Scalable Compression of Massive Data Collections on HPC Systems

Loris Belcastro[1]✉[0000−0001−6324−8108], Paolo Ferragina[2,3][0000−0003−1353−360X], Giovanni Manzini[2][0000−0002−5047−0196], Fabrizio Marozzo[1][0000−0001−7887−1314], Domenico Talia[1][0000−0003−1910−9236], and Paolo Trunfio[1][0000−0002−5076−6544]

[1] DIMES, University of Calabria, Rende, Italy
[2] Department of Computer Science, University of Pisa, Pisa, Italy
[3] Department of L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy
✉ Corresponding author: lbelcastro@dimes.unical.it

Abstract. The exponential growth of digital data poses a significant storage challenge, straining current storage systems in terms of cost, efficiency, maintainability, and available resources. For large-scale data archiving, highly efficient data compression techniques are vital for minimizing storage overhead, communication efficiency, and optimizing data retrieval performance. This paper presents a scalable parallel workflow designed to compress vast collections of files on high-performance computing systems. Leveraging the Permute-Partition-Compress (PPC) paradigm, the proposed workflow optimizes both compression ratio and processing speed. By integrating a data clustering technique, our solution effectively addresses the challenges posed by large-scale data collections in terms of compression efficiency and scalability. Experiments were conducted on the Leonardo petascale supercomputer of CINECA (leonardo-supercomputer.cineca.eu), and processed a subset of the Software Heritage archive, consisting of about 49 million files of C++ code, totaling 1.1 TB of space. Experimental results show significant performance in both compression speedup and scalability.

Keywords: Data Compression · HPC · Parallel Computing · Distributed Processing · Big Data.

1 Introduction

The exponential growth of digital archives, such as repositories containing software development activities, has led to an unprecedented accumulation of huge amounts of source code data. Among the efforts to collect, preserve, and share all publicly available source code, the Software Heritage project[4], initiated in 2016 by INRIA, has emerged as a pivotal initiative. The project aims to create a universal archive of source code, ensuring its long-term availability as a crucial component of our cultural and technological heritage [1,9]. As of January 2025, the Software Heritage Archive includes over 22 billion unique source

[4] See https://www.softwareheritage.org

code files from more than 346 million collaborative development projects. This vast and continually expanding archive underscores the necessity for highly efficient data compression techniques that can handle massive volumes of source code while minimizing storage overhead and optimizing retrieval performance. In addition, high-performance computers, along with appropriate programming models, parallel and distributed algorithms, and frameworks, are essential for efficiently processing and analyzing big data [19].

Traditional compression methods, which apply a single compression algorithm to an entire dataset, fail to exploit inherent (and often occurring) structural redundancies across software projects. To address this challenge, we present a parallel data compression workflow designed to leverage high-performance computing (HPC) systems for the efficient compression of large-scale source code datasets. Our approach builds upon the *Permute-Partition-Compress* (PPC) paradigm [7,8,12,6], which enhances the compressibility of (un)structured data by reordering and segmenting it before applying a commodity compressor (such as Zstd, Brotli, LZMA, etc.). Extensive experiments have been carried out on Leonardo, the petascale supercomputer of CINECA[5], to evaluate the efficiency and scalability of our parallel and distributed compression solution. This paper reports the impressive results we obtained in terms of compression ratio (i.e., the ratio between the compressed size and the original size, which measures how much the data is reduced during compression) and scalability.

The remainder of the paper is structured as follows. Section 2 provides a brief review of related works in the domain of data compression approaches for large collections of data. Section 3 presents the ParSoDA library that has been leveraged to build a parallel and distributed data compression workflow, tailored to run on HPC systems. Section 4 reviews the PPC paradigm that provides an improvement over traditional single-pass compression techniques, allowing better exploitation of data structure and redundancy. Section 5 illustrates the proposed methodology based on a parallel compression workflow. Section 6 evaluates the performance of the proposed solution. Finally, Section 7 draws conclusions and outlines avenues for future research.

2 Related work

In many contexts, efficiently compressing data is crucial for reducing storage requirements and improving retrieval performance. The problem of compressing a collection of files or blobs has been extensively explored in different domains, including web pages [12], genome repositories [21,13], file and storage systems [14], and individual source code repositories [16,15,22,5].

A widely adopted approach for exploiting inter-file redundancies consists of concatenating all files into a single stream and then applying a general-purpose compression algorithm, such as Gzip, which operates by encoding recurring data fragments detected within a fixed-size sliding window that is optimized for minimal memory usage (typically under 1MB) and thus captures repetitions within

[5] See `http://leonardo-supercomputer.cineca.eu`

short distances. More advanced methods, including Google Brotli[6], LZMA[7], Apple LZFSE[8], and Meta Zstd[9], have been recently proposed to efficiently detect and encode redundancies spanning hundreds or even thousands of MBs.

