Lawrence Berkeley National Laboratory

LBL Publications

Title

Performance Analysis and Optimization for Scientific Data Workloads

Permalink

https://escholarship.org/uc/item/0rf47634

Authors

Giannakou, Anna
Ramakrishnan, Lavanya

Publication Date

2022-12-25

Peer reviewed

eScholarship.org

Powered by the California Digital Library
University of California

Performance Analysis and Optimization for Scientific Data Workloads

Anna Giannakou
Lawrence Berkeley National Lab
Berkeley, USA
agiannakou@lbl.gov

Lavanya Ramakrishnan
Lawrence Berkeley National Lab
Berkeley, USA
lramakrishnan@lbl.gov

***Abstract*—Scientific data generated at experimental and observational facilities are increasingly being processed on large-scale compute systems. Most of the experimental data analysis workflows are not designed or implemented to run on large scale environments and take full advantage of HPC compute and storage resources. These applications are unlike the traditional tightly-coupled scientific applications and hence face significant performance and scalability challenges as the volume of data increases exponentially. In this paper, we conduct a performance and scalability analysis for experimental analysis applications and workflows operating on data from light sources. Our analysis detects and quantifies I/O performance, scalability and runtime bottlenecks for three data analysis applications that run on NERSC resources. Based on our analysis we propose and implement a set of optimizations that lead to reducing the amount of time spent on I/O operations by almost 90%.**

***Index Terms*—performance, scalability, HPC, throughput**

I. Introduction

Science experiments are increasingly producing large amounts of data from a diverse set of instruments at increasingly high rates [1]. The data processing and analyses workflows requires experimental data to be analyzed fast, as intermediate results are used to steer and re-calibrate ongoing experiments [2]. Furthermore, the computational, storage and data transfer needs of the analyses necessitate the need for high-performance networking, storage, and compute resources. Scientists have turned to HPC platforms as an attractive solution for running this new class of complex, computationally intensive experimental-data analysis workloads. Although HPC environments grant access to significant compute and storage resources, most of the experimental data analysis workflows (e.g. X-ray imaging analysis) are not designed or implemented to run on such large scale environments. These applications are different from the tightly coupled scientific applications that have run on HPC systems. Oftentimes, a substantial amount of reconfiguration is required both for the underlying system and the workflows's codebase in order to enable reliable, portable execution across HPC environments.

This new class of science workflows requires a deeper understanding of performance and scaling challenges. We need to implement performance oriented optimizations are vital aspects for achieving fast, result turnaround. In order to take advantage of HPC resources and ultimately improve workflows's performance and scalability, timely identification and in-depth analysis of underlying bottlenecks and potential implementation inefficiencies is important. There is limited understanding of how we can be support experimental data analysis workflows on HPC systems. Experimental data workflows have specific characteristics, such as their ability to handle multiple input data streams and have not been studied in detail. This is in contrast to traditional HPC-oriented Message Passing Interface (MPI) workflows that have been thoroughly analyzed and modeled in terms of different performance aspects (e.g. I/O, compute, scalability, etc).

In this paper, we analyze two experimental data analysis workflows executed on resources at National Energy Research Scientific Computing Center (NERSC) focusing on I/O, runtime and scalability aspects. The first one is a Light Source domain workflow that utilizes Free Electron Lasers (XFELs) in order to generate millions of (slightly different) shots of unknown samples. The samples are later analyzed using the Computational Crystallography toolbox (CCTBX). The second workflow, Serial Femtosecond X-ray (SFX) crystalography, analyzes data collected from the Linac Coherent Light Source (LCLS) at SLAC National Accelerator Laboratory, focusing on protein structure discovery and was used in Covid-19 viral protein reconstruction. Our work provides a thorough overview of the and compute and I/O aspects of this new type of experimental data analysis workflows.

In this paper, our research has the following key contributions:

- We perform a analysis of different performance aspects of the two selected workflows (*cctbx* and *SFX*) Our work uses lightweight code instrumentation in order to detect and quantify underlying performance bottlenecks. Our analysis shows that for *SFX* the main bottleneck is redundant read and write operations while *cctbx*'s execution time is affected by the location of input data.
- we designed, implemented underlying system optimizations that aim at resolving or reducing the identified bottlenecks and associated performance penalties.
- Finally, we conduct a detailed evaluation of our performance optimizations and compare the initial and optimized versions of the workflows. For *cctbx* the execution time is reduced up to 36% while for *SFX* we manage to reduce the I/O footprint by 26%.

Identify applicable funding agency here. If none, delete this.

This paper is organized as follows: Section II presents a detailed description of the analysed workflows while and related research projects . Section III outlines our evaluation process along with selected metrics and setup as well as obtained results along with the optimizations that were applied in order to reduce the I/O footprint and overall execution time of each workflow.Section VI concludes the paper outlining important observations.

II. Background

In this section, we describe the individual stages for both workflows and related work on application profiling and performance characterization. A thorough understanding of different aspects of workflow performance is necessary in order to support this new class of experimental data analysis workflows.

