DOI 10.12694/scpe.v17i4.1207
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 369–391. http://www.scpe.org

AQSORT: SCALABLE MULTI-ARRAY IN-PLACE SORTING WITH OPENMP

DANIEL LANGR, PAVEL TVRDÍK AND IVAN ŠIMEČEK *

Abstract. A new multi-threaded variant of the quicksort algorithm called AQsort and its C++/OpenMP implementation are presented. AQsort operates in place and was primarily designed for high-performance computing (HPC) runtime environments. It can work with multiple arrays at once; such a functionality is frequently required in HPC and cannot be accomplished with standard C pointer-based or C++ iterator-based approach. An extensive study is provided that evaluates AQsort experimentally and compares its performance with modern multi-threaded implementations of in-place and out-of-place sorting algorithms based on OpenMP, Cilk Plus, and Intel TBB. The measurements were conducted on several leading-edge HPC architectures, namely Cray XE6 nodes with AMD Bulldozer CPUs, Cray XC40 nodes with Intel Hasswell CPUs, IBM BlueGene/Q nodes, and Intel Xeon Phi coprocessors. The obtained results show that AQsort provides good scalability and sorting performance generally comparable to its competitors. In particular cases, the performance of AQsort may be slightly lower, which is the price for its universality and ability to work with substantially larger amounts of data.

Key words: C++, high performance computing, in-place sorting, many-core, multi-array sorting, multi-core, multi-threaded algorithm, OpenMP, parallel partitioning, parallel sorting

AMS subject classifications. 68P10, 68W10

1. Introduction. The demand for sorting multiple arrays at once emerges not only in high-performance computing (HPC) codes. Such codes are mostly written in C/C++ and Fortran, however, common implementations of sorting algorithms in these languages do not support multi-array sorting. Pointer-based sorting routines—such as `qsort` from the C standard library [10, §7.20.5.2]—and their iterator-based generalizations—such as `std::sort` from the C++ standard library [11, §25.4.1.1]—can operate on a single array only. The Boost library [28] has introduced so-called zip iterators that can work with multiple arrays at once; however, Boost zip iterators are read-only iterators and therefore do not provide a solution for multi-array sorting. Generally, it is not feasible to create a portable standard-compliant implementation of zip iterators that can modify underlying arrays.[1]

The multi-array sorting problem has been frequently addressed by developers; numerous threads of C/C++ mailing lists and web forums have been devoted to this topic.[2] The suggested solution is practically always the same—to transform the *structure of arrays* (SoA) into a single *array of structures* (AoS), then perform sorting, and finally copy data back from AoS to SoA. Such a solution has the following drawbacks:

1. There must be enough free memory to perform the SoA-to-AoS transformation. Namely, the amount of free memory must be at least the same as the amount of memory occupied by the data that need to be sorted. Such a constraint might be inconvenient especially in HPC, where not the computational power but the amount of memory often limits the sizes of problems being solved. Users of HPC programs thus might in practice need to work with multi-array data structures that occupy more than the half of available memory, e.g., with representations of sparse matrices or meshes for spatial discretization of PDEs. Sorting/reordering of these data structures via SoA-to-AoS transformation would be not possible under such conditions.
2. The SoA-to-AoS and back AoS-to-SoA transformations imposes into programs a runtime overhead.

Sorting algorithms can be classified as being either in-place or out-of-place. We define an *out-of-place/not-in-place* sorting algorithm as an algorithm that requires $\Omega(n)$ extra space for auxiliary data structures, where n denotes the number of sorted elements; for a multi-array sorting problem, n denotes the number of sorted elements of each of the arrays. On the contrary, we define an *in-place* sorting algorithm as an algorithm that needs $o(n)$ extra space. Out-of-place sorting algorithms, such as mergesort, have typically the same amortized memory requirements as the SoA-to-AoS transformation. That is, the amount of free memory must be at

*Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 160 00, Praha, Czech Republic (langrd@fit.cvut.cz).

[1]The problem stems from the fact that it is not possible to create a pointer or a C++ reference to multiple data entities. For example, the following code is perfectly valid for an implementation of a sorting algorithm in C++: `auto& next_elem = *(iter + 1); next_elem = std::move(...);`. But in case of multiple arrays, there does not exist any entity that `next_elem` could reference. A detailed discussion about this problem is beyond the scope of this text.

[2]Look, e.g., for posts regarding multiple arrays/vectors sorting in C/C++ on the Stack Overflow community developer site (http://stackoverflow.com).

least the same as is occupied by the sorted data. Within this article, we primarily address multi-array sorting problems where such an amount of free memory is not available. Therefore, for the solution of these problems, neither the SoA-to-AoS transformation nor the out-of-place sorting can be used.

Growing capacities of shared memories of computer hardware architectures and growing numbers of their computational cores have raised the demand for parallel/multi-threaded algorithms; we do not consider distributed-memory parallelism in this text, therefore, we use the terms *parallel* and *multi-threaded* as synonyms. In HPC, the utilization of shared memories via hybrid parallel paradigms, such as the combination of MPI and OpenMP, often outperforms distributed-memory implementations (pure MPI). However, the amount of data that fit shared memories might become so large that their sequential sorting would introduce into applications significant hotspots. For illustration, sorting of an array of 6.4 billions of integers, which fit 64 GB of memory, with sequential `std::sort` took in our measurements over 18 minutes on a contemporary CPU-based system. Using parallel in-place sorting algorithms, we were able to reduce the sorting time to less than 2.5 minutes (speedup 7.5) utilizing all 16 system cores. Similarly, we observed the reduction of runtime from 12 minutes to less than 24 seconds (speedup over 30) when sorting 1.6 billions of integers on an Intel Xeon Phi many-core coprocessor.

Available parallel C/C++ implementations of sorting algorithms usually adopt the pointer-/iterator-based approach of their sequential counterparts. Consequently, they are not able to work with multiple arrays at once. In this article, we present a new parallel variant of the in-place quicksort algorithm called AQsort that does not impose such a constraint. Instead of pointers/iterators, AQsort works with user-provided routines for comparing and swapping sorted data. The drawback of this approach is that it hinders some possibilities to optimize and tune resulting programs by developers and compilers (see Sect. 6 for details). One of the purpose of the presented experimental study is therefore to evaluate the effects of such a restriction.

The structure of this article is as follows. Section 2 introduces the state-of-the-art implementations of parallel sorting algorithms. Sections 3 and 4 describe the AQsort algorithm itself and its C++/OpenMP implementation, respectively. Section 5 presents a study that evaluates AQsort experimentally on modern HPC architectures and compares its performance with its not-multi-array competitors. Section 6 discusses the problem of choosing the most suitable sorting solution according to user's needs and constraints. Finally, Section 7 summarizes our work and concludes the article.

1.1. Motivation. Our research is tightly connected to sparse-matrix computations, mainly to sparse matrix storage formats. These formats determine the way how matrix nonzero elements are stored in computer memory. The simplest format is so-called *coordinate storage format* (COO) [1, 27] that consists of three arrays containing row indexes, column indexes, and values of matrix nonzero elements. COO does not prescribe any order of the elements in these arrays and it is the most suitable format for assembling sparse matrices—generated nonzero elements are simply to the arrays appended. However, COO has high memory requirements; it is therefore not a suitable format for sparse-matrix computations, which are generally bound in performance by memory bandwidth [14, 35].

In practice, likely the most commonly-used format for sparse matrix computations is the *compressed sparse row format* (CSR, CRS), together with its *compressed sparse column* (CSC, CCS) counterpart [1, 27]. The conversion of sparse matrices from COO to CSR consists of two steps: First, the nonzero elements in COO arrays are sorted lexicographically, which represents a multi-array sorting problem. Then, the array of row indexes is substituted by an (hopefully much smaller) array that indicates how many nonzero elements are in each row and where their column indexes and values can be found.

Numerous other formats have been developed in the past that were shown to provide high performance of sparse matrix computations on modern multi-core and many-core architectures, typically in comparison with CSR. Generally, these formats have two common features: (1) they store matrix nonzero elements in memory in some particular order, and (2) they are more or less parametrized. Finding (pseudo)optimal parameters for a given matrix and transforming this matrix into a given format typically involves multiple sorting of the COO arrays. For example, uniformly-blocking formats are parametrized by the block size [13]. To find an optimal block size, matrix nonzero elements need to be sorted repeatedly with respect to different tested block sizes [12, 29].

To amortize the usage of a given format in subsequent matrix-related computations, we thus need a fast scalable sorting algorithm. To allow users to work in their HPC programs with large matrices that occupy more than a half of available memory and thus effectively allow them to solve correspondingly large computational

problems, we need this algorithm to be in-place. Finally, to allow developers to integrate such an algorithm into their codes, we need its efficient portable OpenMP implementation, since the OpenMP threading paradigm [5] prevails in sparse matrix-related and HPC codes in general. All of these requirements AQsort fulfils.

2. Related Work. Many C/C++ implementations of parallel sorting algorithms have been developed in the past. To our best knowledge, we have not found any one capable of generic multi-array in-place sorting. To evaluate AQsort, we therefore compared its performance with the performance of iterator-based sorting functions from forefront libraries provided by GNU, Nvidia, and Intel.

GNU implements parallel sorting algorithms in the scope of its parallel version of the C++ Standard Library, namely the GNU Libstdc++ Parallel Mode [30]. It provides functions for both in-place and out-of-place sorting; the former implements the parallel quicksort algorithm proposed by Tsigas and Zhang [34], the latter implements a parallel multi-way mergesort. Moreover, the parallel quicksort exists in the library in a balanced and an unbalanced variant, which differ in the way of assigning threads to partitions of unequal sizes. The library uses the OpenMP threading paradigm and was originally developed as a standalone software project called the Multi-Core Standard Template Library (MCSTL) [31]. At the time of writing this article, the GNU Libstdc++ Parallel Mode was referred to as "an experimental parallel implementation of many C++ Standard Library algorithms".

Nvidia provides parallel sorting functions in the scope of its library called Thrust. The documentation does not discuss the implementation in detail, but according to the source code, the functions seemingly implement an out-of-place parallel mergesort. Thrust was primarily designed for GPGPU programming, but it also supports OpenMP as an underlying threading paradigm [3].