However, despite their improvements, these state-of-the-art compressors remain inadequate when dealing with large-scale document collections, such as the Software Heritage archive, where repeated patterns may be distributed over vast distances. To overcome this limitation, researchers have proposed three specialized approaches specifically designed to enhance compression efficiency in such large datasets.

The first approach involves using a more powerful compressor that is not constrained by a fixed window or block size. A notable example is the disk-based construction of the Burrows-Wheeler Transform (BWT)[11], which can detect repetitions at unlimited distances while using limited internal memory. However, this method is computationally expensive in both compression and decompression. Despite its high cost, it has demonstrated strong compression effectiveness, as shown for web page collections in [12].

The second approach is *delta-compression*, where a file is encoded by referencing and copying portions of another file. The more similar two files are, the more compact the delta encoding. In large collections, this requires optimally assigning reference files, which can be modeled as finding the best branching in a directed graph, where nodes represent files and edge weights represent compression benefits [18]. Since this approach has a quadratic time complexity, heuristics have been developed to improve scalability [10,17]. A particularly effective method [17] clusters syntactically similar files using MinHash and shingles for similarity detection or Locality-Sensitive Hashing (LSH) for more efficient, albeit non-optimal, grouping.

The third approach, *Permute-Partition-Compress* (PPC)[8], reorganizes files before compression to enhance redundancy detection within general-purpose compressors. It consists of three steps: (*i*) permuting files so that similar ones are adjacent, (*ii*) partitioning the reordered collection into manageable blocks, and (*iii*) compressing each block using SOTA-tools like Zstd and Brotli. This method enables these compressors to use limited memory footprints but still be able to capture long-distance redundancies. For web page collections, simple URL-based ordering heuristics have proven effective, achieving compression rates of up to 4% of the original file size [12].

These approaches have been tested on relatively limited datasets and no parallel or distributed methods have been proposed for compressing massive collections of files. For example, the PPC approach has been evaluated on a sequential machine using up to 200 GB. The main challenge stems from the limited capability of a single disk controller to handle parallel I/O operations on billions of files. Addressing this issue requires specialized distributed and

[6] `https://github.com/google/brotli`
[7] `https://www.7-zip.org/`
[8] `https://github.com/lzfse/lzfse`
[9] `http://facebook.github.io/zstd/`

parallel file systems, such as Lustre or HDFS, combined with efficient parallel and distributed data compression approaches. This paper presents a parallel data compression workflow designed to leverage high-performance computing (HPC) systems for efficiently compressing large-scale source code datasets. In particular, it builds upon the PPC paradigm showing significant performance results in both compression speedup and scalability over a subset of the Software Heritage archive, consisting of about 49 million files containing C++ code, totaling 1.1 TB of space.

3 ParSoDA

ParSoDA (Parallel Social Data Analytics) [4] is a programming library designed to facilitate the development of parallel data analysis and machine learning workflows in high-performance computing systems. Originally implemented in Java, ParSoDA abstracts away the complexity of managing the underlying parallel infrastructure, including built-in scalability mechanisms and support for popular frameworks like Apache Hadoop and Spark.

Building on the Java-based implementation, ParSoDA-Py [3], the Python version of the library, introduces significant improvements in usability, flexibility, and runtime integration. ParSoDA-Py enables seamless execution of applications on various distributed runtimes for HPC systems, such as COMPSs [20], Apache Hadoop and Spark. ParSoDA-Py delegates the execution to an abstract driver, which introduces a new level of abstraction that separates the application code from the underlying execution runtime, improving usability (e.g., the same code can be executed on a different runtime by simply changing the driver in use) and enhancing compatibility with an extensible set of runtimes. The introduction of a specific driver for local multicore setup (i.e., *ParsodaMultiCoreDriver*) enhances the support for multicore environments, including HPC systems, enabling the efficient execution of large-scale data analysis workflows. By abstracting the execution environment, ParSoDA-Py simplifies the transition between different computational setups, allowing users to write applications once and deploy them on local machines, HPC systems, or cloud platforms without modification.

To further optimize performance, ParSoDA-Py includes advanced data crawling and partitioning capabilities. These features ensure that workloads are efficiently distributed, reducing bottlenecks and fully utilizing the resources available in multicore and distributed environments. This design not only improves scalability but also makes the library suitable for dynamic and heterogeneous parallel systems, where resource availability may vary.

