Nowicki *J Big Data* (2020) 7:101
https://doi.org/10.1186/s40537-020-00376-9

Journal of Big Data

RESEARCH Open Access

Comparison of sort algorithms in Hadoop and PCJ

Marek Nowicki*

*Correspondence:
faramir@mat.umk.pl
Faculty of Mathematics and Computer Science, Nicolaus Copernicus University in Toruń, Chopina 12/18, 87-100 Toruń, Poland

Abstract

Sorting algorithms are among the most commonly used algorithms in computer science and modern software. Having efficient implementation of sorting is necessary for a wide spectrum of scientific applications. This paper describes the sorting algorithm written using the partitioned global address space (PGAS) model, implemented using the Parallel Computing in Java (PCJ) library. The iterative implementation description is used to outline the possible performance issues and provide means to resolve them. The key idea of the implementation is to have an efficient building block that can be easily integrated into many application codes. This paper also presents the performance comparison of the PCJ implementation with the MapReduce approach, using Apache Hadoop *TeraSort* implementation. The comparison serves to show that the performance of the implementation is good enough, as the PCJ implementation shows similar efficiency to the Hadoop implementation.

Keywords: Parallel computing, MapReduce, Partitioned global address space, PGAS, Java, PCJ, Sorting, TeraSort

Introduction

Sorting is one of the most fundamental algorithmic problems found in a wide range of fields. One of them is data science—an interdisciplinary field that is focused on extracting useful knowledge from huge amounts of data using methods and tools for data processing and analysis. The basic metrics for data analysis include minimum, maximum, median, and top-K values. It is easy to write simple $O(n)$ algorithms that are not using sorting to calculate the first three of those metrics, but finding the median value and its variants require more work. Of course, Hoare's *quickselect* algorithm [1] can be used for finding the median of unsorted data, but its worst-case time complexity is $O(n^2)$. Moreover, some algorithms, like binary search, require data to be sorted before execution. Sequential implementation of sorting algorithms have been studied for decades. Nowadays, it becomes more and more important to use parallel computing. Here is where the efficient implementation of sorting in parallel is necessary. Existing $O(n)$ sorting algorithms, also adapted for parallel execution, like *count sort* [2, 3] or *radix sort* [4, 5], require specific input data structure, which limits their application to more general cases.

© The Author(s) 2020. This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Processing huge amounts of data, also called *Big Data* processing, is common in data science applications. Using Java, or more general solution based on Java Virtual Machine (JVM), for *Big Data* processing is a well known and established approach. *MapReduce* [6] and its implementation as a part of the Apache Hadoop framework is a good example. However, it has some limitations like disk-based communication that causes performance problem in iterative computations, but works well with the one-pass jobs it was designed for. Another good example of Java framework is Apache Spark [7] that overcomes the Hadoop iterative processing limitations and, instead of using disk for storing data between steps, it uses in-memory caching to increase performance. There are other data processing engines, like Apache Flink [8], that are designed to process data streams in real-time. These solutions outperform MapReduce in real-time streaming applications but have large requirements for memory and CPU speed [9].

The Parallel Computing in Java (PCJ) library [10], a novel approach to write parallel applications in Java, is yet another example of a library that can be used for *Big Data* processing. PCJ implements the Partitioned Global Address Space (PGAS) paradigm for running concurrent applications. The PCJ library allows running concurrent applications on systems comprise one or many multicore nodes like standard workstation, nodes with hundreds of computational threads like Intel KNL processors [11], computing clusters or even supercomputers [12].

The PCJ library won HPC Challenge award in 2014 and has been already successfully used for parallelization of selected applications. One of the applications utilizing PCJ is an application that uses a differential evolution algorithm to calculate the parameters of *C. Elegans* connectome model [13–15]. Another example is the parallelization of DNA (nucleotide) and protein sequence databases querying to find the most similar sequences to a given sequence [16, 17]. This solution was used in the ViraMiner application developing—the deep learning-based method for identifying viral genomes in human biospecimens [18]. The PCJ library was also used for the processing of large Kronecker graphs that imitate real-world networks like online social networks [19, 20]. One more example of the PCJ library usage is the approach to solve *k*-means clustering problem [21] which is a popular benchmark in the data analytics field.

There were works that attempt to implement MapReduce functionality in PGAS languages like X10 [22] or UPC [23, 24]. However, because those approaches extend the base languages with map and reduce functions, their users are required to express parallel algorithms in terms of the MapReduce framework. In this paper, the PGAS approach is directly compared to the MapReduce solution.

This paper presents a sorting algorithm written using the PCJ library in the PGAS paradigm implementation, and its performance comparison with the Hadoop TeraSort implementation [25]. Both codes implement the same algorithm—a variation of the *sample sort* algorithm [26]. The *TeraSort* implementation is a conventional and popular benchmark used in the data analytics domain. It is important to compare exactly the same algorithm, with the same time and space complexity, but having in mind that the algorithm is written using different programming paradigms.

The remainder of this paper is organized as follows. Section 2 introduces the MapReduce and PGAS programming models. Section 3 describes in details the implementation

of the sorting algorithm in both models. Section 4 contains a performance evaluation of the implementation. Last sections 5 and 6 conclude this paper.

Programming models

This section gives an overview of the MapReduce and PGAS programming models.

MapReduce

MapReduce [6] is a programming model for processing large data sets. Processing data in the MapReduce, as the name states, contains two stages: mapping (transforming) values, and reducing (combining) them.

One of the most known MapReduce frameworks is Apache Hadoop. Processing data using Apache Hadoop is composed of five steps: load, map, shuffle, reduce, and store. An example of MapReduce processing is presented in Fig. 1.

PGAS

PGAS [27] is a programming model for writing general-purpose parallel applications that can run on multiple nodes with many Central Processing Units (CPUs).

The main concept of the PGAS model is a global view of memory [27]. The global view is irrespective of whether a machine has a true shared memory or the memory is distributed.

Processors jointly execute a parallel algorithm and communicate via memory that is conceptually shared among all processes [28]. Underneath, the global view is realized by several memories that belong to different processors. In other words, global address space is partitioned over the processors [29] (cf. Fig. 2).

There are many implementations of the PGAS model, like Chapel [30], Co-Array Fortran [31], Titanium [32], UPC [33], X10 [34], APGAS [35] or, presented in this paper, the PCJ library.

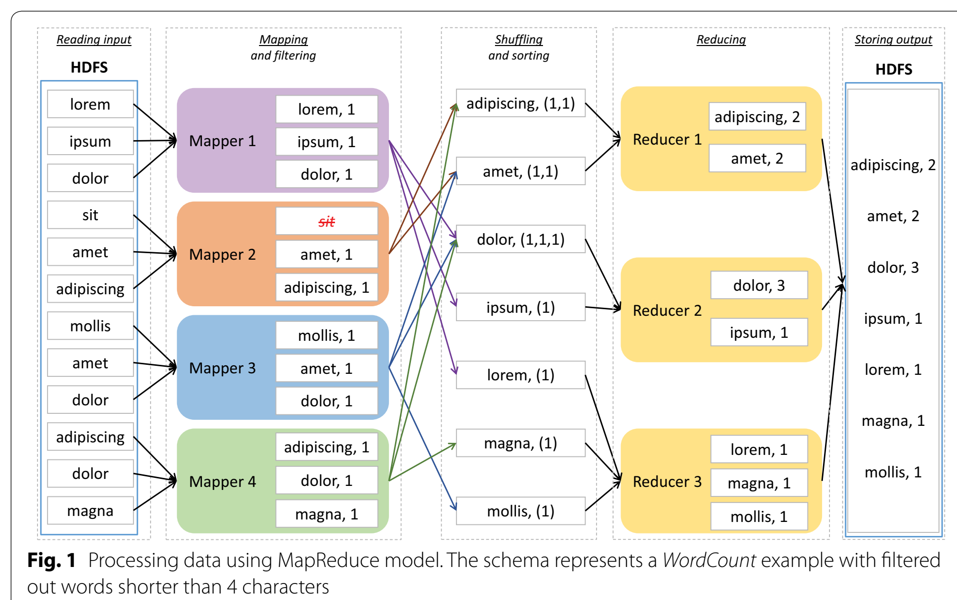

Fig. 1 Processing data using MapReduce model. The schema represents a *WordCount* example with filtered out words shorter than 4 characters

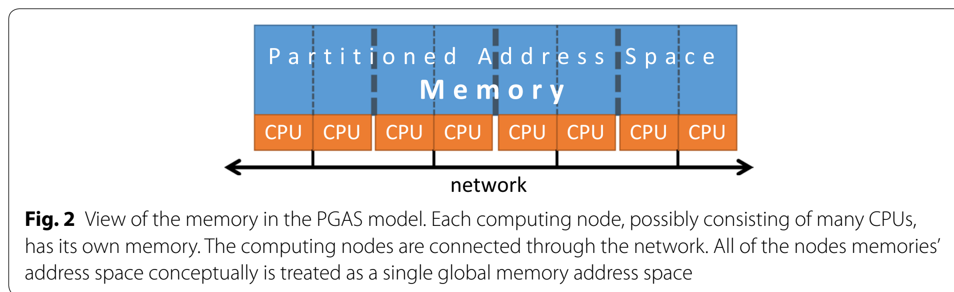

Fig. 2 View of the memory in the PGAS model. Each computing node, possibly consisting of many CPUs, has its own memory. The computing nodes are connected through the network. All of the nodes memories' address space conceptually is treated as a single global memory address space

Each PGAS implementation consists of three basic principles described in [27]. According to the first principle, each processor has its own local memory—storage, and part of it can be marked as *private* to make it not visible to other processors. The second principle is related to the flagging part of the processor's storage as *shared*—available to other processors. Implementation of sharing can be done through the network with software support, directly by hardware shared memory, or by using (RDMA). The *affinity* to a processor of every shared memory location is the third principle. Access time to the local processor's memory is short, whereas access to the memory of other processors, possibly through the network, can lead to high access latency. The information about memory affinity is available to the programmer to help producing efficient and scalable application, as access to other processors memory can be orders of magnitude slower.

The PGAS model implementations vary, i.a. on the way that remote memory can be accessed. For example, the way that the remote memory can be accessed by the threads in the PCJ library is similar to the Co-Array Fortran and the UPC implementations, where each thread can directly access other thread's memory. However, the X10 and APGAS implementations require that the memory can be accessed only at the current *place*—accessing remote *place* requires starting *activity* on the remote *place*.

Some researchers [28] place the PGAS model in-between shared-memory models such as OpenMP [36], and message-passing models like MPI [37]. The idea that all processes of parallel application operate on one single memory is inherited from the shared-memory model, whilst the certain communication cost on accessing data on other processes is inherited from the message-passing model.

The PCJ library

The PCJ library [10] is the novel approach to write parallel applications in Java. The application can utilize both multiple cores of a node and multiple nodes of a cluster. The PCJ library works in Java 8 but can be used with the newest Java version without any problem. It is due to the fact, that the library complies with the Java standards, not using any undocumented functionality, like infamous `sun.misc.Unsafe` class, and does not require any additional library that is not a part of the standard Java distribution.

The PCJ library implements the PGAS programming model. It fulfils the basic principles described in the previous section. Implicitly every variable is marked as *private*—local to the thread. Multiple *PCJ threads*, i.e. PCJ executable units (tasks), can be running on single JVM, and the standard sharing data between threads inside JVM is available. A programmer can mark class fields as *shareable*. The *shareable* variable

value can be accessed by *PCJ threads* through library methods invocation. That makes the second principle fulfilled. The *affinity* is also fulfilled as each *shareable* variable is placed on a specific *PCJ thread*. Diagram of memory affinity and its division into private and shared variables in the PCJ library is presented in Fig. 3.

The main construct of an application using PCJ is a `PCJ` class. This class contains fundamental static methods for implementing parallel constructs like thread numbering, thread synchronization and data exchanging.

The communication details are hidden from the user perspective and the methods are the same when used for intra- and inter-node communication.

Most of the methods use one-sided asynchronous communication that makes programming easy and allows to utilize overlapping communication and computation to large extend. The asynchronousness is achieved by returning a *future object* implementing `PcjFuture<T>` interface that has methods for waiting for a specified maximum time or unbounded waiting for the computation to complete. There exist a synchronous variant of each asynchronous method that is just wrapper for the asynchronous method with an invocation of the unbounded waiting method.

