Critical Evaluation of Existing External Sorting Methods in the Perspective of Modern Hardware*

Martin Kruliš

Parallel Architectures/Applications/Algorithms Research Group
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, Prague, Czech Republic
krulis@ksi.mff.cuni.cz

Abstract. External sorting methods which are designed to order large amounts of data stored in persistent memory are well known for decades. These methods were originally designed for systems with small amount of operating (internal) memory and magnetic tapes used as external memory. Data on magnetic tapes has to be accessed in strictly serial manner and this limitation shaped the external sorting algorithms. In time, magnetic tapes were replaced with hard drives which are now being replaced with solid state drives. Furthermore, the amount of the operating memory in mainstream servers have increased by orders of magnitude and the future may hold even more impressive innovations such as non-volatile memories. As a result, most of the assumptions of the external sorting algorithms are not valid any more and these methods needs to be innovated to better reflect the hardware of the day. In this work, we critically evaluate original assumptions in empirical manner and propose possible improvements.

Keywords: sorting, external sorting, hard disk, parallel

1 Introduction

Sorting is perhaps the most fundamental algorithm which reaches well beyond computer science. Along with relational JOIN operation, it is one of the most often employed data processing steps in most database systems. Despite the fact that the amount of operating memory of current desktop and server computers is increasing steadily, many problems still exist where the sorting operation cannot be performed entirely in the internal memory. In such situations, the data has to be stored in persistent storage such as hard drives and external sorting algorithms must be employed.

External sorting algorithms were designed in rather long time ago, when magnetic tapes were typical representatives of external memory. Even though the magnetic tapes are rarely used at present (except for specialized applications such as data archiving), the fundamental principles of these algorithms are still

* This paper was supported by Czech Science Foundation (GAČR) project P103/14/14292P.

M. Nečaský, J. Pokorný, P. Moravec (Eds.): Dateso 2015, pp. 77–88, CEUR-WS.org/Vol-1343.

being used. These principles are based on assumptions which no longer hold, most importantly the following:

- External memory has to be accessed sequentially or the sequential access is significantly more efficient.
- There is only a small amount of internal memory and the external memory is several orders of magnitude larger.
- External memory is slower than the internal memory by many orders of magnitude.

In this work, we would like to subject these assumptions to an empirical evaluation in order to critically asses their validity. We have performed a number of tests designed to determine performance properties of mainstream systems, hard drives, and solid state drives. Based on the results of these experiments, we propose a modification of existing methods in order to improve their performance.

In the remainder of the paper, we will assume that the data being sorted are represented as a sequence of items, where all items have the same size. An item has a value called *key*, which is used for sorting. We have used only numerical keys in our experiments; however, we have no additional assumptions about the keys than their relative comparability. Furthermore, we expect that a sorting algorithm produces a sequence of items where the keys are arranged in ascending order.

The paper is organized as follows. Section 2 summarizes related work. Traditional approach to the external sorting is revised in Section 3. Section 4 presents the experimental results that asses the individual operations required for external sorting. Finally, Section 5 proposes modifications to existing methods.

2 Related Work

The external sorting (i.e., sorting of the data that does not fit the internal memory and has to be stored in the external persistent memory) is one of the key problems of data processing. Sorting operations are widely employed in various domains [8] and they also form essential parts of other algorithms. Perhaps the first study of problems that require intensive utilization of external memory was presented in thesis of Demuth [5] more than 50 years ago, which focused mainly on the data sorting.

A rather broad study of various problem instances and of various sorting algorithms was conducted by Knuth [8] in 1970s. This study is presented in the *Art of Computer Programming* books, which are still considered to be one of the best educational materials regarding algorithms and data structures. Volume 3 is dedicated to sorting and searching and it describes commonly used methods of external sorting, such as multiway merging, polyphase merging, and various improvements. Many derived algorithms and methods for external data sorting [16] has been published since then, but they all share the fundamental assumptions layed down by Knuth.

