Summer Student Report 2025

on the topic

Performance Impact of Aggregator Configuration on Parallel HDF5 Writes in the MULTIPAC Data Acquisition Software

by

Lars Laurin Baltensperger[1]

Supervisors: Juliana Schell[2]
Björn Dörschel[2]
Ian Chang Jie Yap[2]

Date: July 31, 2025

[1]Eidgenössische Technische Hochschule Zürich (ETHZ), Zürich, Switzerland
[2]European Organization for Nuclear Research (CERN), Geneva, Switzerland

Abstract

This report investigates the impact of ROMIO aggregator configuration on the performance of parallel HDF5 writing in the MULTIPAC data acquisition software at CERN. Synthetic benchmarks were conducted using MPI-IO and h5py to emulate the high-throughput data flows encountered in real experiments. Performance was measured across a range of aggregator configurations, rank counts, and data sizes, with a focus on realistic DAQ scenarios. While the collected data revealed clear trends in the small and large data regimes, the results in the intermediate regime ($k \sim 10^7$) were inconclusive, and no single optimal aggregator configuration could be identified based on the available measurements. To address these ambiguities, a detailed plan for future evaluation is presented, including systematic exploration of advanced ROMIO tuning hints, fine-grained I/O tracing, and the implementation of an automated aggregator selection algorithm. These steps are expected to yield the information necessary to fully understand ROMIO's behavior in the MULTIPAC context and to identify the optimal configuration for future high-rate data taking.

The code used for the evaluation can be found in this github repository [1].

Contents

1 Introduction

The following sections introduce the scientific background, describe the key instrumentation and data acquisition system, and outline the parallel programming concepts and software frameworks relevant to the subsequent performance analysis.

1.1 What is ISOLDE?

The ISOLDE (Isotope Separator On-Line Device) facility, located at CERN, is a laboratory for the production of radioactive ion beams (RIBs) using the isotope separation on-line (ISOL) technique. Although ISOLDE is a relatively small facility, accounting for only about 0.1% of the CERN budget and 7 % of its scientists, it utilizes nearly 50% of the CERN protons. Since its founding in 1967, ISOLDE has continuously evolved to deliver beams to a broad range of physics experiments, spanning nuclear structure, atomic physics, solid-state research, and more.

At the core of ISOLDE's operation is the ISOL method. A pulsed proton beam of 1.4 GeV, with an average intensity of up to 2 μA, is delivered from the CERN Proton Synchrotron Booster onto thick targets. The interactions, spallation, fragmentation, or fission, produce a variety of radioactive nuclides within these targets. The products diffuse out of the heated target material and are then selectively ionized, often using laser ionization for improved elemental and even isomeric selectivity. The resulting ions are extracted and accelerated before entering one of ISOLDE's two mass separators: the General Purpose Separator (GPS) or the High Resolution Separator (HRS). These separators enable high-resolution mass selection, allowing researchers to obtain beams of high isotopic and isobaric purity. [2]

Through these methods, ISOLDE is able to provide access to $\approx$ 1500 different isotopes, including many that are not available elsewhere, supporting fundamental and applied research. The facility is continually upgraded, with improvements in beam quality, reliability, and safety, and is supported by dedicated infrastructure for beam manipulation, radiation handling, and advanced detection systems. The ongoing initiatives of ISOLDE, such as the HIE-ISOLDE upgrade and the MEDICIS project for medical isotope production, ensure that it remains at the forefront of radioactive ion beam science and a crucial resource for the international scientific community.

1.2 What is MULTIPAC?

The MULTIPAC [3] [4] project, installed at the ISOLDE experimental hall at CERN, is a new detector setup designed for the combined study of magnetic and electronic properties of materials. MULTIPAC integrates two techniques: vibrating sample magnetometry (VSM) and time-differential perturbed angular correlation (TDPAC or PAC) spectroscopy, enabling macroscopic and atomic-scale characterization within a single instrument and without moving the sample. A unique feature of MULTIPAC is its ability to operate under external magnetic fields of up to 8.5 Tesla, making it the only detector of its kind worldwide capable of performing TDPAC and VSM measurements at such high magnetic fields.

Vibrating Sample Magnetometer (VSM): The VSM mode is used to measure magnetic properties such as magnetic susceptibility, hysteresis curves, coercivity, and magnetic phase

transitions. In this technique, a sample is vibrated vertically within a strong, uniform magnetic field. This vibration induces a time-varying magnetic flux, which, according to Faraday's law, generates a voltage in nearby pickup coils. The induced voltage is directly proportional to the magnetization of the sample, allowing precise measurement of its magnetic response under varying external conditions.

Time Differential Perturbed Angular Correlation (TDPAC): The TDPAC mode probes the local, atomic-scale environment of implanted radioactive probe isotopes. At ISOLDE, suitable isotopes are produced and implanted into a material sample using the high-purity beams provided by the facility. These isotopes decay by emitting two gamma rays in cascade. The angular correlation and time difference between these two gamma rays are perturbed by the internal magnetic and electric fields of the host material, providing sensitive information about hyperfine interactions, local structural order, electronic environments, and defects. MULTIPAC's design enables these TDPAC measurements to be performed under a wide range of temperatures ($3K - 375K$) and applied magnetic fields, making it possible to investigate phase transitions and coupling phenomena in complex materials.

Detectors and Data Volume: For TDPAC measurements, the MULTIPAC setup utilizes four or six gamma detectors, that are composed of $LaBr_3$(Ce) scintillators coupled to silicon photomultiplier (SiPM) readouts. With six detectors, up to 30 distinct gamma-gamma coincidence spectra can be recorded simultaneously [3].