A. Application and Workflows

1) Reader: For an in depth understanding of the reading process for data format produced by the Laser detectors and analysed both by *cctbx* and *Sfx* we analyze a *Reader* application running on NERSC resources. **Reader**'s sole role is to read produced data in large memory blocks called *Dataframes*. *Dataframes* are stored in a specific file format with the extension *xtc2* (extended tagged container). Since *xtc2* files can be quite large in size occupying up to terabytes (TB) of disk space, they are accompanied by a smaller *smd* file (around 20MB) that acts as an index for each *dataframe*'s location in the *xtc2* file. On NERSC's Cori both *xtc2* and *smd* files are located on permanent storage. Following MPI principles, the **Reader** uses a master process that distributes a user-defined number of events among a configurable set of worker processes. Each worker is responsible for accessing the *xtc2* file and reading the assigned events. No further processing is conducted. The only I/O activity is attributed to the open() and read() calls performed by the master and worker MPI processes. Each worker process utilizes *Psana* [3], the default data analysis framework for *xtc2* datasets, in order to access the assigned events. The **Reader** can be run in both single and multi-node environment depending on available resources (i.e. nodes, cores). Typical data analysis workflows contain a high number of events (e.g. 500 or 1000 events) assigned to each process, making scalability a crucial factor for minimizing both execution time as well as time spent on I/O operations related to data access.

2) XFEL Data Analysis- Cctbx: Free Electron Lasers (XFELs) are used in X-ray scattering experiments in order to determine the structure of unknown samples. Images collected from an XFEL are analyzed through the Computational Crystallography Toolbox (*cctbx*) software package, an open source tool. Since each experimental dataset contains hundreds of thousands of images with varying diffraction patterns, *cctbx* performance greatly depends on optimizing parallel image processing as well as underlying I/O.

We describe the individual stages for the *cctbx* workflow that run on NERSC resources focusing on I/O operations and parallelism. An overview of the *cctbx* pipeline is shown in Figure 1. Red boxes depict I/O intensive phases that are later profiled in our analysis and are executed through a configurable set of containerized workers (using NERSC's Shifter):

Marshaling: The first pipeline phase starts with a master process distributing an equal, configurable number of events and experiment related inputs that are applicable to all images (e.g. metrology, bad pixel mask, gain mask) among MPI worker processes. Each process is running in a dedicated container.

Import: In the second phase, each worker uses *Psana2* in order to read data from the large *xtc2* file (located on permanent storage) in the same manner described in the **Reader** application. This phase is the first of the I/O dependent operations. Once the images are read, a set of dark and gain corrections are applied and a cbf file is assembled per worker.

Spot Finding: The crystallography analysis starts with discovering sets of connected bright pixels using the dark pixel mask. Once these are found (based on input defined parameters) they are isolated for further processing.

Indexing: The discovered sets of bright pixels are indexed and the crystal orientation is defined based on the following factors:*metrology*, *location* and *input defined unit cell*. The successfully indexed images are prepared for the *refinement* phase creating one file per image. Only a subset of the initial images is going to be indexed successfully on each pipeline execution. The different diffraction patterns in images result in varying indexing rates and vastly fluctuating completion times for each worker. The I/O footprint of the indexing phase is attributed to the write calls for generating the .idx files.

Refinement: The model parameters are adjusted in order to ensure better data fit.

Integration: Spots are integrated using a simple summation of foreground pixels and removal of the estimation of background in this area.

Write out Results: Finally, each worker writes out results to a user-defined location that is common amongst all workers. Depending on user's choice the output is going to be comprised by either one set of files per process rank or one set of files per indexed image. The amount of I/O depends on the number of write() calls for the result files.

3) SFX: SFX is from LCLS and is used for analyzing and mapping the structure of **Covid-19** proteins that was successfully ported and executed on Cori. A schematic representation of the workflow stages is shown in Figure 2.

Peak Finding: In the first phase, images of the protein samples are analyzed in order to detect useful defraction patterns. Defraction patterns depend on the number of peaks found in each sample. *SFX* utilizes Psocake [4] for peak finding. While peak finding, the user is able to tune input parameters, launch multiple jobs on all available cores and inspect intermediate results for discovered number of peaks (a job is considered a success if more than 15 hits are found). The results are stored in a cxi file (one cxi file per process). Inspection of intermediate results is crucial in order to reduce

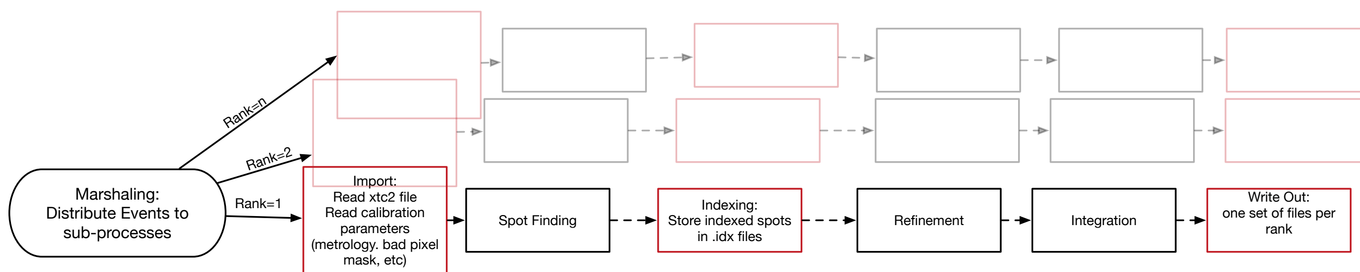


Fig. 1. **Cctbx** pipeline stages for multiple MPI sub-processes.