Despite calling PCJ executable units as *threads*, the execution using PCJ uses a constant number of *PCJ threads* in whole execution. In the current stable version of the PCJ library, the `StartPoint` interface, an entry point for execution, is the parameter of `PCJ.executionBuilder(-)` method. The method returns the *builder* object for setting up the computing nodes with methods for starting execution—the `start()` or `deploy()` methods. The architectural details of the execution are presented in Fig. 4. The multiple *PCJ threads* are part of the JVM that is running on the physical node. Communication between *PCJ threads* within JVM uses *local workers*. Communication between *PCJ threads* on different nodes uses *sockets* to transfer data through the network. The transferred data is handled by *remote workers*.

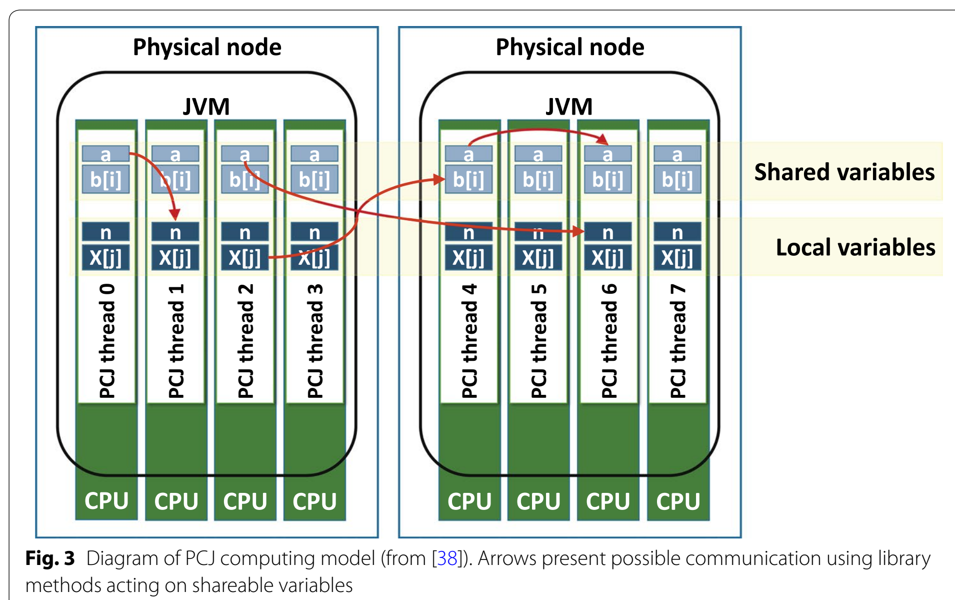

Fig. 3 Diagram of PCJ computing model (from [38]). Arrows present possible communication using library methods acting on shareable variables

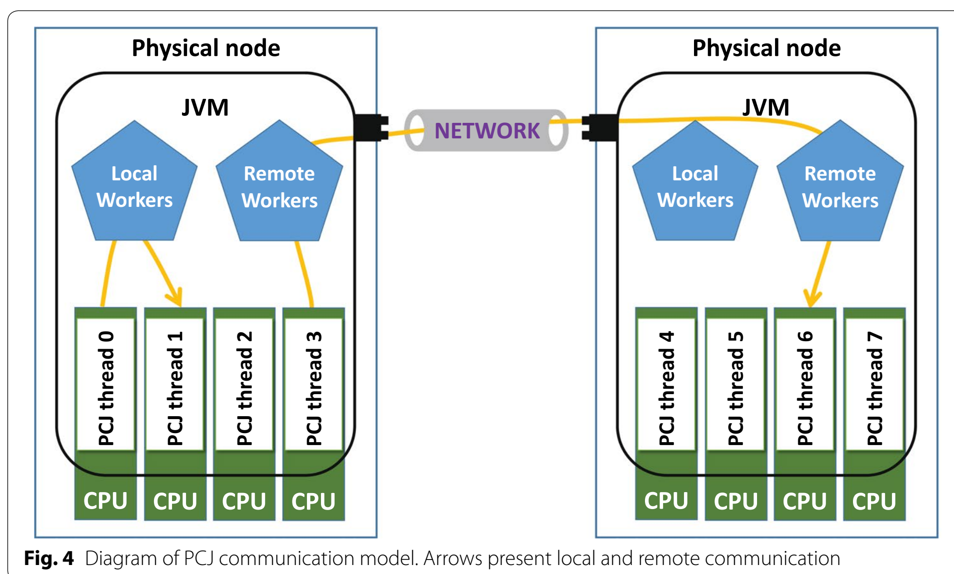

Fig. 4 Diagram of PCJ communication model. Arrows present local and remote communication

Each *PCJ thread* has its own set of *shareable* variables – the variables that can be used for exchanging data between *PCJ threads*. Each *shareable* variable is a field from a regular class. The class with *shareable* variables is called *storage*. To have access to such variable, an *enum class* has to be created with `@Storage` annotation pointing to the class containing the variable, with the name of the variable as an *enum constant*. In one code base, there can be many *storages*, and the ones that will be used in current execution have to be registered using `PCJ.registerStorage(-)` method or, preferably, by annotating `StartPoint` class by `@RegisterStorage` annotation with proper *enum* class name as parameter. To access the *shareable* variable, *PCJ thread* has to provide the *id* of a peer *PCJ thread* and the variable name as an *enum constant* name.

More detailed information about the PCJ library can be found in [38, 39].

Methods

Apache Hadoop is the most widespread and well-known framework for processing huge amount of data. It works well with the non-iterative jobs when the intermediate step data does not need to be stored on disk.

There are papers [40, 41] that show the PCJ implementation of some benchmarks scales very well and outperforms the Hadoop implementation, even by a factor of 100. One of the benchmarks calculates an approximation of π value applying the *quasi-Monte Carlo* method (employing 2-dimensional Halton sequence) using the code included in *Apache Hadoop examples* package. Other application processes large Kronecker graphs that imitate real-world networks with Breadth-First Search (BFS) algorithm. Another was *WordCount* benchmark, based on the code included in *Apache Hadoop examples* package, that counts how often words occur in an input file. However, one could argue that these benchmarks, probably omitting the last one, presented in the aforementioned papers were not perfectly suited for Hadoop processing. For this reason, a conventional, widely used benchmark for measuring the performance of Hadoop clusters, a *TeraSort* benchmark, was selected and evaluated.

The *TeraSort* is one of the widely used benchmarks for Hadoop. It measures the time to sort a different number of 100-byte records. The input file for the *TeraSort* benchmark can be created using `teragen` application from Apache Hadoop package. The application generates a file(s) with random records. Each record is 100-byte long and consists of a 10-byte key and 90-byte value.

Sample sort algorithm

The *TeraSort* is an implementation of a *sample sort* algorithm [26].

The *sample sort* (or *Samplesort*) algorithm is a divide-and-conquer algorithm. It is a generalization of the quicksort algorithm. It uses $p-1$ *pivots* (or *splitters*) whereas quicksort uses only one pivot. The pivots elements are sampled from the input data and then sorted using another sorting algorithm. The input data is divided into p buckets accordingly to pivots values. Then the buckets are sorted. In the original *sample sort* algorithm, the buckets are sorted recursively using the *sample sort* algorithm, but if a bucket's size is below some threshold, the other sorting algorithm is used. Eventually, the concatenation of the buckets produces the sorted output.

The algorithm is well suited for parallelization. The number of pivots is set as equal to the number of computational units (processors)—p. Input data is split evenly among processors. Proper selection of pivots is a crucial step of the algorithm, as the bucket sizes are determined by the pivots. Ideally, the bucket sizes are approximately the same among processors, therefore each processor spends approximately the same time on sorting.

The average-case time complexity of the parallel algorithm, where $p-1$ is the number of pivots and thus there are p processors, and n is the number of input elements, is as follows. Finding $p-1$ pivots cost is $O(p)$, sorting pivots is $O(p\log p)$, broadcasting sorted pivots is $O(p\log p)$, reading input data and placing elements into buckets by p processors is $O(\frac{n}{p}\log p)$, scattering buckets to proper processors is $O(\frac{n}{p})$, sorting buckets by p processors is $O(\frac{n}{p}\log\frac{n}{p})$, concatenation of the buckets is $O(\log p)$. In total, the average-case time complexity of the algorithm is:

$$O\left(\frac{n}{p}\log\frac{n}{p}+p\log p\right)$$

In the worst-case, all but one bucket could have only 1 element, and the rest elements would belong to one bucket. The overall time complexity in the worst-case scenario is:

$$O(n\log n+p\log p)$$

In the previous calculations, it is assumed that the average-case and worst-case time complexity of the inner sorting algorithm is $O(n\log n)$ and of finding the proper bucket is $O(\log n)$.

Hadoop TeraSort implementation

The *TeraSort*, as mentioned before, is an implementation of the *sample sort* algorithm and is written using standard map/reduce sort [42].

The used implementation of *TeraSort* for Hadoop was the one included in the *Apache Hadoop examples* package. This code was used to win annual general-purpose terabyte sort benchmark in 2008 [25].

In the *Apache Hadoop examples* package, there is also the trivial implementation of *Sort* program that uses the framework to fragment and sort the input values. However, it requires the use of `TeraInputFormat` and `TeraOutputFormat` classes from *TeraSort* implementation to properly read and write the generated input data. Removing partitioning code from `TeraInputFormat` and leaving just the code for records (key and value) storing resulted in generating the wrong output sequence—the validation of the output sequence failed.

The *TeraSort* implementation starts with records sampling. The input sampling uses the default number of 100,000 sampled records. The sampled records are sorted and evenly selected as split points and written into a file in Hadoop Distributed File System (HDFS). The sampling is done just before starting mappers tasks.

The benchmark uses a custom partitioner and the split points to ensure that all of the keys in a reducer i are less than each key in a reducer $i + 1$. The custom partitioner uses a *trie* data structure [43]. The trie is used for finding the correct partition quickly. The split file is read by the custom partitioner to fill the trie. In the implementation, the trie has a root with 256 children—intermediate nodes, one for each possible byte value, and each of the children has 256 children – the second level of intermediate nodes, again for each possible byte value. The next level of trie has leaf nodes. Each leaf node contains information about possible index ranges of split points for a given key prefix. Example of the trie is presented in Fig. 5. Figuring out the right partition for the given key is done by looking at first and then second-byte value, and then comparing key with the associated split points.

The *mapping* function is the identity function, as the records are not modified during sorting.

The key/value pairs are sorted before passing them to *reducer*s tasks. Records are sorted comparing key's data using a standard byte to byte comparison technique in the *shuffle* step.

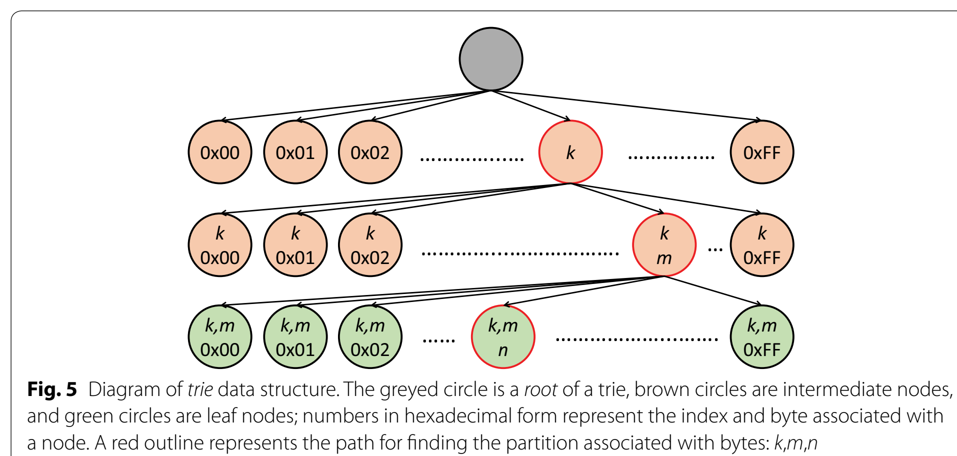

Fig. 5 Diagram of *trie* data structure. The greyed circle is a *root* of a trie, brown circles are intermediate nodes, and green circles are leaf nodes; numbers in hexadecimal form represent the index and byte associated with a node. A red outline represents the path for finding the partition associated with bytes: *k,m,n*

The *reducer* function is also an identity function. However, the reducer receives all values associated with the key as the list, thus it applies the identity function to each value in the input list returning multiple pairs with the same key and various values.

In the end, the sorted data, i.e. the returned key/value pairs, is stored back to HDFS. The directory with results contains multiple output files—one file per *reducer*.

