2. ISO/IEC ISO/IEC 14882:2011: Information Technology — Programming languages — C++. International Organization for Standardization; 2011.
3. Cppreference.com C++ compiler support. Accessed September, 2020. https://en.cppreference.com/w/cpp/compiler_support
4. Hoare CAR. Algorithm 64: quicksort. *Commun ACM*. 1961;4(7):321-322. https://doi.org/10.1145/366622.366644
5. Hoare CAR. Quicksort. *Comput J*. 1962;5(1):10-16. https://doi.org/10.1093/comjnl/5.1.10
6. Goldstine HH, Neumann J. *Planning and Coding of Problems for an Electronic Computing Instrument Institute for Advanced Study*. Princeton; 1947.
7. Langr D, Šimeček I, Dytrych T. Block iterators for sparse matrices. Maria G, Leszek M, Marcin P, eds. *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2016)*. IEEE Computer Society, IEEE Xplore Digital Library; 2016:695-704.
8. Langr D, Šimeček I. On memory footprints of partitioned sparse matrices. Maria G, Leszek M, Marcin P, eds. *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2017)*. Vol 11. IEEE Computer Society, Polish Information Processing Society; 2017:513-512.
9. Langr D, Šimeček I. Analysis of memory footprints of sparse matrices partitioned into uniformly-sized blocks. *Scalable Comput Pract Exper*. 2018;19(3):275-291. https://doi.org/10.12694/scpe.v19i3.1358
10. ISO/IEC *ISO/IEC 14882:2017: Information Technology — Programming languages — C++*. International Organization for Standardization; 2011.
11. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms*. 3rd ed. The MIT Press; 2009.
12. Tsigas P, Zhang Y. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. Bob W, ed. *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*. IEEE; 2003:372-381.
13. Pasetto D, Akhriev A. A comparative study of parallel sort algorithms. Cristina VL, eds. *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. ACM; 2011:203-204.
14. Pasetto D, Akhriev A. A comparative study of parallel sort algorithms; 2011. Accessed July 17, 2015. http://researcher.watson.ibm.com/files/ie-albert_akhriev/sort2011-full.%pdf
15. Mahafzah BA. Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. *J Supercomput*. 2013;66(1):339-363. https://doi.org/10.1007/s11227-013-0910-2
16. Süs M, Leopold C. A user's experience with parallel sorting and OpenMP. Proceedings of the 6th European Workshop on OpenMP; 2004:23-28.
17. Bhan K. Parallel file quicksort. Accessed September 2020. https://github.com/kbuci/parallel-file-quicksort
18. Singler J, Konsik B. The GNU Libstdc++ parallel mode: software engineering considerations. Proceedings of the 1st International Workshop on Multicore Software Engineering; 2008:15-22; ACM, New York, NY.
19. Singler J, Sanders P, Putze F. *MCSTL: The Multi-core Standard Template Library*. Lecture Notes in Computer Science. Vol 4641. Springer; 2007:682-694.
20. Xiao S-y, Li C-l, Guo B-y, Xiao H. A radix sorting parallel algorithm suitable for graphic processing unit computing. *Concurr Comput Pract Exper*. 2021;33(6):e5818. https://doi.org/10.1002/cpe.5818
21. Huang X, Liu Z, Li J. Array sort: an adaptive sorting algorithm on multi-thread. *J Eng*. 2019;2019(5):3455-3459. https://doi.org/10.1049/joe.2018.5154
22. Al-Adwan A, Zaghloul R, Mahafzah BA, Sharieh A. Parallel quicksort algorithm on OTIS hyper hexa-cell optoelectronic architecture. *J Parallel Distrib Comput*. 2020;141:61-73. https://doi.org/10.1016/j.jpdc.2020.03.015

23. Baddar SW, Mahafzah BA. Bitonic sort on a chained-cubic tree interconnection network. *J Parallel Distrib Comput*. 2014;74(1):1744-1761. https://doi.org/10.1016/j.jpdc.2013.09.008
24. Klára S. *Paralelní řazení v C++11 [translated title: Parallel Sorting in C++11]*. Master's thesis. Czech Technical University in Prague, Prague, Czech Republic; 2019. [In Czech].
25. Langr D, Schovánková K. [dataset] CPP11sort. [project ID 20395104. Accessed July 2021]. https://gitlab.com/daniel.langr/cpp11sort

How to cite this article: Langr D, Schovánková K. CPP11sort: A parallel quicksort based on C++ threading. *Concurrency Computat Pract Exper*. 2022;34(4):e6606. https://doi.org/10.1002/cpe.6606