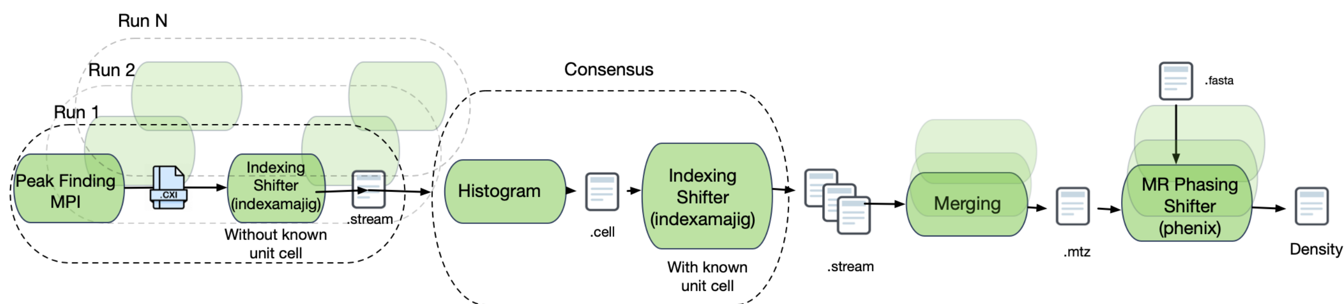


Fig. 2. SFX workflow stages on NERSC resources for Covid-19 protein structure determination. Black arrows represent the data flow for both input data and intermediate results

the computational load for the remaining workflow phases. Each job is running in a container.

Indexing: Provided that enough peaks were found, Psocake will attempt to index the defraction patterns. Although a percentage of jobs will terminate without successful indexing solutions, each job stores the obtained result in a separate stream file. Indexing is conducted through Psocake's *indexamajig*. Multiple jobs can be launched for Indexing.

Histogram and Indexing: In this step, the successfully indexed spots are combined to a single cell file. Then a second round of indexing occurs. The second indexing round is executed inside containers.

Merging: The stream files that contain the results from the second round of indexing are combined in a single *mtz* file after different scaling methods are applied (e.g. monte carlo scaling, simple scaling). Scaling is performed by Psocake's *process hkl*. Users can launch multiple merging jobs for faster phase turnaround time.

Phasing: In the final stage of the pipeline the samples are phased using either **SAD** or **MR** phasing techniques. Both techniques effectively conduct a combinatorial grid search. The main difference between the two techniques is that **MR** starts from a known solution, hence a substantially smaller search space, that leads to lower execution times. In both techniques, a master process launches multiple SLURM jobs as sub-processes that will carry out the search incorporating Phenix [5] and CCP4 [6]. The number of jobs that are launched depends on the size of the search space as well as the actual number of grids. Before each job is launched, the master process creates a sub-directory and copies an identical set of input and executable files. The number of copy operations increases with the number of launched jobs leading to large I/O overhead for increased grid sizes. The jobs are launched sequentially and the computed probabilities are stored in a common file while the master process waits for each job to complete. The result with the highest probability is selected as the phasing solution. The *SFX* workflow was running natively on LCLS resources requiring complex and time consuming configuration of associated software modules (e.g. *Phenix*, *CCP4*, etc)

B. Related work

In this section, we briefly describe research works around workflow profiling and characterization.

Users that execute workflows spawning multiple jobs on supercomputers can use *Darshan* tool [7] to profile the I/O usage of their applications. Darshan instruments I/O functions at multiple levels, primarily MPI-I/O and POSIX I/O. The statistics collected per job include: numbers of processes, bytes read/written, aggregate I/O throughput as well as total runtime and time spent in I/O. On Cori, users can use the dedicated Darshan parser tool in order to easily summarize I/O results. Darshan collects a minimal amount of data making it a suitable solution for monitoring I/O behavior at scale. Existing work [8] analyzes Darshan logs from thousands of applications running on NERSC and ALCF resources over a multi year period in order to discover I/O related bottlenecks and provide useful performance improvements suggestions. However, as previously noted the Darshan module on Cori was unable to track I/O for both applications.

A number of works have addressed performance characterization of workflows running on NERSC resources from different aspects. Previous work [9] focuses on the data transfer part of the workflow and uses decision based techniques to predict the transfer rate of different file sizes from LCLS to NERSC. The decision is based on previously collected performance metrics such as transfer duration transfer start time, source file system, etc. The work clearly recognizes the value of performance characterization for optimizing overall workflow performance. Feature selection and clustering [10] has been used to identify a set of configuration parameters to optimize I/O performance for different workflow types that run on NERSC resources. The authors utilize a variety of feature selection techniques such as F-regression and Mutual Information regression in order to isolate the most important I/O features. The analysis, based on Darshan logs, results to a general recommendation about the use of MPI for aleviating I/O bottlenecks and does not provide application specific optimization strategies. Daley et al. [11] profile a wide set of simulation and data-analytics workloads isolating communication, I/O and compute times and conclude that indeed I/O performance can be improved with the use of intermediate storage mechanisms such as burst buffer.

Betke and kunkel [12] utilize machine learning (i.e. K-means) to analyze past application traces in order to identify workload clusters that demonstrate similar I/O patterns. Although the proposed trace data analysis pipeline was able to produce eight distinct application clusters only three of them had a clear I/O profile (normal, intensive and other) while no application-specific bottlenecks were discovered.

III. Methodology

In this section, we describe the metrics used for evaluating the I/O footprint of each workflow. Furthermore, we provide readers with a detailed description of the hardware and datasets used for our experiments.

A. Metrics

In order to quantify the I/O footprint of each workflow and understand the effect of our design and runtime optimizations, we focus on the following metrics:

Throughput. We calculate throughput by dividing the total number of processed events by the total latency per process. Since *cctbx* is the only event-based workflow the throughput measurements were obtained only for the *cctbx* workflow.