The data communication between fast internal memory and slower external memory is still considered the bottleneck of large data processing [2, 11, 16]. Most algorithms are attempting to avoid this bottleneck by minimizing the number of total input-output operations employing various strategies for internal memory data caching [11]. Another widely utilized technique is to perform the external memory operations asynchronously, so they can overlap with internal data processing [2].

Originally, external sorting employed magnetic tapes as external memory. When magnetic disks arrived, they allowed (almost) random access to the data, so that one disk can store multiple data streams being merged. However, magnetic drives also have their limits and the sequential access to the data is still preferable. Hence, it might be beneficial to employ multiple disks either managed by the sorting algorithm directly or using striping to divide data among the disks evenly (e.g., by RAID 0 technology) [15]. Newest advances in the hardware development, especially the emergence of Solid State Disks (SSD) that employ FLASH memory instead of traditional magnetic recording, have been studied from the perspective of sorting algorithms. The SSD drives are particularly interesting from the energy consumption point of view [1, 14]; however, the introduced papers follow the original approach to external sorting and their main objective is the minimization of I/O operations.

Sorting of large datasets can be also accelerated employing multiple computers interconnected with some networking technology. Clusters of computers may have more combined internal memory than individual servers, thus a distributed solution may even achieve superlinear speedup in some cases. On the other hand, distributed sorting methods introduce new problems, such as communication overhead or load balancing [12]. Furthermore, the distributed sorting is an integral part of Map-Reduce algorithm [4], which is being widely used for distributed data processing on a large scale.

Internal sorting methods are an integral part of the external sorting, since it may be quite beneficial to pre-sort the data partially by chunks that fit the internal memory. Initially, the internal sorting methods have been summarized by Knuth in the Art of Computer Programming (vol. 3) [8]. More recently, Larson et al. [9] made a study of several internal sorting methods and asses their applicability for the initial part of external sorting.

In the past decade, the hardware development has employed parallel processing on many levels. A practical combination of internal sorting methods which takes advantage of parallel features of modern CPUs was proposed by Chhugani [3]. We have also observed an emergence of platforms for massive data parallel processing (like GPGPUs), and so several different algorithms were developed for manycore GPU accelerators [13, 10]. Finally, we would like to point out the work of Falt et al. [6] which proposes an adaptation of Mergesort for in-memory parallel data streaming systems. This work is particularly interesting, since we believe that a similar approach can be adopted for external sorting.

3 Traditional Approach to External Sorting

The traditional algorithms of external sorting have two major parts. The first part employs methods of internal sorting to form *runs* – long sequences of ordered values. The second part merges the runs into one final run, which is also the final result of the sorting problem. The algorithms that employ this approach may vary in details, such as how the runs are generated or how many runs are being merged in each step of the second part.

3.1 Forming Runs

Perhaps the most direct method for creating a run is the utilization of internal sorting algorithms, like Quicksort [7] for instance. The application allocates as large memory buffer as possible and fill it with data from the disk. The data in the buffer are sorted by an internal sorting algorithm and written back to the hard drive as a single run. These steps are repeated until all input data are pre-sorted into runs.

Direct application of internal sorting generates runs, which length corresponds to the internal memory size. Even though the intuition suggests that this is the optimal solution, Knuth [8] describes an improvement, which can be used to increase the average length of the runs. This improvement is based on adjusted version of Heapsort [17] algorithm – i.e., an internal sorting algorithm that employs regular heaps (sometimes also denoted as priority queues).

The algorithm also utilizes as much internal memory as possible. At the beginning, input data are loaded into the internal memory buffer and a d-regular heap is constructed in-place using the bottom-up algorithm. Than the algorithm performs iteratively the following steps:

1. Minimum m (top of the heap) is written to the currently generated run (but not removed from the heap yet).
2. New item x is loaded from the input data file. If $key(m) \leq key(x)$, the x replaces the top of the heap (item m) and the heap is reorganized as if the *increase-key* operation has been performed on the top. Otherwise, the *remove-top* operation is performed on the heap, so its size is reduced by 1, and the x item is stored at the position in the main memory buffer vacated by the heap.
3. If the main heap becomes empty in the previous step, the currently generated run is completed and a new run is started. At the same time, the main memory buffer is already filled with input data values, so a heap structure is constructed from them.