The basic functionality of the detectors in TDPAC is as follows:

Gamma rays first interact with the $LaBr_3$(Ce) scintillator, which absorbs the gamma energy and emits multiple low-energy photons in a cascade. The scintillator features a fast decay time of 25 ns, meaning that two gamma events separated by more than 25ns can be distinguished. In principle, this would allow for a maximum detection rate of 40 million gamma events per second (1/25 ns). These photons are then detected by the silicon photomultiplier (SiPM), which amplifies the signal and converts it into an analog electronic pulse. To make use of these signals for further analysis, a digitizer is needed to convert the fast analog pulses into precise digital data that can be recorded, processed, and interpreted by a computer.

The combination of several detectors and the ability to measure a large number of gamma-gamma coincidences leads to the collection of substantial data volumes in each experiment. This data-intensive nature of the measurements requires powerful digitizers and a high-performance computing setup for efficient acquisition and processing of all detector signals.

1.3 What DAQ hardware are we using?

To efficiently acquire and process the large data volumes generated by the MULTIPAC detector system, a high-performance data acquisition (DAQ) chain is required. This chain consists of one digitizer per detector and a powerful workstation consisting of 64 CPU's.

1.3.1 Digitizers

The MULTIPAC setup uses the Acqiris U5310A digitizer [5] to convert the electronic pulses from the SiPMs into digital signals for further analysis. The analog output from each detector is a continuous voltage signal: it stays at a baseline when no gamma ray is detected and forms

a distinct pulse (a hill-shaped rise and fall) whenever a gamma event occurs. Each such pulse is a segment of the analog signal and contains many data points over time.

The digitizer samples this continuous analog signal at a rate up to 10 GS/s, which corresponds to a time resolution of 100 ps per sample. At each sampling instant, the digitizer records the voltage as a single 10-bit digital number, meaning each data point is quantized into $2^{10} = 1024$ discrete levels over a 1 V input range (voltage resolution of 1/1024 V per level). In other words, for every sample, the digitizer produces a 10-bit number representing the instantaneous voltage at that moment. Over the duration of a pulse, this process creates a sequence of 10-bit numbers (samples) that collectively capture the full shape of each event (as seen on Fig.1), allowing for precise offline analysis of pulse characteristics.

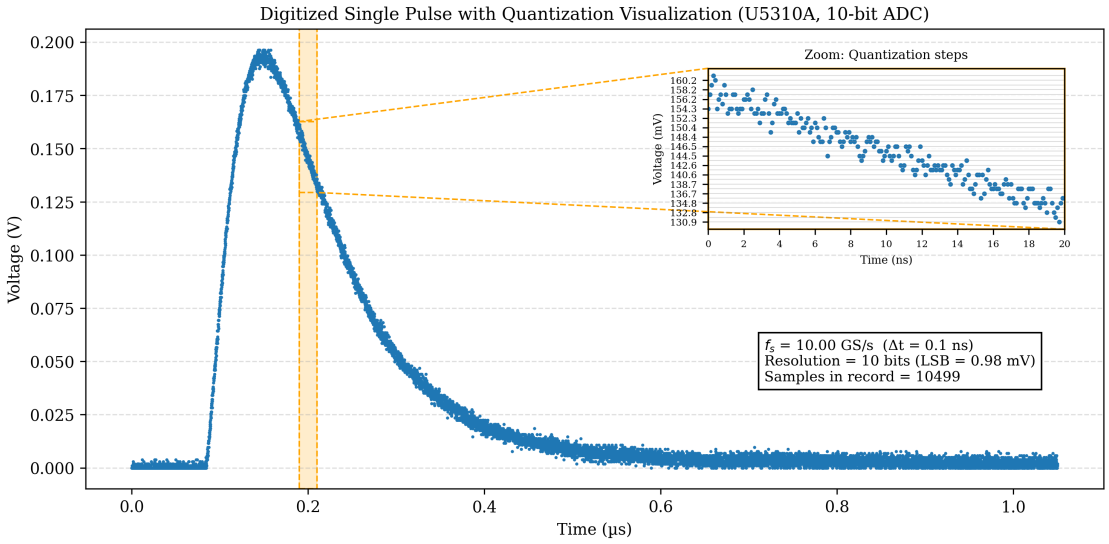

Figure 1: Digitized single pulse acquired with the U5310A digitizer. The signal is sampled at $f_s = 1/\Delta t = 10.00$ GS/s ($\Delta t = 0.1$ ns). The 10-bit ADC resolution results in a quantization step size of LSB = 1 V/2^{10} = 0.98 mV, visible as discrete levels in the zoomed-in inset. A total of 10,499 samples are shown.

At the maximum sampling rate, this results in a raw data acquisition rate of $(10\,\text{GS/s} \times 10\,\text{bits/sample})/8 = 12.5\,\text{GB/s}$ per digitizer, which can be temporarily buffered in the digitizer's 4 GB on-board memory. However, the transfer of digitized data from the digitizer to the host computer is limited by the PCIe Gen2 x8 cables, which support a maximum sustained data rate of about 4 GB/s per digitizer [6]. This means that while short bursts of high-rate data can be accommodated in the internal memory, continuous streaming to the computer is constrained by this PCIe bandwidth.

Furthermore, if the digitizer sampled continuously at 10 GS/s, the vast majority of the 10-bit values recorded would simply correspond to baseline voltages with no gamma event, and only a tiny fraction would capture the brief pulses of interest. This is because the decay time of the scintillators is 25 ns (and thus the actual rate of gamma-ray pulses), which is much longer than the digitizer's time resolution.

Therefore, the MULTIPAC system employs a trigger setup that only records data when a

pulse is detected. This approach further reduces the transport speed to about 2.5GB/s, while ensuring that the output consists primarily of relevant pulse information rather than an overwhelming stream of near-zero baseline values. Nonetheless, the high sampling rate and resolution of the digitizer guarantee that even the fastest detector signals are faithfully captured when they do occur, supporting high-precision TDPAC measurements.