As shown in Figure 1, for each block, developers are required to specify one or more functions. Specifically, ParSoDA-Py defines a general workflow for data analysis applications that is composed by the following steps:

- *Data acquisition*: it is possible to run multiple crawlers in parallel; the collected data items are parsed and converted in an internal format suitable to be processed by the next steps. The driver of a specific runtime will take

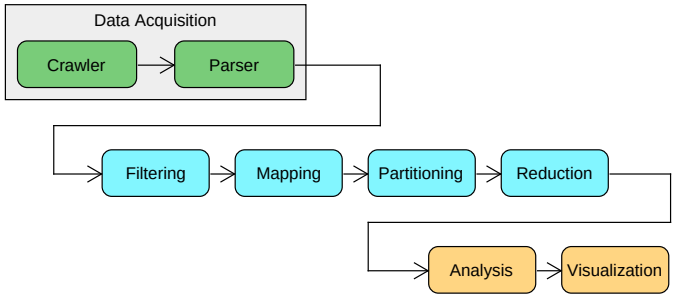


Fig. 1: Execution flow of ParSoDA-Py

care, under the hood, of partitioning the data into chunks and distributing them to workers for parallel processing. ParSoDA-Py also introduces *Parser* classes to convert data items into a structured format. These classes can be easily created or extended by programmers, enabling them to simultaneously read data from various sources and transform them into a standardized object format.

- *Data filtering*: this step applies a set of filtering functions to refine a dataset by removing unwanted or irrelevant data.
- *Data mapping*: each data record is transformed through a series of mapping functions, modifying or augmenting its attributes as needed.
- *Data partitioning*: source data are divided into logical shards based on a primary key (*groupKey*) and then sorted within each shard using a secondary key (*sortKey*) following a secondary sort pattern, which ensures a more efficient processing in later stages.
- *Data reduction*: within each partition, records are aggregated according to a specified reduction function.
- *Data analysis*: after reduction, data is further processed using a given data analysis function to extract insights and meaningful patterns.
- *Data visualization*: in this final step, visualization techniques are applied to present the processed data in a clear and interpretable format.

Some steps, such as data analysis, visualization, and filtering, are optional and may be skipped depending on the specific use case.

4 The PPC paradigm

The PPC (Permute-Partition-Compress) paradigm was introduced in [7,8,12,6] as an improvement over traditional single-pass compression techniques, allowing better data redundancy exploitation. Instead of applying a compressor directly to a dataset, PPC employs a three-stage preprocessing procedure:

- *Permute*: files are reorganized to increase their compressibility by clustering (syntactically) similar elements together. For source code, this may involve grouping files based on filename, syntax similarity, programming language, dependency structure, or code reuse patterns across projects. The goal is to enhance locality in the data, making common patterns more frequent in smaller segments. In a comparative study, the filename-based clustering, though simple, was shown to be effective for compressing code files [6].
- *Partition*: the permuted files are then split into chunks, which are compressed individually. Unlike naive block-based approaches, PPC ensures that each partition contains homogeneous data, resulting in higher compression efficiency. The optimal partitioning strategy is often NP-hard, but heuristic and approximation algorithms can be used to achieve near-optimal segmentation in feasible time.
- *Compress*: at this stage, a standard compression algorithm (e.g., Gzip, Zstd, or Brotli [2]) is applied to each chunk. Due to structured reordering and intelligent partitioning, the base compressor achieves significantly better compression ratios than when applied directly to the raw dataset. Choosing an effective and efficient compressor is crucial, as it must ensure a balance between speed and compression ratio.

5 Proposed solution

To efficiently process large volumes of files in an HPC environment, a parallel workflow has been developed. Specifically, this workflow is implemented using ParSoDA-Py and a variant of the PPC paradigm to enhance data compression efficiency in a parallel environment. The implementation required addressing multiple challenges related to the efficient utilization of HPC system resources, including the minimization of disk accesses, which can represent a scalability bottleneck. While parallel disk I/O is generally faster than sequential I/O, its performance remains constrained by the underlying hardware and can become a limiting factor as data volumes increase. This issue is particularly critical when handling large collections of small files: even when the total data size is not excessive, handling a huge number of files can significantly increase disk overhead, thereby reducing overall throughput.

Figure 2 illustrates the proposed data compression workflow, composed by a number of parallel and distributed phases. During the *data allocation* phase, a metadata file containing information about the files to be compressed is analyzed to determine the optimal allocation strategy, ensuring a balanced distribution of files across computing nodes and preventing workload imbalances. This step prioritizes assigning files with the same name to the same node to enhance compression efficiency. In this phase, files are grouped by filename, and the total size (in bytes) of each group is calculated. Using this information, a data distribution heuristic is applied to minimize load imbalances between nodes and efficiently utilize the CPU cores available on each node, as detailed later in the section.