When the input file ends, the algorithm empties the heap to the current run by repeating the *remove-top* step. After that, the remaining data outside of the heap are sorted by internal sorting and saved as another run.

Let us emphasize two important implementation details. Despite the fact the algorithm operates on individual items, the external I/O operations are always

performed on larger blocks of items and thus both data input and data output is buffered. Furthermore, the explicit construction of a new heap (in step 3) can be amortized into step 2, so that when an item x is placed outside the heap, one step of bottom-up construction algorithm is performed. This way a secondary heap is constructed incrementally as the first heap shrinks.

If the keys exhibit uniform distribution, the two-heap algorithm can generate runs with average length $2N$, where N is the number of items stored in the internal memory buffer. This fact is explained in detail by Knuth [8] and we have also verified it empirically.

3.2 Merging Runs

The second part of the external sorting performs the merging of the runs. The main problem is that the number of runs being merged simultaneously may be limited. In the past, the computers have limited number of magnetic tapes which can be operated simultaneously. At present, operating systems have limited number of simultaneously opened files and we can perform only limited number of concurrent I/O operations to the same disk. For these reasons, we can read or write only N data streams (i.e., runs) at once. In the following, we will use the term data stream as an abstraction, which could be related to a magnetic tape or to a file on a hard disk, in order to simplify technical details such as opening and closing files.

The most direct approach to this problem is called two-phase merging. $N-1$ data streams are used as inputs and one data stream is used as output. At the beginning, the runs are distributed evenly among the input streams. The algorithm then alternates two phases (depicted in Figure 1), until there is only one run in a single data stream. In the first phase, the runs are merged from the input streams into output stream. In the second phase, the runs from the output stream are distributed evenly among the input streams.

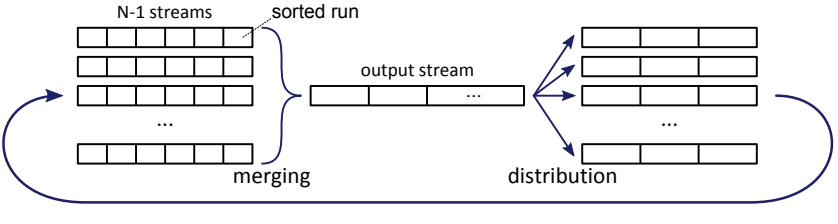

Fig. 1. The principle of two-phase merging

The merging phase takes one run from each fo $N-1$ input streams and merges them together into one run that is written to the output stream. The merging of runs is performed in the internal memory again. It can be implemented hierarchically or by using a priority queue for instance.

The main disadvantage of two-phase merging is the necessity of distributing the runs in the second phase. We can integrate the distributing phase with the merging phase as follows. If the data streams cannot be easily replaced (e.g.,

such as in case of magnetic tapes), we can reduce the number of input streams to $N/2$ and the remaining $N/2$ streams utilize as output streams. Hence, the merged runs are distributed in a round robin manner among the output streams as they are created. Unfortunately, the number of simultaneously merged runs is reduced from $N-1$ to $N/2$, wich increases the total number of iterations performed by the algorithm.

If the data streams can be replaced effectively and efficiently (e.g., a file can be closed and another file can be opened), the distribution of the runs can be performed also in the merging phase whilst $N-1$ input streams are used. Furthermore, it might be possible to utilize more than $N-1$ input streams, if the streams can be opened/closed or sought fast enough.

3.3 Polyphase Merging

There is yet another way how to amortize the distribution phase in the merging phase. This method is called *polyphase* merging, since it repeats only the merging step and the distribution of the runs is performed naturally. The merging of the runs is also performed from $N-1$ input streams into one output stream, but the initial distribution of the runs is not even. When one of the input streams is emptied, this stream becomes new output stream and the output stream is included among the input streams.