1.3.2 Processor

The data acquisition system for MULTIPAC is built around a workstation with an AMD Ryzen Threadripper PRO 5995WX processor [7], featuring 64 physical CPU cores. Each core operates at a base clock of 2.7 GHz, meaning it can execute instructions of up to 2.7 billion cycles per second under standard conditions. In each cycle, the processor may perform one or more fundamental operations, such as arithmetic calculations, data transfers, or control instructions. The clock speed can dynamically increase to up to 4.5 GHz when thermal and power conditions allow.

The wokrstation is equipped with 256 GB of RAM (Random Access Memory), which functions as the computer's working memory. RAM temporarily holds data, instructions, and intermediate results that the processor needs to access rapidly during active computation. This is important since loading data from RAM is much faster than accessing it from the computers long-term storage. This rapid access and large capacity enables the processor to buffer, process, and analyze large volumes of incoming data from the digitizers in real time, before storing results to disk.

The processor architecture consists of eight Core Complex Dies (CCDs), each with eight cores, interconnected via AMD's Infinity Fabric 2. The system is equipped with eight DDR4-3200 memory channels. A memory channel is an independent data path between the processor and RAM, and having multiple channels allows for simultaneous data transfer to and from different memory modules. This configuration achieves a total memory bandwidth of up to 204 GB/s, describing the rate at which data can move between the CPU and RAM.

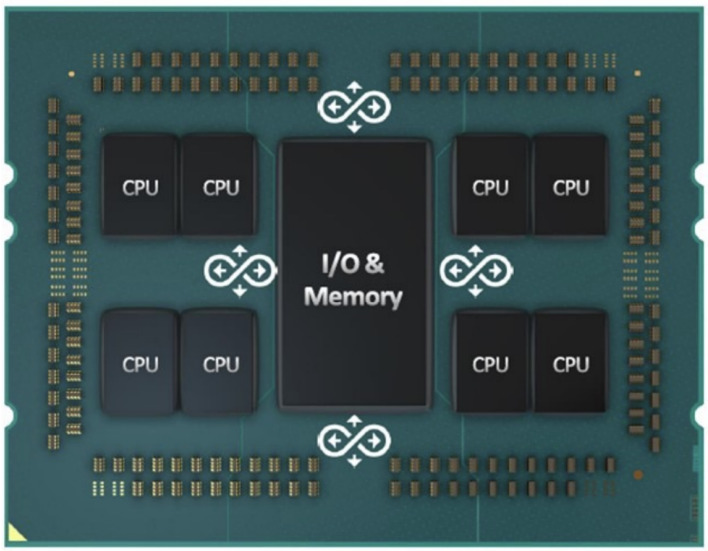

Figure 2: Schematic representation of the AMD Ryzen Threadripper PRO processor architecture. The diagram shows eight Core Complex Dies (CCDs), arranged around a central I/O and memory die. All CCDs are interconnected via AMD's Infinity Fabric (indicated by the white infinity symbols), which enables high-speed data transfer between all processor components [7].

There is also an option to enable Simultaneous Multithreading (SMT) on the Threadripper PRO platform. With SMT, each physical core is split into two logical cores (also known as logical processors), allowing a total of 128 logical processors to be presented to the operating system. A physical core is a dedicated hardware unit capable of executing instructions, whereas a logical core is a virtual processing unit that shares the physical core's resources. Each logical core can execute a sequence of instructions called a thread. Threads represent independent streams of execution within a program, allowing different tasks to be carried out simultaneously. SMT enables each physical core to handle two separate threads at the same time, making more efficient use of the CPU's resources .

1.4 What is Parallel-Programming?

Parallel programming enables computational tasks to be distributed and executed simultaneously across multiple CPU cores, making it possible to solve larger problems or accelerate data processing. In practice, a parallel program is typically written as a single code script. When running the script, the user specifies the number of CPU cores to be used. Each CPU core (or process) enters the program and, depending on how the code is structured, performs designated tasks.

In a parallel program, the individual CPU cores (or processes) are typically referred to as *ranks*. Each rank is assigned a unique integer identifier (`rank`), ranging from 0 to $N-1$, where N is the total number of cores that have been used to launch the script. This naming convention is also used throughout the code examples in this report.

1.4.1 What is MPI?

The Message Passing Interface (MPI) is a widely used standard for developing parallel programs on multi-core processors. MPI provides a portable library for communication between cores, enabling efficient and reliable exchange of data and coordination of tasks. With MPI, computational work can be divided across many processes, which can communicate using both point-to-point (one-to-one) and collective (group-wide) operations. Importantly, the MPI standard also includes advanced features such as parallel I/O (MPI-IO), dynamic process management, and remote memory access.

There are several implementations of MPI available for different platforms. For Windows-based systems, WS-MPI (Windows Subsystem for MPI) is an optimized implementation that provides the standard MPI interface and supports the features required for developing and running parallel programs under Windows. In Python, the `mpi4py` library provides bindings to MPI, making it possible to write and run parallel programs efficiently on clusters or high-core-count workstations.

1.4.2 What is h5py?

When processing large experimental datasets, a common bottleneck is writing data to disk, meaning data can be generated much faster than it can be stored. The HDF5 (Hierarchical Data Format version 5) [8] file format is designed to efficiently manage, store, and organize massive amounts of numerical data. The h5py Python library provides a high-level interface for reading and writing HDF5 files directly from Python, allowing structured storage of large arrays, tables, and metadata in a single portable file.

HDF5 files support parallel access patterns, making them suitable for use in parallel computing environments where data from multiple digitizers or processors needs to be written to the same file concurrently. In our context, we use h5py with MPI-IO to efficiently write measurement data from all our digitizers in parallel into a single HDF5 file, enabling scalable data acquisition and processing for the MULTIPAC experiment.