Execution time. We measure per process execution time by calculating the average execution time among all processes participating in each workflow execution.

Total amount of I/O for read() and write() system calls. We measure the total amount of I/O by summing the amount of I/O for write() and read() system calls for each process. Furthermore, for the *SFX* workflow (**MR** and **SAD** applications), we also calculate the amount of IO spent by the master process.

We performed five runs per workflow and all measurements were obtained for both the initial and optimized workflow versions. For otaining the I/O measurements we utilized *strace* [13] Linux debugging userspace utility. For the *SFX* workflow, we instrumented the code by inserting *strace* calls both at the master process that is responsible for spawning all worker processes as well in the worker processes that conduct grid search and reconstruction. The I/O events recorded by strace were: read(), write(), fseek(), open(), close().

For calculating the per process execution time, we used dedicated time calls placed in the SLURM job submission script. For the *SFX* workflow we obtained timing measurements for both **MR** and **SAD** phasing applications by instrumenting the code of both the master and worker processes by inserting timers using Python's *time* module.

B. Data

For the *cctbx* workflow, we used a single artificial *xtc2* file containing only XFEL images that are successfully indexed. The size of the *xtc2* file is 631 GB and the size of the *smd* file (that acts as an index for the *xtc2* file) is 20MB. For the *SFX* workflow the following input files were used: one *.mtz* file (1.2MB) for storing reflections and one *.fasta* file (1MB) for the *SAD* phasing case while for the *MR* phasing case we utilized: one *.mtz* file (1.2 MB) and a *.fasta* file (507KB).

C. Hardware

We run our experiments on Cori, an XC40 supercomputer at NERSC on two distinct node groups: *KNL* and *Haswell*. Each *KNL* single socket node features an Intel Xeon Phi Processor 7250 (1.4GHz) with 68 cores and 4 hardware threads per core (272 threads total) accompanied by 96GB of DDR4 RAM at 2400MHz. A *Haswell* two socket node features an Intel Xeon Processor E5-2698 v3 (2.3GHz) with a total of 32 cores and 2 threads per core (64 threads in total), that are matched with 128 GB DDR4 RAM at 2133 MHz. We use *KNL* nodes for running the *cctbx* and *Reader* while we run the *SFX* workflow on *Haswell* due to its large per process memory requirements. In order to evaluate the effect of speeding up I/O reads and writes in the execution time and throughput of the *cctbx*, we conduct a set of experiments using NERSC's *Burst Buffer* [14]. Burst buffer is an intermediate storage layer between on-node memory and traditional HDD storage that provides slower than on-node memory reads and writes but faster than HDD-based storage. For our experiments we allocate burst buffer resources using SLURM workload manager. We configure the burst buffer reservation to operate on a stripped mode (in order for data to be scattered across multiple burst buffer nodes) while we opt for a non-permanent reservation (the burst buffer allocation is deleted after the workflow is completed). Our allocation size is 650GB per run.

IV. Experimental Results

This section outlines baseline profiling results for both workflows (*cctbx* and *sfx)* as well as a *Reader* application. We include results for **Reader** in order to obtain a deeper understanding of access patterns related to xtc2-specific input files. Based on the obtained results we identify a series of

bottlenecks which we later alleviate through workflow-specific runtime, implementation and system optimizations. We then quantify the effect of our optimizations through a detailed comparison with the baseline results. Finally, important observations and recommendations are presented.

A. Baseline Results

Reader We measure the execution time per process in the reader workflow in order to obtain a baseline performance for reading large xtc2-specific input files. Results are shown in Figure 3 As the reader workflow distributes events among multiple MPI processes read() calls occur both on *smd* (small-size index file) and *xtc2* (large file containing actual Dataframes) files that are located on *cscratch1* on Cori. Our results show that increasing the number of events per process does impact the average process execution time Similarly, increasing the number of events also increases the total I/O footprint of the workflow bringing it up to 4.5GB (attributed to *read*) calls for 1000 events.

Cctbx We present the baseline execution time and throughput when running *cctbx* on different number of nodes and processes on Table I (100 events per process) and Table II (500 events per process). For our baseline evaluation we use only one node to distribute events between processes (SMD node). As described in Section II-A2 *cctbx*'s I/O intensive stages (i.e. *Import*,*Indexing* and *Write Out*) process large amounts of input data and store results through multiple time consuming *read()* and *write()* system calls. Through our runtime analysis we discover that average process execution time increases due to varying indexing success rates among processes which in turn lead to longer waiting times.

TABLE I
BASELINE PROCESS EXECUTION TIME IN SECONDS AND THROUGHPUT FOR 100 EVENTS PER PROCESS FOR CCTBX.

number of processes	68	680	6800	10200
Execution Time	203.9	244.2	308.6	404.9
Throughput	0.49	0.4	0.32	0.24

TABLE II
BASELINE PROCESS EXECUTION TIME IN SECONDS AND THROUGHPUT FOR 500 EVENTS PER PROCESS FOR CCTBX.

number of processes	68	680	6800	10200
Execution Time	434.5	164.2	308.6	1508.2
Throughput	1.15	3	1.62	0.33

SFX We present the obtained I/O measurements at the master and worker process level granularity focusing on total time dedicated for *read()* and *write()* calls as well as the average per process execution time for the two types of protein phasing (*MR* and *SAD*).

Table III shows the amount of I/O for *read()* and *write()* system calls for the master process in the **SAD** phasing case for *Covid-19* protein samples. Our instrumentation mechanism has shown that the majority of the *read()*, *write()* calls are attributed to redundant operations performed by the master process such as creating multiple copies of the same set of files (one copy set per worker). The copied set includes not only per-job related input but also Python executables that are later launched by the worker processes. Hence, the amount of I/O by the master process follows a linear upward trend to the number of workers.

