Designing an energy-efficient, learning-enhanced algorithm to sort 1TB of ASCII data

Ani Kristo[1], Padmanabhan Pillai[3], and Tim Kraska[2]

[1] Brown University
[2] MIT
[3] Intel Labs

1 Introduction

We present ELSAR, our submission for the 2022 JouleSort 1TB Indy benchmark. Unlike the traditional external merge-sort framework, ELSAR partitions records based on non-overlapping and monotonous ranges of their CDF[4] values. Then, the partitions are sorted using LearnedSort 2.0 - an in-memory sort that has shown great results in numerous evaluations[11]. Finally, the partitions records are simply concatenated consecutively onto a single output file, which is much faster than the traditionally used k-way merge.

In terms of the hardware, we use a desktop computer running on an Intel® Core™ i5 processor with 32GB of RAM and four SSDs. As a result, ELSAR achieves a rate of **158,951 sorted records per Joule**, which is 1.4× the current record held by KioxiaSort[14] (see Figure 1).

In the remainder of this report, we describe the sorting algorithm in detail, list the system components that we use, explain our evaluation method, and present the results.

2 The ELSAR algorithm

ELSAR stands for External LearnedSort for ASCII Records. It is a learning-enhanced, data distribution-based, external sorting algorithm that sorts ASCII records by leveraging small, accurate, and fast linear models. The algorithm combines various techniques to achieve high sorting rates, such as sample-based distribution learning, numerical encoding for ASCII keys, and parallel, buffered, and lock-less file I/O. The central idea of ELSAR is shuffling input records in mutually exclusive, monotonic, and equi-depth partitions. Once sorted, they can simply be concatenated to form an output file, avoiding the expensive merging routine. The sorting algorithm is shown in Figure 2.

[4] The Cumulative Distribution Function (CDF) is calculated as the probability $P(X \leq x)$, where x is a key in the input, and X is the random variable associated with the all the input keys.

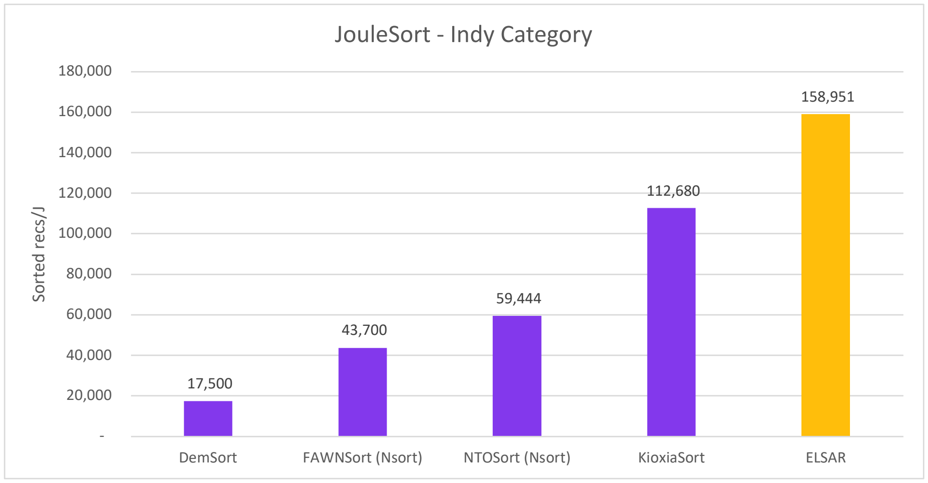

Fig. 1: The energy efficiency (in sorted records per Joule) of the ELSAR algorithm as compared to previous winners of the JouleSort 1TB Indy category[14,8,13,7]

2.1 Model training

Initially, the algorithm reads 10GB of the input file in memory and picks 10% of the records uniformly-at-random to form a training set for the prediction model. The model uses an RMI architecture[9] and approximates the distribution function (CDF) of the input dataset, thus estimating the position of a particular record in the final sorted order. The RMI architecture consists of layered linear models in a directed acyclic structure whose leaf nodes output a value $x \in [0, 1]$ that represents the percentile rank of the record among the estimated population. The algorithm will use the model's rank predictions to shuffle records into logical partitions so that the partitions will eventually be evenly-sized, regardless of the input's distribution skew.