1.4.3 What is ROMIO?

ROMIO [9] is the part of the code in MPI that coordinates the parallel writing and reading of data into a shared file. It is essential because, without ROMIO, each core would either have to write to its own individual file, making data management complex and unscalable, or multiple cores might try to write to the same file independently, potentially overwriting each other's data or causing file corruption and inconsistent results. ROMIO solves these problems by providing a mechanism that allows multiple cores to efficiently and safely access a single shared file. It coordinates the splitting of data across the cores, organizes and merges their access patterns, and determines the order in which data is written. This ensures that parallel file I/O is both correct and highly performant.

A key concept in ROMIO's design is that of the **aggregator**. Rather than having every core interact directly and independently with the file system, leading to many small, inefficient I/O operations, ROMIO designates a subset of the available cores as aggregators. These aggregators act as representatives, gathering data from the other processes, combining and arranging it into larger contiguous chunks, and then performing the actual file I/O on behalf

of the group.

Under the hood, ROMIO achieves this parallel file I/O through the following main steps [10]:

a) **Choosing Aggregators:** ROMIO determines which cores will act as aggregators based on user-provided hints (such as `cb_nodes` and `cb_config_list`) or default values. While `cb_nodes` sets the requested number of aggregators, the `cb_config_list` parameter ultimately determines which ranks are selected, overwriting the value of `cb_nodes` in the code [11]. By default, `cb_config_list = "*:1"` [12], meaning that only one aggregator is used; thus, even if many cores participate in parallel I/O, only a single core will actually write to the file.

b) **Determining Chunk Size:** The total data is split among the aggregators so that each is responsible for a contiguous domain of the file. The `file_domain` size per aggregator is typically given by `file_domain_size = total_file_size / num_aggregators`. The `cb_buffer_size` hint then determines how much of that domain is written in a single I/O operation per aggregator, allowing for further tuning of parallel file access behavior. The default value is given by `cb_buffer_size = "16777216"` (16 MB) [13].

c) **Two-Phase I/O:**

   i) **Aggregation phase:** All participating cores send their data to the designated aggregator. Each aggregator collects the incoming data and merges it into a large, contiguous buffer corresponding to its assigned file domain.

   ii) **I/O phase:** Each aggregator writes its buffered data as a large, contiguous chunk to the underlying file system. This approach significantly reduces the number of small, inefficient I/O operations, increasing throughput and reducing filesystem contention.

2 Evaluation

The main goal of the evaluation is to systematically measure how the number of ROMIO aggregators affects the performance of parallel writing of data into an HDF5 file using `h5py` with MPI-IO. Two different code implementations were compared: the *all_ranks* version, where all cores used to launch the script participate in I/O collectives, and the *split_ranks* version, where only a subset of cores perform the actual file writing.

To facilitate this evaluation, minimal MPI Python scripts were developed to save data from multiple cores into a single HDF5 file in parallel. To realistically simulate the data flow from multiple digitizers, the total number of cores used to launch the scripts is divided into two groups: *creator ranks* and *writer ranks*. The creator ranks mimic digitizer output by generating random integer data. Each creator then sends its generated data to a designated writer rank. The writer ranks, in turn, are responsible for performing the actual file write operations into the shared HDF5 file. This setup was tested using 4, 8, and 12 ranks, corresponding to scenarios with 2, 4, and 6 simulated digitizers, respectively.

2.1 Timing Strategy and Implementation

For each experiment, the total elapsed time for parallel I/O was measured. Below are code excerpts illustrating the main difference in timing and rank setup for both implementation variants:

```python
# ── TIMING START ──────────────────────────────────────────
t0 = MPI.Wtime()

# ── PARALLEL HDF5 I/O ON COMM_WORLD ──────────────────────
with h5py.File(filename, 'w', driver='mpio', comm=comm, info=info) as f:
    # collective create
    dset = f.create_dataset('dataset', (num_of_core_pairs * k,), dtype=np.uint16)

    if rank >= num_of_core_pairs:
        start = writer_id * k
        end   = start + k
        dset[start:end] = data

# ── TIMING END & REDUCE ──────────────────────────────────
t1 = MPI.Wtime()
```

(a) all_ranks version

```python
# ── TIMING START ──────────────────────────────────────────
t0 = MPI.Wtime()

# ── SPLIT OUT WRITER COMMUNICATOR ─────────────────────────
# everyone calls Split, but non-writers get COMM_NULL
color = 0 if rank >= num_of_core_pairs else MPI.UNDEFINED
writer_comm = comm.Split(color=color, key=rank)

if writer_comm != MPI.COMM_NULL:
    wrank = writer_comm.Get_rank()

    # ── PARALLEL HDF5 I/O ON writer_comm ──────────────────
    with h5py.File(filename, 'w', driver='mpio', comm=writer_comm, info=info) as f:
        # collective create on writer_comm
        dset = f.create_dataset('dataset', (num_of_core_pairs * k,), dtype=np.uint16)

        start = wrank * k
        end   = start + k
        dset[start:end] = data

    # ── TIMING END & REDUCE ──────────────────────────────
    t1 = MPI.Wtime()
```

(b) split_ranks version

Figure 3: Comparison of the timed file writing sections for the *all_ranks* and *split_ranks* implementations.

In the *all_ranks* version, all cores that have been used to launch the script (including both the creator and writer ranks) are used to open the hdf5 file. As a result, ROMIO can select any of these ranks as aggregators, although only the writer ranks actually possess data to write.

In contrast, in the *split_ranks* version, only the writer ranks participate in opening the hdf5 file. Consequently, ROMIO is restricted to choosing aggregators exclusively from this subset, reducing the pool of potential aggregator ranks compared to the *all_ranks* setup.

