Aggregating Columnar Data on Cluster File Systems

Jonas Hahnfeld[1,2], Jakob Blomer[1], and Thorsten Kollegger[2]

[1] CERN, Geneva, Switzerland
{jonas.hahnfeld,jakob.blomer}@cern.ch
[2] Goethe University Frankfurt, Institute of Computer Science, Frankfurt, Germany
kollegger@em.uni-frankfurt.de

Abstract. Data in High Energy Physics (HEP) is primarily stored in binary columnar formats. This enables efficient reading of select columns for the physics analysis use case while employing transparent compression to reduce storage space. However, this combination of requirements makes it challenging to write HEP data in parallel. This has been partially addressed with multithreaded writing, but efficient usage in cluster environments remained an open question. In this paper, we extend the concepts to distributed scenarios and parallel writing of columnar data into a single file on cluster file systems. For this, we present different strategies based on a central aggregator, using MPI one-sided communication, and relying on synchronization with file locks. We evaluate the scalability of these approaches on two cluster file systems, and discuss the impact of different parameters on performance. Finally, we motivate a real-world use case for resource-intensive Monte Carlo simulation jobs.

Keywords: Distributed parallel writing · Cluster file systems · Columnar data format · High Energy Physics · ROOT.

1 Introduction

The Large Hadron Collider (LHC) at CERN is currently the largest particle accelerator used for High Energy Physics (HEP) research. It hosts a number of experiments that record and analyze particle collisions to improve our understanding of the fundamental structure of the universe. This requires handling of data at exabyte scale, which commonly relies on the ROOT framework for storage and analysis [7].

After the currently ongoing Run 3, until 2030, the LHC will be upgraded to the High-Luminosity LHC (HL-LHC). This will enable even more collisions and consequently result in a steep increase of data rates. To prepare for this future, the HEP community is currently adopting the RNTuple format [6]. It was developed as an evolution of the presently used TTree columnar format. Early results confirm smaller file sizes and better performance when exploiting modern storage systems [5].

In previous work, we added support for scalable multithreaded parallel writing to RNTuple [13]. This has the potential to avoid the merge step for combining

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

smaller files from multiple threads. The evaluation shows perfect scalability up to the storage bandwidth limit on a single server. This will be useful on traditional High Throughput Computing resources provided by the Worldwide LHC Computing Grid [4].

However, as data volumes and compute requirements grow, capacities are increasingly complemented by resources on existing High Performance Computing (HPC) systems. This requires efficient usage of cluster environments, with distributed parallelism and output to cluster file systems. A particular challenge is that distributed writers must be able to make independent progress without relying on collective operations. This is critical for load balancing between events of different sizes and varying processing times.

The main contributions of this work are as follows: In Section 3, we discuss concepts to aggregate columnar data on cluster file systems and possibilities to implement distributed parallel writing. We experimentally evaluate the scalability of the alternative implementation strategies in Section 4. In a second step, we study the best strategies on two different cluster file systems in Section 5. Finally, we present a possible use case of our work for Monte Carlo simulations in Section 6.

2 Related Work

There exist multiple formats and libraries to store scientific data on HPC systems: NetCDF was designed to exchange data in atmospheric research [17]. Its parallel interface PnetCDF is built on MPI-IO [14]. HDF5 supports flexible data models and parallel I/O via MPI [10]. However, neither of these two widespread libraries is suitable for parallel writing of HEP data as discussed in [13]: While NetCDF supports record variables with an unlimited dimension, it assumes they grow together which is impractical for nested data of variable size. For HDF5, resizing an extendible dataset is a collective operation, which would prevent independent writer progress. Variable-length array datatypes on the other hand are not supported at all with parallel writing.

More recently, ADIOS 2 tackles scalable storage of scientific data as part of the Exascale Computing Project (ECP) [11]. The library is based on MPI and the framework supports a number of *engines* and *transports*. ADIOS 2 *variables* allow for flexible data layouts, including different array sizes per rank needed to support nested data. However, the library is fundamentally built around *steps* which are collective synchronization points. Altogether, none of the existing libraries support distributed parallel writing of HEP data and a direct comparison is not possible.

MPI-IO refers to the set of I/O functions defined as part of the MPI standard [15]. It is portably implemented by ROMIO [21] and OMPIO [8], and underpins both PnetCDF and HDF5 mentioned before. MPI-IO can significantly improve application performance with data sieving and collective I/O [20]. However, this does not apply for columnar HEP data where events are processed in parallel and writers should progress without synchronization. More-

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

over, HEP data is grouped into clusters that are non-overlapping and several tens of megabytes in size.

Singh and Gabriel introduce parallel I/O directly on compressed files [19]. In contrast, we integrate distributed parallel writing for columnar data. This requires aggregating metadata to allow reading of individual columns as if the data was written sequentially. Furthermore, our solution focuses on independent write operations instead of collective writes.

Beyond multithreading, the ROOT framework also provides the two classes `TParallelMergingFile` and `TMPIFile`. They use incremental merging to produce a single output file, which was also adopted by the ATLAS experiment [18]. However, these approaches require sending all data to a server or collector rank, which does not scale as we will show. For this reason, we do not present an explicit comparison to these classes.