Full code of the benchmark is available at GitHub [44]. The directory contains many files, but the benchmark consists of 5 Java files: `TeraSort.java`, `TeraSortConfigKeys.java`, `TeraInputFormat.java`, `TeraOutputFormat.java`, and `TeraScheduler.java`. Those files in total contain 831 physical lines of code as reported by `cloc` application [45] and 617 logical lines of code as reported by `lloc` application [46].

PCJ implementation

The PCJ implementation of *TeraSort* benchmark is a variation of the *sample sort* algorithm.

The algorithm used here is almost the same as the one used for the *TeraSort* algorithm. It samples 100,000 records and evenly selects one pivot per *PCJ thread* (thus the implementation name is *OnePivot*). There exists a simpler pivots selecting algorithm, where instead of sampling 100,000 records, each *PCJ thread* takes the constant number (e.g. 3 or 30) of pivots, but it generates not as good data distribution (the implementation name is *MultiplePivots*). However, the splits calculating time in both algorithms is negligible comparing to total execution time. Moreover, the performance is not much worse as presented in [47].

The execution is divided into 5 steps, similar to Edahiro's *Mapsort* described in [48]. Figure 6 presents the overview of the algorithm as a flow diagram. Table 1 contains a description of the algorithm steps. A detailed description of the basic algorithm, but in the multiple pivots per *PCJ thread* variant, with code listings is available in [47].

The basic PCJ implementation uses a single file as an input file and writes the result to a single output file. A total number of records is derived from the input file size. Every thread reads its own portion of the input file. The number of thread's records is roughly equal for all threads. If the total number of records divides with a remainder, the threads with id less than remainder have one record more to process.

Implementation variants

The basic implementation was optimized to obtain better performance, resulting in new variants of the implementation. The descriptions of optimized implementations are presented in Sect. 4.

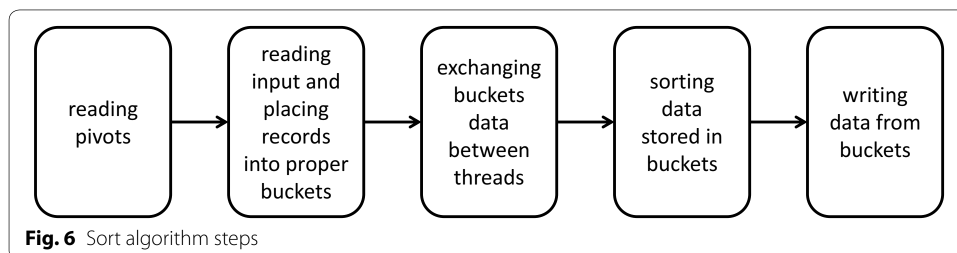

Fig. 6 Sort algorithm steps

Table 1 Description of algorithm steps

Step	Description
Reading pivots	Pivots are read evenly from a specific portion of the input file by each thread. Then ***PCJ Thread-0*** performs the reduce operation for gathering pivots data from other threads. The list is being sorted using standard Java sort algorithm [49]. The possible duplicate records are removed from the list. Then the evenly placed pivots are taken from the list and broadcasted to all the threads. A thread starts reading the input file when it receives the list
Reading input	Pivots are the records that divide input data into buckets. Each thread has to have its own set of buckets that will be used for exchanging data between threads. Each bucket is a list of records. While reading input, the record's bucket is deducted from its possible insert place in pivots list by using Java built-in binary search method. The record is added to the right bucket
Exchanging buckets	After reading the input file, it is necessary to send the data from the buckets to the threads that are *responsible for* them. The responsibility here means sorting and writing to the output file. After sending buckets data to all other threads, it is necessary to wait for receiving data from all of them
Sorting	After receiving every buckets' data it is time to sort. Each bucket is shredded into smaller arrays—one array per source thread. It is necessary to flatten the array and then sort the whole big array of records. Standard Java sort algorithm [49] for non-primitive types is used for sorting the array. The sort algorithm, called *timsort*, is a stable, adaptive, iterative mergesort, which implementation is adapted from Tim Peters's list sort for Python [50], that uses techniques from [51]
Writing output	Writing buckets data to a single output file in the correct order is the last step. This is the most sequential part of the application. Each thread has to wait for its turn to write data to the output file

Full source codes of all benchmark implementations are available at GitHub [52]. Each PCJ implementation is just one file that contains, depending on the variant, 330–410 physical lines of code as reported by `cloc` application [45] and 226–282 logical lines of code as reported by `lloc` application [46].

Results

The performance results presented in the paper have been obtained using the Hasso-Plattner Institute (HPI) Future SOC Lab infrastructure.

Hardware and software

The performance results presented in the paper has been obtained using the 1000 Core Cluster of HPI Future SOC Lab. Table 2 contains an overview of the used hardware. Table 3 contains information about the software used for benchmarks.

Apache Hadoop configuration Previous Hadoop benchmarks were done using the dedicated Hadoop cluster. However, the HPI Future SOC Lab cluster, used to obtain data for this paper, is a general-purpose cluster. To compare PCJ with Apache Hadoop it was necessary to properly set up and launch the Hadoop cluster on the Future SOC Lab cluster with SLURM submission system. The standard mechanism of starting up the Hadoop cluster uses an *ssh* connection that is unavailable between nodes of the cluster. A job that requests eight nodes was selected to workaround the issue. The job master node was selected to be the *Hadoop Master* that starts *namenode*, *secondarynamenode* and *resourcemanager* as daemon processes on the node, and *datanode*s (and for some benchmarks *nodemanager*s) daemons on all allocated nodes by executing *srun* command in the background. Thanks to the cluster configuration, there was no time limit for a job, and thus the job could run indefinitely.

Table 2 1000 Core Cluster of HPI Future SOC Lab Hardware overview

Hardware	
Cluster	25 computational nodes
	using at most 8 nodes (job resource request limit)
Nodes	4 Intel Xeon E7-4870 processors
	with maximum 2.40 GHz clock speed
Processor	10 cores (each 2 threads in hyper-threading mode)
	in total 80 threads per node
RAM	1 TB of RAM on each node more than 820 GB free, i.e. available for the user
Network	10-Gigabit ethernet
	Intel 82599ES 10-Gigabit Ethernet Controller
Storage	`/home` directory—is a NAS mounted drive with 4 TB capacity (about 1.5 TB free); available from all nodes
	`/tmp` directory—1 TB SSD drive; exclusively mounted on each node

Table 3 1000 Core Cluster of HPI Future SOC Lab Software overview

Software	
Operating system:	*Ubuntu* Linux 18.04.4 LTS *(Bionic Beaver)*
Job scheduler:	*SLURM*, version 18.08.7
Java Virtual Machine:	*Oracle JDK 13*
Apache Hadoop:	Stable version: 3.2.1
PCJ:	Stable version: 5.1.0

The *Hortonworks Documentation* [53], as well as *IBM Memory calculator worksheet* [54], describe the way of calculating the memory settings for Apache Hadoop. However, it is not well suited for the Future SOC Lab cluster, as there is only 1 disk, and the calculated values cause `InvalidResourceRequestException` exception to be thrown while submitting a job to the Hadoop cluster. Manipulating the calculated values can fix the issue, but generating input with 10^7 records takes more than 8 minutes while it can take less than 2 minutes on better configurations. Eventually, the memory configuration values were selected differently.

Almost all of the default values of Apache Hadoop configuration were left unmodified. The most important changes to configuration files are presented below.

In the *yarn-site.xml* file, the physical memory available for containers (*yarn.nodemanager.resource.memory-mb*) and the maximum allocation for every container (*yarn.scheduler.maximum-allocation-mb*) were set to 821600 MB, the minimum allocation for every container (*yarn.scheduler.minimum-allocation-mb*) was set to 128 MB, the enforcement of virtual memory limits were turned off (*yarn.nodemanager.vmem-check-enabled*) and the number of vcores that can be allocated for containers (*yarn.nodemanager.resource.cpu-vcores*) and the maximum allocation of vcores (*yarn.scheduler.maximum-allocation-vcores*) was set to 80.

The value of memory requested for all map tasks and reduce tasks is set to 40,000 MB (*mapreduce.map.resource.memory-mb* and *mapreduce.reduce.resource.memory-mb* in *mapred-site.xml* file) and application master memory is set to 128,000 MB (*yarn.app.mapreduce.am.resource.memory-mb* in *mapred-site.xml* file). The arbitrarily selected value 40,000 MB allows for full utilization of the memory, even if not all assigned tasks to cores use the whole accessible memory. The value also does not force the scheduler to use the selected number of tasks but allows to dynamically set the proper number of mappers and reducers.

The *dfs.replication* value of *hdfs-site.xml* file is set to 1, as presented tests do not need to have resilient solution and benchmarks should measure *TeraSort* implementations, not HDFS.

PCJ configuration The PCJ runtime configuration options were left unmodified except a *-Xmx820g* parameter, which means that maximum Java heap size is set to 820 GB, a *-Xlog:gc*:file=gc-{*hostname*}.log* parameter that enables printing all massages of garbage collectors into the separate output files, and a *-Dpcj.alive.timeout=0* parameter, that disables PCJ mechanism for active checking of nodes liveness.

Benchmarks

The results were obtained for a various number of records (from 10^7 up to 10^{10} records), i.e. size of input files is from about 1 GB up to about 1 TB. The input files were generated using `teragen` application from Apache Hadoop package. The application generates the official GraySort input data set [55]. Both the PCJ implementation and the *Hadoop* implementation were sorting exactly the same input sequences and produces single or multiple output files. Of course, generated output files for both implementations produce the same output sequence.

Results are based on the total time needed for the benchmark execution excluding the time needed to start the application. It is the easiest measurement to test, and it clearly presents the effectiveness of the implementation. The PCJ implementation outputs total time, while for the *Hadoop* implementation the total time was calculated as the time elapsed between *terasort.TeraSort: starting* and *terasort.TeraSort: done* log messages written with millisecond precision.

As the PCJ and Apache Hadoop both run on JVM, to mitigate garbage collection, warming-up and just-in-time compilation influences on the measurements, benchmark applications had been run several times (at least 5 times), and the shortest execution time was taken as a result.

OnePivot scaling

Figure 7 presents the total time needed to execute the basic sort implementation in PCJ depending on the total thread used. The benchmark was run on 1, 2, 4 and 8 nodes with 80 threads per node (80, 160, 320 and 640 threads in total).

The small data sizes do not show scalability. It is visible for larger records count—10^9. With the higher count of records to sort, the scalability of the execution is better. Unfortunately, due to insufficient disk space, there was no possibility to check the scalability for 10^{10} records. When sorting 10^7 and 10^8 records, the time needed for execution is almost constant, irrespective to the number of used threads. It is an

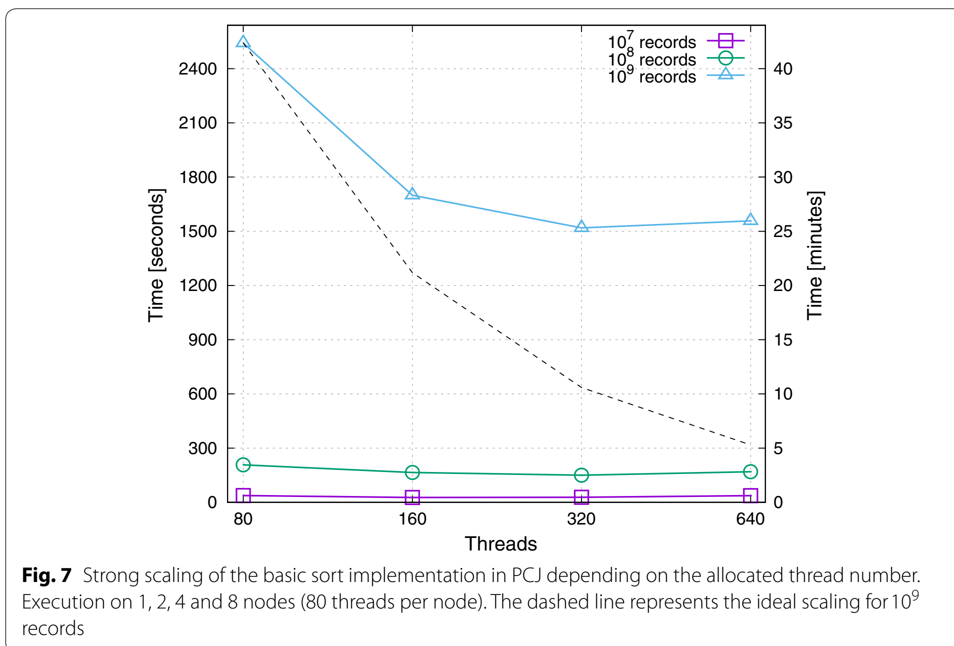

Fig. 7 Strong scaling of the basic sort implementation in PCJ depending on the allocated thread number. Execution on 1, 2, 4 and 8 nodes (80 threads per node). The dashed line represents the ideal scaling for 10^9 records