TABLE III
BASELINE AMOUNT OF I/O IN MB FOR MASTER PROCESS FOR READ() AND WRITE() SYSTEMS CALLS FOR **SAD** PHASING.

number of processes	100	300	500
Read()	134.3	379.4	624.5
Write()	120.9	362.9	604.9

TABLE IV
BASELINE AMOUNT OF I/O IN MB FOR MASTER PROCESS AND WORKER PROCESSES FOR READ() AND WRITE() SYSTEMS CALLS FOR **MR** PHASING.

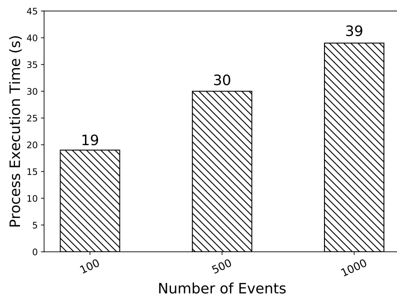
type of process	master	worker
Read()	0.9	0.45
Write()	1.15	0.03

I/O measurements for *read()* and *write()* system calls performed by each sub-process in the **MR** phasing case are shown in Table IV. Through our instrumentation mechanism we identify that the majority of *read()* and *write()* system calls for the master process is copying a set of input files similar to the **SAD** phasing trend. For worker processes the *read()* calls are attributed to accessing temporary files for input data. In constrast with **SAD** phasing were the number of worker processes varies, **MR** phasing has a fixed number of workers that is the determined by the grid size.

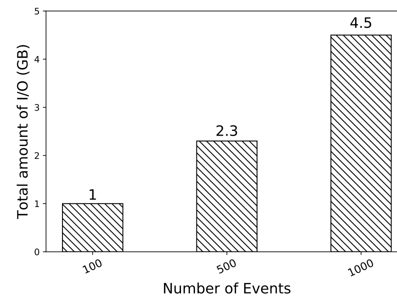
B. Optimizations

[*Utilizing Burst Buffer for storing input data Cctbx*]

After determining that the I/O intensive stages in each MPI process of the *cctbx* application refer to reading large amounts of input data and writing out results, we considered Burst Buffer, a faster storage solution, for temporarily storing input data. Figures 4a and 4b demonstrate the average execution time of each process while increasing the number of processes and nodes for two different event set configurations (100 and 500 events). Burst Buffer significantly reduces process execution time especially for large scale runs (150 nodes=10200 processes) up to 13% and 32% (for the 500 event per process scenario) respectively. In contrast with the overall observed trend, the 10 node (680 processes and 500 events per process) execution time using traditional HDD setup where data is located on *cscratch1* is significantly shorter than the one using Burst Buffer. We discovered that the difference in small node runs is attributed to the fact that approximately 1/6 of the images were not able to be indexed hence causing processes to exit earlier and overall shorter waiting times. Our results prove that utilizing intermediate storage solutions like burst buffer reduces the total time spent on I/O operations for read intensive applications.

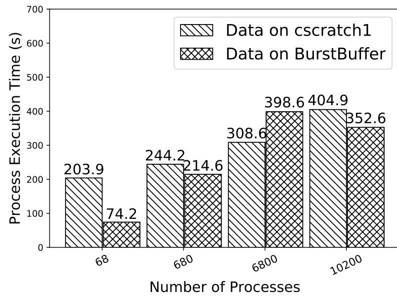


(a) Execution time for 100,500 and 1000 events per process. As the number of events increases process execution time follows the same trend.

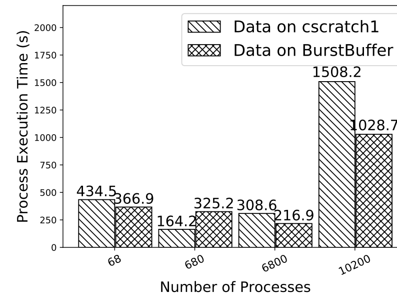


(b) Total amount of I/O for 100,500 and 1000 events per process. The total ammount of I/O increases due to more Read() calls

Fig. 3. **Reader**: Execution time and total amount of I/O increase with the number of events due to more Read() calls.

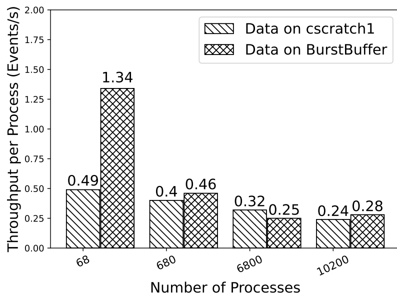


(a) Execution time for 100 events/process. Faster access to input data reduces execution time.

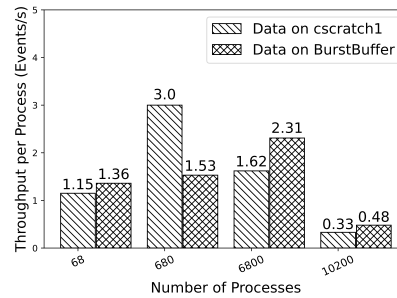


(b) Execution time for 500 events per process. Time is reduced up to 32%.

Fig. 4. **Cctbx**: Burst Buffer reduces execution time in runs that include large number of nodes.



(a) Throughput for 100 events per process. Highest increase is observed in single node (68 processes) runs.