3 Aggregating Columnar Data

Columnar data formats are optimized for reading subsets of the stored columns. Notable examples include the original ROOT Trees [7] currently used by LHC experiments and Apache Parquet [22]. In this work, we focus on RNTuple, which is the designated successor of the TTree format [6].

3.1 Overview of Multithreaded Parallel Writing

All three columnar data formats mentioned above support nested data and transparent compression. As a result, rows (called *entries* in RNTuple) have variable size. This makes parallel writing challenging because data cannot be laid out in a regular grid. However, for HEP use cases, entries or *events* are often independent and can be reordered.

Based on this observation, we previously described concepts for parallel writing of columnar data [13]. The key contribution is the identification of relocatable *units of writing* that can be prepared and compressed independently. In the case of RNTuple, a *cluster* is a natural unit of writing, which spans a consecutive range of entries. This allows to implement parallel writing with cooperating threads: After the cluster size is known, a lock is taken and the writer reserves an appropriate buffer in the file. Finally, the metadata is updated to append the cluster and its entries to the dataset. We note that this approach trivially enables independent progress outside the critical section, and allows different number of clusters written per thread.

3.2 Distributed Parallel Writing with Aggregator

In this work, we aim to extend the concepts developed for multithreaded parallel writing to distributed environments. The goal is to have multiple *writers* produce a single file with columnar data on a shared file system. A naive strategy is to send compressed clusters to a single *aggregator* that serializes the write

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

operations. In a potential implementation with MPI, this can be realized with a thread on a root rank. Doing so requires support from the MPI implementation and an application that is initialized with `MPI_THREAD_MULTIPLE`.

In the HEP context, an equivalent approach is used by the Athena framework of the ATLAS experiment [18]. However, for distributed environments, the approach has two disadvantages that limit scalability: First, it requires sending a large amount of data, where a single aggregator can become a bottleneck. Moreover, since writing is serialized at the aggregator, it is limited by the bandwidth of a single process. This means it cannot efficiently utilize the full potential of parallel cluster file systems.

A better approach is to send only the metadata, which is small compared to the total size of compressed cluster data. Nevertheless it is enough information for the aggregator to compute the cluster size and reserve a buffer in the file. It then returns the buffer's byte offset to the writer, which can flush out the cluster data. The advantage is that this approach allows distributed parallel writing from all processes into the assigned parts of the file. Still, all metadata is collected as before and written by the aggregator when closing the file.

For implementations, there are two important details to optimize performance: First, the container format may require anchor objects when reserving byte buffers. For example, ROOT files are organized on disk as a linked list of *keys*. Therefore, when embedding RNTuple data into ROOT files, each buffer is preceded by a `TKey` header. In a straightforward implementation, this header is directly written on the aggregator. However, this can introduce increased latency if other processes on the same node write at the same time. Alternatively, the aggregator can send the header to the writer, which can prepend it to the cluster data. We will demonstrate the advantage of this optimization in Section 4.2.

Additionally, cluster file systems may use striping to distribute a file to multiple storage servers. If multiple nodes write to parts of the file in the same stripe, this will result in lock contention. It may therefore be required to align the reserved buffers to improve performance. If the stripe size is small compared to a unit of writing, one method is padding to ensure alignment. While this introduces an overhead in file size, we found it acceptable compared to the gain in write bandwidth. We will discuss the performance impact of alignment for different cluster file systems in Section 5.

3.3 Maintaining a Global Offset without Aggregator

In this section, we discuss concepts for distributed parallel writing without a central aggregator. The key insight is that metadata aggregation can be delayed until committing the dataset. Instead, each writer tracks its local metadata and the collective aggregation is done when closing the file. For the cluster data, the writers only need to reserve and write to buffers in the file. These can be linearly allocated using a global offset that is atomically incremented. The problem of implementing such global offset is similar to that of maintaining a shared file pointer in MPI-IO.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

The authors of ROMIO mention different methods to implement shared file pointers [21]. One possibility is to have a central entity, either a thread or dynamic process, own the file pointer. A writer can access and update the value by communicating with this entity. However, such active entity is also able to directly aggregate the metadata as discussed in the previous section. For this reason, it is not useful to reduce the aggregator's role and only maintain the file offset.

A method without active involvement of the central entity is remote memory access, for example via MPI one-sided communication. It allows a process to share a "window" of its memory and either uses "active" or "passive" target synchronization. While the former explicitly involves the target process, the MPI standard mentions that the passive "communication paradigm is closest to a shared memory model" [15]. Using accumulate functions, it is then possible to implement a counter to maintain a global offset.

The final method used by some portable MPI-IO implementations relies on file locks [21]: They store the shared file pointer in a file and protect reads and atomic updates with advisory record locks. The same technique can be used for distributed parallel writing of columnar data. We note that instead of a second, separate file, it is also possible to identify a part of the final file that can be temporarily used.