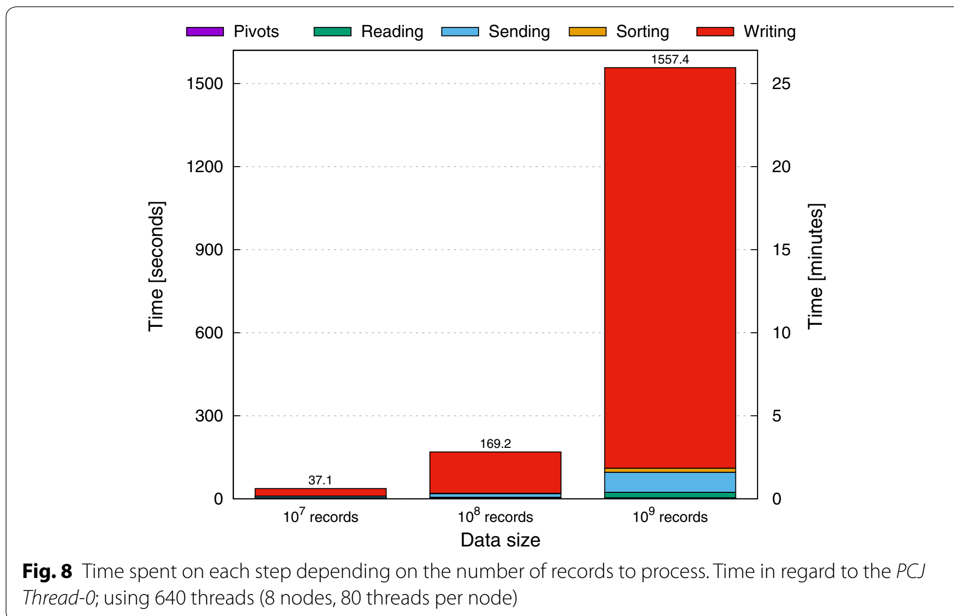

Fig. 8 Time spent on each step depending on the number of records to process. Time in regard to the *PCJ Thread-0*; using 640 threads (8 nodes, 80 threads per node)

expected behaviour as the bucket sizes are small and maintenance, as reading pivots or exchanging data, consumes time.

Moreover, the vast amount of time is spent on sequentially writing output file as presented in Fig. 8. The figure shows the time spent on each step for execution on 8 nodes, 80 threads per node, in regard to *PCJ Thread-0*, as different steps can be executed on *PCJ threads* at the same point of time.

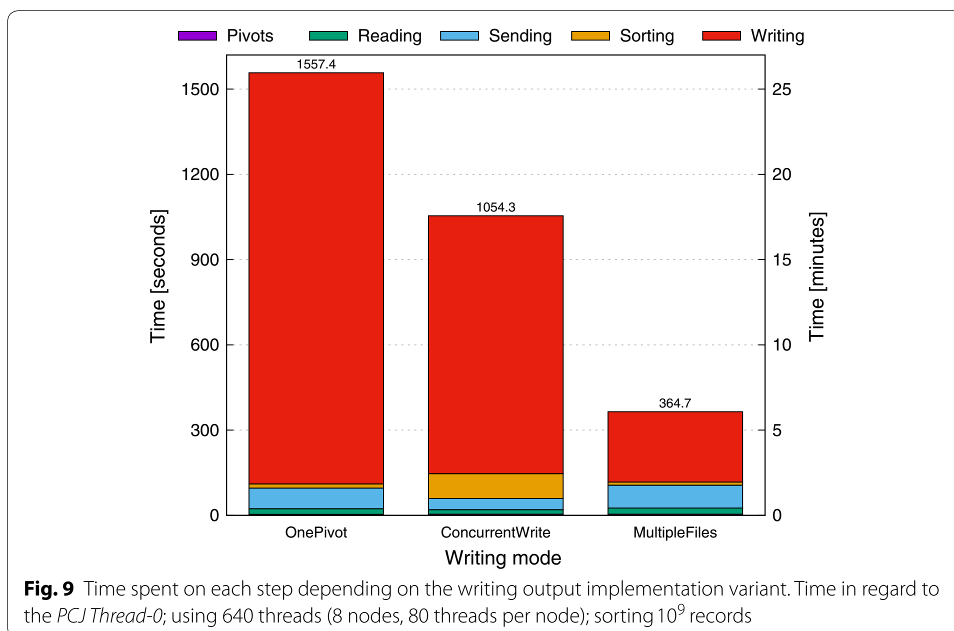

Fig. 9 Time spent on each step depending on the writing output implementation variant. Time in regard to the *PCJ Thread-0*; using 640 threads (8 nodes, 80 threads per node); sorting 10^9 records

Various writing modes

To reduce the negative performance of sequentially writing data into the output file, the next two variants of implementation were created.

The first variant of implementation, called *ConcurrentWrite*, uses memory-mapped file technique to concurrently write data to a single output file without the need of self-managing of sequenced access to the file. At the beginning of execution, the output file with proper size was created by *PCJ Thread-0* using `setLength(-)` method of `RandomAccessFile` class. This is an important step, as concurrently changing size by many threads using `FileChannel` class, used to map the file into memory, causes exceptions of system error to be thrown. In the writing step, the adequate file portion was mapped into memory in read-write mode, modified and finally saved. Size of the mapped region of the file in the benchmarks was set to contain maximum 1,000,000 records. Using this technique, the operating system is responsible for synchronization and writing data into the disk.

In the second implementation variant, called *MultipleFiles*, each *PCJ thread* creates one output file that the thread is exclusively responsible for. In this scenario, no additional synchronization is required, as the filenames rely only on *threadId*. This variant produces not exactly the same output as *OnePivot* and *ConcurrentWrite*, as it consists of multiple output files that have to be concatenated to produce a single output file. The concatenation time, using standard `cat` command, of 640 files, each of 156,250,000 bytes size (in total about 100 GB—the size of 10^9 records), takes at least $9m53s$ (median: $12m21s$, mean: $13m49s$, maximum: $23m38s$) based on 50 repeats. However, the Hadoop benchmark is also producing multiple output files, so the concatenation time is not calculated for the total execution time in the presented results.

Figure 9 presents the time spent on each step when sorting 10^9 records, in regard to *PCJ Thread-0*, using 640 threads (8 nodes, 80 threads per node). The results demonstrate, that when less synchronization is required, the time needed for writing is shorter. However, adding concatenation time to *MultipleFiles* results in similar or slightly worse

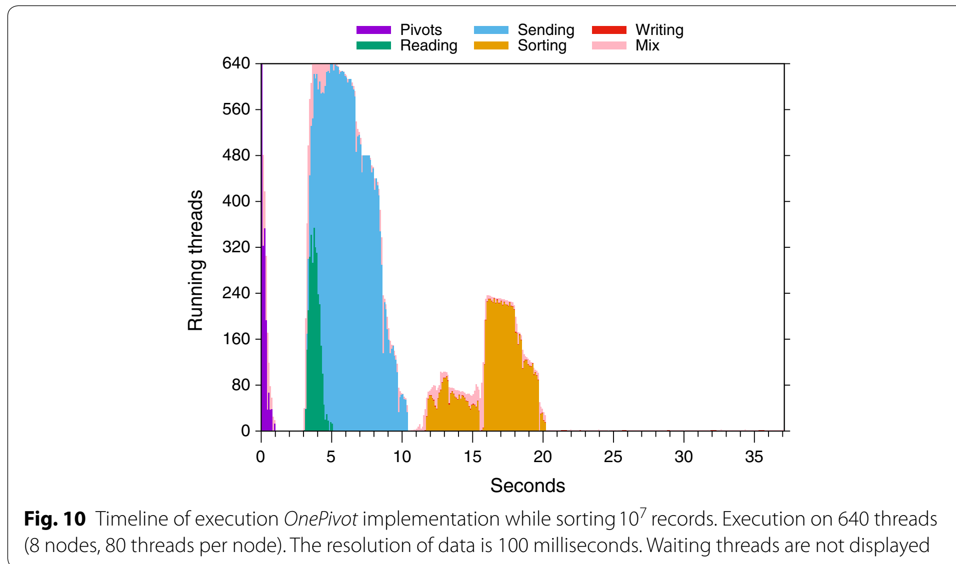

Fig. 10 Timeline of execution *OnePivot* implementation while sorting 10^7 records. Execution on 640 threads (8 nodes, 80 threads per node). The resolution of data is 100 milliseconds. Waiting threads are not displayed

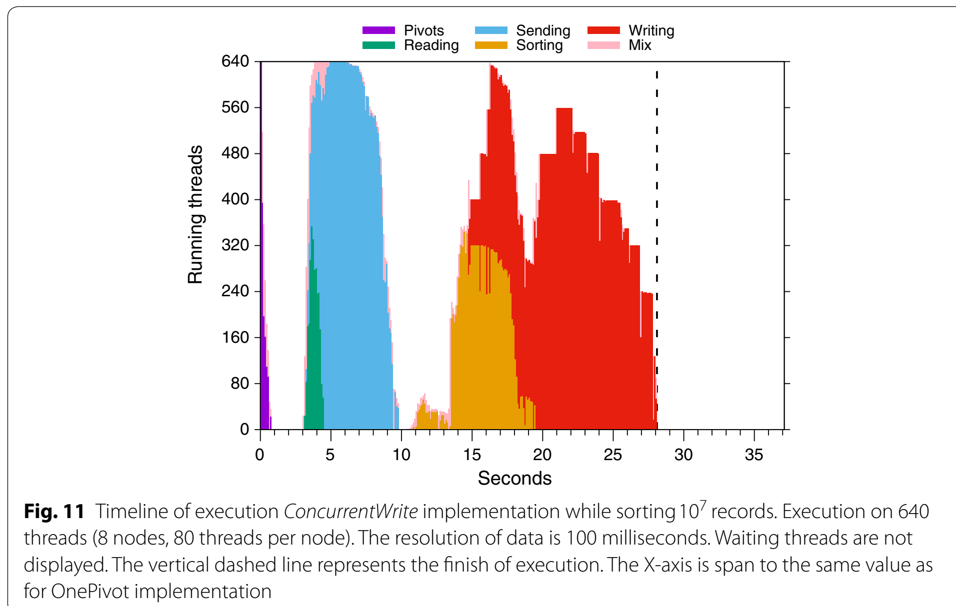

Fig. 11 Timeline of execution *ConcurrentWrite* implementation while sorting 10^7 records. Execution on 640 threads (8 nodes, 80 threads per node). The resolution of data is 100 milliseconds. Waiting threads are not displayed. The vertical dashed line represents the finish of execution. The X-axis is span to the same value as for OnePivot implementation

total time than *ConcurrentWrite*. Moreover, the concurrently writing data into the same output file is about 33% faster than waiting for own turn by the *PCJ thread* to write its own portion of output data.

Figures 10, 11 and 12 presents a timeline of execution *OnePivot*, *ConcurrentWrite* and *MultipleFiles* benchmark implementation, respectively, sorting 10^7 records using 640 threads (8 nodes, 80 threads per node). The bars represent the number of threads executing adequate algorithm step without displaying waiting threads. The *mix* step is when a thread executes more than one algorithm step in the same timestep. The resolution of data is 100 milliseconds. Each figure' x-axis is span to the same value for better comparison, and the vertical dashed line represents the finish of execution.