Intel provides parallel sorting functions for their own threading paradigms/frameworks Cilk Plus [26] and Thread Building Blocks (TBB) [24]. Cilkpub—a library of community-contributed Cilk Plus code—contains functions that implement both a parallel in-place quicksort and a parallel out-of-place samplesort. Intel TBB contains an implementation of an in-place parallel quicksort. Some details of these functions are provided by McCool et al. [18].

We have chosen the above mentioned (iterator-based) implementations since, thanks to their providers, we presumed highly efficient and optimized codes targeting modern hardware architectures. However, note that there are numerous other parallel implementations of sorting algorithms as well, available either in the form of standalone codes or within some more generic libraries/frameworks. These include, e.g., Intel PSS [25], OMPTL [2], Parallel STL [19], ParallelSort [21], psort [16], STAPL [33], and STXXL [7].

2.1. Parallel Quicksort. Current implementations of generic in-place sorting algorithms are mostly based on quicksort [8,9]. Quicksort partitions sorted data according to a so-called pivot element and then recursively calls itself for both resulting parts. In multi-threaded environments, the recursive calls are natural candidates for task parallelism, which is available, e.g., in OpenMP since version 3.0, Intel TBB, and Intel Cilk Plus. However, partitioning needs to be parallelized as well; sequential partitioning at top levels of recursion would significantly hinder the scalability of parallel quicksort.

Tsigas and Zhang [34] proposed a parallel quicksort with efficient parallel partitioning that is widely used in practice and frequently mentioned in literature. Alternative solutions and their comparison has been presented by Pasetto and Akhriev [22,23]. Within they work, they also proposed a new approach to parallel partitioning, however, they defined it only by words and did not provide a corresponding algorithm [23, Section 2.4].

Another parallel quicksort have been proposed by Mahafzah [15], however, he does not provide experimental comparison with other solutions. Moreover, he presents results only for small data up to 80 MB of memory footprint and small number of threads up to 8. Süs and Leopold [32] compared several parallel implementations of quicksort based on OpenMP and POSIX threads (Pthreads).

In practice, efficient implementations frequently combine quicksort with other sorting algorithms. One reason is the quicksort's worst-case complexity $O(n^2)$. To deal with the worst cases, such implementations allow the recursive process happen only to some maximum depth, commonly set to $\lfloor 2 \log_2 n \rfloor$. If it is exceeded, the quicksort is abandoned and the rest of the not-yet sorted data is processed by another sorting algorithm with the $O(n \log n)$ worst-case complexity, typically by heapsort [6]; such a combination of quicksort and heapsort is referred to as *introspective sort* or shortly *introsort* [20]. Another reason for combining quicksort with other algorithms is that applying quicksort to very small partitions might be inefficient; recursive calls are relatively expensive here. When partitions of sizes below some cutoff parameter are reached, they are thus usually sorted with some simple $O(n^2)$ algorithm, typically with insertion sort [6].

3. Algorithm Design. Available iterator-based C++ sorting functions—both sequential and parallel—typically adhere to the following declaration pattern:

```
template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

The `first` and `last` parameters represent random-access iterators that determine data to be sorted. The `comp` binary function decides whether its first argument should appear before the second in the sorted data. Unfortunately, there is no portable way how to construct writable random-access iterators for multiple arrays. To solve the in-place multi-array sorting problem, we thus need to give up the iterator-based approach.

Let us introduce some terminology used in the text below. Up to now, we have spoken about multi-array sorting, i.e., sorting of multiple arrays at once. However, by the term *array*, we generally mean any sequence of data, i.e., any data structure that contains multiple elements accessible via indexes; in C++, such data structures can be represented, e.g., by the `std::vector` and `std::array` STL containers, or ordinary C-style one-dimensional arrays. We assume that all the arrays have n elements indexed from 0 to $n-1$. We refer to the elements with the same index in all these arrays as to a *multielement*.

In a multi-array sorting problem, we thus want to sort n multielements according to some order given by *sorting keys* that can be derived from multielements data. We say that a multielement M_i is *less than* another multielement M_j if M_i should take place before M_j in the sorted arrays, according to their sorting keys. Otherwise, we say that M_i is *greater than or equal to* M_j.

Let us consider reordering of sparse matrix nonzero elements; a multielement is then represented by a single matrix nonzero element, i.e., by its row index, column index, and value. If we want to sort matrix nonzero elements, e.g., lexicographically (such as for conversion to CSR), we might define sorting keys for the ith multielement as

```
std::size_t key(std::size_t i) { return rows[i] * n_cols + cols[i]; }
```

where `n_cols` equals the number of matrix columns and the arrays `rows`/`cols` contain row/column indexes of matrix nonzero elements.

The AQsort implementation declares sorting functions as follows:

```
template <typename Comp, typename Swap>
void sort(std::size_t length, Comp* const comp, Swap* const swap);
```

The parameter `length` represents the number of multielements that need to be sorted, i.e., n. The parameter `comp` is a pointer to a binary function that takes two arguments and returns `true` if the multielement indexed by the first argument is less than the multielement indexed by the second argument; otherwise, it returns `false`. Finally, the parameter `swap` is a pointer to a function that takes two arguments and swaps multielements indexed by these arguments.

Thanks to this approach, AQsort has no information about data being sorted. The algorithm does not know how many arrays are sorted at once, it cannot recognize the types of the elements in these arrays, and there is no way how it could access these elements. It also does not work with sorting keys, only indirectly through the `comp` function. This function needs to derive sorting keys for both indexed multielements, compare them, and return the appropriate binary value. It is up to AQsort users to provide the expected functionality of `comp` and `swap` over the sorted arrays.

Let us now present the AQsort algorithm itself. Parallel quicksort is carried out by the `ParallelQuickSort` procedure presented by Algorithm 1. Its input consists of the following parameters:

1. n: total number of sorted multielements,
2. *numthreads*: total number of threads,
3. *start*: index of the first multielement to be sorted by this call,
4. *count*: number of multielements to be sorted by this call,
5. `Comp`: reference to a binary function that compares multielements with two different indexes and returns true if the first multielement is less than the second one; otherwise returns false,
6. `Swap`: reference to a procedure that swaps multielements with two different indexes,
7. *level*: auxiliary parameter for preventing quicksort worst case complexity $O(n^2)$.

Initially, `ParallelQuickSort` is called from a single thread with the following arguments:

$$start = 0, \quad count = n, \quad level = 2 \cdot \lfloor \log_2(n) \rfloor, \tag{3.1}$$

Algorithm 1 Main recursive procedure of AQsort

```
1: procedure ParallelQuickSort(n, numthreads, start, count, Comp, Swap, level)
2:     while true do
3:         if level = 0 then
4:             SequentialSort(start, count, Comp, Swap, level)
5:             return
6:         end if
7:         level ← level − 1
8:         T ← ⌊numthreads × count/n⌋
9:     end while
10:    if T < 2 then
11:        SequentialSort(start, count, Comp, Swap, level)
12:        return
13:    end if
14:    pivot ← SelectPivotMoM(start, count, Comp)
15:    Swap(pivot, start + count − 1)
16:    pivot ← start + count − 1
17:    lessthan ← ParallelPartition(start, count, pivot, Comp, Swap, T)
18:    Swap(start+ lessthan, pivot)
19:    greaterthan ← count − lessthan − 1
20:    while greaterthan > 0 and Comp(start + lessthan, start + count − greaterthan) = false and Comp(start +
       count − greaterthan, start + lessthan) = false do
21:        greaterthan ← greaterthan − 1
22:    end while
23:    if lessthan > greaterthan then
24:        run ParallelQuickSort(n, numthreads, start + count − greaterthan, greaterthan, Comp, Swap, level)
       in a new parallel task
25:        count ← lessthan
26:    else
27:        run ParallelQuickSort(n, numthreads, start, lessthan, Comp, Swap, level) in a new parallel task
28:        start ← start + count − greaterthan
29:        count ← greaterthan
30:    end if
31: end procedure
```

and *numthreads* set to the number of threads that a user wants to use for parallel sorting.

The `ParallelQuickSort` procedure works as follows:

1. If the maximum allowable depth of recursion is reached, `SequentialSort` is called, which further immediately proceeds to heapsort.
2. The number of threads that proportionally falls on the number of multielements processed by this call (*count*) is calculated and stored in T; the total number of threads as well as the total number of multielements therefore need to be passed to `ParallelQuickSort` as arguments. When T drops below 2, the processed multielements are sorted sequentially by calling the `SequentialSort` procedure.
3. Otherwise, a pivot multielement is selected and the processed multielements are partitioned with respect to this pivot using T threads by calling the `ParallelPartition` function.
4. In `ParallelQuickSort`, the pivot is chosen using the *median of medians* (MoM) strategy referred also to as *ninther*, which is usually recommended for large arrays; see, e.g., [4]. The MoM pivot selection is represented by the `SelectPivotMoM` function.
5. After parallel partitioning, multielements that lays behind the pivot and are equal to it are excluded from further processing, since they are already at their final positions. This might considerably improve the algorithm efficiency in cases when sorted arrays contain only few unique sorting keys.
6. So-called *tail call elimination* is applied to reduce the required call stack space (see, e.g., [6, Sect. 7-4]). `ParallelQuickSort` thus recursively calls itself only once instead of twice, and uses a while loop to

Algorithm 2 Parallel partitioning—Part 1