Partitioned data are then processed by each node using a ParSoDA-based workflow. In particular, each node loads the metadata of the assigned partition

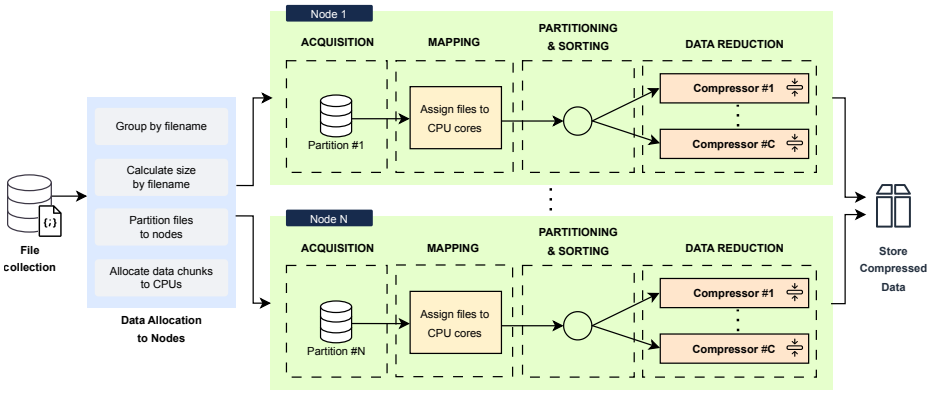


Fig. 2: Parallel compression workflow.

and begins the *mapping phase*. During this phase, files in the partition are associated with a composite key $\langle \texttt{C}, \texttt{NAME} \rangle$, where C represents the assigned CPU core, and `NAME` denotes the filename. Through a secondary sorting process, files are grouped by the primary key C and ordered by the secondary key `NAME` before being delivered to the corresponding CPU core. In the data reduction phase, files are concatenated in chunks of K bytes and compressed using a standard compressor (i.e., Zstd). This process ensures that all files with the same name are processed sequentially and compressed contiguously. To reduce the number of I/O operations, compression is performed using in-memory buffers with periodic flushing of compressed data to storage.

It is worth noting that, in certain cases, the requirement to process all files with the same name together in one CPU core must be relaxed to achieve a balanced workload distribution and ensure scalability. In fact, a critical scenario arises when specific filenames appear significantly more frequently than others, leading to potential workload imbalances that can negatively impact scalability. For example, as shown in Figure 3, some filenames, such as *main.cpp* or *main.c*, occur far more frequently than others, often resulting in a total data length that necessitates distribution across multiple workers to preserve scalability.

Algorithm 1 describes the proposed data compression workflow, using the following notation:

- F, a collection of files, characterized by a filename, a SHA1 digest to identify the file uniquely, and a length in bytes;
- N, the number of nodes;
- C, the number of CPU cores per node;
- K, the maximum size (in bytes) of a data chunk to be compressed;

The files in F are grouped by filename (line 1) to compute the total length B for each filename (lines 2–4) and determine the mean length of these groups of files (line 5). The inter-node allocation strategy is then applied as follows: for each group of files G_i (line 6), we iterate until all files in G_i have been allocated.

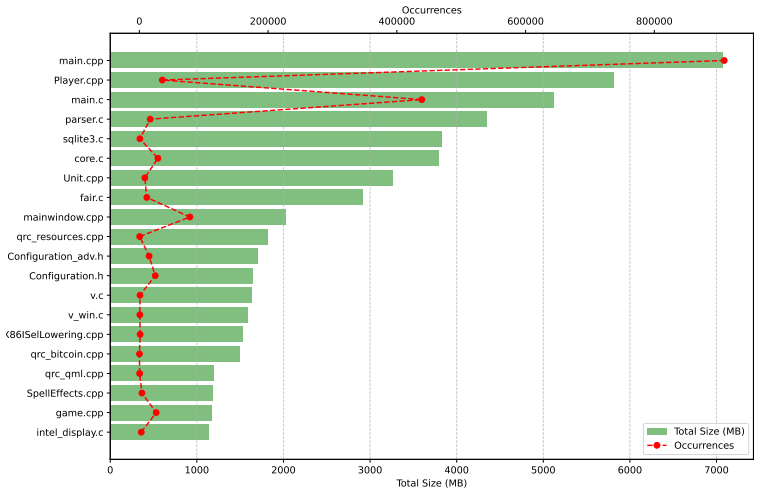


Fig. 3: Top 20 filenames by total size with occurrences in a collection including 1.1 TB of C++ source code files.