It is important to note that, even if the input was drawn from a uniformly-distributed population, we are interested in modeling the *empirical* distribution, i.e., the distribution of the *observed* data, rather than the theoretical one. The difference is that the observed sample does not follow a smooth distribution that could be modeled with a single linear model. At a fine scale, the data will behave like a step-function, with more structure, noise, and irregularities. That is why it is necessary to use a more complex architecture than a single linear model, which can capture and encode the empirical distribution's subtle characteristics.

The original paper[10] provides a detailed explanation of the CDF model's architecture, complexity, training algorithm, and analysis.

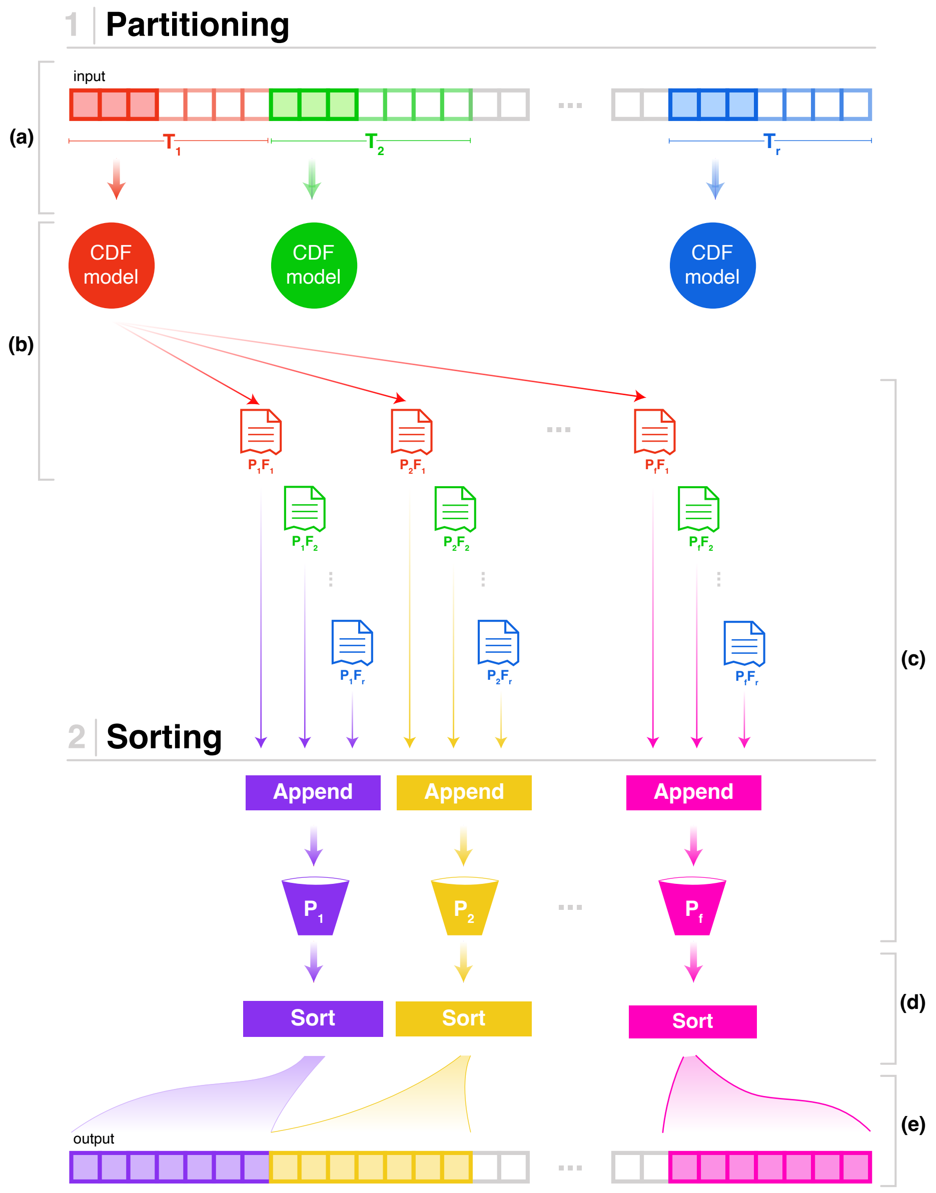

Fig. 2: The ELSAR algorithm.