When merging the metadata, implementations have to decide on the order of clusters. If appended by process, their order may not match the structure in the file. This can result in non-linear access patterns and seeking when reading the produced file. It is unclear if this has a measurable impact since each cluster is several tens of megabytes in size. If found important, a linear order can be restored by sorting based on buffer offsets when aggregating the cluster metadata.

4 Scalability of Distributed Parallel Writing

Based on the described approaches, we implement distributed parallel writing of columnar data in the RNTuple format. The prototype relies on MPI for easy integration into cluster environments on HPC systems. However, some aggregation strategies such as the metadata aggregator can also be implemented with other communication libraries. For example, some HEP experiment frameworks have a distributed architecture based on ZeroMQ [3,9].

In a first step, we evaluate the scalability of our prototype on the Vega supercomputer [2]. Each node of the system has two AMD Epyc 7H12 processors for a total of 128 physical cores, and 256 GB of DDR4-3200 memory. Nodes are interconnected with Infiniband HDR, which provides a bandwidth of up to 200 Gb/s. We build ROOT 6.36.06, the latest version with long-term support, from source with GCC 14.2.0. Our prototype is compiled using OpenMPI 5.0.9 and UCX 1.20.0 with support for `MPI_THREAD_MULTIPLE`. For execution, we use entire nodes allocated with SLURM and request two `cpus-per-task` to allow using hyperthreading for the aggregator on the root rank.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

In the remainder, we use a variation of the synthetic benchmark described in our previous work [13]: Instead of threads, each MPI rank writes entries of random data into two fields, a 64-bit integer and a vector of `float`s. The vector size follows a Poisson distribution with $\mu = 5$ while the elements are uniformly distributed in $[0, 100)$. This results in a relatively small entry size of only 36 bytes on average. Unless noted otherwise, each rank writes 10 million entries, amounting to 360 MB before compression, in a weak scaling setup. The columnar data is staged in a 4 MiB buffer before being passed to the operating system. We experimented with bigger buffer sizes up to 32 MiB, but did not see improved bandwidths. For each data point, we repeat the benchmark five times and compute the arithmetic mean of the run times. Then we calculate the storage bandwidth based on the written data volume of all ranks. If compressing data using `zstd` (level 10), the bandwidth is based on the compressed data volume written to storage. Error bars are computed with one standard deviation of the measured run times.

4.1 Comparison of Aggregation Strategies

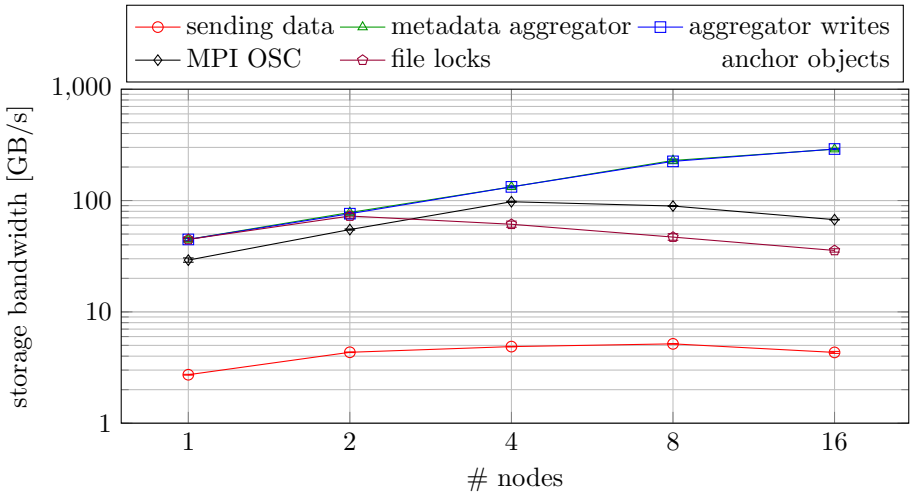

Fig. 1. Measured bandwidth writing to `/dev/null` without compression.

By writing to `/dev/null`, we compare the strategies for infinitely fast storage. This allows to assess their communication behavior when writing the data is not the bottleneck. Figure 1 shows the measured bandwidth without compression with up to 16 nodes and 128 ranks per node. The first strategy is to send all data to an aggregator rank using MPI. It is clear that this approach is the slowest, with a maximum bandwidth of 5 GB/s. This was measured in a strong scaling experiment at a constant total data volume of 46 GB, to reduce run

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

time. In previous experiments, weak scaling with reduced repetition counts did not result in improved bandwidths.

For the second strategy, our prototype sends only metadata to the aggregator and will write data directly into the shared file. This approach scales much better and achieves more than 290 GB/s on 16 nodes. By default, writing is fully distributed and each rank writes the anchor objects necessary for its data. We evaluate this approach with a variation where the aggregator is instructed to write these anchor objects. In the case of writing to `/dev/null`, this does not make a difference since data is discarded by the operating system. This is why the two top lines (green triangles and blue squares) are perfectly overlapping.