The distribution of the runs has to be carefully planned in order to achieve optimal workload. The optimal run distribution for the last merging step is that each input stream has exactly one run. One step before the last step, the optimal distribution is that $N-2$ input streams have two runs and the remaining input stream has one run. Using this pattern, the initial distribution can be planed retrospectively. An example of the merging plan for 21 runs and three streams ($N=3$) is presented in Table 1.

	beginning	#1	#2	#3	#4	#5	#6
stream 1	13	5	0	3	1	0	1
stream 2	8	0	5	2	0	1	0
stream 3	0	8	3	0	2	1	0

Table 1. Polyphase merging of 21 runs if $N=3$

At the beginning, the first stream holds 13 runs and the second 8 runs. The third stream is used as the output stream. Each phase merges as many runs as possible and merges them (e.g., 8 runs are merged in the first step). We may observe, that if there are two input streams, the distribution of the runs follow the Fibonacci sequence. If the total number of the runs is not suitable for optimal distribution, some streams may be padded with virtual runs wich do not require merging.

4 Experiments

In this section, we present performance experiments that asses individual operations of external sorting – especially the internal sorting, the performance of the heap data structure, and the disk I/O operations.

The internal sorting experiments were conducted on a high-end desktop PC with Intel Core i7 870 CPU comprising 8 logical cores clocked at 2.93 GHz and 16 GB of RAM. Additional tests were performed on 4-way cache coherent NUMA server with four Intel Xeon E7540 CPUs comprising 12 logical cores clocked at 2.0 GHz and 128 GB RAM. The RHEL (Red Hat Enterprise Linux) 7 was used as operating system on both machines.

The I/O tests were conducted on three storages. Tests denoted *HDD* were measured using one Segate Barracuda HDD (2 TB, 7,200 rpm) connected via SATA II interface. Tests denoted *SSD* were conducted on 6 solid state drives Samsung 840 Pro (512 GB each) in software RAID 0 configuration. Finally, tests denoted *array* used Infortrend ESDS 3060 enterprise disk array which comprised 2 SSD disks (400 GB each) and 14 HDDs (4 TB, 7,200 rpm) connected in RAID 6. The array was connected via dual-link 16 Gbit Fiber Channel. Again, the RHEL 7 was used as operating system and XFS was used as the file system.

4.1 Internal Sorting

In order to asses the benefits of the heap data structure for both generating the runs and for multiway merging, we compare Heapsort with conventional sorting methods. Before we can do that, we need to select optimal heap degree (i.e., branching degree of the tree that represents the heap) for this task.

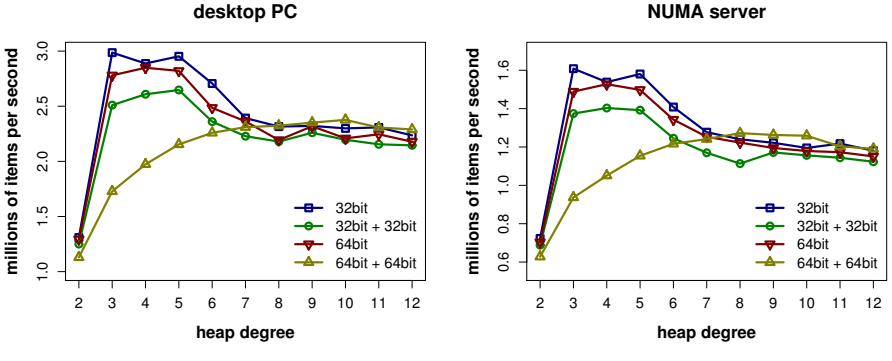

Fig. 2. Heapsort throughput results for various degrees of regular heaps

Figure 2 presents the measured throughput of Heapsort algorithm with various heap degrees on 1 GB of data. The experiments were conducted using four different item/key sizes: 32-bit (integer) keys, 32-bit integers with additional 32-bit payload, 64-bit integer keys, and 64-bit keys with 64-bit payload. The payload simulates additional index or pointer associated with the key which refers to additional (possibly large) data properties of the item.