2.2 Evaluation Parameters

Parameter Selection The evaluation was conducted using a systematic sweep across several key parameters:

- **Number of MPI ranks:** Experiments were performed with 4, 8, and 12 total ranks, corresponding to 2, 4, and 6 simulated digitizers, respectively.
- **Number of aggregators:** The number of aggregators in each script was controlled via the `cb_config_list` ROMIO hint. The following values were tested: `"*:1"`, `"*:2"`, `"*:3"`, `"*:4"`, `"*:6"`, `"*:8"`, `"*:12"`. In total, 7 different aggregator configurations were used for each run, subject to the constraint that the number of aggregators does not exceed the number of participating writer ranks.
- **Data size per writer:** The number of elements written per writer rank, denoted as k, was varied across 6 distinct values (logarithmically spaced) to simulate a wide range of realistic data acquisition scenarios, ranging from $k = 1000$ up to $k = 100\_000\_000$.

Data Type Consideration All synthetic data saved in the evaluation was stored as `uint16` (unsigned 16-bit integer) arrays. This choice was motivated by the characteristics of the Ac-

qiris U5310A digitizer, which produces 10-bit output per sample. Since 10 bits is not a native data size in common computing architectures, the next largest power-of-two data type, 16 bits (2 bytes), was selected. This ensures that every recorded data value fits comfortably within a single array element, avoiding the need for inefficient bit packing or custom serialization routines.

2.3 Repeatability

To ensure that our evaluation results were as repeatable and robust as possible, several measures were taken to minimize the effects of system noise and uncontrolled background activity.

By default, the operating system distributes all system and background tasks across all available CPU cores. This means that when launching an evaluation script, there could be background processes running on the same cores used for measurement, leading to variability in processing times from run to run. In our tests, the operating system was not restricted to running exclusively on certain cores, and thus some level of background interference was unavoidable.

To improve repeatability, we utilized a program called `Process Lasso` to manage CPU affinities. CPU affinity refers to binding a given process or thread to specific CPU cores, thereby restricting where the operating system is allowed to schedule that process. By setting affinities, we aimed to ensure that our evaluation scripts would only run on selected cores, minimizing interference from other system activities. Specifically, we set all system processes to run on the first Core Complex Die (CCD), i.e., cores 0–7, while our evaluation scripts were launched on the last two CCDs (cores 52–63) using the `psutil` Python library. This approach was intended to isolate our measurement processes from most background system activity, thereby reducing measurement noise as much as possible.

Additionally, Simultaneous Multithreading (SMT) was disabled during all evaluations. This ensured that only the 64 physical cores were available for computation, providing one fully isolated core per MPI rank and minimizing the risk of resource contention between logical siblings.

Although this method significantly improved repeatability, it is important to note that CPU affinity assignments are not strict guarantees: certain system processes cannot be pinned to specific cores and may still run on the same cores as our evaluation scripts. Therefore, complete isolation could not be achieved, but the frequency and severity of interference were minimized.

To further enhance statistical rigor, each measurement was repeated 300 times for every parameter configuration. The mean elapsed times were then used for analysis and reporting, providing a robust average that accounts for any residual variability.

2.3.1 Verification of Threadripper Topology

In addition to process isolation, we sought to verify the actual core layout and topology of our AMD Threadripper system. To this end, we created a core-to-core latency map 4 using the library [14] by measuring the communication latency between each pair of cores. This mapping allowed us to double-check the processor's CCD structure and to quantify the latency between CCDs as well as between individual cores within the same CCD.