(b) Throughput for 500 events per process. Highest increase is observed in single node (68 processes) runs

Fig. 5. **cctbx** Utilizing Burst Buffer increases throughput per process due to faster access to input data.

Figures 5a and 5b show the normalized throughput while increasing the number of worker processes and nodes for two different sets of events. The results show that staging data in Burst Buffer increases throughput in terms of events processed at an individual process level. However as the number of nodes and processes increase a downward trend is observed independently of the input data location.

[*Increasing the number of load balancing nodes Cctbx*]. By default, *cctbx* features one node for distributing events among MPI processes resulting to delays in the *Import* stage of each process. n order to reduce the time spent in the first of the three I/O intensive stages (*Import stage* in Figure 1) we opt for increasing the number of event distributing nodes. Figures 6a and 6b demonstrate the execution time and throughput for each process while increasing the number of processes and nodes for two different SMD configurations (1 node which is the default setting and 4 nodes). we opt for increasing the number of event distributing nodes. The results confirm that increasing the number of SMD nodes does have a positive effect in the process execution time especially for large node

runs (100 and 150 nodes). In these two cases the execution time is reduced by 27% and 36% respectively. However, in single node runs increasing the load balancing nodes has a negative effect since the number of processes left to process the event streams is lower. Throughput in terms of events processed per second exhibits the same trend. Summary. Overall we have observed that for I/O intensive workflows where large input data needs to be processed and distributed among different processes, utilizing Burst Buffer's faster intermediate storage and increasing the number of load balancing nodes has a positive effect in both throughput and execution time per process for larger scale runs.

[*Eliminating redundant operations for reducing total amount of I/O SFX.*] Our instrumentation mechanism has shown that the majority of the *read()*, *write()* calls are attributed to redundant operations performed by the master process such as creating multiple copies of the same set of files (one copy set per worker). The copied set includes not only per-job related input but also Python executables that are later launched by the worker processes. Since the copied files were used for read only purposes, we were able to eliminate all copy related system calls (e.g. open(), read(), write()) and reduce the associated I/O cost of the phasing step. Figures 7a and 7b show the amount of I/O for *read()* and *write()* system calls before and after our optimization for the master process in the **SAD** phasing case for *Covid-19* protein samples. Our results show that the I/O cost reduction depends on the number of worker processes which is related to the size of the to-be-reconstructed grid.

I/O measurements for *read()* and *write()* system calls performed by each sub-process in the **MR** phasing case are shown in Figure 8. Eliminating intermediate reading from a temporary file reduces the amount of MBs dedicated to *read()* calls by approximately 26% while the amount of *writes()* remains unchanged.

Figure 10 shows the effect of our optimization on the **time dedicated to I/O** by both master and worker processes. We measure time spent on I/O operations both in the master process before and after eliminating the redundant copy operations. In the optimized *MR* phasing case the time is reduced to under 20s for a full grid reconstruction (675 total sub-processes were spawned for a full *Covid-19* protein grid).

[*Increase parallelism SFX.*] Our runtime analysis of the *SFX* pipeline has shown that did not support parallel job execution. We increase workflow scalability by enabling parallel job execution. In order to ensure parallel job execution on multiple Cori nodes we redesigned and re-implemented the job submission pipeline enabling the master process to launch multiple asynchronous grid searches through autonomous worker instances (one group of workers per grid search). The master process keeps a record of all the worker's process id (pid) as well as the time that they were launched. The pids can be used for sanitizing purposes as well as provide intermediate visibility in cases where some worker instances hang during grid search. Furthermore, for providing visibility in the intermediate phases of the grid search non-blocking inter-process communication was enabled through shared pipes. The newly enabled communication allows the user to review worker error messages during execution instead of waiting for the workflow to complete and conduct a post-mortem analysis. Our design opts for collocating multiple worker processes per node in order to take advantage of available cores and memory resources. Finally, we were able to facilitate individual grid reconstruction for improving parallelism and ultimately reducing overall phasing execution time.

Figure 11 shows the effect of enabling parallel job execution on the average execution time per job. Each sub-process has specific memory requirements (10GB/process) which limits the number of processes that can simultaneously run at any time on a Haswell node to 12. Limiting the number of simultaneously running processes increases the average wait time per process significantly, however the overall **MR** phasing workflow execution time is reduced.

Overall, the obtained results demonstrate how our improvements enabled successful reconstruction of two *Covid-19* protein samples on Cori, while significantly reducing the overall I/O footprint of the that stage.

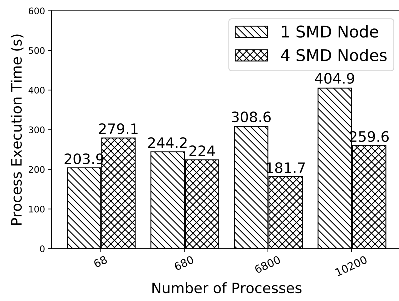
Summary. For *SFX*, by eliminating redundant copy operations and enhancing multi-level parallelism during grid reconstruction we managed to reduce I/O footprint by 26%.

V. Discussion

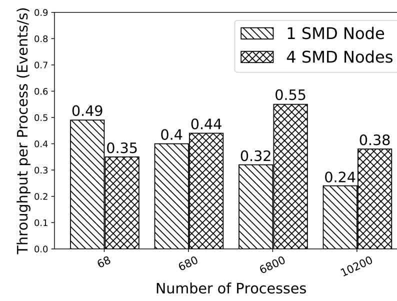
In this section, we provide some insights and recommendations for improving performance when running experimental data analysis workflows that require fast result turnaround on HPC environments. The recommendations stem from our experiences in detecting and alleviating performance bottlenecks for both *cctbx* and *SFX*.