The results indicate, that 2-regular heaps (which are often used as a default) have poor performance. Much better choice are degrees between 3-5, which reduce the height of the tree. On the other hand, the largest (16 B) items have exhibited a slightly different behaviour, which we were unable to explain so far. A more extensive study of this data structure has yet to be conducted. In the following experiments, we have used the optimal heap degree for each situation.

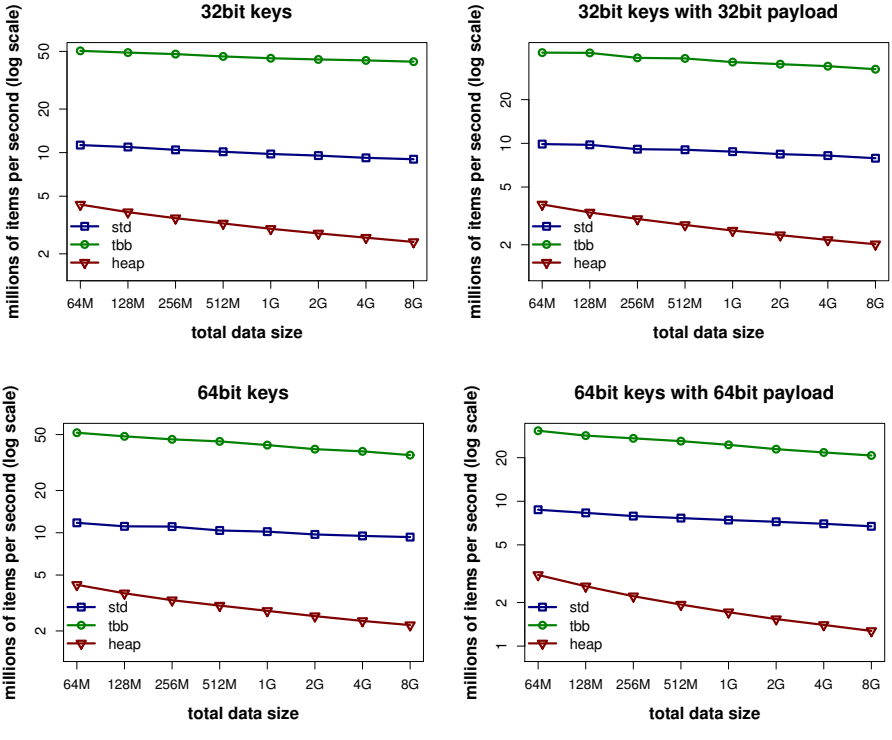

Fig. 3. Internal sorting throughput results

In the internal sorting experiments, we compare the Heapsort algorithm with serial Quicksort implemented in C++ STL library (`std::sort`) and parallel sorting algorithm implemented in the Intel Threading Building Blocks (TBB) library. These algorithms may not be the fastest possible, but they present an etalon, which can be used for further comparisons.

The results depicted in Figure 3 show that the Heapsort is several times slower than the two other methods and its performance degrade more rapidly on larger datasets. Furthermore, the TBB sort is expected to scale with increasing number of cores, but no parallel implementation of Heapsort exists to our best knowledge and we believe that it is not feasible with current CPU architectures.

We have conducted the experiments on the NUMA server as well. The server CPUs have lower serial throughput, which is compensated by the number of cores. Hence, the sorting throughput of TBB algorithm is comparable, but the `std::sort` and the Heapsort exhibit significantly lower performance.

4.2 Sequential Data Reads

The existing external sorting methods are designed to access data sequentially. Hence, our first set of experiments measure the sequential read operations. Let us emphasize, that we use sequential access to the file that holds the data; however, the underlying storage device may fragment the file or otherwise scatter the data.