We also implement two strategies to maintain a global offset as introduced in Section 3.3. The first option uses MPI one-sided communication (OSC) and reaches a maximum of 100 GB/s at four nodes, which is slower than the metadata aggregator. To also test synchronization via file locks, we maintain the global offset in a file on the Lustre file system and use byte-range locking via `fcntl`. In this case, the performance peaks at 70 GB/s with 2 nodes and then decreases with more nodes.

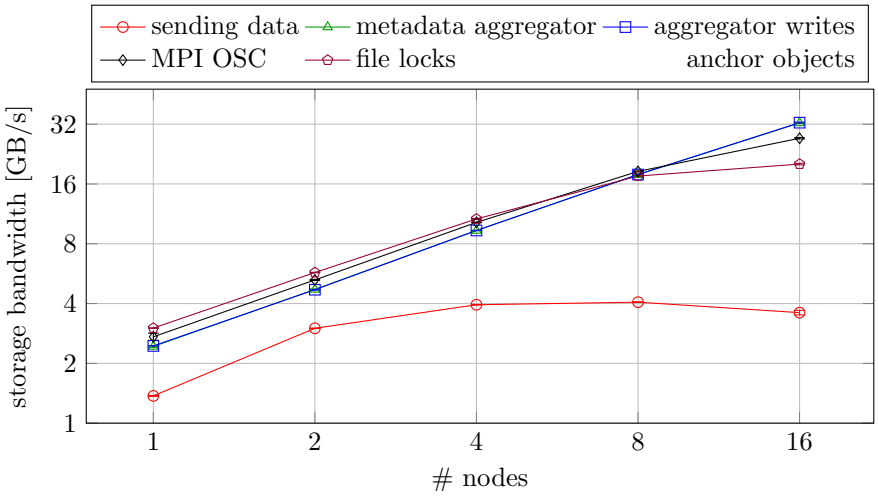

Fig. 2. Measured bandwidth writing to `/dev/null` with `zstd` compression.

Figure 2 shows the results when compressing the columnar data with `zstd`. While sending the data is still the slowest, the other strategies are limited by the compression throughput. This results in a maximum bandwidth of 32.5 GB/s with 16 nodes at a parallel efficiency of 68 % compared to a maximum bandwidth of 3 GB/s with a single node. It should be noted that these bandwidths are achieved with different strategies, as synchronization via file locks is the fastest for fewer nodes. This is because it uses the file system as additional external

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

resource for synchronization. The metadata aggregator and OSC on the other hand need to compete for CPU resources.

4.2 Distributed Parallel Writing on Cluster File Systems

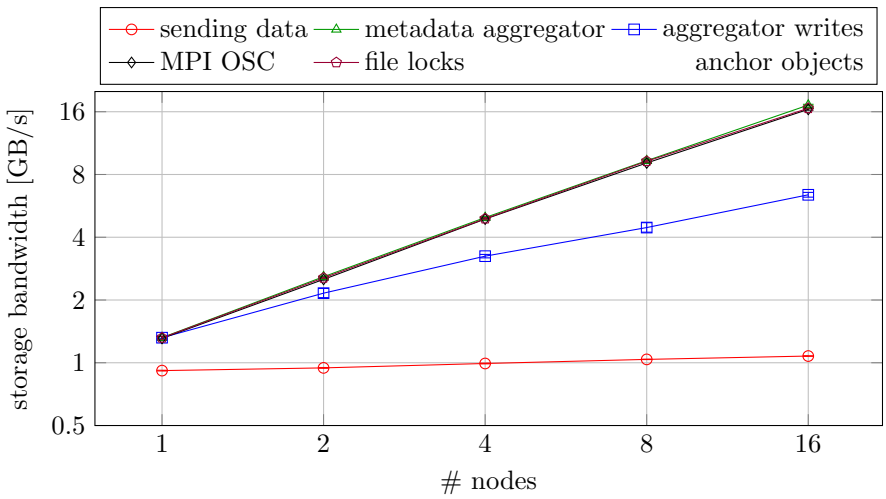

Fig. 3. Measured bandwidth writing to Lustre without compression.

Next, we write the RNTuple data into a single file on the Lustre file system provided on Vega. The file system is backed by 10 DDN Exascaler ES400NVX with NVMe disks, running version 2.15.2 of Lustre. In this section, we configure a write alignment of 1 MiB, matching the default stripe size, and a stripe count of 8. We will explore different choices for these parameters in Section 5.1 and discuss their impact. The measurements follow the same methodology as before, with a strong scaling setup for the first strategy that sends all data. For the others, the benchmark writes a constant amount of data per rank, and all results are plotted in Figure 3. Writing to Lustre, the metadata aggregator achieves a maximum bandwidth of 17.2 GB/s. MPI OSC and file locks scale up to 16.4 GB/s and 16.6 GB/s, respectively. If the aggregator writes the anchor objects, the bandwidth is significantly lower with a maximum of 6.4 GB/s. As discussed before in Section 3.2, this is because of increased latency on the aggregator.