Provide containerized environment for cross-facility execution. Container solutions like Shifter and Singularity [15] are becoming popular in HPC environments due to their ability to facilitate reproducibility and enable seamless execution across facilities regardless of the underlying system's features. Containers also provide a straightforward solution to setting complex environments with many interdependent software layers. Both *cctbx* and *SFX* come with a set of complex software dependencies and oftentimes, depending on the features and architecture of the underlying system, require considerable amounts of reconfiguration in order to be successfully executed. We have created container images for both workflows using NERSC's Shifter and resolved core dependency issues (such as *Psocake* installation). Users were able to deploy containers on multiple NERSC compute nodes or locally at LCLS. Furthermore, using containers actually reduced the load time especially for large python programs as shown in [16] resulting to performance enhancements for both *cctbx* and *SFX*. Our experience has shown that data analytics workflows with complex dependencies should opt for using containers in order to benefit from reduced loading times.

Reducing amount of I/O. For data analysis workflows with large input datasets eliminating redundant I/O operations by sharing files and balancing I/O operations between processes

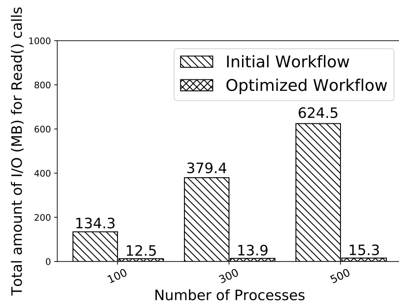


(a) Execution time for 100 events per process. Increasing the number of SMD nodes reduces execution time up to 27%.

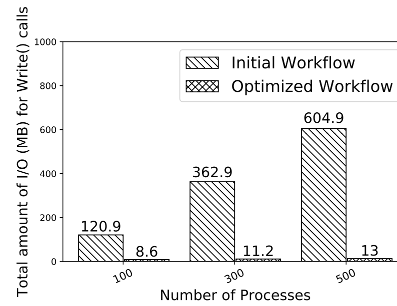


(b) Throughput for 100 events per process. Throughput increases for 100 and 150 node runs.

Fig. 6. **Cctbx**: Increasing the number of master nodes improves load balancing and reduces execution time for large scale runs.



(a) I/O for read calls in master process before and after eliminating redundant copy system calls while increasing the number of sub-processes for a grid reconstruction in the **SAD** phasing case



(b) I/O for write calls in master process before and after eliminating redundant copy system calls while increasing the number of sub-processes for a grid reconstruction in the **SAD** phasing case.

Fig. 7. **SFX SAD phasing**: Eliminating redundant copy operations reduces overall amount of I/O since fewer read() and write() calls are performed.

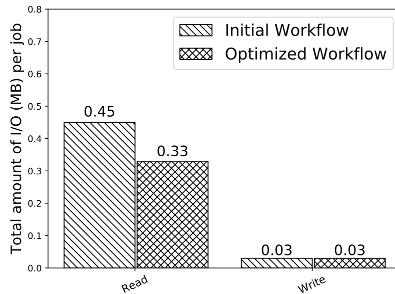


Fig. 8. **SFX MR phasing**: Amount of I/O in MB for Read() and Write() system calls per sub-process before and after eliminating redundant copy operations. Eliminating copy operations reduces I/O due to fewer read() calls.

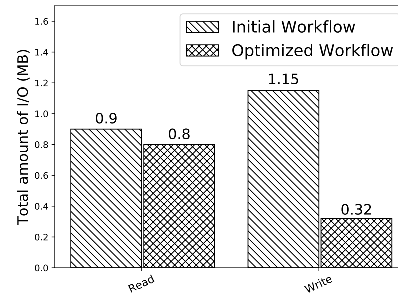


Fig. 9. **SFX MR phasing**: Amount of I/O in MB for read() and write() system calls in the master process before and after eliminating redundant copy operations for a full grid reconstruction. Eliminating copy operations reduces overall amount of I/O in master process due to fewer read() calls being performed.

can lead to significant reductions in overall amount of I/O as well as time spent in I/O operations. Through our work we were able to verify that eliminating redundand copy operatations for both cases (MR and SAD) of *SFX* workflow lead to a significant reduction in the MBs dedicated to read calls() as well as the overall time spent in I/O operations. Furthermore, Workflows with significant I/O would benefit from enhanced parallelism and even I/O distribution among different processes. Our experience with *cctbx* has proven that increasing load balancing can have a positive effect both on execution time and throughput.

Use of intermediate storage solutions in order to speed up input data reading. In the *cctbx* workflow we have observed that staging input data in BurstBuffer significantly reduces runtime in large node runs (up to 32% for 150 nodes). Opting for intermediate faster storage should be preferred

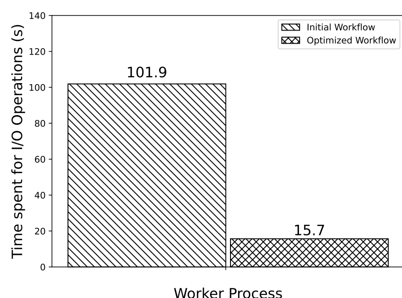


Fig. 10. **SFX MR phasing**: Time spent for I/O in master process before and after eliminating redundant copy operations and increasing parallelism. I/O time is reduced significantly.