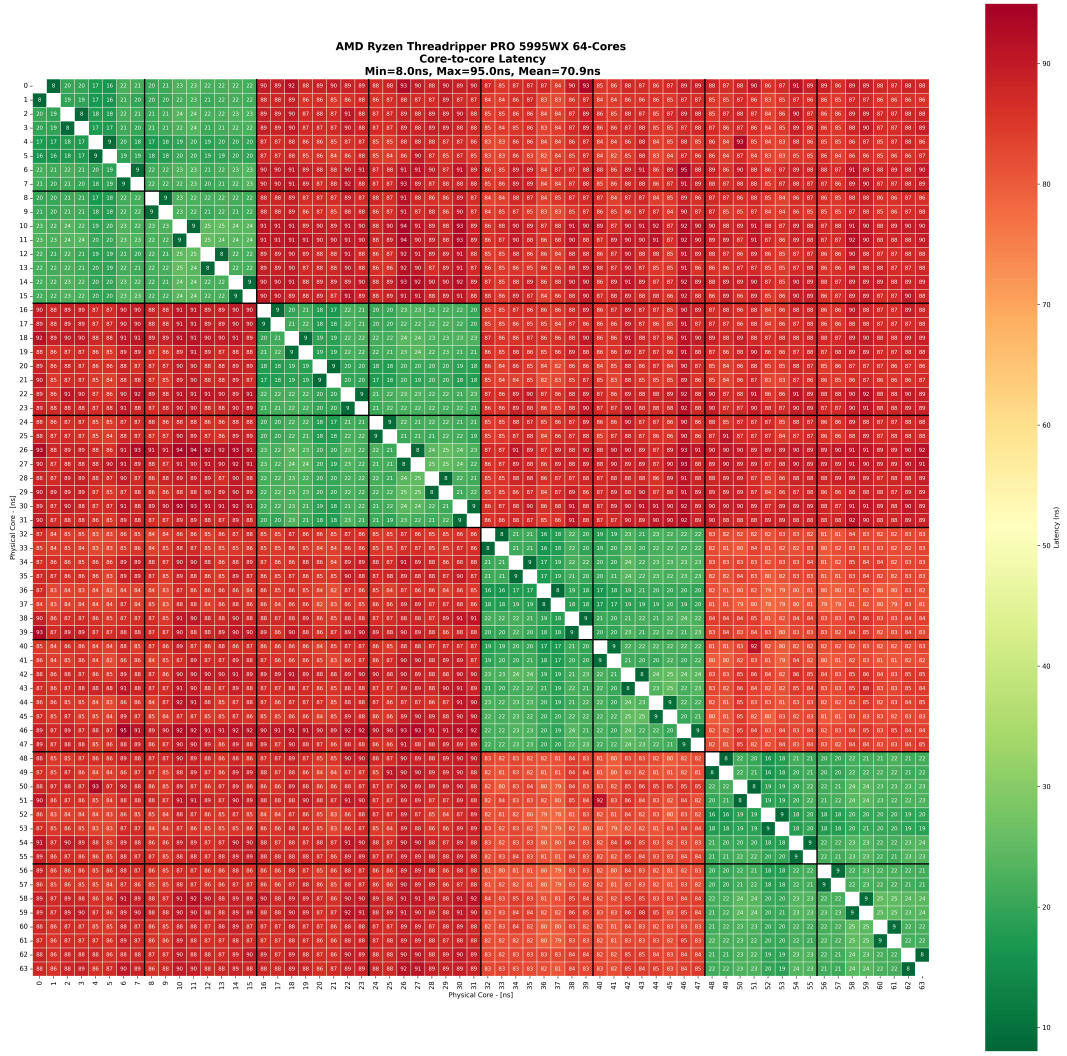

Figure 4: Core-to-core latency heatmap for the AMD Ryzen Threadripper PRO 5995WX 64-core processor. Each cell indicates the measured communication latency (in nanoseconds) between a pair of physical cores. The eight green blocks along the diagonal correspond to the eight Core Complex Dies (CCDs), with low intra-CCD latencies (∼10–20 ns). The green blocks just above and below the diagonal indicate faster communication paths provided by the Infinity Fabric between these CCDs. Higher latencies (yellow/red) are observed between cores located on different CCDs without a direct fast interconnect. The heatmap validates the expected structure and topology of the processor.

3 Results

The results of the performance evaluation are presented in Figure 5, which displays the speed-up factor (ratio of median I/O times) between the `all_ranks` and `split_ranks` versions as a function of the number of datapoints per writer, k. In this context, values above 1 indicate that the `split_ranks` version achieves faster I/O, while values below 1 signify that the `all_ranks` version is faster. Each curve corresponds to a different aggregator configuration, allowing direct comparison of the impact of aggregator selection for fixed k and rank count. The vertical spread between the curves at a given value of k quantifies the effect of varying the aggregator configuration on parallel write performance.

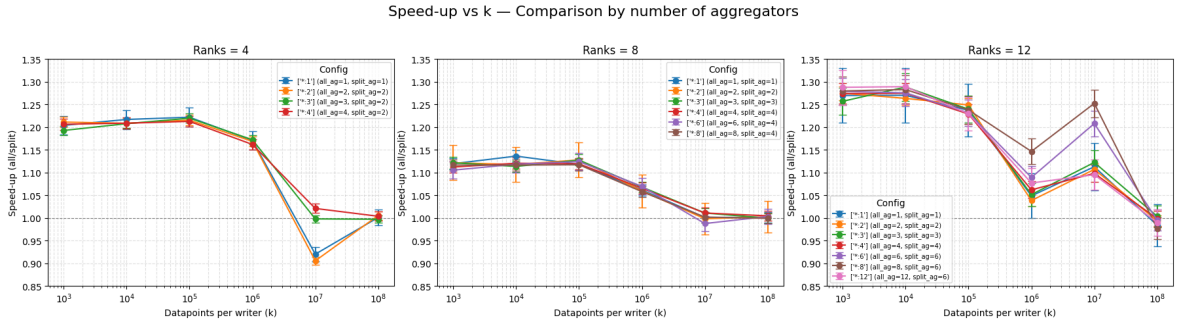

Figure 5: Speed-up (all/split) as a function of datapoints per writer (k), shown for three different numbers of MPI ranks (4, 8, 12). Each curve represents a specific aggregator configuration (`cb_config_list`), where the legend indicates the number of aggregators used in the all_ranks and split_ranks variants. Error bars denote the propagated standard error of the median, computed from repeated runs as described in Section 3.1. The y-axis shows the ratio of median I/O times between the all_ranks and split_ranks versions, while the x-axis is on a logarithmic scale.

3.1 Error Propagation

The error bars shown in all performance plots represent the propagated standard error of the mean (SEM) for the speedup ratio between the `all_ranks` and `split_ranks` versions. For each experimental configuration (number of ranks, aggregator setting, and data size), we conduct 300 independent measurement runs and record the timing for each run.

For each set of timings (e.g., all_ranks, split_ranks), we compute:

- The median run time, used as the central value for plotting.
- The standard error of the mean (SEM), defined as $\text{SEM} = \sigma/\sqrt{N}$, where σ is the standard deviation of the timing measurements and N is the number of runs.

To compute the error bars for the speedup ratio $S = t_{\text{all}}/t_{\text{split}}$, we propagate the SEMs for the individual timings using the formula for the Gaussian error propagation in the ratio of two independent variables:

$$\text{SEM}_S = S \cdot \sqrt{\left(\frac{\text{SEM}_{\text{all}}}{t_{\text{all}}}\right)^2 + \left(\frac{\text{SEM}_{\text{split}}}{t_{\text{split}}}\right)^2}$$

where t_{all} and t_{split} are the median timings for the `all_ranks` and `split_ranks` versions, and SEM_{all} and $\text{SEM}_{\text{split}}$ are their corresponding SEMs. This approach allows us to visualize the uncertainty in the observed speedup arising from variability in both measurements.

4 Conclusion

The analysis of the performance data reveals three distinct behavioral regimes, each characterized by different scaling properties and sensitivities to the aggregator configuration. The following sections summarize the key observations and explanations for each of these regimes.

4.1 Small number of datapoints ($k < 10^7$)

For small values of k, the `split_ranks` version consistently outperforms the `all_ranks` version (i.e., speed-up > 1), as can be seen in Figure 5. This effect occurs because ROMIO internally uses collective operations (such as barriers and communicator setup) whose overhead increases with the number of participating ranks. In the `all_ranks` variant, twice as many ranks participate, resulting in substantially higher fixed collective costs. When the amount of data per process is small, these setup and synchronization overheads dominate the total runtime.