2.2 Encoding ASCII records

In order to model the distribution of the input, the ASCII records have to be projected onto a numerical space on which the CDF model can make linear regression calculations to predict the rank of the records. Therefore, the algorithm operates on the key's numerical encoding and a pointer to the ASCII record while in memory.

The encodings are calculated using the binary values of each character in the key represented as base-95 numbers. Since printable ASCII characters have codes between 32 and 127, the encoding of a character in position x_i of the key of length l $(1 \leq i \leq l)$ is $(\text{ASCII}(x_i) - 32) \times 95^{l-i}$. Then, the numerical encoding of the entire key is:

$$\sum_{i=1}^{l} \left((\text{ASCII}(x_i) - 32) \times 95^{l-1}\right)$$

With a 64-bit primitive type, we can encode up to the 9^{th} byte of the key. Nevertheless, the LearnedSort routine has an Insertion Sort-based touch-up step that performs last-mile sorting (on the 10^{th} byte). This step also handles prediction inaccuracies from the CDF model and is quite fast since Insertion Sort works very well for nearly-sorted arrays[10].

2.3 Partitioning

Then, ELSAR spawns r threads, each responsible for reading a specific, non-overlapping range of records from the input file (Figure 2a). The best choice for the value of r is the number of cores in the processors (10 in our case). Each thread reads the input records in batches of 1MB at a time. Each of these reader threads initializes a set of f thread-local partition fragments, one for each logical partition. For an average partition size of 1GB, each thread i will have to create 1000 temporary fragment files $(P_1F_i, P_2F_i, ..., P_{1000}F_i)$. The set of fragments across all reader threads (e.g., $P_1F_1, P_1F_2, ..., P_1F_{10}$) forms a logical partition.

Next, the algorithm processes the keys in the batch through a trained CDF model, which estimates the rank of a record among all records (Figure 2b). Each reader thread has its read-only copy of the model. Then, threads distribute the records into their predicted partitions using the model's output.

After the reader threads have processed their batches, they append the records in each fragment to their corresponding temporary files and continue reading the next batch of records. This results in 10K temporary files, each having an average size of 100MB. Note that each reader thread maintains a thread-local partition fragment instead of concurrently writing to the same physical partition file, to avoid using mutexes and locks, which are costly. Since the reader threads have mutually disjoint working sets, we can perform file I/O with the non-locking versions of the STDIO library (i.e., `fread_unlocked()`). These functions omit the file pointer's lock check and run slightly faster.

2.4 Sorting partitions

After the entire input file has been read, ELSAR creates s threads, each responsible for sorting a logical partition. The number s is calculated as the minimum of 1) the number of cores and 2) the maximum number of partitions that can be in memory simultaneously. In our system $s = 10$. The sorter threads read all the fragment files that belong to the logical partition assigned to them and append their records into a single large buffer (Figure 2c).

Then, the algorithm calls LearnedSort 2.0[11] as an in-memory sorting routine for the partition buffer (Figure 2d). LearnedSort also uses a CDF model to learn the distribution of the partition data and quickly sort the records. Since we are dealing with ASCII keys, ELSAR first encodes them into a numerical representation which can be used to train the CDF model. At this stage, the algorithm only uses the numerical encodings (8 bytes) and pointers to the records (4 bytes) for the sorting procedure, hence reducing the size of memory copying and moving operations, especially for such large records (otherwise, 100 bytes).

Finally, after the partition contents have been sorted, they are concatenated sequentially with the neighboring partitions' records to form a single continuous output file (Figure 2e). Each sorter thread maintains an open descriptor for the output file, and, for each partition i that it will flush, it seeks to the offset location pre-calculated as the sum of the sizes of the partitions $[1..(i-1)]$.

Since the sorted buffer contains only encodings and pointers to the records, it is not possible to flush the entire buffer in one sequential call. Therefore, the thread first coalesces the records in batches of 100KB by dereferencing the sorted pointers in order. Then, it performs a buffered write of the coalesced buffer using `fwrite_unlocked()`, thus optimizing the write performance.