The following experiments perform sequential reads from binary data files using blocks of fixed size. The data are read using 1-8 threads concurrently, where each thread has its own file in order to avoid explicit synchronization on application level. In every case, the application reads total amount of 64 GB (each thread has an equal share). We have used three APIs for file management – traditional blocking API (*sync*), asynchronous API (*async*), and memory mapped files (*mmap*) – in order to asses their overhead in different situations.

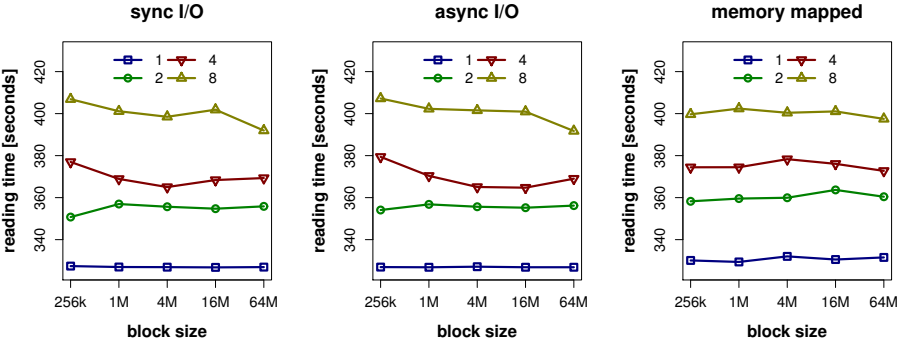

Fig. 4. Reading 64 GB sequentially from a single magnetic hard drive

Figure 4 presents the results measured using commodity hard drive. In all cases, the single-threaded version outperforms the multi-threaded versions, since one thread is capable of sufficiently utilize the drive. Furthermore, the serial reading throughput was reaching 200 MBps for all selected block sizes.

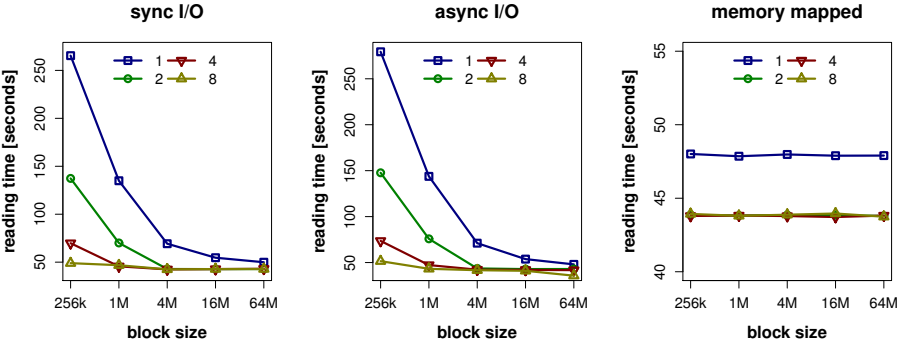

Fig. 5. Reading 64 GB sequentially from 6 SSD disks in RAID 0

Figure 5 presents the results measured on a RAID 0 array of six SSD disks. Since these disks have much greater combined throughput and much lower latency, both sync and async API demonstrate significant overhead on smaller blocks. Furthermore, multiple threads may take advantage of the high processing speed of the disks and achieve better performance. The best reading speed was approaching 1.5 GBps.