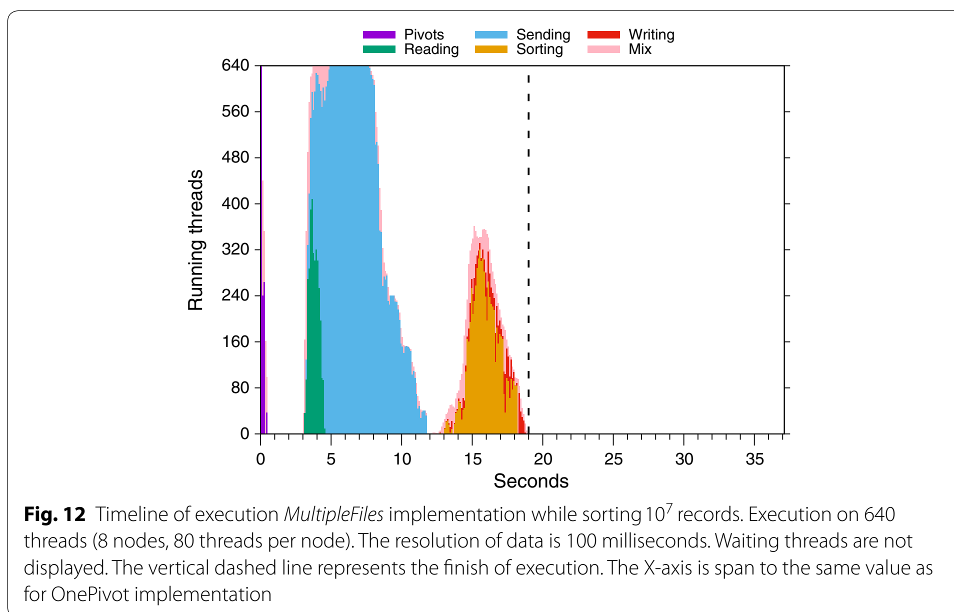

Fig. 12 Timeline of execution *MultipleFiles* implementation while sorting 10^7 records. Execution on 640 threads (8 nodes, 80 threads per node). The resolution of data is 100 milliseconds. Waiting threads are not displayed. The vertical dashed line represents the finish of execution. The X-axis is span to the same value as for OnePivot implementation

The seemingly empty plot in Fig. 10 starting at 20 s means sequentially writing results into the output file. The time needed for writing data by single *PCJ thread* in *ConcurrentWrite* implementation (cf. Fig. 11) takes much longer, as many *PCJ threads* are executing the writing step for several seconds. The best implementation is *MultipleFiles* (cf. Fig. 12) and also allows for writing output data into the not-shared file system.

One more thing worth noting is overlapping execution of the steps among threads. Some threads may still wait for buckets data from all other threads, but at the same time, the threads that have received all buckets proceed with sorting and writing. It is thanks to the asynchronous communication that is one of the main principles and advantages of the PGAS programming model, and thus especially the PCJ library.

Output drive

Up to now, the output data in the benchmarks was written into the shared folder on NAS drive. However, thanks to the *MultipleFiles* implementation, each *PCJ threads* can use its own path, that is not shared anymore. That led to the next performance measurement presented in Fig. 13. The `/home` path indicates the directory on NAS mounted drive, whereas the `/tmp` directory is located on the only locally accessible SSD drive.

At first glance, there is a strange data point, when processing 10^9 records on 80 threads (1 node) and writing to local SSD drive, as the total execution time is very similar to the processing using 2 nodes. It is the result of the internal processing of the bucket data exchanging inside a node that can be done concurrently without the bottleneck of the network connection. The behaviour does not occur when writing to the NAS drive, as the writing cannot be done in a true parallel way.

Figure 14 presents the total execution time of the *MultipleFiles* implementation, saving output data into `/tmp` directory, using in total 80 threads on 1, 2 and 4 nodes, as well as 8 nodes with 10, 20, 40 and 80 threads per node. The *XnYt* label means Y threads per node on X nodes.

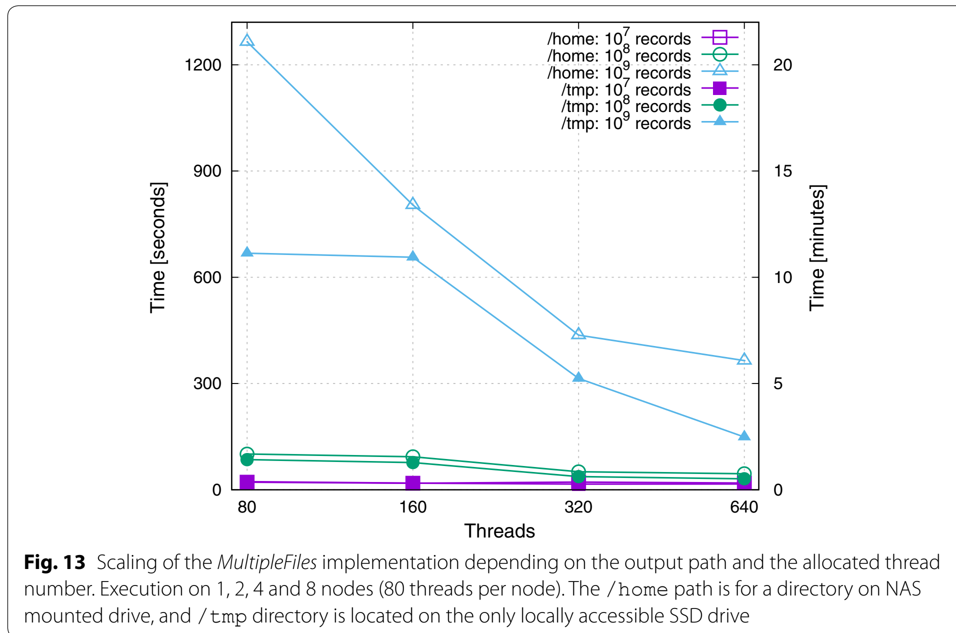

Fig. 13 Scaling of the *MultipleFiles* implementation depending on the output path and the allocated thread number. Execution on 1, 2, 4 and 8 nodes (80 threads per node). The `/home` path is for a directory on NAS mounted drive, and `/tmp` directory is located on the only locally accessible SSD drive

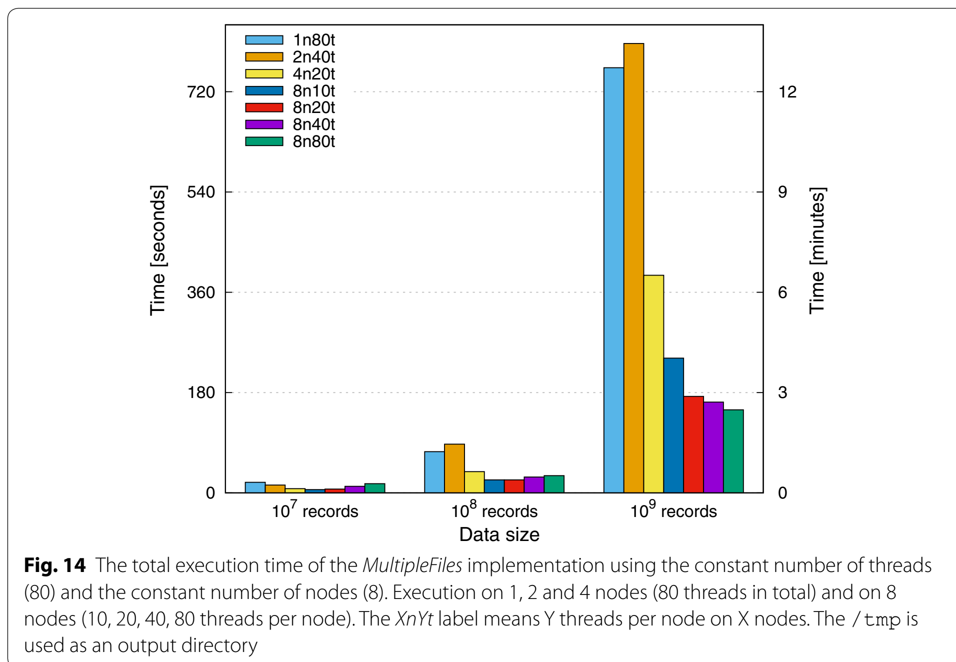

Fig. 14 The total execution time of the *MultipleFiles* implementation using the constant number of threads (80) and the constant number of nodes (8). Execution on 1, 2 and 4 nodes (80 threads in total) and on 8 nodes (10, 20, 40, 80 threads per node). The *XnYt* label means Y threads per node on X nodes. The `/tmp` is used as an output directory

Concurrent send

The concurrent writing of multiple output files onto local SSD drive gives about tenfold performance gain compared to sequentially writing a single output file onto the NAS mounted drive. This consideration resulted in the fourth implementation called *ConcurrentSend*. The implementation is based on concurrently sending data from the buckets while reading input data. Bucket data is sent when the bucket is full, i.e. bucket contains a predetermined number of items. Figure 15 shows the total execution time of the

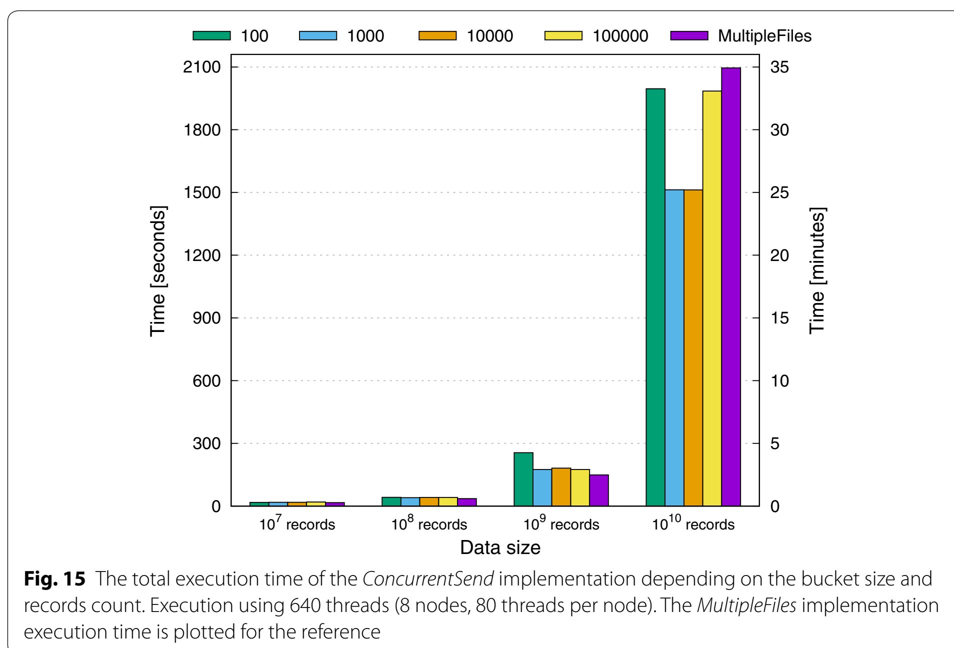

Fig. 15 The total execution time of the *ConcurrentSend* implementation depending on the bucket size and records count. Execution using 640 threads (8 nodes, 80 threads per node). The *MultipleFiles* implementation execution time is plotted for the reference

ConcurrentSend implementation depending on the records count using 640 threads (8 nodes, 80 threads per node). The *MultipleFiles* implementation execution time is plotted for the reference.

There is no visible increase in performance over *MultipleFiles* execution for smaller input data sizes. The visible gain is for input data with 10^{10} records. In this scenario, the overlap of reading input data and sending filled buckets outperforms the time losses on checking bucket size, preparing a message to send, and processing incoming data.

Selecting the proper bucket size is very important for the performance—for the next benchmarks, the bucket size was set to 1000 records.

Writing into HDFS

All previous benchmarks were focused on the performance of the PCJ *TeraSort* algorithm implementation that uses NAS or locally mounted drives to store input and output data. However, one of the main advantages of Apache Hadoop is the usage of HDFS that can store really big files in a resilient way across many *datanode*s. Having that in mind, the fifth implementation, called *HdfsConcurrentSend*, has been created. The implementation is based on the *ConcurrentSend* version, but instead of using standard Java IO classes to read and write files, the HDFS mechanism was used. Each *datanode* uses the only locally accessible `/tmp` directory on the SSD drive for storing data blocks.