3 The hardware

Table 1: Summary of hardware components for ELSAR.

Type	**Part Name**	**Qty.**	**Cost**
Motherboard	ASUS PRIME Z690-A	1	\$279.99
CPU	Intel® Core™ i5-12600K	1	\$278.99
CPU Cooler	Noctua NH-L12S	1	\$54.95
Memory	G.Skill Trident Z5 16GB	2	\$759.98
Storage	WD_BLACK™ SN850 2TB	4	\$1,199.96
Power Supply	Corsair SF Series® SF450	1	\$138.89
Case	NZXT H510 Mid-Tower	1	\$89.99
Total			**\$2,802.75**

In Table 1 we have listed the hardware components that we use to perform the benchmark measurements and their associated cost in USD. All parts are

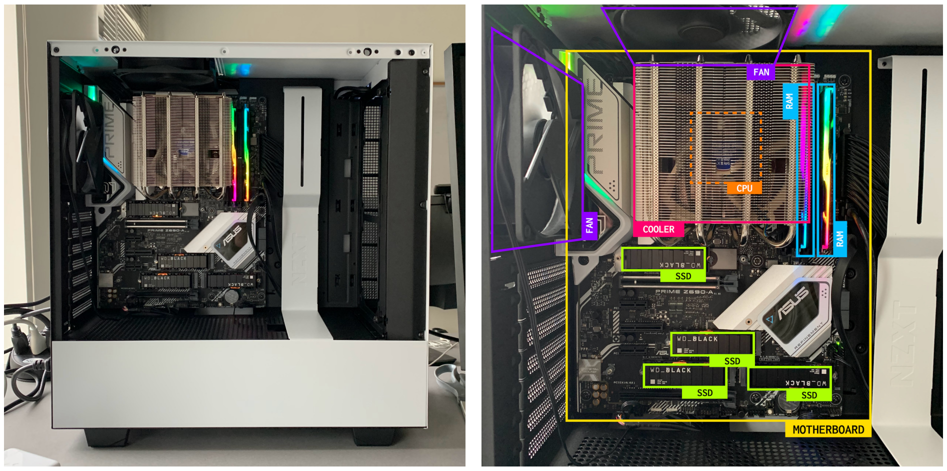

Fig. 3: Photos of the desktop computer used for ELSAR.

available for retail in all major online marketplaces. Below we describe the hardware in more detail.

Processor We chose to use the Intel® Core™ i5-12600K as it belongs to the newest, 12th generation processors, and provides a great balance between performance and power consumption. While some other desktop CPUs, such as Intel® Core™ i7 and Intel® Core™ i9 series processors, are faster, they also consume more power. In contrast, the Intel® Core™ i5 series processors have demonstrated the lowest energy readings among 12th generation CPUs in several benchmarks [6].

An exciting feature of these new-generation processors is the hybrid architecture that utilizes high-performance cores (P-Cores) and power-efficient ones (E-Cores). The i5-12600K processor has ten cores (6P + 4E) and 16 threads (E-Cores do not support SMT). The P-Cores can achieve speeds up to 4.90GHz, whereas the E-Cores up to 3.70GHz, operating at a maximum power of 150W. Our software currently does not specifically exploit differences in the cores, though, and simply utilizes 16 threads on the processor.

Note that our processor is the clock unlocked variant. However, we have disabled overclocking in the BIOS. At the time of this report, the base model (i5-12600) is not available in retail in the US.

Memory This generation of processors also introduces support for DDR5 memory, which can achieve 1.5$\times$ higher speeds than its predecessor, DDR4. This new generation memory offers improved performance with greater power efficiency[2]. Therefore, we decided to use the G.Skill Trident Z5 DDR5 RAM with a total capacity of 32GB, operating at 4800MT/s in our system.

Storage Given that external sorting is disk IO-bound, investing in high-speed storage technology is paramount. This does not come at the expense of higher power consumption since M.2 NVMe™ SSDs wattage is roughly the same as the mSATA and 2.5" counterparts, while still operating at less than 5W[15]. We used four WD_BLACK™ SN850 2TB SSDs, which can reach sequential read speeds of up to 7000MB/s, and sequential writes up to 5100 MB/s[5].