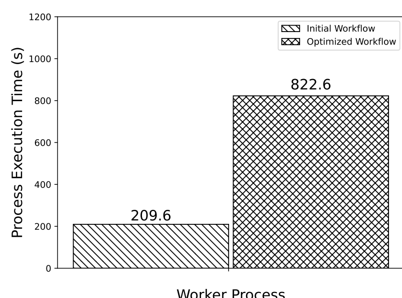


Fig. 11. **SFX MR phasing**: Per Job execution time before and after enhanced parallelism including waiting time. Average execution time increases because a number of worker processes need to wait due to node memory requirements limitations.

for scientific workflows with significantly large input datasets where each workflow process in a pool conducts an initial reading step before further processing. Employing a solution where input, intermediate and output data is automatically managed across different HPC storage layers based on a workflow-specific data plan would significantly shorten overall execution time [17]. Our results have proven that data location at different HPC storage layers significantly affects workflow performance. Integrating automatic data management with runtime optimizations and containerized workflow execution would significantly improve end-to-end workflow performance as well as overall user experience.

VI. Conclusion

Performance and scalability are critical factors for fast result turnaround for a new class of processing and analyses workflows that process vast amounts of data generated at experimental facilities. However, these applications are not designed or implemented originally for HPC systems and hence are unable to take full advantage of HPC resources in terms of compute and I/O. Furthermore, although traditional HPC applications have been extensively profiled, understanding of experimental workloads remains largely unexplored. In this paper, we conduct a thorough performance analysis for two experimental data workflows,*cctbx* and *SFX*, focusing on unveiling I/O, runtime and scalability bottlenecks. Based on our findings, we design and implement a number of runtime, I/O and parallelism optimizations for both workflows reducing their overall I/O footprint and execution time significantly. Our work addresses the issue of cross-facility execution by providing dedicated container instances in order to achieve seamless runtime. .

Acknowledgments

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors of this work. The authors would like to thank Debbie Bard, Johannes Blaschke, Aaron Brewster, Chun Hong Yoon, Jana Thayer, Wilko Kroeger and Frederic Poitevin for their help for accessing data, their insightful suggestions throughout this work and assistance in result interpretation.

References

[1] J. Thayer, "Building a data system for lcls-ii," 2018, last accessed September 2021. [Online]. Available: https://www.esrf.fr/files/live/sites/www/files/events/conferences/2018/IFDEPS/S8.02.Thayer.pdf

[2] V. Mariani, A. Morgan, C. H. Yoon, T. J. Lane, T. A. White, C. O'Grady, M. Kuhn, S. Aplin, J. Koglin, A. Barty, and H. N. Chapman, "*OnDA*: online data analysis and feedback for serial X-ray imaging," *Journal of Applied Crystallography*, vol. 49, no. 3, pp. 1073–1080, Jun 2016. [Online]. Available: https://doi.org/10.1107/S1600576716007469

[3] D. e. a. Damiani, "Linac coherent light source data analysis using psana," *Journal of Applied Crystallography*, vol. 49, no. 2, 3 2016. [Online]. Available: https://www.osti.gov/biblio/1256493

[4] H. Shin, S. Kim, and C. H. Yoon, "Data Analysis using Psocake at PAL-XFEL," *Journal of the Korean Physical Society*, vol. 73, no. 1, pp. 16–20, Jul. 2018. [Online]. Available: https://doi.org/10.3938/jkps.73.16

[5] P. D. e. a. Adams, "*PHENIX*: a comprehensive Python-based system for macromolecular structure solution," *Acta Crystallographica Section D*.

[6] M. D. e. a. Winn, "Overview of the *CCP*4 suite and current developments," *Acta Crystallographica Section D*.

[7] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–10, 2009.

[8] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A multiplatform study of i/o behavior on petascale supercomputers," ser. 2015 - 24th International Symposium on High-Performance Parallel and Distributed Computing, pp. 33–44.

[9] M. Jin, Y. Homma, A. Sim, W. Kroeger, and K. Wu, "Performance prediction for data transfers in lcls workflow," in *Proceedings of the ACM Workshop on Systems and Network Telemetry and Analytics*, ser. SNTA '19, 2019, p. 37–44.

[10] J. Bang, C. Kim, K. Wu, A. Sim, S. Byna, S. Kim, and H. Eom, "Hpc workload characterization using feature selection and clustering," in *Proceedings of the 3rd International Workshop on Systems and Network Telemetry and Analytics*, ser. SNTA '20, 2020, p. 33–40.

[11] C. S. Daley, Prabhat, S. Dosanjh, and N. J. Wright, "Performance analysis of emerging data analytics and hpc workloads," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage Data Intensive Scalable Computing Systems*, 2017, p. 43–48.

[12] E. Betke and J. Kunkel, "Footprinting parallel i/o – machine learning to classify application's i/o behavior," in *High Performance Computing*. Cham: Springer International Publishing, 2019, pp. 214–226.

[13] "Strace the linux syscall tracer," https://strace.io, accessed: 2021-04-14.

[14] " NERSC Burst Buffer," https://docs.nersc.gov/performance/io/bb/, accessed: 2021-04-14.

[15] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[16] A. Giannakou, J. P. Blaschke, D, D. Bard, and L. Ramakrishnan, "Experiences with cross-facility real-time light source data analysis workflows," *2021 Urgent HPC: HPC for Urgent Decision Making Workshop*, pp. 1–8, 2021.

[17] D. Ghoshal and L. Ramakrishnan, "Madats: Managing data on tiered storage for scientific workflows," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 41–52. [Online]. Available: https://doi.org/10.1145/3078597.3078611