Specifically, at the beginning, the smallest data partition p and the least loaded CPU core c within that partition are selected (lines 7-8), and a counter for the current allocation is initialized (line 9). Then, we iterate through all files in G_i, assigning them to partition p and CPU core c (lines 11-12), while updating the current allocation counter (line 13). The assignment to core c continues until at least K bytes have been assigned to the current core or until the partition size exceeds the calculated mean length (line 14). If either of these conditions is met, a new partition and a new core are chosen to balance file distribution across cores and partitions, avoiding harmful imbalances that could impact scalability (lines 15–17). Once file distribution across partitions and cores is complete, the algorithm proceeds with the parallel compression phase based on the PPC approach (lines 21-29). The workflow instance running on node n then loads the files associated with its assigned partition D (line 22). Next, the application groups files by assigned core and sorts each group by filename (line 23). In parallel, across the different cores, the data is then compressed into chunks of size K and stored (lines 24-27).

6 Experimental results

To assess the effectiveness of the proposed data compression application, an extensive experimental campaign has been carried out on Leonardo[10], a petascale supercomputer hosted by CINECA, the Italian inter-university consortium

[10] See `http://leonardo-supercomputer.cineca.eu`

Algorithm 1 Scalable Compression Workflow

Require: A collection of files $F = \{f_1, f_2, ..., f_n\}$, number of nodes N, number of CPU cores C, and a compression chunk size K

$\mathcal{G} = groupByFilename(F)$ ▷ **Step 1 (Permute):** Group files by filename
for $G_i \in \mathcal{G}$ **do** ▷ Compute total file length for each filename
  $B(G_i) = \sum_{f \in G_i} len(f)$
end for
$mean \leftarrow sum(B)/N$ ▷ Mean length of filename's groups
for $G_i \in \mathcal{G}$ **do** ▷ **Step 2 (Partition):** Distribute file groups to CPU cores
  $p \leftarrow findSmallestPartition()$
  $c \leftarrow findLeastLoadedCore(p)$
  $curr_allocation \leftarrow 0$
  for $f \in G_i$ **do**
    f[partition] $\leftarrow p$ ▷ Assign file to a partition
    f[core] $\leftarrow c$ ▷ Assign file to a CPU core
    $curr_allocation \leftarrow curr_allocation + len(f)$;
    if $curr_allocation \geq K \lceil getPartitionLen(p) \geq mean$ **then**
      $curr_allocation \leftarrow 0$ ▷ Move to another partition for balancing
      $p \leftarrow findSmallestPartition()$
      $c \leftarrow findLeastLoadedCore(p)$
    end if
  end for
end for
for $n \in 1, ..., N$ **do in parallel**
  $D \leftarrow loadPartitionData(n)$
  $S \leftarrow groupByCPUCoreAndSortByFilename(D)$
  for $s \in S$ **do in parallel**
    while $z \leftarrow getNextChunk(s, K)$ **do** ▷ **Step 3 (Compress):** Compression with chunking
      $compressChunkAndStore$(z)
    end while
  end for
end for

for high-performance computing. Specifically, experiments were conducted on a subset of the Software Heritage archive, consisting of 1.1 TB and about 49 million files from C++ code repositories. Although the total volume of data may not appear excessively large, the enormous number of files that must be read and written in parallel overwhelms traditional disk controller capabilities, often resulting in disk failures and errors on common hardware. Consequently, it becomes imperative to employ HPC systems equipped with more advanced storage solutions.

The experiments were carried out on nodes equipped with 64 CPU cores and 384 GB of RAM. These nodes use the Lustre file system, which offers significant advantages for handling large amounts of data, such as high throughput and scalability across multiple storage devices. However, Lustre is not well suited for storing a vast collection of small files due to its high metadata overhead and suboptimal performance for random small-file access. Additionally, Lustre applies a lock on the parent directory to manage metadata when multiple processes attempt to access files within the same folder, limiting concurrent access and significantly reducing parallel performance. To mitigate these limitations, we organized the data into a structured directory hierarchy, distributing files across multiple directories to reduce metadata contention and improve I/O efficiency.