The results using `zstd` compression in Figure 4 are similar to those without compression: Using the metadata aggregator, we measure a maximum bandwidth of 12.9 GB/s. Maintaining a global offset with MPI OSC or synchronizing via file locks is slightly slower at 11.9 GB/s and 11.6 GB/s, respectively. We conclude that these three strategies are able to make efficient use of cluster file systems for writing columnar data.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

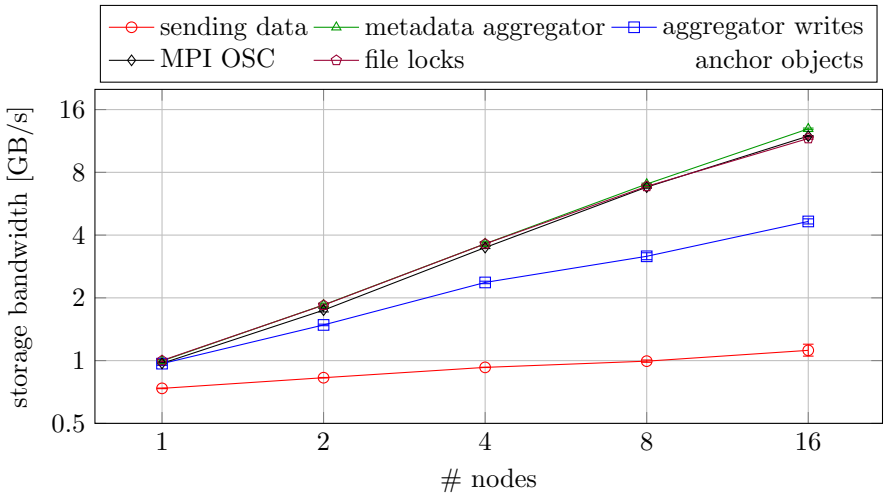

Fig. 4. Measured bandwidth writing to Lustre with `zstd` compression.

5 Performance Studies of Cluster File Systems

In this section, we continue to study the performance of the most scalable strategies as determined before. For that, we focus on the use case of writing compressed data because it is more relevant for the HEP community. Additionally, it involves some compute and is therefore closer to the real use of HPC resources. In the following, we look at the two file systems provided on the Vega supercomputer: Lustre and CephFS. We compare different write and file system configurations and measure their impact on the attained bandwidth.

5.1 Lustre

As a parallel file system, Lustre is able to *stripe* a file to multiple Object Storage Targets (OSTs). If used properly, this allows an application to achieve a higher bandwidth than a single OST can provide. Striping can be configured with two primary parameters: The stripe count prescribes how many objects are allocated on different OSTs. Data is written round-robin to each object according to the configured stripe size. In our measurements, we do not see an advantage for larger stripe sizes. We therefore keep this parameter at its default value of 1 MiB and set a matching write alignment. As before, data is staged in a 4 MiB buffer on the application side before being written.

For the Lustre file system, we already compared the different strategies in Section 4.2. We therefore restrict ourselves to the metadata aggregator and compare different configurations as shown in Figure 5. The line with green triangles corresponds to the previous measurement using a stripe count of 8, with a maximum of 12.9 GB/s. When instead configuring a lower $stripe\_count = 1$, the write bandwidth is limited to around 3 GB/s.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

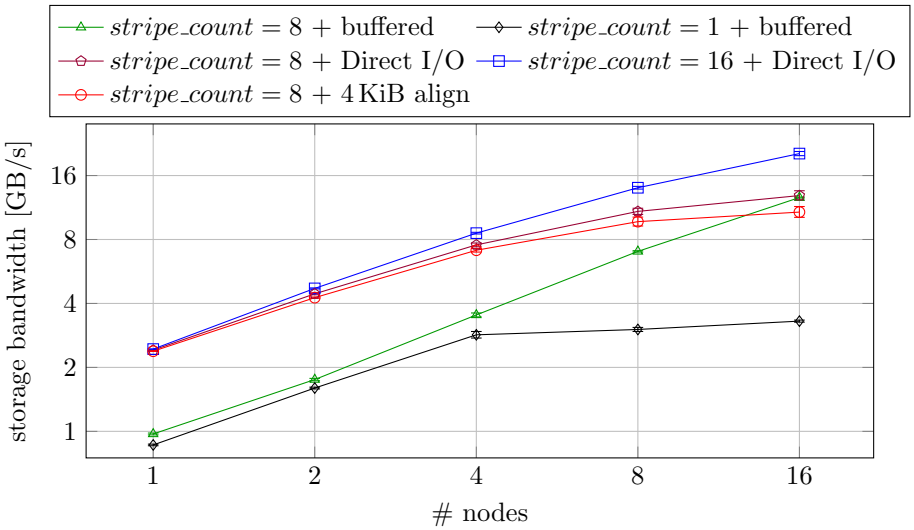

Fig. 5. Measured bandwidth writing to Lustre using the metadata aggregator and `zstd` compression, with different stripe counts and alignments.