```
1:  function ParallelPartition(start, count, pivot, Comp, Swap, T)
2:      m ← ⌊count/PBS⌋
3:      tleft[] ← integer array of size T
4:      tstart[] ← integer array of size T + 1
5:      for t = 0 to T − 1 do
6:          tstart[t] ← start + PBS × ⌊t × m/T⌋
7:      end for
8:      tstart[T] ← start + PBS × m
9:      for T threads do in parallel
10:         t ← actual thread number
11:         left ← tstart[t]
12:         right ← tstart[t + 1] − 1
13:         tleft[t] ← left+ SequentialPartition(left, right − left + 1, pivot, Comp, Swap)
14:         perform parallel barrier synchronization
15:     end for
16:     i ← 0
17:     j ← T − 1
18:     while i < j do
19:         imod ← (tleft[i] − start) mod PBS
20:         jmod ← (tleft[j] − start) mod PBS
21:         if imod = 0 then
22:             i ← i + 1
23:             continue
24:         end if
25:         if jmod = 0 then
26:             j ← j − 1
27:             continue
28:         end if
29:         ilast ← tleft[i] − imod + PBS − 1
30:         jfirst ← tleft[j] − jmod
31:         while tleft[i] ≤ ilast and tleft[j] − 1 ≥ jfirst do
32:             Swap(tleft[i], tleft[j] − 1)
33:             tleft[i] ← tleft[i] + 1
34:             tleft[j] ← tleft[j] − 1
35:         end while
36:     end while
```

process the second partition. The recursive call is always performed for the smaller partition.

The key for the good scalability of AQsort is efficient parallel partitioning at the high levels of the quicksort's recursive process accomplished by the `ParallelPartition` function. Its functionality stems from the solution described by Pasetto and Akhriev that was introduced in Section 2.1. We elaborated their concept into an fully-defined efficient blocking-based algorithm that is introduced by Algorithm 1.

The `ParallelPartition` function takes the following arguments:

1. *start*: index of the first multielement to be partitioned,
2. *count*: number of multielements to be partitioned,
3. *pivot*: index of a pivot multielement,
4. `Comp`, `Swap`: see Algorithm 1,
5. T: number of threads to be used for partitioning,

and returns the number of multielements that are less than the pivot. The functionality of `ParallelPartition` is as follows: It first splits the processed multielements into T parts of the same size, where T is the number of threads that are required to participate in parallel partitioning (lines 2–8). Each part is then independently partitioned by a single thread with respect to the pivot by calling the `SequentialPartition` function (lines 9–

Algorithm 3 Parallel partitioning—Part 2

```
37:   lessthan ← 0
38:   for k ← 0 to T − 1 do
39:       lessthan ← lessthan + tleft[k] − tstart[k]
40:   end for
41:   temp ← (tleft[i] − start) mod PBS
42:   if temp ≠ 0 and (start + lessthan < tleft[i] − temp or start + lessthan ≥ tleft[i] − temp + PBS) then
43:       if Comp(start + lessthan, pivot) then
44:           q ← PBS − temp
45:           while q > 0 do
46:               Swap(tleft[i], start + lessthan + q − 1)
47:               tleft[i] ← tleft[i] + 1
48:               q ← q − 1
49:           end while
50:       else
51:           q ← temp
52:           while q > 0 do
53:               Swap(tleft[i] − 1, start + lessthan − q)
54:               tleft[i] ← tleft[i] − 1
55:               q ← q − 1
56:           end while
57:       end if
58:   end if
59:   lthread ← 0
60:   rthread ← T − 1
61:   gleft ← tleft[0]
62:   gright ← tleft[T − 1]
```

15), which represents a standard sequential partitioning algorithm. For efficiency, multielements are processed in blocks, and the size of blocks represents a global AQsort parameter called *PBS*; such a blocking approach is inevitable for multi-core and many-core environments to avoid cache contention. The size of split parts is chosen to be an exact multiple of the block size.

We further distinguish 3 different types of blocks. A block is called *black* if all its multielements are less than the pivot. A block is called *white* if all its multielements are greater than or equal to the pivot. A block is called *grey* if it contains multielements of both types and the multielements less than the pivot are placed on its left side, thus have lower indexes, than the multielements greater than or equal to the pivot. After performing `SequentialPartition` by each thread, the corresponding part of multielements contains at most one grey block; the other blocks are either black or white. The next step is to "neutralize" these at most T grey blocks by swapping their multielements such that as a result, either only one or no grey block exists (lines 16–36). If it does, it is placed to its final position, which is already known (lines 37–58). The neutralization of grey blocks is performed sequentially; this step is very fast and there would be only little or no benefit from its parallelization.

After neutralization of the grey blocks—up to at most the single one—all black and white blocks are swapped in parallel so that all black blocks are finally placed left from white blocks (lines 59–105). Thought the most of the pseudocode of this step consists of a critical section, the most runtime is spent outside of it (lines 100–102).

In the end, `ParallelPartition` needs to process multielements that did not fit the blocking scheme. This consists of placing the remaining not-yet-processed multielements that are less than the pivot to the left side of the position where the pivot finally belongs (lines 106–111). This last step is performed sequentially (again, it is very fast and there would be only little or no benefit from its parallelization).

Now, it remains to show how sequential sorting is performed. AQsort uses the combination of quicksort, insertion sort, and heapsort. Heapsort is conditionally applied to prevent quicksort's worst case complexity $O(n^2)$. Insertion sort is applied to partitions smaller than a threshold given by an AQsort global parameter called *IST*. The `SequentialSort` procedure therefore first checks the number of multielements to be sorted. If it is greater than *IST*, it calls the `SequentialQuickSort` procedure, which performs a standard recursive

Algorithm 4 Parallel partitioning—Part 3

```
63:  for T threads do in parallel
64:      done ← false
65:      while true do
66:          enter critical section
67:              if gleft ≥ gright or gleft ≥ start + lessthan or gleft − PBS < start + lessthan then
68:                  done ← true
69:                  break
70:              end if
71:              while gleft ≥ tstart[lthread + 1] do
72:                  lthread ← lthread + 1
73:                  if lthread ≥ T then
74:                      done ← true
75:                      break
76:                  end if
77:                  gleft ← tleft[lthread]
78:              end while
79:              myleft ← gleft
80:              gleft ← gleft + PBS
81:              while gright > PBS and gright − PBS < tstart[rthread] do
82:                  if rthread = 0 then
83:                      done ← true
84:                      break
85:                  end if
86:                  rthread ← rthread − 1
87:                  gright ← tleft[rthread]
88:              end while
89:              if gright ≤ PBS then
90:                  done ← true
91:                  break
92:              end if
93:              myright ← gright − PBS
94:              gright ← gright − PBS
95:          exit critical section
96:          if done = true then
97:              break
98:          end if
99:          if myleft < myright then
100:             for k ← 0 to PBS − 1 do
101:                 Swap(myleft + k, myright + k)
102:             end for
103:         end if
104:     end while
105: end for
```

quicksort. Then, insertion sort is executed to finally sort multielements not sorted by quicksort.

The `SequentialQuickSort` procedure works as follows:

1. If the maximum allowable depth of recursion is reached, the quicksort is abandoned and the processed multielements are sorted with the `HeapSort` procedure.
2. Otherwise, a pivot multielement is selected and the processed multielements are partitioned with respect to this pivot by calling the `SequentialPartition` function.
3. The pivot is chosen using so-called *median of three* (Mo3) strategy, which is faster than MoM.
4. As in `ParallelQuickSort`, after partitioning, multielements that are equal to the pivot are excluded

Algorithm 5 Parallel partitioning—Part 4

```
106:     for k ← tstart[T] to start + count do
107:         if Comp(k, pivot) then
108:             Swap(k, start + lessthan)
109:             lessthan ← lessthan + 1
110:         end if
111:     end for
112:     return lessthan
113: end function
```

from further processing.

5. As in `ParallelQuickSort`, the *tail call elimination* is applied to reduce the required call stack space.
6. Quicksort is performed only for partitions with sizes greater than IST. Smaller partitions are left to be sorted by insertion sort.

4. Implementation. We provide an AQsort implementation in the form of an open-source GitHub project[3]. It is written in C++ and uses OpenMP as a threading paradigm; OpenMP version 3.0 or higher is required because of task parallelism. Its usage consists of:

1. including the main AQsort header file `aqsort.h`,
2. defining functions for comparing and swapping multielements,
3. calling one of the provided sorting functions.

The contents of `aqsort.h` is shown in Figure 4.1.

There are three sorting functions defined, all within the `aqsort` namespace. The first function called `parallel_sort` performs sorting in parallel by using OpenMP. The second function called `sequential_sort` performs sequential sorting. The third function called `sort` is a simple "switch" that proceeds to `parallel_sort` if OpenMP is available and to `sequential_sort` otherwise. Note that `parallel_sort` is accessible only if the `_OPENMP` preprocessor macro is defined. This macro is typically provided automatically by compilers in case that OpenMP threading is activated (as, e.g., when the `-fopenmp` command line argument is passed to the GNU C/C++ compilers).

The `sequential_sort` function can be employed in codes where OpenMP threading is not available, such as sequential programs. In HPC, these may also be pure MPI-based programs; their MPI processes are typically mapped to all available system cores and, consequently, there is no room for shared-memory parallelism.

Both `parallel_sort` and `sequential_sort` functions are only simple wrappers. The implementation of sorting algorithms defined in Section 3 is hidden in the `aqsort::impl` namespace and the corresponding codes are stored in the `impl` subdirectory. There is no need for building and linking the AQsort library; its functionality is provided purely in the form of C++ header files.

In Section 3, we introduced two AQsort global algorithm parameters, namely PBS and IST. These parameters are represented within the implementation by preprocessor macros called `AQSORT_PARALLEL_PARTITION_BLOCK_SIZE` and `AQSORT_INSERTION_SORT_THRESHOLD`, respectively. If users want to change their default values for their codes, they need to define these macros before the `aqsort.h` header file is included.

Let us now show how to define comparison and swapping functions for AQsort. Primarily, we can define them either as *function objects*, or as *lambda functions*, which were introduced by the C++11 standard. Both options are illustrated by Figures 4.2 and 4.3, respectively; note that the latter is considerably clearer and more concise. These example codes show how to sort nonzero elements of a sparse matrix A, where:

$$A = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 5 & 0 & 0 & 6 \end{bmatrix} \quad \text{and} \quad n = 6. \tag{4.1}$$

Note that it would have no sense to perform such sorting in parallel; multi-threaded functionality of AQsort was designed for very large arrays. The reasonable condition for efficient parallel sorting with AQsort is

$$n \gg PBS \times numthreads, \tag{4.2}$$

[3] `https://github.com/DanielLangr/AQsort`