Power Supply When choosing the power supply unit, an important factor is the 80 Plus™ efficiency rating. We chose Corsair SF Series® SF450 80 Plus™ Platinum-certified, which is a fully-modular, high-end PSU that is said to achieve an energy efficiency of 92% on the 100-200W range[12] where our system operates. For the same workloads, a lower-efficiency-rated PSU would draw more power from the wall than the Corsair SF450, increasing the system's total energy consumption. Our PSU's maximum power is 450W, which is well above the observed maximum of 170W during the sorting task.

Motherboard Our motherboard's model is ASUS PRIME Z690-A, which is compatible with the processor's LGA 1700 socket and supports DDR5 memory. It has an ATX form factor with four PCIe® Gen4 M.2 slots.

CPU Cooler Since the processor does not have a stock cooler, we purchased the Noctua NH-L12S. This is a single-fan, 120mm cooler geared towards compact builds, thus providing high-efficiency cooling at less than 2W[3].

Case Finally, all the components are mounted onto an NZXT H510 Mid-tower case compatible with ATX boards. The case has two Aer F 120mm fans, one for intake and one for exhaust. Each of these fans consumes less than 2W[1].

The total cost of the system is \$2,802.75, which represents a 14.7% decrease from KioxiaSort's build cost (\$3,286.66[5]), and the lowest cost to date among all JouleSort entries in the last ten years (see Figure 4).

4 System Configuration

Table 2 summarizes the system configuration of ELSAR. The computer runs on a Ubuntu 20.04 operating system with a text-only interface. As per the benchmark requirements, we have disabled Intel®'s Extreme Memory Profile (XMP) in the BIOS, which would otherwise allow CPU and memory overclocking. The code is written in C++ and compiled with g++ 9.4.

[5] We estimated KioxiaSort's build cost based on the lowest available prices on online US marketplaces and the current conversion rate of 1USD = 122JPY for the parts only available from retailers in Japan.

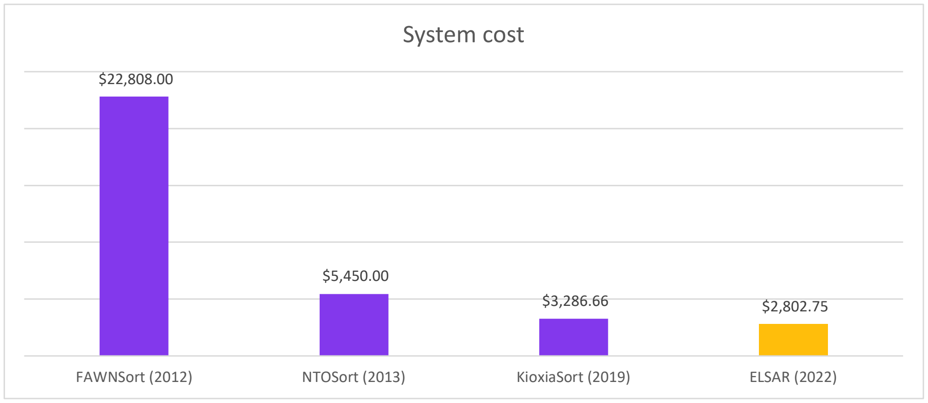

Fig. 4: System cost comparison with previous winners of the JouleSort Indy Category benchmark (in USD)[14,8,13]

Table 2: System configuration for ELSAR.