The achievable bandwidth is not only limited by the hardware, but also by buffered writing and caching in software. This can be avoided by opening the file with the additional flag `O_DIRECT`. In previous work, we showed the benefit for multithreaded writing [12]. For distributed parallel writing to Lustre, it results in an increased bandwidth with fewer nodes and a stripe count of 8. This can be further improved to 20.3 GB/s with a stripe count of 16, which is not beneficial for buffered writing. We also tested increasing the application buffer to 32 MiB, which however does not change the measured bandwidths.

For the final measurement, we revert to a stripe count of 8 and decrease the write alignment to 4 KiB. This is lower than the stripe size of 1 MiB and results in contention when multiple ranks write into the same stripe. With Direct I/O, we measure a bandwidth of 10.8 GB/s, which is 16 % lower than 12.9 GB/s measured using an alignment of 1 MiB. For buffered writing, the effect is smaller and the bandwidth is only reduced to 12 GB/s, which we omit from the plot. A possible explanation is that writing with `O_DIRECT` is synchronous while buffering is able to hide latency resulting from contention.

5.2 CephFS

The second cluster file system available on Vega is CephFS. It provides a POSIX-compatible layer on top of the RADOS object storage. The installation on Vega is backed by 61 servers with HDDs and runs Ceph version 18.2.4. The object pool is erasure-coded with $m = 16$ data chunks and a default size of 4 KiB. This multiplies to a *stripe_width* of 64 KiB, which we choose as write alignment. On

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

the CephFS side, we keep the default file layout with an object size of 32 MiB and *stripe_count* = 1. For all measurements, we bypass caching by opening the file with `O_DIRECT`. Otherwise, with buffered writing, we observed bad scaling behavior with more than one node.

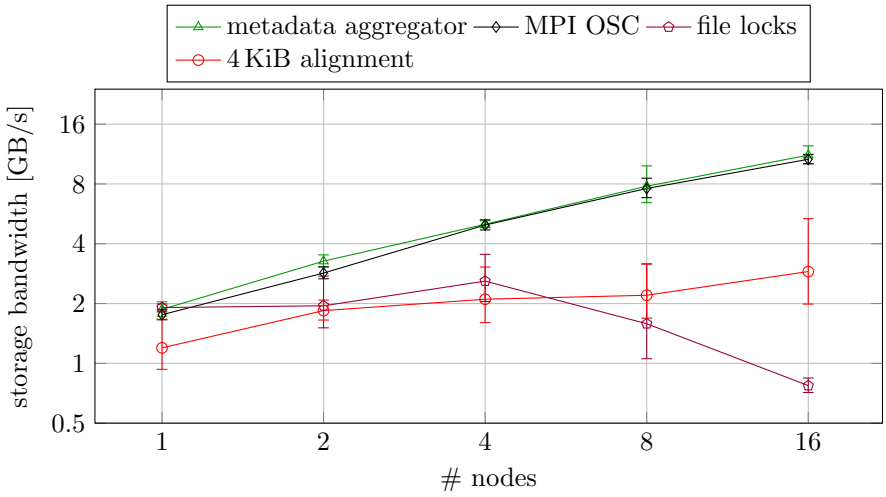

Fig. 6. Measured bandwidth writing to CephFS with `zstd` compression.

Figure 6 shows the bandwidths measured with different strategies writing compressed data to CephFS. Compared to previous results, the variation between runs is much larger, as reflected by the error bars. This is because the CephFS mount is also used for the home directories, which likely generates constant background load. Nevertheless, we measure up to 11.2 GB/s with the metadata aggregator and 10.6 GB/s with MPI OSC.

When synchronizing via file locks, the bandwidth does not scale beyond 2.6 GB/s. This can have multiple reasons, one of which is the general load on the Metadata Servers (MDS) from the home directories. More detailed measurements would be required to fully explain this bad scalability. Finally, we measure the metadata aggregator with a write alignment of 4 KiB. It is clear that the achieved bandwidth is much lower with a maximum of 2.9 GB/s.

6 Application: Monte Carlo Simulation

In this last section, we consider a possible real-world use case for distributed parallel writing: HEP experiments heavily rely on Monte Carlo simulations of the expected detector response. Then, physicists compare these predictions with actual detector data to find signals of new physics or precisely measure observables. To address the required computational needs, experiments would benefit

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

from the ability to use HPC systems. However, this requires to make efficient use of cluster environments.

In the following, we consider the CMS experiment, one of the two general-purpose experiments at the LHC. Similar techniques will also be applicable for other experiments. Monte Carlo simulation in CMS is handled by its offline software framework CMSSW [1]. Recently, CMS framework developers started integrating RNTuple as an alternative to TTree. We build on this work and locally add our prototype as an option for distributed parallel writing. This requires an extension of our implementation to support writing multiple RNTuple datasets into a single file. We achieve this by starting one metadata aggregator per dataset and synchronizing buffer reservation on the root rank.

Table 1. Monte Carlo simulation steps of 1000 TTbar events using CMSSW with 1 process and 8 threads. The final row is a combination of the last two steps, directly outputting reconstructed events into the MiniAOD format.