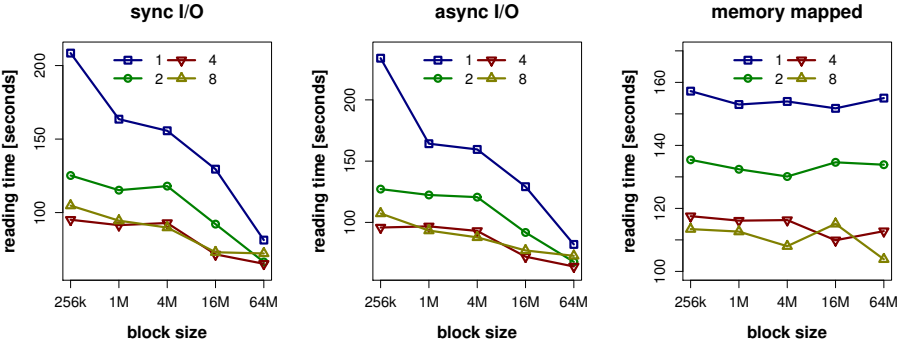

Fig. 6. Reading 64 GB sequentially from an enterprise disk array

The last set of experiments conducted on an enterprise disk array is depicted in Figure 6. The RAID nature of the array provide much better performance than a single drive, but the additional overhead imposed by the data redundancy and internal storage control makes the array less efficient than the SSD drives. The array is also much less predictable, since the I/O scheduling algorithm of the host system is probably clashing with the internal scheduler of the array.

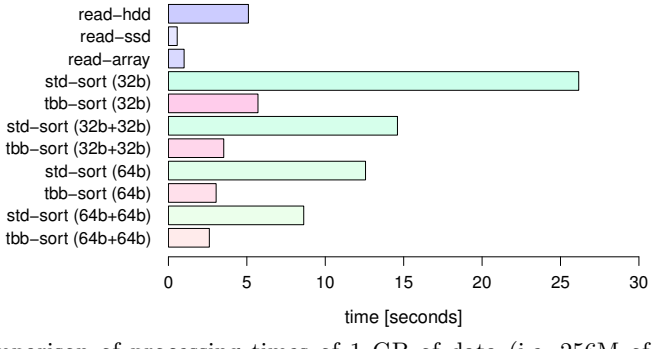

Fig. 7. Comparison of processing times of 1 GB of data (i.e. 256M of 32-bit items, 128M of 32 + 32-bit and 64-bit items, and 64M of 64 + 64-bit items

Finally, let us compare the data reading times with internal sorting times. Figure 7 summarizes the results of reading and sorting operations performed on 1 GB of data. In general, the internal sorting is slower than reading from persistent storage. Furthermore, the reading operation is expected to scale linearly with the data size, while the sorting has time complexity $\Theta(N \log N)$ and so it will get even slower (w.r.t. reading) when larger data buffers are used.

4.3 Random Data Access

In the merging phase, the data streams are usually loaded from the same storage device. In such case, the device must load data from multiple locations in short order. Hence, we would like to compare random access with the sequential access.

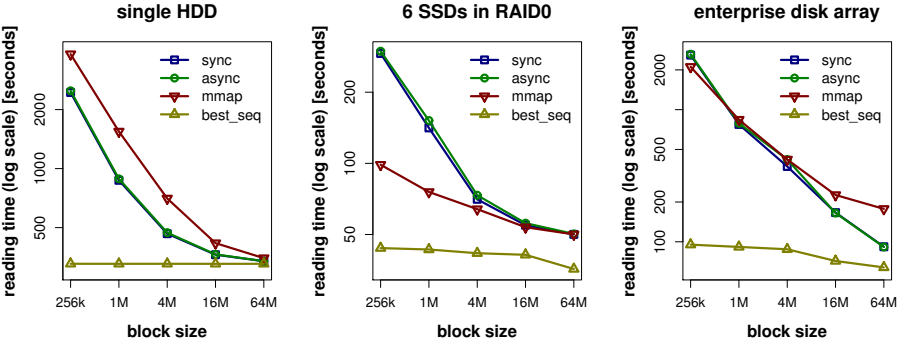

Fig. 8. Reading 64 GB as a random permutation of blocks

Figure 8 presents results of reading times, when the data are loaded in compact blocks, which are accessed in random order. The *best_seq* values are reading times of the best sequential method measured in the previous experiments, so that we can put the random access times in the perspective. The results suggests that in every case, the random access overhead can be minimized if we use sufficiently large blocks. In case of a single HDD, blocks of tens of megabytes are sufficient. In case of faster devices, large blocks are more advisable.