4.2 Large number of datapoints ($k = 10^8$)

For very large k, both the number of aggregators and the variant (`all_ranks` or `split_ranks`) have negligible impact; all curves collapse to a speed-up of 1, as can be seen in Figure 5. In this regime, the total runtime is dominated by the time required to transfer large data blocks to storage, so the system is limited by the hardware's maximum bandwidth (e.g., the SSD). Here, any overhead from communicator setup, collective operations, or aggregator configuration is insignificant compared to the time spent on I/O. This trend is consistent with previous studies in parallel I/O, which observe that tuning aggregator counts yields diminishing returns at very large data sizes [15].

4.3 Intermediate datapoints ($k = 10^7$)

For intermediate data sizes, the performance trends are more complex and vary with the number of ranks:

- For 4 ranks: With `cb_config_list` $= *:1, *:2$, the `all_ranks` variant is faster despite both having the same number of aggregators; with `cb_config_list` $= *:3, *:4$, the speed-up approaches 1, indicating little difference.
- For 8 ranks: All data points are near 1, suggesting that both variants and all aggregator configurations perform similarly.
- For 12 ranks: The `split_ranks` version is notably faster, with speed-ups above 1 for all aggregator settings; again, the precise number of aggregators appears to have minimal effect.

At $k = 10^7$, these differences cannot be conclusively explained by the currently available data. To better understand the behavior in this regime, more detailed system-level information would be required, including:

1. **Aggregator selection and communication patterns:** It is necessary to know which ranks are chosen as aggregators and how data from writer ranks is distributed among them, as this affects data movement efficiency and load balancing.
2. **Buffer size configuration:** The performance can depend strongly on the size of the buffer each aggregator uses for I/O, as mismatched sizes can introduce overhead or underutilize hardware. This can be tuned by using the ROMIO hint `cb_buffer_size`.
3. **Processor topology:** On multi-CCD processors, core placement becomes important: if writers and aggregators are distributed across different CCDs, cross-CCD data transfers may introduce extra latency.
4. **Additional ROMIO hints:** Further ROMIO hints than the ones mentioned so far, such as for data sieving, deferred file opens, and collective buffering options, could be used to explain the current behaviour or further optimize performance.
5. **More detailed tracing:** Gathering fine-grained information on aggregator assignments, data distribution, communication timing, and resource utilization is necessary to attribute observed performance effects more precisely.

Without this information, the causes of performance variations at $k = 10^7$ remain unclear.

5 Future Work

The present study has highlighted both the complexity and significance of ROMIO aggregator configuration for optimal parallel HDF5 write performance in the MULTIPAC DAQ software. Building on these findings, several promising directions for future research can be identified. These span the systematic exploration of additional ROMIO tuning hints, the application of advanced I/O tracing and analysis tools, and the implementation of automated aggregator selection algorithms. In the following, each direction is discussed in detail.

5.1 Systematic Exploration of ROMIO Hints

While the current evaluation focused primarily on the effects of aggregator count and buffer size, ROMIO exposes a broader set of hints that may have a substantial impact on I/O performance. These include:

- `romio_no_indep_rw`: Disables independent read and write capabilities, enabling *deferred file opens*. This reduces metadata operations and may improve scalability when only collective I/O is performed.
- `romio_cb_write`: Controls whether collective buffering is used for collective writes, which can impact synchronization overhead and parallel efficiency.
- `cb_buffer_size`: Sets the buffer size used by aggregators in collective operations. This can influence the size and number of I/O requests as well as memory usage on aggregator processes.
- `cb_config_list`, `romio_aggregator_list`: Allow explicit selection and placement of aggregator ranks, enabling topology-aware optimization (e.g., aligning aggregators with CCDs).

- `romio_cb_alltoall`: Specifies the algorithm used to redistribute data from non-aggregators to aggregators during collective I/O, for example, using `MPI_Alltoallv` versus point-to-point communication.
- `romio_cb_fr_alignment`, `romio_cb_ds_threshold`: Control file region alignment and thresholds for collective buffering, which can optimize I/O performance by aligning accesses to file system or hardware boundaries.

Given the observed sensitivity of performance to aggregator settings at intermediate data sizes, particular emphasis should be placed on the datapoint range around $k \sim 10^7$, which both corresponds to the volume of data expected in MULTIPAC experiments and exhibits the most pronounced differences between ROMIO configurations. For each experimental scenario, all active ROMIO hints should be recorded, and the corresponding parameter values documented. Where possible, topology-aware aggregator placement (e.g., mapping aggregators to the same CCD) should be explored using `cb_config_list` or `romio_aggregator_list`. Moreover, the feasibility and effect of enabling collective-only I/O (disabling independent operations) should be tested by setting `romio_no_indep_rw` to `true`, particularly in the split ranks variant.

Such a comprehensive exploration would not only enable a more thorough understanding of performance anomalies observed at intermediate data sizes but may also yield new configurations that maximize throughput and scalability for future MULTIPAC data taking campaigns.

5.2 Fine-Grained I/O Analysis Using Recorder

The present evaluation measured only aggregate I/O times, which is insufficient to reveal the internal mechanisms, communication patterns, and bottlenecks that ultimately determine performance. To address this limitation, future studies should incorporate advanced I/O tracing tools such as Recorder [16]. Recorder enables transparent, low-overhead capture of all I/O calls and metadata at the HDF5, MPI-IO, and POSIX (Linux file system) levels, including function parameters, timestamps, aggregator assignments, and collective operation details.