```
#ifndef AQSORT_INSERTION_SORT_THRESHOLD
   #define AQSORT_INSERTION_SORT_THRESHOLD 16
#endif

#include "impl/sequential_sort.h"

#ifdef _OPENMP
   #ifndef AQSORT_PARALLEL_PARTITION_BLOCK_SIZE
      #define AQSORT_PARALLEL_PARTITION_BLOCK_SIZE 1024
   #endif

   #include "impl/parallel_sort.h"
#endif

namespace aqsort
{
#ifdef _OPENMP
   template <typename Comp, typename Swap>
   inline void parallel_sort(std::size_t length, Comp* const comp, Swap* const swap)
   {
      impl::parallel_sort(length, comp, swap);
   }
#endif

   template <typename Comp, typename Swap>
   inline void sequential_sort(std::size_t length,
                               Comp* const comp, Swap* const swap)
   {
      impl::sequential_sort(length, comp, swap);
   }

   template <typename Comp, typename Swap>
   inline void sort(std::size_t length, Comp* const comp, Swap* const swap)
   {
#ifdef _OPENMP
      parallel_sort(length, comp, swap);
#else
      sequential_sort(length, comp, swap);
#endif
   }
}
```

Fig. 4.1: Contents of the AQsort main header file `aqsort.h`. Comments and some unimportant preprocessor directives are omitted.

where *numthreads* denotes the number of utilized OpenMP threads.

5. Experiments. We have conducted an extensive experimental study to evaluate the performance and scalability of AQsort and to compare it with existing forefront implementations of parallel sorting algorithms. Measurements were run on modern HPC hardware architectures, namely 3 types of multi-core computational nodes of large-scale HPC systems and a many-core Intel Xeon Phi coprocessor/accelerator; see Table 5.1 for details.

The utilized Intel Xeon processors of the Cray XC40 system supported hyper-threading, i.e., running two threads per a single physical core. However, within preliminary test measurements, hyper-threading brought

[4]Formerly known as Hornet (the system was renamed/upgraded during our work).
[5]High Performance Computing Center Stuttgart, University of Stuttgart, Stuttgart, Germany.
[6]National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign, USA.
[7]Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic.
[8]Jülich Supercomputing Centre, Institute for Advanced Simulation, Jülich, Germany.
[9]Non-uniform memory access.
[10]Symmetric multiprocessing.

```
#include <algorithm>
#include <vector>

#include <aqsort.h>

template <typename T> struct Comp {
   Comp(std::vector<T>& rows, std::vector<T>& cols) : rows_(rows), cols_(cols) { }

   inline bool operator()(std::size_t i, std::size_t j) const {
      // lexicographical ordering:
      if (rows_[i] < rows_[j]) return true;
      if ((rows_[i] == rows_[j]) && (cols_[i] < cols_[j])) return true;
      return false;
   }

   /* private: */ std::vector<T> &rows_, &cols_;
};

template <typename T, typename U> struct Swap {
   Swap(std::vector<T>& rows, std::vector<T>& cols, std::vector<U>& vals)
      : rows_(rows), cols_(cols), vals_(vals) { }

   inline void operator()(std::size_t i, std::size_t j) {
      std::swap(rows_[i], rows_[j]);
      std::swap(cols_[i], cols_[j]);
      std::swap(vals_[i], vals_[j]);
   }

   /* private: */ std::vector<T> &rows_, &cols_;
   /* private: */ std::vector<U> &vals_;
};

int main() {
   typedef unsigned int uint32_t;

   // sparse matrix in COO
   std::vector<uint32_t> rows, cols;
   std::vector<double> vals;

   // matrix assembly in reverese lexicographical order
   rows.push_back(0); cols.push_back(0); vals.push_back(1.0);
   rows.push_back(3); cols.push_back(0); vals.push_back(5.0);
   rows.push_back(1); cols.push_back(1); vals.push_back(3.0);
   rows.push_back(2); cols.push_back(2); vals.push_back(4.0);
   rows.push_back(0); cols.push_back(3); vals.push_back(2.0);
   rows.push_back(3); cols.push_back(3); vals.push_back(6.0);

   // sorting in lexicographical order:
   Comp<uint32_t> comp(rows, cols);
   Swap<uint32_t, double> swap(rows, cols, vals);

   aqsort::sort(rows.size(), &comp, &swap);
   // matrix elements are now sorted lexicographically
}
```

Fig. 4.2: Custom reordering of sparse matrix nonzero elements with AQsort using function objects.

no speedup in comparison with a default single-thread-per-core configuration. The AMD Opteron processors of the Cray XE6 system did not support hyper-threading. Therefore, within the presented study, we did not use more than a single thread per core on these architectures. On the contrary, on IBM BlueGene/Q (BG/Q) nodes and Intel Xeon Phi coprocessors, we usually achieved the highest sorting performance when running multiple threads per a single core. Both IBM BG/Q and Intel Xeon Phi support up to 4 simultaneously running threads

```cpp
#include <cstdint>
#include <utility>
#include <vector>

#include <aqsort.h>

int main() {
   // sparse matrix in COO, matrix elements in reverese lexicographical order
   std::vector<uint32_t> rows { 0, 3, 1, 2, 0, 3 };
   std::vector<uint32_t> cols { 0, 0, 1, 2, 3, 3 };
   std::vector<double> vals { 1.0, 5.0, 3.0, 4.0, 2.0, 6.0 };

   // sorting in lexicographical order:
   auto comp = [&rows, &cols] (std::size_t i, std::size_t j) /* -> bool */ {
      if (rows[i] < rows[j]) return true;
      if ((rows[i] == rows[j]) && (cols[i] < cols[j])) return true;
      return false;
   };

   auto swap = [&rows, &cols, &vals] (std::size_t i, std::size_t j) {
      std::swap(rows[i], rows[j]);
      std::swap(cols[i], cols[j]);
      std::swap(vals[i], vals[j]);
   };