Submitting Hadoop *TeraSort* job to the Hadoop cluster was done with additional parameters, that set up the default number of possible map tasks and reduce tasks (*-Dmapreduce.job.maps* $= N_{\text{tasks}}$ and *-Dmapreduce.job.reduces* $= N_{\text{tasks}}$, where N_{tasks} was calculated as a product of nodes count and 80). The value N_{tasks} means the upper limit of mappers and reducers is N_{tasks}, so concurrently there can be $2N_{\text{tasks}}$ threads used. The limit works for *reduce tasks* as stated in *Job Counters* at the end of job executions,

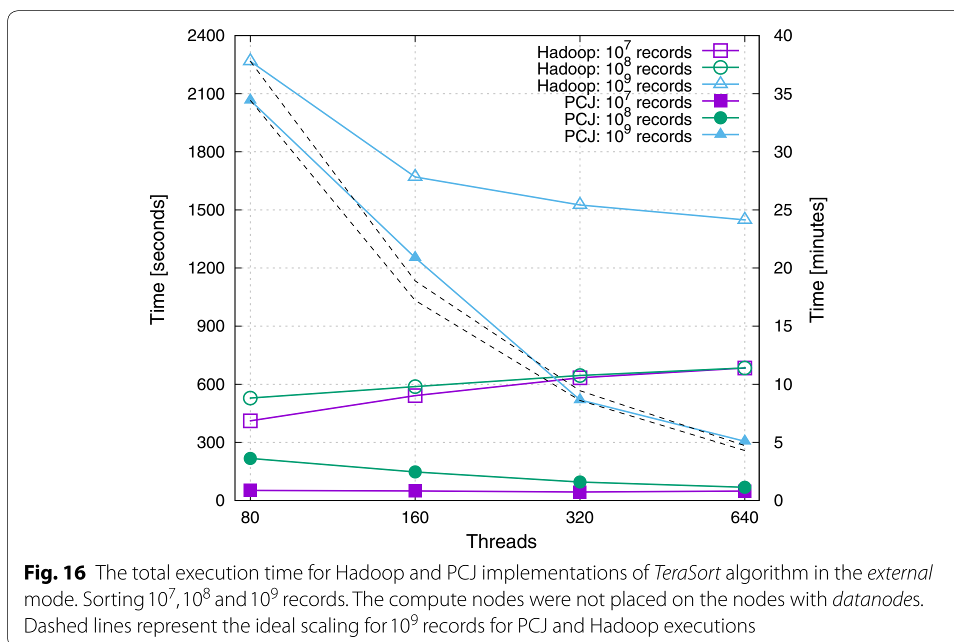

Fig. 16 The total execution time for Hadoop and PCJ implementations of *TeraSort* algorithm in the *external* mode. Sorting 10^7, 10^8 and 10^9 records. The compute nodes were not placed on the nodes with *datanodes*. Dashed lines represent the ideal scaling for 10^9 records for PCJ and Hadoop executions

whilst the number of *map tasks* depends on the input file size: for 10^7 and 10^8 records it is 640 mappers; for 10^9 records it is 1280 mappers; for 10^{10} records it is 7680 mappers.

External mode Fig. 16 presents the total execution time of sorting 10^7, 10^8 and 10^9 records using Apache Hadoop and PCJ implementations of *TeraSort* algorithm. The compute nodes were not placed on the nodes with *datanode*s (the *external* mode). In that situation, for Apache Hadoop, the proper number of *nodemanager*s were run on the nodes without *datanode*s before submitting sorting job to the Hadoop cluster. The executions using the PCJ library were done as usual by starting *PCJ threads* on the nodes without *datanode*s. This behaviour is natural in the SLURM submission system, as there was one job with allocated nodes for *master* and *datanode*s and the new jobs allocate different nodes.

The results show that the PCJ execution performance is better in comparison to the Hadoop execution. This mode is *unnatural* for Hadoop, and it cannot take advantages of data placement. For the 10^9 records both solutions scale, but the scaling of PCJ is much better. The results obtained for a smaller number of records gives lower scaling for PCJ and even the higher execution time for the Hadoop execution.

Internal mode As aforementioned, the *external* mode is not natural for Hadoop. The following benchmark results were obtained in the *internal* mode. The *internal* means that the computing nodes were placed on the same nodes as *datanode*s and the total number of *nodemanager*s was constant during all of the benchmarks. It means that the containers which execute tasks could be placed on every node without limit. The executions using the PCJ library were done using an unnatural way for SLURM by externally attaching the execution to the *Hadoop Cluster* job.

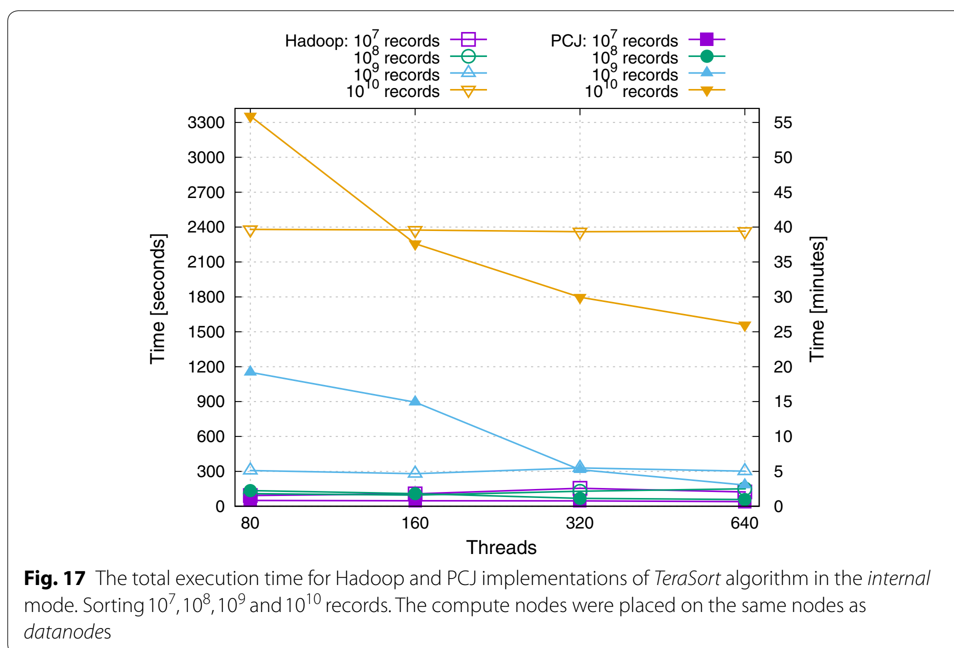

Fig. 17 The total execution time for Hadoop and PCJ implementations of *TeraSort* algorithm in the *internal* mode. Sorting $10^7, 10^8, 10^9$ and 10^{10} records. The compute nodes were placed on the same nodes as *datanodes*

The total execution times of sorting 10^7, 10^8, 10^9 and 10^{10} records using Apache Hadoop and PCJ implementations of *TeraSort* algorithm are presented in Fig. 17. The compute nodes were placed on the same nodes as *datanodes* (the *internal* mode).

The *internal* results of Hadoop sorting are approximately constant what suggests that there is no limit of nodes the containers are running on. The Hadoop cluster has chosen to run the map and reduce tasks on all of the nodes to minimize the data that has to be transferred from *datanodes* to computing nodes. In the PCJ case for 10^7, 10^8 and 10^9 records, PCJ uses exactly 80 threads per node. However, PCJ could not process 10^{10} in that configuration on less than 4 nodes, as Garbage Collector was constantly pausing the execution. The presented results for 10^{10} was collected using exactly 8 nodes with 10, 20, 40 and 80 threads per node. The PCJ execution is also taking advantage of the execution on the same nodes as *datanodes* and eventually, using 640 threads, the performance is much better for PCJ.

Figure 18 presents the total execution time of the PCJ *ConcurrentSend* implementation while writing results directly into the `/tmp` directory and through HDFS in the *internal* mode. The HDFS is also using the `/tmp` directory to store data. The data in the non-HDFS variant is read from the shared drive while the HDFS variant also read data from HDFS.

Hadoop memory setting The big gap in the total execution time for PCJ and Hadoop causes the necessity to verify the proper value of maximum memory for map tasks and reduce tasks. Up to now, the value was set to 40 GB. The maximum number of map and reduce tasks are set to 80 for each type of task resulting in a total of 160 tasks. Taking that into account, the benchmarks with 5 GB, 10 GB, 20 GB and 40 GB maximum memory values were executed. Figure 19 shows the total execution time using $N_{\text{tasks}} = 640$ on 8 nodes depending on the maximum memory for a map and reduce tasks.

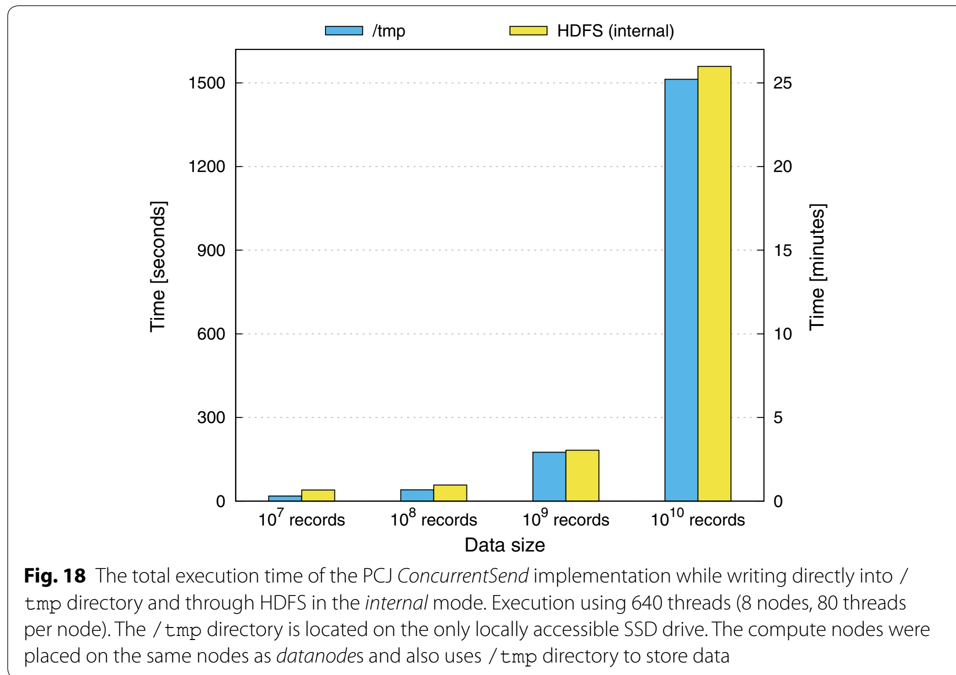

Fig. 18 The total execution time of the PCJ *ConcurrentSend* implementation while writing directly into `/tmp` directory and through HDFS in the *internal* mode. Execution using 640 threads (8 nodes, 80 threads per node). The `/tmp` directory is located on the only locally accessible SSD drive. The compute nodes were placed on the same nodes as *datanodes* and also uses `/tmp` directory to store data

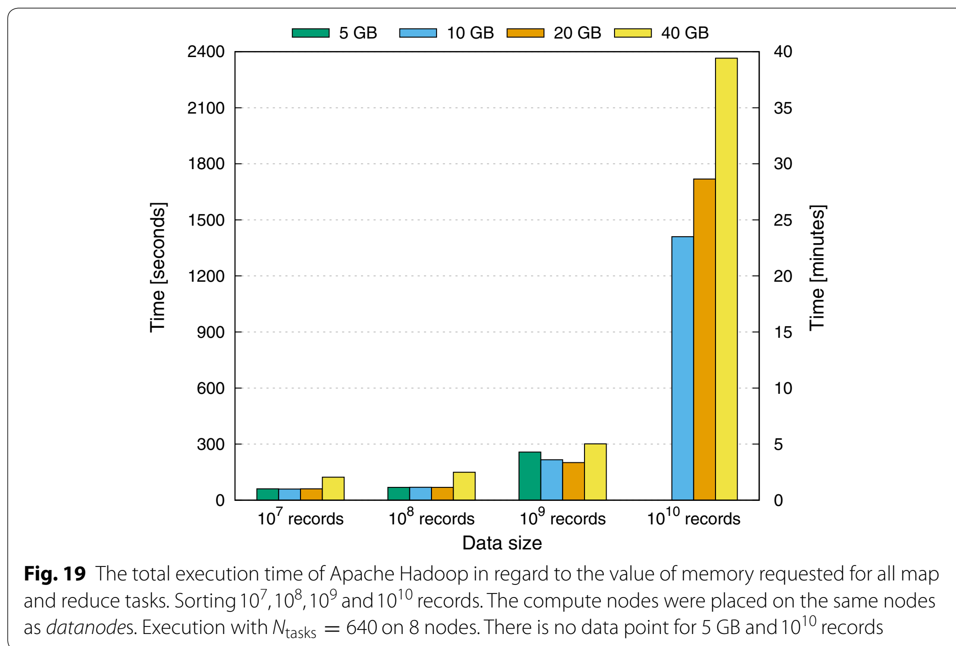

Fig. 19 The total execution time of Apache Hadoop in regard to the value of memory requested for all map and reduce tasks. Sorting $10^7, 10^8, 10^9$ and 10^{10} records. The compute nodes were placed on the same nodes as *datanodes*. Execution with $N_{\text{tasks}} = 640$ on 8 nodes. There is no data point for 5 GB and 10^{10} records

The sort execution time is shortest for the memory limit set to 10 GB and 20 GB. For the largest input 10^{10}, the memory limit set to 10 GB gives the best performance. Having in mind that on each node there is at least 820 GB RAM and there are 40 physical cores (80 logical cores with hyper-threading), setting the maximum memory allocation pool for each JVM to 10 GB, utilizes the whole memory without oversubscription. The processing of 10^{10} records, with the value of memory requested for all map tasks and reduce tasks set to 5 GB, failed due to `OutOfMemoryError`.