We used Zstd as the base compression algorithm, as it offers fast compression/decompression speeds while maintaining a better compression ratio. Specif-

ically, the compression task was configured with two compression levels, 12 and 22, where the latter represents the maximum compression level supported by Zstd. The choice of these two compression levels was motivated by the need to evaluate the trade-off between compression efficiency (speed) and efficacy (space). Level 12 provides a good balance between compression ratio and speed, making it suitable for scenarios where performance is a priority, such as fast decompression or limited memory usage. In contrast, level 22 offers a better compression, making it ideal when minimizing data size is critical, despite the larger computational cost. A compression chunk size K of 4MB was used to balance fast decompression of single files with good compression of individual chunks (see the Conclusion section for further discussion on this issue). While most Python-based Zstd libraries support multi-threaded compression, we opted for a single-core compressor to ensure consistent compression ratios, reduce complexity, and prioritize scalability and predictability in HPC environments. This choice avoids the higher per-operation overhead, increased memory usage, and potential output size inflation associated with multi-threading[11]. However, our workflow is designed to support multi-threaded compressors if needed.

	Cores	**Compressed size [GB]**		**Compression ratio [%]**	
		PPC	**Random**	**PPC**	**Random**
Level 22	64	77.88	172.76	7.43	16.48
	128	78.48	173.10	7.48	16.51
	256	78.98	173.23	7.53	16.52
	512	79.34	173.25	7.57	16.52
	1024	79.67	173.26	7.60	16.52
Level 12	64	86.69	188.33	8.27	17.96
	128	87.32	188.47	8.33	17.97
	256	87.84	188.71	8.38	18.00
	512	88.21	188.79	8.41	18.00
	1024	88.55	188.81	8.44	18.01

Table 1: Comparison of compression ratios (i.e., the ratio between the compressed size and the original size) for a 1.1 TB file collection using the PPC and random approach, with compressor Zstd adopting the compression levels 22 and 12 and chunk size $K = 4MB$.

Table 1 shows the results obtained using two compression levels (12 and 22), which highlight the significant advantages of the PPC approach over basic random-order processing of the files. Several configurations have been tested with a variable number of CPU cores, ranging from 64 to 1024. In all configurations, PPC allows for a significant improvement in the compression ratio compared to the random approach, demonstrating its effectiveness in optimizing data storage. The compression ratio is calculated as the ratio between the compressed size and the original size of the processed data, where lower values indicate better

[11] For more details about this issue, see `https://python-zstandard.readthedocs.io`

compression efficiency. Specifically, using the compression level 22, with 64 CPU cores, PPC reduces the compressed size to 77.88 GB (i.e., a compression ratio of 7.43%), compared to 172.76 GB using the random-order approach (compression ratio of 16.48%), achieving an improvement of about 9% in terms of compression ratio. This trend remains consistent as the number of nodes increases, with PPC consistently producing compressed files of less than 80 GB at 16 nodes, compared to the significantly higher 173 GB required by the random-order approach.

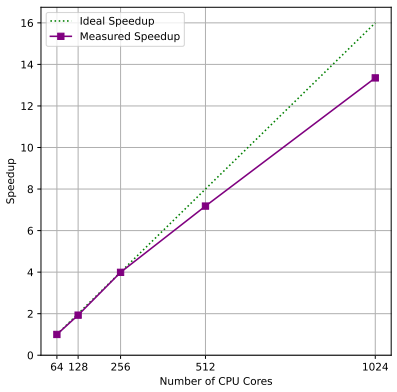
Even when using compression level 12, the results in Table 1 highlight a clear advantage of the PPC approach over the random one. However, as expected, the compression ratio is slightly higher, by about 0.84%, compared to that obtained with compression level 22. This means that using the lower compression level results in an overhead of approximately 8.8 GB but offers faster compression speeds (see Figure 4(d)). Regarding compression size, with 64 cores, PPC reduces the original data to 86.69 GB, compared to 188.33 GB with the random-order approach; notably, this is also better than adopting the random-order approach with compression level-22.

The higher performance of PPC is due to its ability to group similar files before compression, enhancing the capability of the compression algorithm to exploit local redundancy in the data. In contrast, the random approach processes files in an arbitrary order, leading to a loss of locality and reduced compression efficiency. The slight increase in compressed ratio with higher CPU counts in the PPC method is expected, as distributing data across nodes slightly reduces locality within each partition (the greater the number of partitions, the more distributed the data). However, this increase is marginal, remaining below 2% compared to the 64-core case, confirming that PPC maintains high compression efficiency even in highly data-parallel environments.

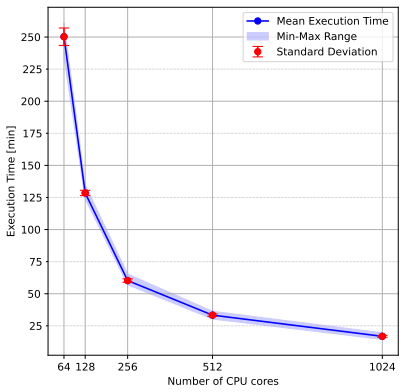
As shown in Figure 4, the workflow exhibited excellent execution times and scalability, with speedup values close to ideal values. In particular, as depicted in Figure 4(a), compared to the 64 core execution, we observed a linear speedup up to 256 cores. The speedup slightly decreased to 7.18 with 512 cores and 13.35 when scaling to 1024 cores. This small decrease is likely due to bottlenecks related to the hundreds of parallel I/O operations on the file system.