   aqsort::sort(rows.size(), &comp, &swap);
   // matrix elements are now sorted lexicographically
}
```

Fig. 4.3: Custom reordering of sparse matrix nonzero elements with AQsort in C++11 using lambda functions.

Table 5.1: Configurations of HPC systems and their run-time environments used for experiments.

Architecture:	Cray XC40	Cray XE6	Intel Xeon Phi	IBM BlueGene/Q
System:	Hazel Hen[4]	Blue Waters	Star	Juqueen
Provider:	HLRS[5]	NCSA[6]	CTU[7]	JSC[8]
Processor:	Intel Xeon E5-2680 v3	AMD Opteron 6276	Xeon Phi 7120P	IBM PowerPC A2
Frequency:	2.5 GHz	2.3 GHz	1.238 GHz	1.6 GHz
Cores per node:	24	16	61	16
Node memory:	128 GB	64 GB	16 GB	16 GB
Memory access:	NUMA[9]	NUMA	SMP[10]	SMP
Used compilers:	GNU g++ 4.9.2 Intel icpc 15.0.2	GNU g++ 4.8.2 Intel icpc 15.0.3	Intel icpc 15.0.2	GNU g++ 4.8.1

per core, therefore, we present measurements for up to 64 and 244 threads for these architectures, respectively. All the measurements on Intel Xeon Phi were performed in the native mode, i.e., test programs were run directly on the coprocessor.

Please note that we do not provide results of all the measurements for IBM BG/Q, since the Juqueen's runtime environment was not intended for such types of experiments. There was neither a possibility to obtain an interactive access to computational nodes nor a possibility to use a single node only. The smallest allocation unit for a job consisted of 32 nodes and such an allocation had wall clock time limited to 30 minutes. We therefore performed only selected experiments on IBM BG/Q.

Within measurements, we used 3 types of data sets for sorting:

1. *integer numbers* (IN) represented with the 64-bit unsigned integer data type,
2. *binary numbers* (BN) represented with the 8-bit unsigned integer data type,
3. *sparse matrix nonzero elements* (SM) in the COO format represented with the 32-bit unsigned integer indexes and double-precision 64-bit values.

Characteristics of these data sets and experiments performed with them are shown in Table 5.2. Although

Table 5.2: Characteristics of data sets and experiments performed with them.

Data:	IN	BN	SM
Number of arrays:	1	1	3
(Multi)element memory footprint:	8 bytes	1 byte	$2 \cdot 4 + 8 = 16$ bytes
Data generation:	random	random	random
Random distribution:	uniform	uniform	uniform
Initial ordering:	none	none	reverse lexicographical
Final ordering:	natural	natural	lexicographical

AQsort is primarily intended for multi-array sorting, we chose the IN and BN single-array sorting problems for this study since it allowed us to directly compare AQsort with its iterator-based competitors. In case of the SM data sets, such a comparison had to be performed indirectly by running the SoA-to-AoS and back AoS-to-SoA transformations.

For sake of readability, we generally refer to the elements of IN and BN arrays as to multielements, even though they are in fact "single elements" only. For all types of data sets, multielements were generated randomly; we used the implementation of the Mersene Twister pseudorandom number generator [17] provided by C++11 and Boost. In the IN and BN cases, data were sorted directly, which corresponded to sorting of randomly shuffled integer/binary numbers. The matrix nonzero elements (SM) were first sorted in the reverse lexicographical order and the measurements were performed while sorting them in the lexicographical order.

Each particular measurement within this study can be characterized by the following *input parameters*:

1. the algorithm/its implementation used for sorting,
2. the utilized hardware architecture,
3. the type of sorted data (IN, BN, SM),
4. the number of sorted multielements (n),
5. the number of utilized threads.

Due to the randomness in the input data, sorting times generally differ for the same input parameters across different algorithm runs. Let algorithm's *performance stability* denote a degree of its ability to sort data with the same input parameters in the same runtime; we call it *stability* only if the context is clear. (We could quantify stability, e.g., as an inverse of the standard deviation of the sorting time from multiple algorithm runs. Note that the defined stability has nothing to do with sorting algorithms being either stable or unstable in the sense of preserving the order of elements with equal keys. Quicksort, and therefore AQsort as well, in inherently unstable in this sense.) The reported results represent average sorting times from multiple measurements. Typically, we used between 6 and 12 measurements for the same input parameters, depending mostly on the algorithm's stability and required computational resources.

The multielements of the IN, BN, and SM data sets have different memory footprints. Their different counts thus fit memories of the architectures presented by Table 5.1. For instance, $2^{30} \approx 1.07 \cdot 10^9$ SM elements can be stored in a memory of an Intel Xeon Phi coprocessor or an IBM BG/Q node. However, we could not work with such large data sets, since even in-place variants of quicksort require some call stack space to perform the recursion. Moreover, there has to be some amount of memory reserved for system processes. We therefore decided to work within our measurements with data sets of sizes $n = 2^k \cdot 10^8$, where $k \in \mathbb{Z}$. Such a choice allowed us to cover a wide range of data set sizes and also guaranteed enough memory for system processes and the call stack. On Intel Xeon Phi coprocessors and IBM BG/Q nodes, we thus sorted at most $8 \cdot 10^8$ SM multielements. However, AQsort was the only implementation that was able to sort such an amount. If the SoA-to-AoS transformation and/or out-of-place sorting was performed, the maximum number of sorted SM multielements was only $4 \cdot 10^8$.

All the presented results were obtained for default values of the AQsort parameters, i.e., $IST = 16$ and $PBS = 1024$ (see Figure 4.1), if not specified otherwise.

5.1. Evaluation of AQsort. First, we measured the *strong scalability* of AQsort, i.e., the response of the sorting time to a different number of OpenMP threads with constant n. Results of these experiments are shown in Figure 5.1. Due to different memory footprints of multielements, we set different n for different types of sorted data (IN, BN, SM). However, for each particular type, n was fixed for all the considered architectures to allow their mutual comparison with respect to AQsort performance.