Description	**Value**	**Notes**
Intel® XMP	Disabled	
OS	Ubuntu 20.04.4 LTS (focal)	
Kernel	5.4.0-105-generic	
Volume Manager	LVM 2.03.07(2)	2 LVs $\times$ 2 PVs each
Logical Volumes	`/data`, `/tmp-data`	Striped (size = 64KiB)
File system	XFS V5	Opts: `defaults, noatime`
Compiler	g++ 9.4.0	Opts: `-Ofast -march=native`

In order to maximize the disk IO bandwidth, we use RAID-0 with the Linux Volume Manager (LVM). As shown in Figure 5, all four SSDs are connected via PCIe® Gen4 M.2 connectors to the CPU socket. We separated the four physical volumes into two volume groups (`vg0` and `vg1`), each containing a logical volume (`vg0/lvol0` and `vg1/lvol0`) with a stripe size of 64KiB. Both logical volumes use the XFS file system, and they are of size 3.45TiB and 3.15TiB, respectively, leaving some space for the boot and root partitions. We use `vg0/lvol0` (mounted on `/data`) for the input and output files, and `vg1/lvol0` (mounted on `/tmp-data`) for temporary files created during the sort. The input file will be read and temporaries created during the first phase, while the temporaries are read and the the output file is written during the second phase of the algorithm. This drive configuration ensures the drives do not experience concurrent reads and writes.

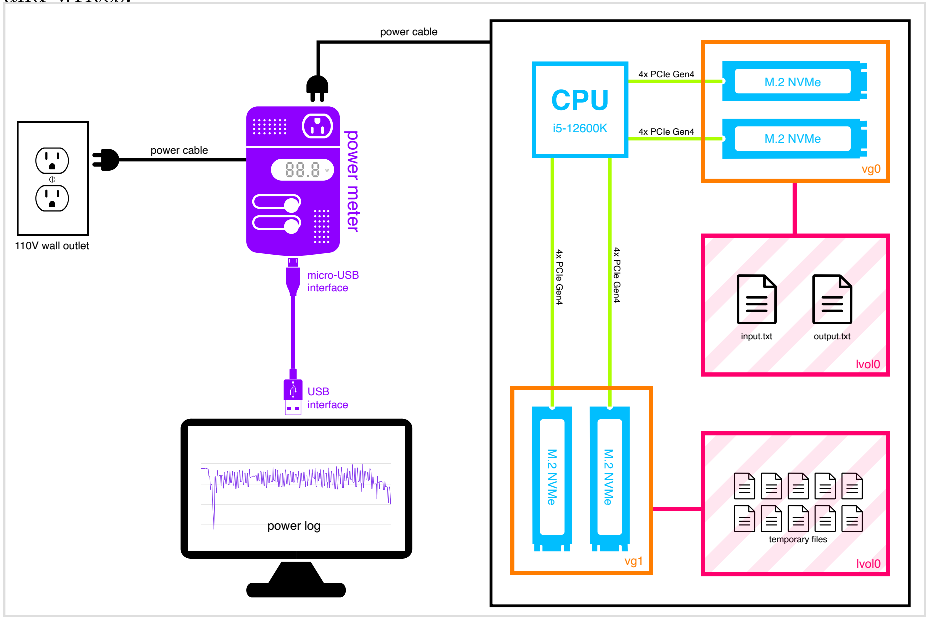

Fig. 5: The system configuration of ELSAR.

5 Measurement procedure

5.1 Power logging

We used the Watts Up! PRO power meter as several previous JouleSort submissions. The meter logs every 1 second with 0.1W precision and 1.5% accuracy[4]. As shown in Figure 5, the computer is plugged into the power meter, which is, in turn, plugged into the wall. The meter is connected to a separate monitoring computer via a USB cable. The monitoring computer uses a publicly-available Python utility[16] that reads the meter's data from the given serial port and saves them to a CSV file. Each reading is associated with the respective timestamps in the log file. Both machines use NTP and are synchronized to `time.nist.gov`.

5.2 Input file generation

We generate the input file only once at the experiments' beginning and reuse it for each run. Per the requirements, we use the `gensort` program with the `-c` option to display the checksum and the `-a` option to generate ASCII keys. The checksum obtained for the 1TB file is `12a06cd06eeb64b16`.

5.3 Performing measurements

We perform five consecutive runs to measure the energy consumption. The measurements follow the process below:

1. Memory cache is cleared using the command:
   ```
   $ sync; echo 1 > /proc/sys/vm/drop_caches;
   ```
2. The execution is paused for 5 seconds to allow the computer to go back to idle power consumption after the cache clearing.
3. The current timestamp is displayed.
4. The ELSAR algorithm is executed using the `time` utility which will report the total elapsed time after the sorting process terminates.
5. The current timestamp is displayed again. (These timestamps are necessary for accurately calculating the average power later on.)
6. The output's checksum and sortedness are verified with the `valsort` program. For a successfully sorted output file, valsort should report no duplicate keys, the same checksum as the input (`12a06cd06eeb64b16`), and the following message: `SUCCESS - all records are in order`.
7. After verification, the output file is deleted from the disk, and all unused blocks in the filesystem are discarded using the `fstrim -a` command.