Step	Run Time	Output File Size	Event Throughput
GEN-SIM	1199.2 s	787.2 MB	0.83 ev/s
DIGI	286.6 s	1269.3 MB	3.49 ev/s
RECO	163.5 s	123.0 MB	6.12 ev/s
MiniAOD	72.8 s	40.6 MB	13.73 ev/s
RECO → MiniAOD	172.3 s	40.5 MB	5.80 ev/s

Table 1 lists the steps of a Monte Carlo simulation using CMSSW: In a first step, collision events are generated and simulated in the detector. For this study, we choose to generate top-antitop pairs ($t\bar{t}$ or "TTbar"). Then the detector response is digitized and trigger information is added. The result are events in the same form as they would be recorded by the actual detector. Afterwards, the events are reconstructed by computing particle momenta and energies. Finally, the event content is reduced into the "MiniAOD" format for physics analyses [16].

The columns in Table 1 show the run time, the output file size, and the event throughput for handling 1000 TTbar events using a single process with 8 threads. It can be seen that the first GEN-SIM step takes the most time, while the output of the second DIGI step is the largest. We therefore use the latter to test our integration, generating 1.2 GB of output per rank. However, many analyses only require the reconstructed events in the smaller MiniAOD format, which is produced at a much lower rate. To cover this use case, we run a combination of the last two steps, reconstructing the events and directly producing MiniAODs. For our benchmarks, we run the generation, simulation, and digitization once ahead of time. We then reuse the produced sample as inputs where each MPI rank processes the same 1000 events in a weak scaling setup. While this neglects load balancing problems, it is sufficiently realistic to evaluate distributed parallel writing.

Figure 7 plots the event throughput when scaling up to 16 nodes. As we configure 8 threads per rank, it is possible to fit 16 ranks per node on Vega.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

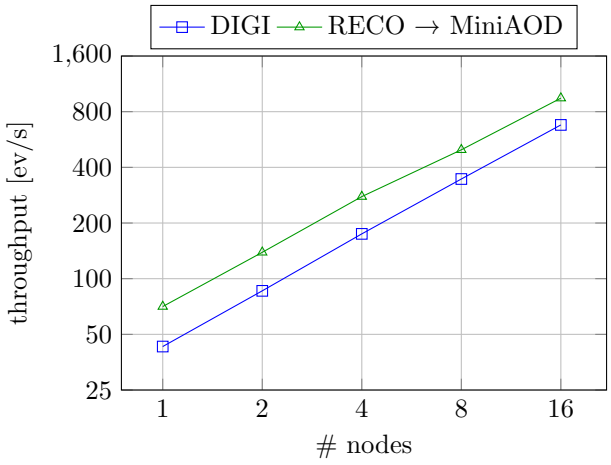

Fig. 7. Measured throughput of TTbar events using CMSSW with 16 ranks per node and 8 threads per rank.

This results in 19.2 GB of output data per node for the DIGI step, and around 650 MB per node of MiniAOD. As the variation between runs is negligible at less than 5 % of the run time, we omit the error bars.

It can be seen that the event throughput increases almost linearly with the number of nodes. For the DIGI step, it scales from 43 events per second with one node to 678.4 events per second with 16 nodes. This corresponds to 256 ranks that write into a single output file of 327.2 GB at a sustained average bandwidth of 867 MB/s. When producing MiniAOD, the data volume is much lower with a total of 10.6 GB written by 256 ranks. The event rate increases from 70.7 events per second with one node to 946.6 events per second with 16 nodes. We note that in both cases, the achieved throughput is lower than the ideal extrapolation based on Table 1. However, scaling to multiple nodes works well with a parallel efficiency of 98.7 % and 83.7 % compared to one node, respectively.

7 Conclusions

In this paper, we introduced the concepts for distributed parallel writing of columnar data. We presented multiple strategies for aggregating columnar data on cluster file systems and compared their communication behavior. Using a synthetic benchmark, we studied their scalability and found that three approaches perform well: the metadata aggregator, maintaining a global offset via MPI one-sided communication and synchronizing via file locks.

We then looked into the performance of different configurations on two cluster file systems: First, we showed that opening the file with `O_DIRECT` increases the bandwidth when writing to Lustre. For best performance, it is necessary to tune the stripe configuration and ensure proper alignment. Additionally, we presented

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

measurements on CephFS, where alignment is even more important. Further experiments should investigate the performance on other cluster file systems commonly used in HPC systems.

Finally, we tested our prototype for Monte Carlo simulation with the CMSSW experiment framework. This confirmed the feasibility of our approaches for aggregating realistic data in cluster environments. The results demonstrated good scalability for the studied use cases with up to 16 nodes. As such, the possibility of distributed parallel writing will be useful for efficiently exploiting computing resources during HL-LHC. Avenues for future research and development include fault tolerance and crash recovery not discussed in this work.

Acknowledgments. This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant 13E18CHA). We acknowledge the EuroHPC Joint Undertaking for awarding this project access to the EuroHPC supercomputer LUMI, hosted by CSC (Finland) and the LUMI consortium, and Vega, hosted by the Institute of Information Science (Slovenia) and the HPC RIVR consortium. We thank Chris Jones and Matti Kortelainen from Fermilab for their work on integrating RNTuple into CMSSW that we built on.