Table 5.3: Maximum measured speedup of parallel AQsort with respect to `std::sort` (*left*) and sequential AQsort (*right*).

	Speedup with respect to `std::sort`			Speedup with respect to sequential AQsort		
Architecture	IN	BN	SM	IN	BN	SM
Cray XC40	8.8	20.4	8.8	9.9	6.5	8.8
Cray XE6	5.2	17.6	4.6	6.9	3.8	5.2
Intel Xeon Phi	30.7	16.5	28.1	41.8	4.1	41.3
IBM BG/Q	11.8	36.9	14.3	15.6	5.6	16.2

To illustrate the benefits of parallel sorting, Figure 5.1 also shows sorting times for the sequential `std::sort` function from the C++ Standard Library. Clearly, AQsort reduced these sorting times in all cases considerably. The maximum obtained speedup with respect to `std::sort`, and also with respect to the sequential version of AQsort, are shown in Table 5.3.

The results for the IN and SM data sets are similar—the maximum obtained speedup with respect to `std::sort` falls between about 1/4 and 1/3 of the number of threads. The BN case is specific—the total memory footprint of the data being sorted was the same as in the IN case, but the number of sorted multielements was 8 times higher, which caused much longer sorting times of `std::sort`. On the contrary, sorting times of the sequential version of AQsort were actually lower in the BN case than in the IN case. The speedup of parallel AQsort with respect to sequential AQsort was relatively low in the BN case, especially on Intel Xeon Phi and IBM BG/Q. We attribute this effect to the saturation of the memory subsystems; on all utilized architectures, the number of memory controllers/channels was much lower than the number of their computational cores.

Note that for AMD and Intel CPU-based nodes, the lowest sorting times were always obtained when the number of OpenMP threads equaled the number of node cores (one-thread-per-core configuration). For Intel Xeon Phi and IBM BlueGene/Q, the lowest sorting times mostly occurred for the two-threads-per-core configuration, and, even if not, the differences were minimal. We can thus generally recommend these configurations for AQsort maximum speedup on given architectures.

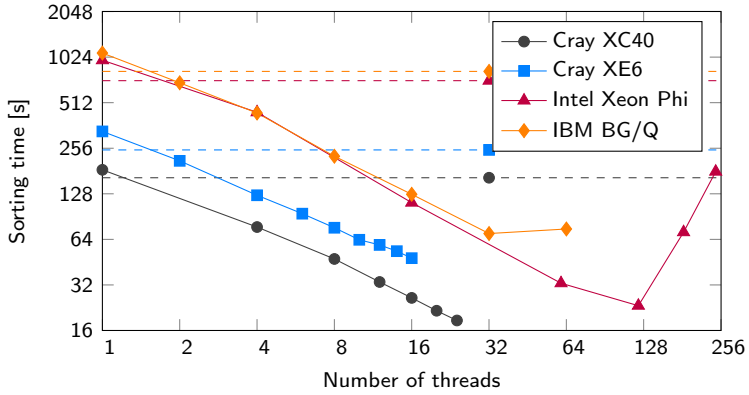
Second, we measured the relation between AQsort sorting times and the number of sorted multielements n; results are shown in Figure 5.2. On all architectures, we performed experiments up to the maximum number of multielements that fitted the available memory. The number of threads was always set to a value that provided highest speedup within the strong scalability results. The results show that the sorting time grew linearly with n and that the rate of this growth was close to 1 in all cases. Such a growth clearly does not correspond to a quicksort complexity $O(n \cdot \log n)$. In auxiliary measurements, we have observed similar linear growth with all the considered implementations of sorting algorithms, including `std::sort`.

The third experiment evaluated the dependence of AQsort sorting time on the algorithm global parameter *PBS*; results are presented in Figure 5.3. They show that, as might have been expected, small block sizes provided low algorithm performance. As the block size grew, the sorting time decreased accordingly. At some point sorting time established and no longer improved with higher block sizes; small variances in sorting times were caused by the instability of the algorithm. Generally, setting a higher block size is seemingly preferable. However, users need to be sure that there is enough data to be sorted efficiently according to condition (4.2).

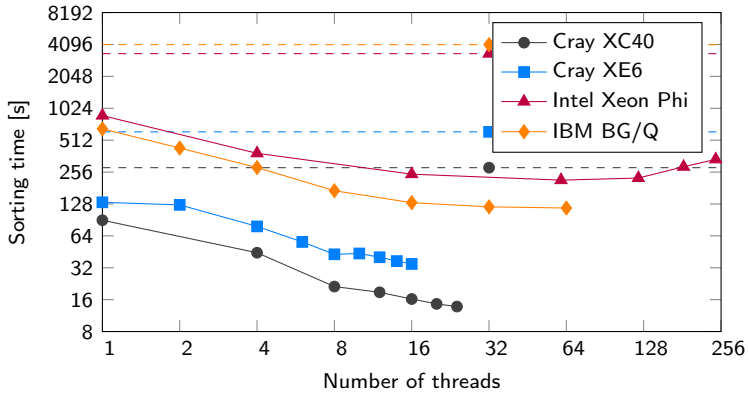
The fourth experiment evaluated the dependence of AQsort sorting time on the second algorithm global parameter *IST*; results are presented in Figure 5.4 (for space reasons, we show the measurements for the IN data sets only). According to these results, users should not set too high values of the threshold for insertion sort; again, small variances in sorting times were caused by the instability of the algorithm.

The last experiment evaluated the stability of AQsort. We performed 200 runs of the algorithm with each particular set of input parameters and processed the results statistically, see Table 5.4. For easier comparison, we show *relative sorting times* that are sorting times in percents normalized by the their average values. Obviously, the sorting times can vary significantly in practice, especially for the BN data sets. However, standard deviations reveal that typical sorting times might be expected much closer to the average than the measured extrema.

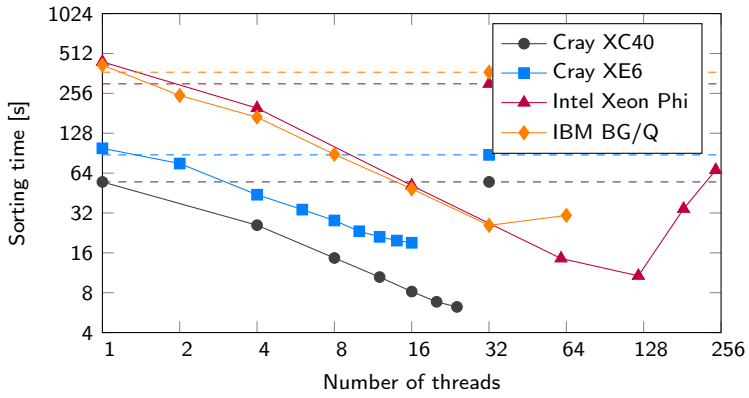
For the above described experiments, we build test programs with the GNU C++ compiler for both AQsort and `std::sort`, with the exception of Intel Xeon Phi, where we used the Intel C++ compiler.



(a) integer numbers (IN), $n = 1.6 \cdot 10^9$

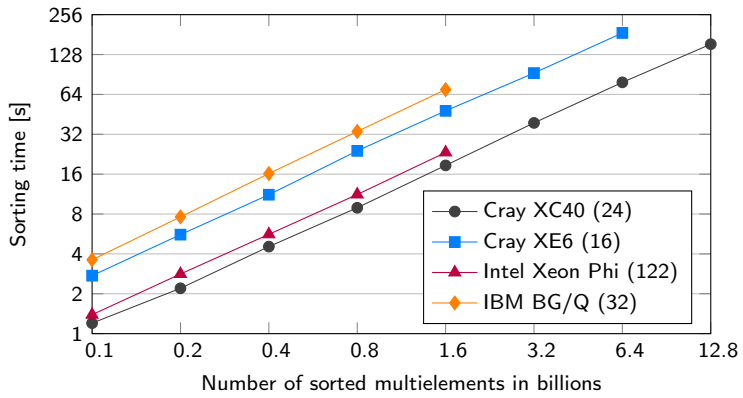


(b) binary numbers (BN), $n = 12.8 \cdot 10^9$

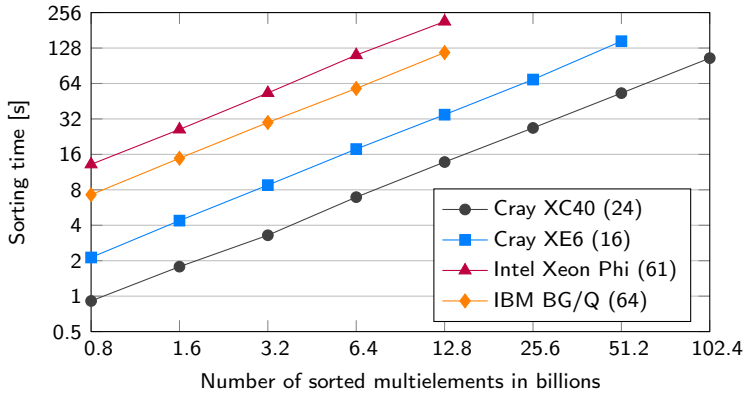


(c) sparse matrix nonzero elements (SM), $n = 0.4 \cdot 10^9$

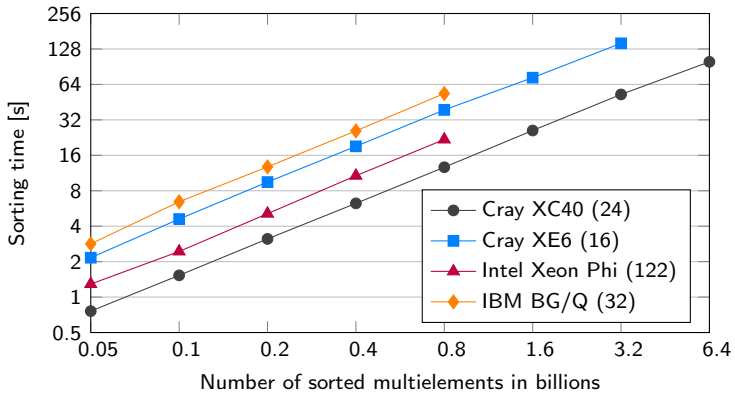
Fig. 5.1: Strong scalability of AQsort measured on different architectures. Dashed lines show the sorting times of the sequential `std::sort` function.



(a) integer numbers (IN)

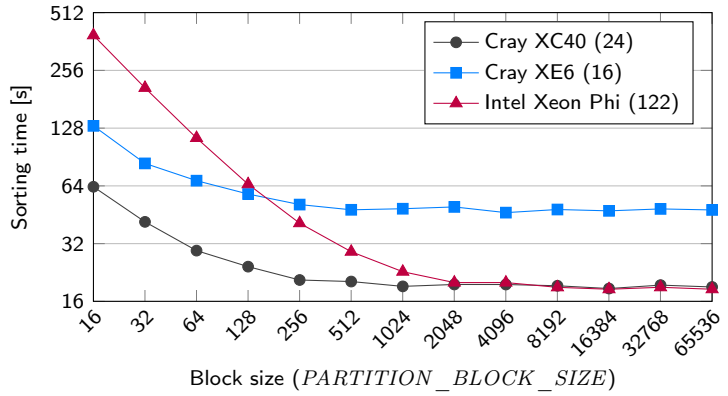


(b) binary numbers (BN)

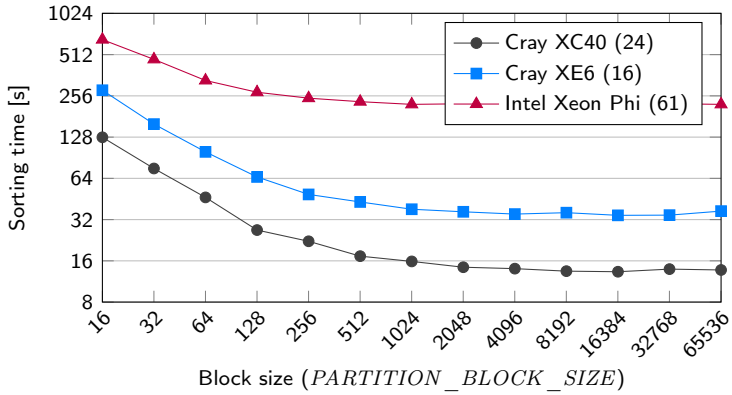


(c) sparse matrix nonzero elements (SM)

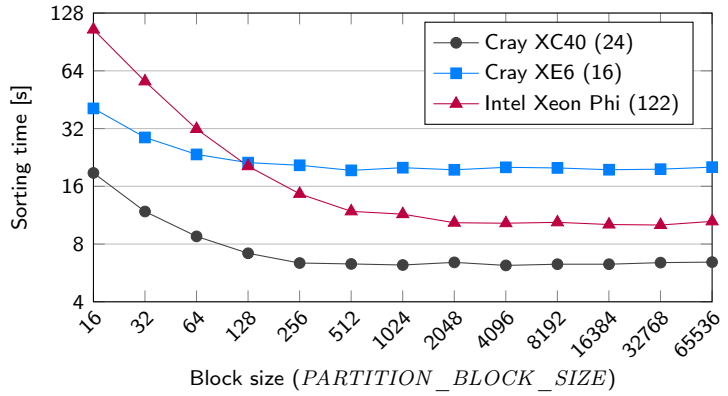
Fig. 5.2: Relation between AQsort sorting times and the number of multilements n, measured on different architectures. Numbers in parentheses in the legend denote the number of utilized OpenMP threads.



(a) integer numbers (IN), $n = 1.6 \cdot 10^9$



(b) binary numbers (BN), $n = 12.8 \cdot 10^9$



(c) sparse matrix nonzero elements (SM), $n = 0.4 \cdot 10^9$

Fig. 5.3: AQsort sorting times for different partitioning block sizes (parameter *PBS*) measured on different architectures. Numbers in parentheses in the legend denote the number of utilized OpenMP threads.

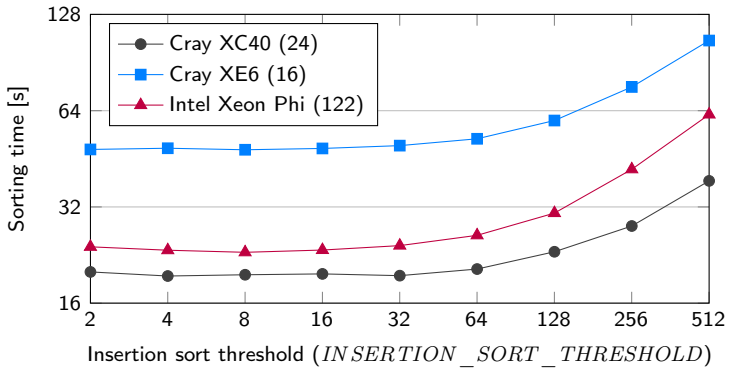