It should be noted, however, that Recorder is implemented for POSIX-compliant (Linux/Unix) systems and relies on features such as `LD_PRELOAD` for dynamic function interception. As a result, Recorder cannot be used on native Windows systems. For fine-grained I/O tracing with Recorder, experiments would therefore need to be conducted in a Linux environment, either on dedicated hardware, a virtual machine, or a suitable cluster.

By enabling Recorder during parallel I/O benchmarks, one can obtain detailed traces that reveal the full sequence of I/O operations, aggregator assignments, buffer usage, and synchronization events. Such traces enable in-depth analysis of performance bottlenecks, including unaligned accesses, metadata overhead, and suboptimal aggregator placement, and make it possible to directly correlate I/O behavior with ROMIO configuration settings. This insight is especially valuable for interpreting complex performance trends in the intermediate data size regime ($k \sim 10^7$). Furthermore, Recorder preserves a comprehensive record of I/O activity for each experiment, supporting reproducibility and enabling retrospective analysis as system conditions or research questions evolve.

5.3 Automated Aggregator Selection Using the AutoIO Algorithm

A promising approach to eliminating the need for exhaustive manual tuning is the use of runtime algorithms that can dynamically determine the optimal number of aggregators for collective I/O operations. The AutoIO algorithm [15], originally implemented in the OMPIO component of Open MPI, provides such a method by considering both the system-specific write saturation point and the application's data distribution.

Adapting the AutoIO algorithm for use with ROMIO-based MPI-IO on the MULTIPAC platform would involve several steps. First, a simple microbenchmark would be used to determine the per-core write saturation point k for the current system and storage stack. Then, for each collective I/O operation in the DAQ workflow, the total data volume to be written would be computed, and the number of aggregators set according to $N_{\text{agg}} = \lceil S/k \rceil$, where S is the total data size for the operation. Where possible, process topology (e.g., cartesian communicators or data decomposition) could be used to further optimize the grouping and assignment of aggregator ranks.

Because ROMIO does not currently support automatic runtime aggregator selection, the AutoIO logic would need to be implemented as a wrapper or preprocessing layer in the experiment control scripts. For each experimental run, the wrapper would compute the recommended aggregator count and configure the appropriate ROMIO hints (e.g., `cb_nodes`, `cb_config_list`) before launching the parallel I/O job. The performance of this auto-selected configuration could then be compared directly to both hand-tuned and default settings, across the full range of data sizes relevant for MULTIPAC.

Such an approach promises not only to reduce the configuration burden for end-users and operators but also to generalize performance optimization across changing hardware and experimental conditions.

References

[1] Lars LaurinBaltensperger. *Evaluation code*. URL: https://github.com/lbaltensp/CERN_report.

[2] R Catherall et al. "The ISOLDE facility". In: *Journal of Physics G: Nuclear and Particle Physics* 44.9 (Aug. 2017), p. 094002. DOI: 10.1088/1361-6471/aa7eba. URL: https://dx.doi.org/10.1088/1361-6471/aa7eba.

[3] Björn Dörschel. "ISOLDE summer student report, Commissioning of the MULTIPAC detectors". In: *CERN records* (June 2023). URL: https://repository.cern/records/s1a3g-p7v93/preview/ISOLDE%20summer%20student%20report%20Comissioning%20of%20the%20MULTIPAC%20detectors.pdf.

[4] J. H.-Schell et al. "MULTIPAC-Setup for Perturbed Angular Correlation Experiments in Multiferroic (and Magnetic) Materials: proof-of-concept". In: *CERN indico* (Jan. 2023). URL: https://indico.cern.ch/event/1246149/attachments/2580711/4451042/INTC-I-249.pdf.

[5] Acqiris. *U5310A Digitizer*. URL: https://acqiris.com/hardware/10-bit/.

[6] Wikipedia. *PCIe*. URL: https://en.wikipedia.org/wiki/PCI_Express#History_and_revisions.

[7] AMD Ryzen. *AMD Ryzen Threadripper PRO 5995WX*. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/white-papers/threadripperpro-white-paper-third-wave-workstation-computing.pdf.

[8] *h5py docs*. URL: https://docs.h5py.org/en/stable/.

[9] Rajeev Thakur. *Users Guide for ROMIO: A High-Performance,Portable MPI-IO Implementation*. note. 2004. URL: https://ftp.mcs.anl.gov/pub/romio/users-guide.pdf.

[10] *ROMIO: code repository*. URL: https://github.com/microsoft/Microsoft-MPI/tree/master/src/mpi/msmpi/io.

[11] *ROMIO: where cb_config_list overwrites cb_nodes*. URL: https://github.com/microsoft/Microsoft-MPI/blob/master/src/mpi/msmpi/io/ad_file.cpp#L100.

[12] *ROMIO: where default value of aggregators is implemented*. URL: https://github.com/microsoft/Microsoft-MPI/blob/master/src/mpi/msmpi/io/adioi.h#L132.

[13] *ROMIO: where default value of write size per aggregator is implemented*. URL: https://github.com/microsoft/Microsoft-MPI/blob/f2d849ff483b36add9bc2d18c6d7bbc9ce38f4ad/src/mpi/msmpi/io/adioi.h#L122C1-L123C53.

[14] *Core to core latency map github library*. URL: https://github.com/nviennot/core-to-core-latency?tab=readme-ov-file.

[15] Jialu Li, Wei-keng Liao, and Alok Choudhary. "Automatically Selecting the Number of Aggregators for Collective I/O Operations". In: *Proceedings of the 7th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE 2011)*. ACM, 2011, pp. 1–8. DOI: 10.1145/1996109.1996111. URL: https://www.researchgate.net/publication/221202255_Automatically_Selecting_the_Number_of_Aggregators_for_Collective_IO_Operations.

[16] Chen Wang et al. "Recorder: Comprehensive Parallel I/O Tracing and Analysis". In: *arXiv preprint arXiv:2501.04654* (2024). URL: https://arxiv.org/abs/2501.04654.