Figure 4(b) illustrates how execution times across different nodes provide key insights into the computational balancing and scalability of the compression workload. As expected, increasing the number of nodes significantly reduces execution time, confirming that the compression process benefits from parallelism. For example, scaling from 64 cores to 1024 cores, the execution time decreased from 4.34 hours to 0.32 hours (19.2 minutes). Although the overall trend indicates improved performance with increased parallelism, data distribution across cores can lead to execution time imbalances. Following the PPC approach, the presence of highly frequent filenames may cause certain cores to process significantly more data than others, resulting in workload imbalances during compression tasks. To mitigate this issue, the algorithm imposes a limit on the maximum amount of data assigned to each core, helping to balance the workload (see Algorithm 1, line 14). Analyzing the average execution times per core along with the

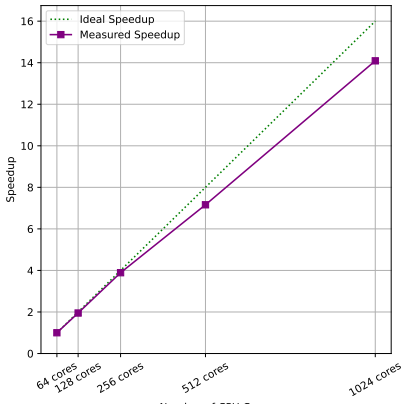
standard deviation reported in Figure 4(b), it is possible to note that execution time variations between cores are minimal. This behavior results in the entire application being scalable.



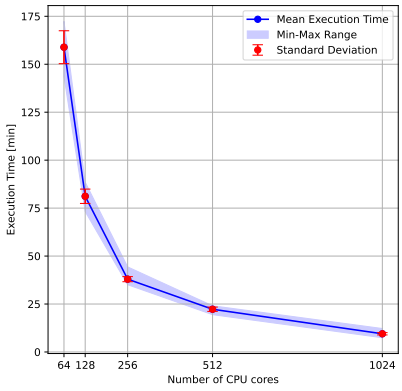
(a) Speedup (level 22).



(b) Execution time (level 22)



(c) Speedup (level 12).



(d) Execution time (level 12)

Fig. 4: Speedup and execution times for a 1.1 TB file collection, with compressor Zstd adopting the compression levels 22 and 12 and chunk size K = 4MB

Finally, as illustrated in Figures 4(c)-4(d), the proposed solution exhibits a similar behavior at compression level 12, but with a speedup closer to the ideal value and slightly better in the case of 1024 cores (14.09) compared to compression level 22. Additionally, execution times—ranging from 172 to 12

minutes with 64 and 1024 cores, respectively—are reduced by an average of 34% (compression level 12) compared to the previous configuration (compression level 22) due to better efficiency of Zstd.

7 Conclusion

This paper presented a parallel data compression workflow for large-scale source code repositories, leveraging HPC environments and the Permute-Partition-Compress (PPC) paradigm. Implemented using ParSoDA-Py, our parallel approach significantly improves the compression ratio by reorganizing and segmenting the data before applying compression. Specifically, experiments showed that our approach significantly improves the compression ratio over random-order compression, from 16.5% to 7.5%, while maintaining strong scalability up to 1024 CPU cores. As future work, we plan to explore alternative PPC heuristics for file clustering, scaling up to tens of terabytes of data. Further, we aim to assess the trade-off between compression ratio and decompression throughput by varying the chunk size (here set to 4 MB), investigate the impact of other compression-level settings for Zstd (here set to 12 and 22) or other compressors, and implement dynamic workload distribution to avoid a preliminary data partitioning phase. Additionally, we plan to investigate decompression and random access performance, which are critical for real-world archival and retrieval applications.