Fig. 5.4: AQsort runtimes for different values of the insertion sort threshold (parameter *IST*) measured with the IN data sets on different architectures. Numbers in parentheses in the legend denote the number of utilized OpenMP threads.

Table 5.4: Statistical values for 200 sorting times of parallel AQsort in percents normalized by their average values. The number in parentheses denote the number of utilized OpenMP threads.

	Cray XC40 (24)			Cray XE6 (16)			Intel Xeon Phi (122)		
Statistics	IN	BN	SM	IN	BN	SM	IN	BN	SM
Minimum	87.0	77.6	90.9	87.8	80.9	88.2	94.0	93.1	92.1
Average	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Maximum	108.4	135.7	108.9	110.6	129.9	112.6	109.6	110.0	116.7
Stddev	4.7	6.6	3.8	4.7	7.9	5.2	2.6	4.8	3.5

5.2. Comparison of AQsort with Modern Implementations of Sorting Algorithms. In this section, we show the results of experiments that were designed to compare the performance of AQsort and modern implementations of sorting algorithms introduced in Section 2. Namely, we considered the following parallel solutions:

1. in-place `std::__parallel::sort(..., quicksort_tag())` function from GNU Libstdc++ Parallel Mode (further referred to as GNU-QS),
2. in-place `std::__parallel::sort(..., balanced_quicksort_tag())` function from GNU Libstdc++ Parallel Mode (GNU-BQS),
3. out-of-place `std::__parallel::sort(..., multiway_mergesort_tag())` function from GNU Libstdc++ Parallel Mode (GNU-MWMS),
4. in-place `tbb::parallel_sort` function from Intel TBB (TBB),
5. in-place `cilkpub::cilk_sort_in_place` function from Cilkpub (CP-IP),
6. out-of-place `cilkpub::cilk_sort` function from Cilkpub (CP-OoP),
7. out-of-place `thrust::sort` function from Nvidia Thrust (Thrust).

The `cilkpub::cilk_sort` function calls `cilkpub::cilk_sort_in_place` if there is not enough memory to perform out-of-place sorting; we avoided this approach within our study. Cp-OoP thus always refer to the out-of-place sorting.

To evaluate the benefits of parallel sorting, we show the sorting times for the sequential `std::sort` function as well. The version of GNU Libstdc++ was given by the version of the GNU C++ compiler, see Table 5.1. As for other libraries, we compiled test programs against Intel TBB version 4.3 Update 4, Cilkpub version 1.05, and Nvidia Thrust version 1.8.0.

AQsort was the only implementation that could sort multiple arrays. To compare its performance with other implementations, we thus either measured runtime of sorting single arrays (the IN and BN cases), or we had to encapsulate sorting with the SoA-to-AoS and back AoS-to-SoA transformations (the SM case). Presented measurements were on all architectures made over the largest possible data sets, i.e., data sets that fitted the half

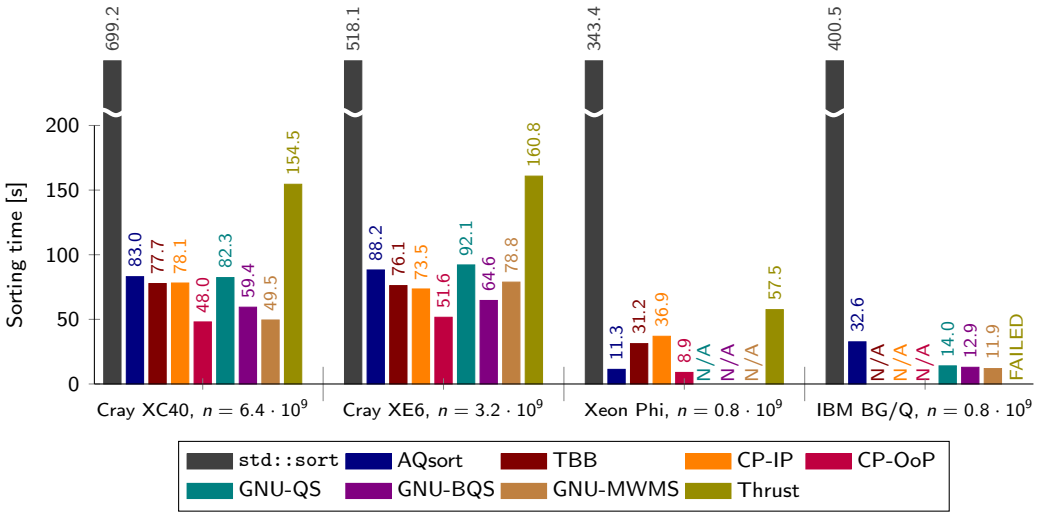


Fig. 5.5: Comparison of sorting times for different implementations of sorting algorithms for IN data sets.

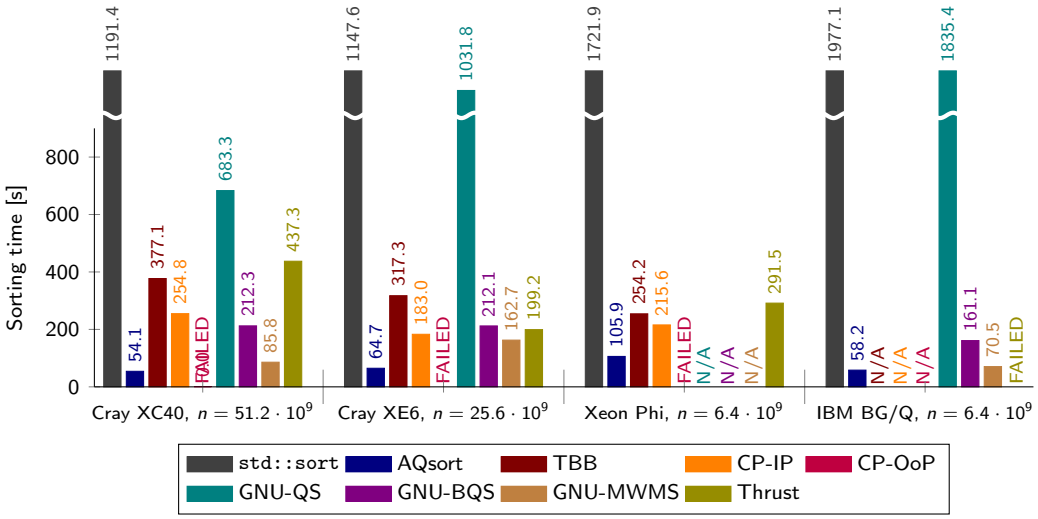


Fig. 5.6: Comparison of sorting times for different implementations of sorting algorithms for BN data sets.

of the available memory because of either out-of-place sorting and/or the SoA-to-AoS transformation. When both the SoA-to-AoS transformation and out-of-place sorting took place, the memory of the original arrays had to be released before sorting and reallocated afterwards.

All the mentioned implementations were available in the runtime environments of Cray XC40 and Cray XE6 nodes. The runtime environment of available Intel Xeon Phi accelerator did not contain the GNU Libstdc++ Parallel Mode functionality (GNU-QS, GNU-BQS, GNU-MWMS). The runtime environment of IBM BG/Q did not contain solutions provided by Intel (TBB, CP-IP, CP-OoP).

For experiments run on Intel Xeon Phi, we used exclusively the Intel C++ compiler. Otherwise, we used the GNU C++ compiler for `std::sort`, AQsort, GNU-QS, GNU-BQS, GNU-MWMS, and Thrust; and the Intel C++ compiler for TBB, CP-IP, and CP-OoP.

The obtained results are shown in Figures 5.5–5.7. The "N/A" labels denote an absence of a given implementation on a given architecture. The "FAILED" labels denote measurements that were repeatedly terminated due to runtime errors, typically segmentation faults; we have not performed analyses of these errors. Based on these results, we can make the following observations:

1. All the parallel solutions considerably reduced sorting times in comparison with sequential `std::sort`.

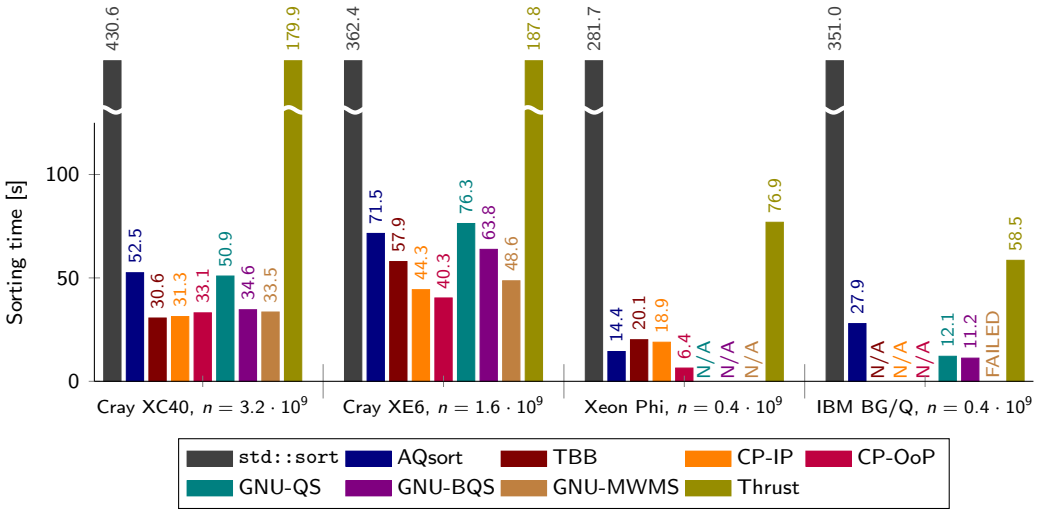


Fig. 5.7: Comparison of sorting times for different implementations of sorting algorithms for SM data sets.

An exception was only GNU-QS in combination with the BN data sets.

2. Thrust performed poorly in comparison with other parallel solutions. Recall that though Thrust supports OpenMP, it was primarily designed as a library for Nvidia GPU accelerators. It thus seems that the optimization level for OpenMP is in Thrust relatively low, at least for its sorting functionality.
3. The other out-of-place sorting implementations, i.e., CP-OoP and GNU-MWMS, mostly, but not always, provided the lowest sorting times.
4. GNU-BQS always outperformed GNU-QS.
5. AQsort outperformed for BN data sets all other implementations—even the out-of-place ones—on all architectures.
6. For IN and SM data sets, AQsort run slowly on IBM BG/Q in comparison with GNU-QS, GNU-BQS, and GNU-MWMS. This might have been caused by not-so-well optimized OpenMP environment for task-based parallelism on this architecture; AQsort was the only implementation build upon OpenMP tasks.
7. AQsort provided lowest sorting times of all in-place sorting implementations on Intel Xeon Phi.
8. For IN data sets and Cray architectures, AQsort performed slightly worse than its in-place competitors (TBB, CP-IP, GNU-QS, and GNU-BQS).
9. For SM data sets, AQsort provided higher sorting times than most of other implementations. However, these sorting times do not include the overhead given by the SoA-to-AoS and back AoS-to-SoA transformations; see Section 5.3 for details.

5.3. SoA-to-AoS and AoS-to-SoA Transformations. The reason why we did not include the SoA-to-AoS and AoS-to-SoA transformations in measurements presented by Figure 5.7 was that the performance and efficiency of theses transformations are highly implementation-dependent. For illustration, consider the following C++ solution for the SoA-to-AoS transformation that we used within our codes:

```
// sparse matrix in COO
std::vector<uint32_t> rows, cols;
std::vector<double> vals;
...
struct element_t {
   uint32_t row, col;
   double val;
};
const std::size_t n = rows.size();
std::vector<element_t> elements(n);
#pragma omp parallel for
for (std::size_t k = 0; k < n; k++) { elements[k].row = rows[k]; ... }
```

Profiling of this code on a Cray XC40 node for $n = 3.2 \cdot 10^9$ and 24 threads revealed that the parallel `for` loop took only 1.58 seconds, while the initialization of the `elements` vector took 23.15 seconds (subsequent sorting with GNU-MWMS took 32.75 seconds). A similar problem arose in the case of out-of-place sorting, where memory occupied by the `rows`, `cols`, and `vals` arrays had to be released prior to sorting and reallocated afterwards. This consisted of 3 calls of the `std::vector::resize` function, which took in this experiment 24.61 seconds, while the final parallel AoS-to-SoA transformation took only 2.13 seconds. A detailed analysis of this problem is beyond the scope of this article. Let us just note that it is caused by so-called *zero-initialization* of elements during resizing of C++ `std::vector`s, which is inherently carried out sequentially.

6. Discussion. A straightforward conclusion of our work is that though speedups are far from being linear, parallel/multi-threaded sorting considerably reduces sorting times on modern multi-core and many-core hardware architectures. The question that remains to be answered is which implementation to choose for particular sorting problems.

If we need to sort multiple arrays at once and suppose that there might not be enough available memory for the SoA-to-AoS transformation, then AQsort is the only option of all the considered solutions. This might happen, e.g., when developing scientific or engineering applications where multi-array sorting is required for possibly very large data, such as large sparse matrices or discretization meshes. In such cases, we do not want to limit the sizes of users' problems being solved just because of sorting. AQsort is the only implementation that does not need to run the SoA-to-AoS transformation. Therefore, it is the only implementation that can be straightforwardly applied to multiple arrays large enough to fill more than the half of the available memory.

On the contrary, if we certainly know that there is enough memory available, there will be no simple answer. Generic sorting of a single array with a lot of distinct sorting keys would be likely most efficient with some parallel out-of-place implementation. However, one might need to implement the SoA-to-AoS and back AoS-to-SoA transformations carefully to prevent its large runtime overhead.

Another aspects that play a crucial role in the selection of sorting implementation are threading paradigms and portability. Generally, OpenMP prevails in the domains of HPC, mathematical, scientific, and engineering software. Of all the considered implementations, only AQsort and Nvidia Thrust provide portable OpenMP solutions. GNU Libstdc++ Parallel Mode implementations are tightly bound to the GNU C++ compiler; except of this compiler, we succeeded in compiling programs that called the GNU Libstdc++ Parallel Mode sorting functions only with the newest versions of the Intel C++ compiler, namely versions 14 and 15. PGI, Cray, IBM, and older versions of Intel C++ compilers failed in such a compilation. The portability of sorting implementations based on Intel TBB and Intel Cilk Plus is relatively low in comparison with OpenMP. Moreover, though it is technically possible to integrate TBB or Cilk Plus routines into OpenMP code, such a combination of threading paradigms might result in malformed parallelism; we encountered such a behavior in the Cray runtime environments. Generally, it is not recommended to combine multiple threading paradigms within a single code.

Moreover, the interface of AQsort allows its integration with C and Fortran codes through explicit instantiation of its sorting functions with fixed footprints of comparison and swapping functions. Such an option is not feasible with other iterator-based sorting solutions.

AQsort generally seems to provide somewhat longer sorting times when compared with the sorting implementation that is most suitable for each particular case. We attribute this fact to the AQsort's universality represented by the usage of user-provided compare and swap functions. Recall that within AQsort implementation, we cannot directly touch sorted data. We therefore cannot optimize and tune the code with respect to cache subsystems, which is a common practice, e.g., in the form of software prefetching, vectorization, or accessing data with respect to their alignment. Or, for instance, we cannot store a pivot during partitioning in a temporary variable, which might cause its storage in registers during runtime for suitable types of data. Moreover, calling compare and swap functions via pointers might generally hinder many optimizations carried out by compilers on a regular basis. Users/developers therefore need to decide themselves whether they require maximum sorting performance with all that restrictions given by current implementations or whether they are willing to abandon a bit of performance for the possibility to work with larger data sets in a fully portable way.

7. Conclusions. The contribution of this article is a new parallel/multi-threaded version of the quicksort sorting algorithm and its implementation called AQsort. This implementation, written in C++ and using OpenMP, is available in the form of an open-source software project. It is primarily intended to be integrated into scientific and engineering HPC codes that operate in modern multi-core and many-core runtime environments

where OpenMP parallel paradigm is dominant.

AQsort fills some gaps in modern implementations of parallel sorting algorithms. Moreover, it provides also sequential sorting functionality that can be exploited either in sequential codes or in HPC codes based on pure MPI parallelism.

The main features of AQsort are:

1. *Generality*—it works with user-provided functions for comparing and swapping sorted data. Consequently, in contrast to pointer-based/iterator-based sorting functions, it can directly sort multiple arrays at once (see Section 3).
2. *Space-efficiency*—it operates in place and for multiple arrays it does not require the SoA-to-AoS and back AoS-to-SoA transformations. Therefore, it is the only implementation that can be straighforwardly applied to multiple arrays large enough to fill more than the half of the available memory (see Section 1).
3. *Scalability*—it considerably reduces sorting time with respect to optimized implementations of sequential sorting algorithms and it efficiently utilizes all the cores of modern multi-core and many-core hardware architectures (see Section 5.1).
4. *Portability*—it is implemented with standard OpenMP pragmas and functions. It is build upon the combination of nested parallelism and tasking, which are supported by majority of modern C++ compilers (see Section 4). Moreover, AQsort allows to create wrappers for C and Fortran programming languages. Of all the considered parallel implementations, only AQsort and Thrust were available on all the tested architectures. In comparison with AQsort, Thrust provided longer runtimes in all measurements and required more memory due to the implementation of an out-of-place sorting algorithm (see Section 5.2).
5. *Efficiency*—its performance is generally comparable with modern implementations of sorting algorithms when running on forefront HPC hardware architectures (see Section 5.2). It seems to be especially suitable for Xeon Phi coprocessors and for data that contain only few distinct sorting keys. In other cases, AQsort might be outperformed by its competitors, which is a price for its universality (see Section 6).

In addition to the presentation of AQsort, this article also:

1. serves as a brief updated survey of existing implementations of parallel sorting algorithms and their experimental comparison on leading-edge shared-memory hardware architectures (see Sections 2 and 5.2),
2. presents a parallel partitioning algorithm briefly proposed by Pasetto and Akhriev [23] in terms of detailed pseudocode (see Section 3),
3. generally evaluates the universal approach to parallel sorting that uses custom compare and swap functions with traditional solutions (see Section 6).

Acknowledgements. The authors would like to thank T. Dytrych of the Louisiana State University for providing an access to the Blue Waters system. The authors would like to thank M. Václavík of the Czech Technical University in Prague for providing an access to the Star university cluster. The authors acknowledges support from M. Pajr of CQK Holding and International HPC Initiative.

We acknowledge PRACE for awarding us access to resource JUQUEEN based in Germany at the Gauss Center for Supercomputing. We acknowledge PRACE for awarding us access to resource Hornet based in Germany at the High Performance Computing Center Stuttgart. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

REFERENCES

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 2nd ed., 1994.

[2] F. Beekhof, *OMPTL: OpenMP multi-threaded template library*, 2012. Accessed July 17, 2015 at `http://tech.unige.ch/omptl/`.

[3] N. Bell and J. Hoberock, *Thrust: A productivity-oriented library for CUDA*, in GPU Computing GEMs: Jade Edition, W.-M. Hwu, ed., Morgan Kaufmann, 2011.

[4] J. L. Bentley and M. D. McIlroy, *Engineering a sort function*, Software: Practice and Experience, 23 (1993), pp. 1249–1265.

[5] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, The MIT Press, 2007.
[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, Cambridge, Massachusetts, USA, 3rd ed., 2009.
[7] R. Dementiev, L. Kettner, and P. Sanders, *STXXL: standard template library for XXL data sets*, Software: Practice and Experience, 38 (2008), pp. 589–637.
[8] C. Hoare, *Algorithm 64: Quicksort*, Communications of the ACM, 4 (1961), pp. 321–322.
[9] ———, *Quicksort*, The Computer Journal, 5 (1962), pp. 10–16.
[10] ISO/IEC, *ISO/IEC 9899:1999: Information Technology — Programming languages — C*, 1999.
[11] ———, *ISO/IEC 14882:2011: Information Technology — Programming languages — C++*, 2011.
[12] D. Langr, I. Šimeček, and T. Dytrych, *Block iterators for sparse matrices*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2016), IEEE Xplore Digital Library, 2016. Accepted for publication.
[13] D. Langr, I. Šimeček, P. Tvrdík, T. Dytrych, and J. P. Draayer, *Adaptive-blocking hierarchical storage format for sparse matrices*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2012), IEEE Xplore Digital Library, 2012, pp. 545–551.
[14] D. Langr and P. Tvrdík, *Evaluation criteria for sparse matrix storage formats*, IEEE Transactions on Parallel and Distributed Systems, 27 (2016), pp. 428–440.
[15] B. A. Mahafzah, *Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture*, The Journal of Supercomputing, 66 (2013), pp. 339–363.
[16] D. Man, Y. Ito, and K. Nakano, *An efficient parallel sorting compatible with the standard qsort*, International Journal of Foundations of Computer Science, 22 (2011), pp. 1057–1071.
[17] M. Matsumoto and T. Nishimura, *Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Transactions on Modeling and Computer Simultaion, 8 (1998), pp. 3–30.
[18] M. McCool, R. A. D., and R. James, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, 2012.
[19] Microsoft, *Parallel STL*, 2014. Accessed July 17, 2015 at `https://parallelstl.codeplex.com/`.
[20] D. R. Musser, *Introspective sorting and selection algorithms*, Software: Practice and Experience, 27 (1997), pp. 983–993.
[21] R. Nair, *ParallelSort: Parallel sorting algorithm using OpenMP*, 2014. Accessed July 17, 2015 at `https://github.com/rsnair2/ParallelSort`.
[22] D. Pasetto and A. Akhriev, *A comparative study of parallel sort algorithms*, in Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '11, New York, NY, USA, 2011, ACM, pp. 203–204.
[23] ———, *A comparative study of parallel sort algorithms*. Accessed July 17, 2015 at http://researcher.watson.ibm.com/files/ie-albert_akhriev/sort2011-full.pdf, 2011.
[24] J. Reinders, *Intel Threading Building Blocks*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.
[25] A. D. Robinson, *A parallel stable sort using C++11 for TBB, Cilk Plus, and OpenMP*, 2014. Accessed July 17, 2015 at `https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp`.
[26] A. Robison, *Composable parallel patterns with Intel Cilk Plus*, Computing in Science Engineering, 15 (2013), pp. 66–71.
[27] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd ed., 2003.
[28] B. Schäling, *The Boost C++ Libraries*, XML Press, 2nd ed., 2014.
[29] I. Šimeček and D. Langr, *Efficient parallel evaluation of block properties of sparse matrices*, in Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2016), IEEE Xplore Digital Library, 2016. Accepted for publication.
[30] J. Singler and B. Konsik, *The gnu libstdc++ parallel mode: Software engineering considerations*, in Proceedings of the 1st International Workshop on Multicore Software Engineering, IWMSE '08, New York, NY, USA, 2008, ACM, pp. 15–22.
[31] J. Singler, P. Sanders, and F. Putze, *MCSTL: The multi-core standard template library*, in Proceedings of the Euro-Par 2007 Parallel Processing, vol. 4641 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 682–694.
[32] M. Süß and C. Leopold, *A user's experience with parallel sorting and OpenMP*, in Proceedings of the Sixth European Workshop on OpenMP, EWOMP'04, 2004, pp. 23–28.
[33] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, *A framework for adaptive algorithm selection in STAPL*, in Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '05, New York, NY, USA, 2005, ACM, pp. 277–288.
[34] P. Tsigas and Y. Zhang, *A simple, fast parallel implementation of Quicksort and its performance evaluation on SUN Enterprise 10000*, in Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003), 2003, pp. 372–381.
[35] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76.

Edited by: Dana Petcu
Received: July 22, 2016
Accepted: August 31, 2016