5.4 Calculating energy consumption

For each run, the monitoring computer generates a separate power log file. We calculate the mean power of each run using all the rows in the CSV file whose timestamps are between the starting and ending timestamps displayed in Steps 3 & 5 above, exclusive. We do not include the readings that correspond to the starting and ending timestamps above in order to avoid fractional readings.

Finally, we obtain the energy consumption of each run by multiplying the mean power with the execution time (in seconds) reported by the `time` utility. The results are shown in the next section.

Table 3: ELSAR performance results

Run	**Time**	**Power**	**Energy**
Run 1	617.4s	101.82W	62,862J
Run 2	623.8s	100.79W	62,869J
Run 3	620.9s	101.71W	63,153J
Run 4	618.3s	102.42W	63,325J
Run 5	610.4s	102.14W	62,346J
Mean	**618.1s**	**101.78W**	**62,912J**
Stdev	**5.0s**	**0.62W**	**372J**

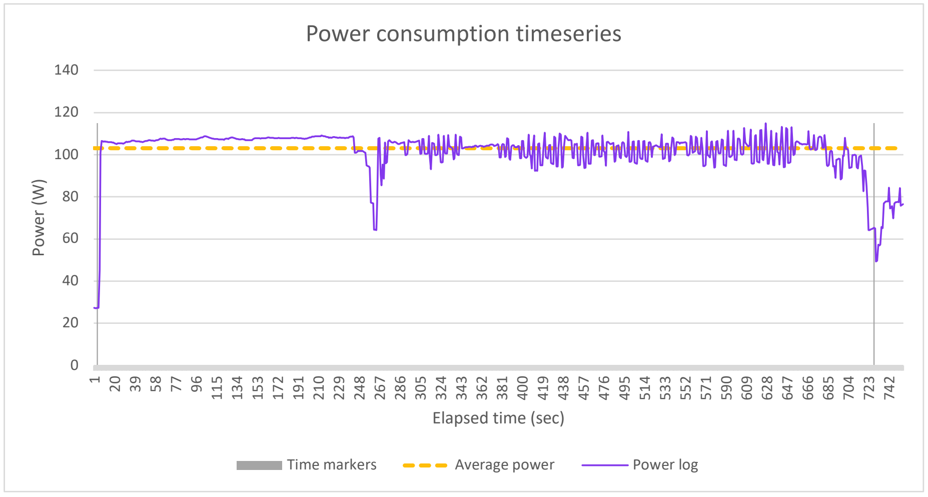

Fig. 6: A typical example of the power measurements at every second of the execution. The mean power is shown in the dashed yellow line, and the vertical grey lines indicate the timestamps for the beginning and end of the execution.

6 Results

The execution times, average power, and energy consumption of all five runs are displayed in Table 3. Figure 6 displays a run's typical power readings at 1-second intervals. The mean execution time is 618.1 seconds, and the mean power is 101.78W, hence **resulting in an average of 62,912J** ±372 **for sorting the 1TB ASCII dataset**. This represents a 29.1% decrease from the current winner of the JouleSort Indy Category (KioxiaSort).

On the other hand, Figure 7 shows the break-down of the time required by each phase of the algorithm, as well as their proportional energy consumption. The *Load* phase refers to the read operation performed by each reader thread, which brings a batch of records to memory. Whereas, *Combine* refers to the operation that brings together the records across fragment files that belong to the same partition (i.e., PxF0, PxF1, . . . , PxFr=999), which will be used by the sorter threads. The CDF model training only accounts for a small fraction of the algorithm (<1%), whereas the biggest time and energy consumer is the partitioning phase that splits the input file onto multiple fragments for each logical partition. Record coalescing takes up roughly 7%, however, it is an optimization that brings down the total execution time of the algorithm, specifically helping reduce the Concatenation phase.