Acknowledgments. The work was supported by the research project "National Centre for HPC, Big Data and Quantum Computing" - "Spoke 0: FutureHPC & BigData", CN00000013 - CUP H23C22000360005 and CUP I53C22000690001. The work of P.F. was also partially supported by the EU – Horizon 2020 Program under the scheme "INFRAIA-01-2018-2019 – Integrating Activities for Advanced Communities", Grant Agreement n. 871042, "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics", and by the NextGenerationEU – Project: "SoBigData.it - Strengthening the Italian RI for Social Mining and Big Data Analytics" (Prot. IR0000013, M.4 - C.2 – I 3.1 – Avviso n. 3264 del 28/12/2021, CUP: B53C22001760006).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abramatic, J.F., Di Cosmo, R., Zacchiroli, S.: Building the universal archive of source code. Commun. ACM **61**(10), 29–31 (sep 2018)
2. Alakuijala, J., Farruggia, A., Ferragina, P., Kliuchnikov, E., Obryk, R., Szabadka, Z., Vandevenne, L.: Brotli: A general-purpose data compressor **37**(1) (Dec 2018)
3. Belcastro, L., Giampà, S., Marozzo, F., Talia, D., Trunfio, P., Badia, R.M., Ejarque, J., Mammadli, N.: Boosting HPC data analysis performance with the ParSoDA-Py library. Journal of Supercomputing (2024)
4. Belcastro, L., Marozzo, F., Talia, D., Trunfio, P.: ParSoDA: high-level parallel programming for social data mining. Social Network Analysis and Mining **9**(1) (2019)

5. Bhattacherjee, S., Chavan, A., Huang, S., Deshpande, A., Parameswaran, A.G.: Principles of dataset versioning: Exploring the recreation/storage tradeoff. Proc. VLDB Endow. **8**(12), 1346–1357 (2015)
6. Boffa, A., Di Cosmo, R., Ferragina, P., Guerra, A., Manzini, G., Vinciguerra, G., Zacchiroli, S.: On the compressibility of large-scale source code datasets. Journal of Systems and Software (2025)
7. Buchsbaum, A.L., Caldwell, D.F., Church, K.W., Fowler, G.S., Muthukrishnan, S.: Engineering the compression of massive tables: an experimental approach. In: Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 175–184 (2000)
8. Buchsbaum, A.L., Fowler, G.S., Giancarlo, R.: Improving table compression with combinatorial optimization. J. ACM **50**(6), 825–851 (nov 2003)
9. Cosmo, R.D., Zacchiroli, S.: Software Heritage: why and how to preserve software source code. In: Proc. 14th International Conference on Digital Preservation (iPRES) (2017)
10. Douglis, F., Iyengar, A.: Application-specific delta-encoding via resemblance detection. In: Proc. General Track 2003 USENIX Annual Technical Conference. pp. 113–126 (2003)
11. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica **63**(3), 707–730 (Jul 2012)
12. Ferragina, P., Manzini, G.: On compressing the textual web. In: Proc. 3rd ACM International Conference on Web Search and Data Mining (WSDM). pp. 391–400 (2010)
13. Hosseini, M., Pratas, D., Pinho, A.J.: A survey on data compression methods for biological sequences. Information **7**(4) (2016)
14. Hu, X., Wang, F., Li, W., Li, J., Guan, H.: QZFS: QAT accelerated compression in file system for application agnostic and cost efficient data storage. In: Proc. 2019 USENIX Annual Technical Conference (USENIX ATC). pp. 163–176 (2019)
15. Hunt, J.J., Vo, K.P., Tichy, W.F.: Delta algorithms: an empirical analysis. ACM Trans. Softw. Eng. Methodol. **7**(2), 192–214 (apr 1998)
16. Molfetas, A., Wirth, A., Zobel, J.: Using inter-file similarity to improve intra-file compression. In: Proc. 2014 IEEE International Congress on Big Data. pp. 192–199 (2014)
17. Ouyang, Z., Memon, N.D., Suel, T., Trendafilov, D.: Cluster-based delta compression of a collection of files. In: Proc. 3rd International Conference on Web Information Systems Engineering (WISE). pp. 257–268 (2002)
18. Suel, T., Memon, N.: Algorithms for delta compression and remote file synchronization. Academic Press, United Kingdom (Aug 2002)
19. Talia, D., Trunfio, P., Marozzo, F., Belcastro, L., Cantini, R., Orsino, A.: Programming big data applications: Scalable tools and frameworks for your needs. World Scientific (2024)
20. Tejedor, E., Becerra, Y., Alomar, G., Queralt, A., Badia, R.M., Torres, J., Cortes, T., Labarta, J.: Pycompss: Parallel computational workflows in python. The International Journal of High Performance Computing Applications **31**(1), 66–82 (2017)
21. Wandelt, S., Bux, M., Leser, U.: Trends in genome compression. Current Bioinformatics **9**(3), 315–326 (2014)
22. Xia, W., Li, C., Jiang, H., Feng, D., Hua, Y., Qin, L., Zhang, Y.: Edelta: A word-enlarging based fast delta compression approach. In: Proc. 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage). USENIX Association (2015)