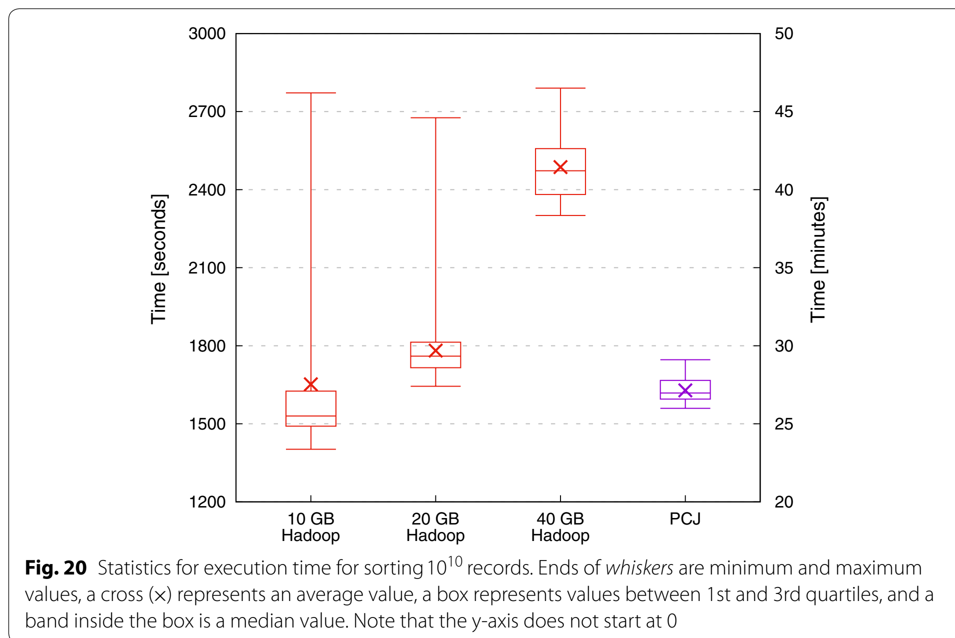

Fig. 20 Statistics for execution time for sorting 10^{10} records. Ends of *whiskers* are minimum and maximum values, a cross (×) represents an average value, a box represents values between 1st and 3rd quartiles, and a band inside the box is a median value. Note that the y-axis does not start at 0

Figure 20 presents total execution time in a form of statistical box plots for sorting 10^{10} records collected in *internal* mode using 640 threads ($N_{\text{tasks}} = 640$ for Hadoop execution), based on 50 executions. The values for Hadoop are presented for memory requested value for all mappers and reducers set to 10 GB, 20 GB and 40 GB. The PCJ execution has set the *-Xmx820g* JVM parameter on each node. The ends of *whiskers* are minimum and maximum values, a cross (×) represents an average value, a box represents values between 1st and 3rd quartiles, and a band inside the box is a median value.

The performance of PCJ is similar to the Hadoop when the memory is set to 10 GB and 20 GB. The averages are very similar, but the minimal execution time is lower for the Hadoop execution with the 10 GB value of maximum requested memory for all map and reduce tasks. Still, the PCJ execution times are consistent, whereas the Hadoop gives a broad range of total execution time.

Figure 21 contains the plots with the strong scalability of the Apache Hadoop and PCJ implementations of *TeraSort* benchmark for a various number of records (i.e. 10^7, 10^8, 10^9 and 10^{10} records). The Hadoop version was run in the *internal* mode with the 10 GB value of maximum requested memory for each mapper and reducer. The Hadoop scheduler independently could allocate tasks to the nodes, as the 8 nodes were constantly running *nodemanager*s. The PCJ results were gathered in two scenarios. In the first scenario PCJ used 1, 2, 4 and 8 nodes (80 threads per node; *Xn80t*) like in Fig. 17. In the second scenario PCJ used 8 nodes with 10, 20, 40 and 80 threads per node (*8nXt*).

The total execution time depends largely on the choice of the number of nodes and threads in regards to the size of the data to process. Generally, for the constant number of processing threads, the application is more efficient using more nodes. The PCJ implementation outperforms Hadoop for smaller input data sizes (10^7 and 10^8 records), the performance is roughly the same for 10^9 records. Hadoop is more efficient for processing

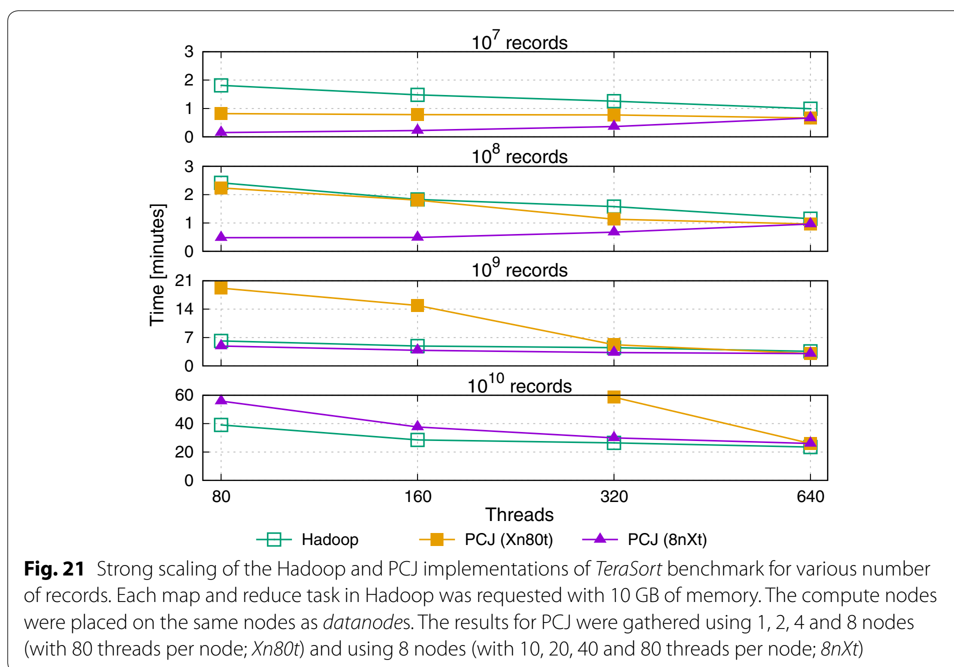

Fig. 21 Strong scaling of the Hadoop and PCJ implementations of *TeraSort* benchmark for various number of records. Each map and reduce task in Hadoop was requested with 10 GB of memory. The compute nodes were placed on the same nodes as *datanodes*. The results for PCJ were gathered using 1, 2, 4 and 8 nodes (with 80 threads per node; *Xn80t*) and using 8 nodes (with 10, 20, 40 and 80 threads per node; *8nXt*)

large data sets (10^{10} records) with a lower number of threads, but when looking at the total execution time for the largest number of threads (640) the difference is almost not visible.

Selecting a proper number of mappers and reducers is the role of Hadoop scheduler and the user does not have to think about that. The PCJ solution requires the user to start the execution on a right number of nodes and using a proper number of threads per node. A good rule of thumb is executing the application on all of the available nodes and starting as many threads as the number of logical cores on nodes having in mind the size of the input to process per thread. The performance results in that situation should not be worse than for other solutions.

Discussion

In this study, the PCJ library was used to prepare the parallel implementation of the *TeraSort* algorithm in the PGAS paradigm. Preparing a parallel application using the PCJ library was straightforward. It could be done iteratively. When the current performance was not satisfying, it was relatively easy to change a part of the code to make the application run more effectively. The user expertise in programming in Java was the only potentially limiting factor for the implemented optimization. There was no upfront special knowledge about the underlying technology that the user had to know, to write a parallel code. The resulted code is easy to read and maintain. Each step of the algorithm is clearly visible and each executed action is directly written.

The basic implementation performance was not satisfactory, as only a small fraction of execution was done in a parallel way. The writing step consumed most of the run time.

That led to next implementations which wrote results concurrently using a single memory-mapped output file and multiple output files. The latter performed better as no synchronization was involved in writing. Moreover, the possibility to write data into multiple output files allowed for the use of the local drive. Writing onto only locally accessible drive makes the implementation even more efficient. Presented results show that writing data onto the only locally accessible drive results in a better performance. This occurs regardless of the number of records to process. However, the more records to process, the performance gain is higher.

The execution using the same number of threads on a different number of nodes shows that the performance is the highest on the largest number of nodes. It is due to the high copying mechanism contention when data is being exchanged on a node with a larger number of threads. Moreover, doubling the number of threads on a node does not decrease the execution time by a factor of two. For the smaller input data sizes, the execution time takes even more than two times longer when using 640 threads (more than 16 s) than when using 80 threads (less than 6 s).

An overlapping of the read and exchange buckets' data resulted in the subsequent sort implementation. The possibility of asynchronous sending and receiving of data, almost like when using local machine memory, is one of the main advantages of the PGAS programming model. The implementation outperformed the nonoverlapping read and exchange implementation version for large data size. However, selecting the proper bucket size is crucial for the performance. Too small bucket size causes a lot of small messages and the latency consumes the eventual performance gain. On the other hand, too big bucket size results in the sending of the bucket data at the end of the data reading stage, like in nonoverlapping version, but with the additional work during every insert—checking if the bucket is full.

The last presented implementation used HDFS for reading input data and writing output data. The implementation was a little harder to write than the previous ones as it required additional libraries and setup for HDFS connection and special treatment of input and output files in HDFS. The comparison of the performance result of reading data from the shared drive and writing directly onto the locally accessible drive, with the performance when reading and writing data using HDFS shows the overhead of using the HDFS. The overhead is big for small data sizes (doubles the execution time for 10^7 elements) and decreases with increasing the data size—execution takes about 3-4% longer for 10^9 and 10^{10} elements.

In contrast to the PCJ implementations, the Hadoop implementation assumes upfront knowledge about the way that the Hadoop processes the data. The main part of the algorithm—dividing data into buckets and passing data between map and reduce steps—was taken care of internally by Hadoop by providing a special user-defined partitioner. Moreover, the record sorting in the partition is also done internally by Hadoop before passing the key/value pairs to reducers.

The Hadoop cluster configuration plays a crucial role in the possible maximum performance. The best performing Hadoop configuration performs similar or only slightly better to the PCJ implementation of *TeraSort* algorithm. However, there was almost no configuration change for the PCJ execution.

Previous studies [40, 41] show that the PCJ implementation of some benchmarks outperforms the Hadoop implementation, even by a factor of 100. Other studies that compare PCJ with APGAS and Apache Spark [56] show that the PCJ implementation has the same performance as Apache Spark, and for some benchmarks it can be almost 2-times more efficient. However, not all of the studies were perfectly suited for MapReduce processing. The present study shows similar performance for Hadoop and PCJ while running a conventional, widely used benchmark for measuring the performance of Hadoop clusters. Similar performance results can be obtained by specific problems' solutions that utilize HDFS, which may suggest that the benchmark is too highly dependent on I/O.

Presented benchmarking results for the sorting algorithm's PCJ and Apache Hadoop implementations were based on the time spent on each processing step and the total execution time. This paper does not deal with the algorithms in a wider context on real applications, where sorting is only one of the many steps needed to get the desired results. Moreover, the work does not try to determine if the usage of the PCJ library is easier or harder than using the Apache Hadoop, as it involves a personal judgement based on the user's knowledge and experience.

Conclusion

This paper described the PCJ library sorting algorithm implementation and compared its performance with Apache Hadoop *TeraSort* implementation.

The implementation using the PCJ library was presented in an iterative way that shows the possible performance problems and the ways to overcome them. The reported results of the concluding implementation show a very good performance of the PCJ implementation of the *TeraSort* algorithm. The comparison of *TeraSort* implementations indicates that PCJ performance is similar to Hadoop for a properly configured cluster and even more efficient when using on clusters with drawbacks in the configuration. Additionally, the source code written using PCJ is shorter in terms of physical and logical lines of code, and more straightforward—e.g. shuffling the data between threads is directly written into the code. Understanding of the code does not require a deep knowledge about underneath actions taken by background technology, as it is needed for the Hadoop MapReduce framework. Improper partitioning of the Hadoop *input format class* can produce incorrect results. Moreover, PCJ can also benefit from using HDFS as a standalone resilient filesystem. The advantages of HDFS do not necessarily force use of Hadoop MapReduce as a processing framework.