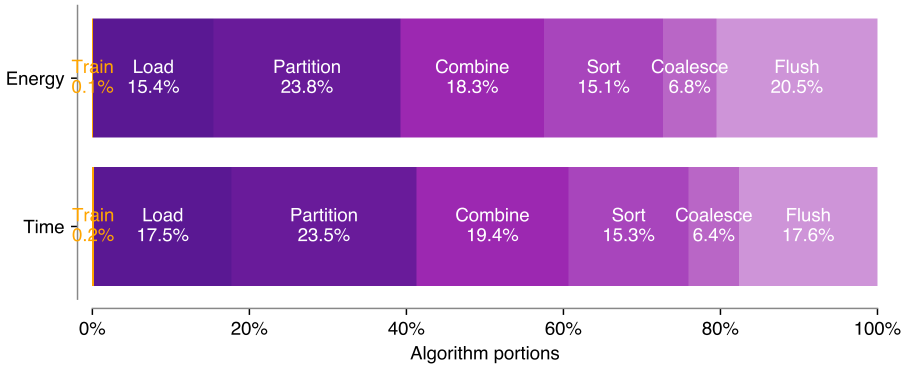

Fig. 7: A break-down of the time and energy consumption for each phase of the ELSAR algorithm.

7 Summary

We presented ELSAR, a new external sorting algorithm for the 1TB JouleSort Indy dataset that works by partitioning records based on their keys' CDF values, thus bypassing the typical file merging phase employed by the typical external sorting algorithms. Our evaluations indicate 29.1% more energy savings than the current winner of the this benchmark, hence setting a new record in this category.

References

1. Aer F 120mm Specifications, `https://nzxt.com/product/aer-f-120mm`
2. JEDEC DDR5 & NVDIMM-P Standards Under Development, `https://www.jedec.org/news/pressreleases/jedec-ddr5-nvdimm-p-standards-under-development`
3. NH-L12S || Specifications, `https://noctua.at/en/nh-l12s/specification`
4. Watts Up! PRO Specifications, `http://www.idlboise.com/sites/default/files/WattsUp_Pro_ES.pdf`
5. WD_BLACK SN850 NVMe™ SSD Specifications, `https://www.westerndigital.com/products/internal-drives/wd-black-sn850-nvme-ssd`
6. Alcorn, P.: Intel 12th-Gen Alder Lake Pricing, Benchmarks, Specs and All We Know, `https://www.tomshardware.com/news/intel-alder-lake-specifications-price-benchmarks-release-date`
7. Beckmann, A., Meyer, U., Sanders, P., Singler, J.: Energy-Efficient Sorting using Solid State Disks. Tech. rep. (2010)
8. Ebert, A.: NTOSort. Tech. rep. (2013)
9. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The Case for Learned Index Structures. In: Proceedings of the 2018 international conference on management of data. pp. 489–504 (2018)
10. Kristo, A., Vaidya, K., Çetintemel, U., Misra, S., Kraska, T.: The Case for a Learned Sorting Algorithm. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 1001–1016 (2020)
11. Kristo, A., Vaidya, K., Kraska, T.: Defeating duplicates: A re-design of the LearnedSort algorithm (2021)
12. Mpitziopoulos, A.: Corsair SF450 Platinum SFX PSU Review: Best of the Best, `https://www.tomshardware.com/reviews/corsair-sf450-platinum-sfx-psu,5917-5.html`
13. Pillai, P., Kaminsky, M., Kozuch, M.A., Andersen, D.G.: FAWNSort: Energy-efficient Sorting of 10GB, 100GB, and 1TB. Tech. rep. (2012)
14. Sano, S., Mahmoud, Z., Suzuki, T.: KioxiaSort: Sorting 1TB by 89K Joules. Tech. rep. (2019)
15. Webster, S.: WD Black SN750 SE SSD Review: Cost-Effective Storage for Gamers, `https://www.tomshardware.com/reviews/wd-black-sc570-se-review/2`
16. Yoon, Y.: Watts Up! power logging software (2015), `https://github.com/anikristo/wattsup`