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

References

1. CMSSW – CMS Offline Software, `https://github.com/cms-sw/cmssw`, accessed 2025-04-24
2. Vega, `https://izum.si/en/vega-en/`, accessed 2025-04-24
3. Al-Turany, M., Klein, D., Manafov, A., Rybalchenko, A., Uhlig, F.: Extending the FairRoot framework to allow for simulation and reconstruction of free streaming data. Journal of Physics: Conference Series **513**(2), 022001 (Jun 2014). `https://doi.org/10.1088/1742-6596/513/2/022001`
4. Bird, I., et al.: LHC Computing Grid: Technical Design Report. Tech. Rep. LCG-TDR-001, CERN, Geneva (Jun 2005), `https://cds.cern.ch/record/840543`
5. Blomer, J., Canal, P., de Geus, F., Hahnfeld, J., Naumann, A., Lopez-Gomez, J., Lazzari Miotto, G., Padulano, V.E.: ROOT's RNTuple I/O Subsystem: The Path to Production. EPJ Web of Conf. **295**, 06020 (2024). `https://doi.org/10.1051/epjconf/202429506020`
6. Blomer, J., Canal, P., Naumann, A., Piparo, D.: Evolution of the ROOT Tree I/O. EPJ Web Conf. **245**, 02030 (2020). `https://doi.org/10.1051/epjconf/202024502030`
7. Brun, R., Rademakers, F.: ROOT — An object oriented data analysis framework. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **389**(1), 81–86 (1997). `https://doi.org/10.1016/S0168-9002(97)00048-X`
8. Chaarawi, M., Gabriel, E., Keller, R., Graham, R.L., Bosilca, G., Dongarra, J.J.: OMPIO: A Modular Software Architecture for MPI I/O. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) Recent Advances in the Message Passing Interface. pp. 81–89. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6

9. Eulisse, G., Konopka, P., Krzewicki, M., Richter, M., Rohr, D., Wenzel, S.: Evolution of the ALICE Software Framework for Run 3. EPJ Web Conf. **214**, 05010 (2019). https://doi.org/10.1051/epjconf/201921405010
10. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An Overview of the HDF5 Technology Suite and its Applications. In: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases. p. 36–47. AD '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1966895.1966900
11. Godoy, W.F., et al.: ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. SoftwareX **12**, 100561 (2020). https://doi.org/10.1016/j.softx.2020.100561
12. Hahnfeld, J., Blomer, J., Canal, P., Kollegger, T.: Direct I/O for RNTuple Columnar Data. EPJ Web Conf. **337**, 01034 (2025). https://doi.org/10.1051/epjconf/202533701034
13. Hahnfeld, J., Blomer, J., Kollegger, T.: Parallel Writing of Nested Data in Columnar Formats. In: Carretero, J., Shende, S., Garcia-Blas, J., Brandic, I., Olcoz, K., Schreiber, M. (eds.) Euro-Par 2024: Parallel Processing. pp. 18–31. Springer Nature Switzerland, Cham (2024)
14. Li, J., Liao, W.k., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., Zingale, M.: Parallel netCDF: A High-Performance Scientific I/O Interface. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing. p. 39. SC '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/1048935.1050189
15. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 5.0 (Jun 2025), https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf
16. Petrucciani, G., Rizzi, A., Vuosalo, C., on behalf of the CMS Collaboration: Mini-AOD: A New Analysis Data Format for CMS. Journal of Physics: Conference Series **664**(7), 072052 (dec 2015). https://doi.org/10.1088/1742-6596/664/7/072052
17. Rew, R., Davis, G.: NetCDF: An Interface for Scientific Data Access. IEEE Computer Graphics and Applications **10**(4), 76–82 (1990). https://doi.org/10.1109/38.56302
18. Serhan Mete, A., van Gemmeren, P.: Shared I/O Developments for Run 3 in the ATLAS Experiment. In: Proceedings of 41st International Conference on High Energy physics — PoS(ICHEP2022). vol. 414, p. 219 (2022). https://doi.org/10.22323/1.414.0219
19. Singh, S.P., Gabriel, E.: Parallel I/O on Compressed Data Files: Semantics, Algorithms, and Performance Evaluation. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). pp. 192–201 (2020). https://doi.org/10.1109/CCGrid49817.2020.00-74
20. Thakur, R., Gropp, W., Lusk, E.: Data Sieving and Collective I/O in ROMIO. In: Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation. pp. 182–189 (1999). https://doi.org/10.1109/FMPC.1999.750599
21. Thakur, R., Gropp, W., Lusk, E.: On Implementing MPI-IO Portably and with High Performance. In: Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems. p. 23–32. IOPADS '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/301816.301826
22. Vohra, D.: Apache Parquet, pp. 325–335. Apress, Berkeley, CA (2016). https://doi.org/10.1007/978-1-4842-2199-0_8

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_6