5 Conclusions

The experimental results indicate that using heaps for increasing the length of initial runs has negative impact on the performance. The internal sorting takes more time than data loads, thus the optimization of the internal sorting methods become a priority. Another important fact yielded by the empirical evaluation is that the random access could be nearly as fast as sequential access when the data are transmitted in large blocks. Hence, we can place as many streams as required on the same storage (even in the same file) and access them simultaneously. Based on the evidence, we propose the following:

- The internal sorting used for generating runs should utilize the fastest (parallel) algorithm possible. The length of the runs are no longer first priority.
- Modern servers of the day have hundreds of GB operating memory and tens of TB storage capacity. Hence, if the sorted data fit the persistent storage, the first phase will generate hundreds of runs at most.
- The merging phase should process all generated runs in one step. Current operating systems can work with hundreds separate files and the storage can handle simultaneous (and thus random) access to all these streams without a significant drop in performance.

We have established that traditional methods of external sorting presented by Knuth are obsolete. New methods should focus more on optimizing the internal sorting algorithms and efficient hierarchial merging than on the number of I/O operations. In the future work, we would like to adopt a data stream sorting algorithm of Falt et al. [6] for external merging.

References

1. Beckmann, A., Meyer, U., Sanders, P., Singler, J.: Energy-efficient sorting using solid state disks. Sustainable Computing: Informatics and Systems 1(2), 151–163 (2011)
2. Bertasi, P., Bressan, M., Peserico, E.: psort, yet another fast stable sorting software. Journal of Experimental Algorithmics (JEA) 16, 2–4 (2011)
3. Chhugani, J., Nguyen, A.D., Lee, V.W., Macy, W., Hagog, M., Chen, Y.K., Baransi, A., Kumar, S., Dubey, P.: Efficient implementation of sorting on multi-core simd cpu architecture. Proceedings of the VLDB Endowment 1(2), 1313–1324 (2008)
4. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
5. Demuth, H.B.: Electronic data sorting. Dept. of Electrical Engineering (1956)
6. Falt, Z., Bulánek, J., Yaghob, J.: On parallel sorting of data streams. In: Advances in Databases and Information Systems. pp. 69–77. Springer (2013)
7. Hoare, C.: Quicksort. The Computer Journal 5(1), 10 (1962)
8. Knuth, D.E.: Sorting and Searching. Addison-Wesley (2003)
9. Larson, P.A.: External sorting: Run formation revisited. Knowledge and Data Engineering, IEEE Transactions on 15(4), 961–972 (2003)
10. Merrill, D.G., Grimshaw, A.S.: Revisiting sorting for gpgpu stream architectures. In: Proceedings of the 19th international conference on Parallel architectures and compilation techniques. pp. 545–546. ACM (2010)
11. Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., Lomet, D.: Alphasort: A cache-sensitive parallel external sort. The VLDB Journal – The International Journal on Very Large Data Bases 4(4), 603–628 (1995)
12. Rasmussen, A., Porter, G., Conley, M., Madhyastha, H.V., Mysore, R.N., Pucher, A., Vahdat, A.: Tritonsort: A balanced large-scale sorting system. In: Proceedings of the 8th USENIX conference on Networked systems design and implementation. pp. 3–3. USENIX Association (2011)
13. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore gpus. In: Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. pp. 1–10. IEEE (2009)
14. Vasudevan, V., Tan, L., Kaminsky, M., Kozuch, M.A., Andersen, D., Pillai, P.: Fawnsort: Energy-efficient sorting of 10gb. Sort Benchmark final (2010)
15. Vengroff, D.E., Scott Vitter, J.: Supporting i/o-efficient scientific computation in tpie. In: Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on. pp. 74–77. IEEE (1995)
16. Vitter, J.S.: External memory algorithms and data structures: Dealing with massive data. ACM Computing surveys (CsUR) 33(2), 209–271 (2001)
17. Williams, J.W.J.: Algorithm-232-heapsort. Communications of the ACM 7(6), 347–348 (1964)