The PGAS programming model, represented by the PCJ library in this case, can be very competitive with the MapReduce model. Not only when considering possible performance gains but also in productivity. It provides a simple abstraction for writing a parallel application in terms of a single global memory address space and gives control over data layout and its placement among processors. Moreover, the PGAS model is a general-purpose model that suits many problems, in contrast to the MapReduce model, where the problem has to be adapted for the model (cf. [9, 40, 57, 58]).

The plan for a future study is to compare sort algorithms implemented using Apache Spark and various PGAS model implementations. Future approaches should also investigate the performance of other sort algorithms.

Abbreviations
BFS: Breadth-first search; CPU: Central processing unit; HDFS: Hadoop distributed file system; JVM: Java virtual machine; PCJ: Parallel computing in java; PGAS: Partitioned global address space; RDMA: Remote direct memory access.

Acknowledgements
The work presented in the paper is a results of the HPI grants [47, 59, 60]. The author would like to thank Future SOC Lab, Hasso Plattner Institute for Digital Engineering for awarding access to the 1000 Core Cluster and the provided support.

Authors' contributions
The sole author did all programming and writing. The author read and approved the final manuscript.

Funding
No funding has been received for the conduct of this work and preparation of this manuscript.

Availability of data and materials
All the source codes are included on the websites listed in the ***References*** section. The processed data were generated using the `teragen` application from publicity available Apache Hadoop package.

Competing interests
The author declares that he has no competing interests.

Received: 30 July 2020 Accepted: 1 November 2020
Published online: 16 November 2020

References

1. Hoare CA. Algorithm 65: find. Commun ACM. 1961;4(7):321–2.
2. Sun W, Ma Z. Count sort for GPU computing. In: 2009 15th international conference on parallel and distributed systems. IEEE; 2009. p. 919–924.
3. Kolonias V, Voyiatzis AG, Goulas G, Housos E. Design and implementation of an efficient integer count sort in CUDA GPUs. Concurr Comput. 2011;23(18):2365–81.
4. Merrill D, Grimshaw A. High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing. Parallel Processing Letters. 2011;21(02):245–72.
5. Gogolińska A, Mikulski Ł, Piątkowski M. GPU Computations and Memory Access Model Based on Petri Nets. In: Transactions on Petri Nets and Other Models of Concurrency XIII. Springer; 2018: 136–157.
6. Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM. 2008;51(1):107–13.
7. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association; 2012:2.
8. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache Flink: stream and batch processing in a single engine. Bull IEEE Comput Soc Tech Committ Data Eng. 2015;36:4.
9. Mishra P, Mishra M, Somani AK. Applications of Hadoop Ecosystems Tools NoSQL. New York: Chapman and Hall; 2017. p. 173–90.
10. PCJ homepage. https://pcj.icm.edu.pl. Accessed 26 Nov 2019.
11. Nowicki M, Górski Ł, Bała P. Evaluation of the parallel performance of the Java and PCJ on the Intel KNL based systems. In: International conference on parallel processing and applied mathematics. 2017; p. 288–97.
12. Nowicki M, Górski Ł, Bała P. Performance evaluation of parallel computing and Big Data processing with Java and PCJ library. Cray User Group. 2018;.
13. Rakowski F, Karbowski J. Optimal synaptic signaling connectome for locomotory behavior in Caenorhabditis elegans: design minimizing energy cost. PLoS Comput Biol. 2017;13(11):e1005834.
14. Górski Ł, Rakowski F, Bała P. Parallel differential evolution in the PGAS programming model implemented with PCJ Java library. In: International conference on parallel processing and applied mathematics. Springer; 2015. p. 448–58.
15. Górski Ł, Bała P, Rakowski F. A case study of software load balancing policies implemented with the PGAS programming model. In: 2016 International conference on high performance computing simulation (HPCS); 2016. p. 443–8.
16. Nowicki M, Bzhalava D, Bała P. Massively parallel sequence alignment with BLAST through work distribution implemented using PCJ library. In: Ibrahim S, Choo KK, Yan Z, Pedrycz W, editors. International conference on algorithms and architectures for parallel processing. Cham: Springer; 2017. p. 503–12.
17. Nowicki M, Bzhalava D, Bała P. Massively Parallel Implementation of Sequence Alignment with Basic Local Alignment Search Tool using Parallel Computing in Java library. J Comput Biol. 2018;25(8):871–81.
18. Tampuu A, Bzhalava Z, Dillner J, Vicente R. ViraMiner: deep learning on raw DNA sequences for identifying viral genomes in human samples. BioRxiv. 2019:602656.
19. Ryczkowska M, Nowicki M, Bała P. The performance evaluation of the Java implementation of Graph500. In: Wyrzykowski R, Deelman E, Dongarra J, Karczewski K, Kitowski J, Wiatr K, editors. Parallel processing and applied mathematics. Cham: Springer; 2016. p. 221–30.
20. Ryczkowska M, Nowicki M, Bała P. Level-synchronous BFS algorithm implemented in Java using PCJ library. In: 2016 International conference on computational science and computational intelligence (CSCI). IEEE; 2016. p. 596–601.

21. Istrate R, Barkoutsos PK, Dolfi M, Staar PWJ, Bekas C. Exploring graph analytics with the PCJ toolbox. In: Wyrzykowski R, Dongarra J, Deelman E, Karczewski K, editors. Parallel processing and applied mathematics. Cham: Springer International Publishing; 2018. p. 308–317.
22. Dong H, Zhou S, Grove D. X10-enabled MapReduce. In: Proceedings of the fourth conference on partitioned global address space programming model; 2010. p. 1–6.
23. Teijeiro C, Taboada GL, Tourino J, Doallo R. Design and implementation of MapReduce using the PGAS programming model with UPC. In: 2011 IEEE 17th international conference on parallel and distributed systems. IEEE; 2011. p. 196–203.
24. Aday S, Darkhan AZ, Madina M. PGAS approach to implement mapreduce framework based on UPC language. In: International conference on parallel computing technologies. Springer; 2017. p. 342–50.
25. O'Malley O. TeraByte Sort on Apache Hadoop. Yahoo. http://sortbenchmark.org/YahooHadoop.pdf. 2008. p. 1–3.
26. Frazer WD, McKellar AC. Samplesort: a sampling approach to minimal storage tree sorting. JACM. 1970;17(3):496–507.
27. Almasi G. PGAS (Partitioned Global Address Space) Languages. In: Padua D, editor. Encyclopedia of parallel computing. Boston: Springer; 2011. p. 1539–1545.
28. De Wael M, Marr S, De Fraine B, Van Cutsem T, De Meuter W. Partitioned Global Address Space languages. ACM Comput Surv. 2015;47(4):62.
29. Culler DE, Dusseau A, Goldstein SC, Krishnamurthy A, Lumetta S, Von Eicken T, et al. Parallel programming in Split-C. In: Supercomputing'93. Proceedings of the 1993 ACM/IEEE conference on supercomputing. IEEE; 1993. p. 262–73.
30. Deitz SJ, Chamberlain BL, Hribar MB. Chapel: Cascade High-Productivity Language. An overview of the chapel parallel programming model. Cray User Group. 2006.
31. Numrich RW, Reid J. Co-array Fortran for parallel programming. In: ACM SIGPLAN Fortran Forum, vol. 17. ACM; 1998:1–31.
32. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, et al. Titanium: a high-performance Java dialect. Concurr Comput. 1998;10(11–13):825–36.
33. Consortium U, et al. UPC Language Specifications v1.2. Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US); 2005.
34. Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, et al. X10: an Object-oriented approach to nonuniform cluster computing. In: ACM SIGPLAN Notices, vol. 40. ACM; 2005. p. 519–38.
35. Tardieu O. The APGAS library: resilient parallel and distributed programming in Java 8. In: Proceedings of the ACM SIGPLAN workshop on X10; 2015. p. 25–6.
36. Dagum L, Menon R. OpenMP: an industry-standard API for shared-memory programming. Comput Sci Eng. 1998;1:46–55.
37. Clarke L, Glendinning I, Hempel R. The MPI message passing interface standard. In: Programming environments for massively parallel distributed systems. Springer; 1994. p. 213–18.
38. Nowicki M, Ryczkowska M, Górski Ł, Szynkiewicz M, Bała P. PCJ-a Java library for heterogenous parallel computing. Recent Adv Inf Sci. 2016;36:66–72.
39. Nowicki M, Górski Ł, Bała P. PCJ–Java Library for Highly Scalable HPC and Big Data Processing. In: 2018 international conference on high performance computing and simulation (HPCS). IEEE; 2018. p. 12–20.
40. Ryczkowska M, Nowicki M. Performance comparison of graph BFS implemented in MapReduce and PGAS programming models. In: International conference on parallel processing and applied mathematics. Springer; 2017. p. 328–37.
41. Nowicki M, Ryczkowska M, Górski Ł, Bała P. Big Data analytics in Java with PCJ library: performance comparison with Hadoop. In: Wyrzykowski R, Dongarra J, Deelman E, Karczewski K, editors. International conference on parallel processing and applied mathematics. Cham: Springer; 2017. p. 318–27.
42. Apache Hadoop TeraSort package. https://hadoop.apache.org/docs/r3.2.1/api/org/apache/hadoop/examples/terasort/package-summary.html. Accessed 26 Nov 2019.
43. Sahni S. Tries. In: Mehta DP, Sahni S, editors. Handbook of data structures and applications. New York: CRC; 2004.
44. Hadoop implementation of the *TeraSort* benchmark. https://github.com/apache/hadoop/tree/780d4f416e3cac3b9e8188c658c6c8438c6a865b/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples/terasort. Accessed 10 Jan 2020.
45. AlDanial/cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages. https://github.com/AlDanial/cloc. Accessed 28 Jan 2020.
46. Artur Bosch / lloc - Logical Lines of Code. https://gitlab.com/arturbosch/lloc/tree/7f5efaf797d33a5eebb338c21637807571022fab. Accessed 28 Jan 2020.
47. Nowicki M. Benchmarking the Sort Algorithm on Ethernet Cluster. Technical Report. In: HPI Future SOC Lab: Proceedings 2019 **(in press)**.
48. Pasetto D, Akhriev A. A comparative study of parallel sort algorithms. In: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM; 2011:203–204.
49. Arrays (Java SE 13 & JDK 13). https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Arrays.html#sort(java.lang.Object%5B%5D). Accessed 07 Jul 2020.
50. Python timsort. http://svn.python.org/projects/python/trunk/Objects/listsort.txt. Accessed 07 Jul 2020.
51. McIlroy P. Optimistic sorting and information theoretic complexity. In: Proceedings of the fourth annual ACM-SIAM symposium on discrete algorithms; 1993. p. 467–74.
52. PCJ implementations of the *TeraSort* benchmark. https://github.com/hpdcj/PCJ-TeraSort/tree/a1c2cb339511e9bcd3befb892f82c522c7fbd1c3/src/main/java/pl/umk/mat/faramir/terasort. Accessed 01 July 2020.
53. Hortonworks Documentation: 11. Determine YARN and MapReduce memory configuration settings. https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.0.6.0/bk_installing_manually_book/content/rpm-chap1-11.html. Accessed 5 Nov 2020.

54. IBM Knowledge Center: Memory calculator worksheet. https://www.ibm.com/support/knowledgecenter/en/SSPT3X_4.0.0/com.ibm.swg.im.infosphere.biginsights.dev.doc/doc/biga_caching_worksheet.html. Accessed 5 Nov 2020.
55. GraySort Benchmark. Sort Benchmark Home Page. http://sortbenchmark.org. Accessed 6 Oct 2020.
56. Posner J, Reitz L, Fohry C. Comparison of the HPC and big data Java libraries spark, PCJ and APGAS. In: 2018 IEEE/ACM parallel applications workshop, alternatives To MPI (PAW-ATM). IEEE; 2018. p. 11–22.
57. Menon RK, Bhat GP, Schatz MC. Rapid Parallel Genome Indexing with MapReduce. In: Proceedings of the second international workshop on MapReduce and its applications; 2011. p. 51–8.
58. Wodo O, Zola J, Pokuri BSS, Du P, Ganapathysubramanian B. Automated, high throughput exploration of process-structure-property relationships using the MapReduce paradigm. Mater Disc. 2015;1:21–8.
59. Nowicki M. Benchmarking Java on Ethernet Cluster. Technical Report. In: HPI Future SOC Lab: Proceedings 2019 **(in press)**.
60. Nowicki M. Benchmarking the TeraSort algorithm on Ethernet Cluster. Technical Report. In: HPI Future SOC Lab: Proceedings 2020 **(in press)**.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com