

Big Data Staging with MPI-IO for Interactive X-ray Science

Justin M. Wozniak,^{*†} Hemant Sharma,[‡] Timothy G. Armstrong,[§] Michael Wilde,^{*†} Jonathan D. Almer,[‡] Ian Foster^{*†§}

^{*}Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

[†]Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA

[‡]X-ray Science Division, Argonne National Laboratory, Argonne, IL, USA

[§]Dept. of Computer Science, University of Chicago, Chicago, IL, USA

Abstract—New techniques in X-ray scattering science experiments produce large data sets that can require millions of high-performance processing hours per week of computation for analysis. In such applications, data is typically moved from X-ray detectors to a large parallel file system shared by all nodes of a petascale supercomputer and then is read repeatedly as different science application tasks proceed. However, this straightforward implementation causes significant contention in the file system. We propose an alternative approach in which data is instead staged into and cached in compute node memory for extended periods, during which time various processing tasks may efficiently access it. We describe here such a big data staging framework, based on MPI-IO and the Swift parallel scripting language. We discuss a range of large-scale data management issues involved in X-ray scattering science, and we measure the performance benefits of the new staging framework for high-energy diffraction microscopy, an important emerging application in data-intensive X-ray scattering. The use of our framework has been shown to accelerate scientific processing turnaround from three months to under 10 minutes, and our I/O technique reduces input overheads by a factor of 5 on 8K Blue Gene/Q nodes.

I. INTRODUCTION

Many branches of science face a big data challenge as experimental sensors and simulators produce ever larger data sets. X-ray scattering experiments at facilities such as the Advanced Photon Source (APS) at Argonne National Laboratory (ANL) are no exception. Improvements in detector technology can produce ~ 15 TB per week or more of raw image data; subsequent processing can more than double that quantity. These increases in data sizes outpace increases in computer performance, creating a crisis situation. These datasets will either be used to advance X-ray science in a transformative way, or be discarded.

Big Data tools must be applied to the scientific workflow to make these data sets manageable and useful. This includes all aspects of the data management cycle: ingest, metadata management, bulk data movement and storage, and accessibility for processing and analysis. Further, we desire solutions that can be run *interactively*: that is, while the scientist is operating the X-ray equipment, data processing operations proceed on available clusters and high-performance computing (HPC) resources, so that experiment time is used optimally. Thus, any errors in the experimental setup may be quickly detected and corrected, and interesting features may be

identified. Today, data analysis commonly is performed weeks or months after data is collected; we have demonstrated the ability to accelerate the scientific cycle to minutes.

The first stage of X-ray scattering analysis is essentially an image processing problem. X-ray detectors produce large quantities of images in standard formats. Scientific tools already exist for various types of analysis but are constantly under development; they are designed for execution on small-scale resources such as a laptop. The analysis tasks are generally independent. Thus, a *many-task* framework that can rapidly compose applications from these existing codes and run them at high concurrency levels could address many computing problems in X-ray image analysis. As shown in our results, this approach attains high performance without additional low-level coding (such as MPI messaging).

In this paper, we consider all aspects of the X-ray science data management cycle, but we emphasize a key problem: presenting large data sets for many-task processing on supercomputers such as the IBM Blue Gene/Q (BG/Q). These supercomputers can be used by a many-task framework such as Swift, but must provide high-efficiency access to data sets. As our performance results show, individual tasks cannot independently access the shared file system and achieve high performance.

Our approach is to stage input data to compute-node-local filesystems, such as the RAM disk on the BG/Q (or a solid-state disk on a hypothetical system). We load this data using a simple technique built around MPI-IO shared file operations for maximal efficiency. Once data staging is complete, scientific workflow tasks are distributed to processors where they can perform local data operations with high I/O rates. Our approach thus combines a *collective* phase for big I/O operations with a *loosely coupled* phase consisting of independent data analysis tasks. For interactive analysis, the staged data could be reused over several human-in-the-loop cycles (although we do not address that here).

Figure 1 is a simplified diagram of our framework. Scientific instruments, such as X-ray detectors, produce large quantities of data that are streamed to the parallel filesystem (e.g., GPFS) of the HPC machine. When an analysis run is triggered, this data is staged to RAM disks (e.g., `/tmp`) on each compute node (not each core). This operation uses available collec-

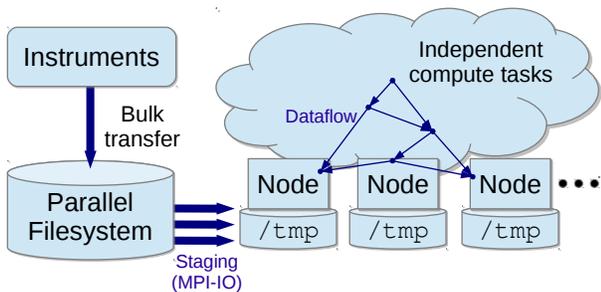


Figure 1: Overview of I/O staging; many-task workflow.

tive I/O libraries (e.g., MPI-IO `MPI_File_read_all()`). Then, the workflow proceeds, with independent tasks executing to process local data.

This approach offers many benefits. First, the underlying *scientific codes are not modified*; no new library used by the codes. They simply operate on data in the local directory instead of a shared filesystem directory. Second, we provide a high-level interface to the data staging operation—the user does not use MPI-IO directly. Third, the user workflow is a full-featured language, capable of expressing MapReduce and more general patterns, including loops, conditionals, and functions.

The remainder of this paper is organized as follows. In §II, we present an overview of a feature application in this paper: high-energy diffraction microscopy. In §III, for completeness, we present an overview of the Swift language, used to program the workflows described here. In §IV, we describe the new Swift I/O hook feature that we introduce for I/O enhancement. In §V, we describe the HEDM workflows in more detail. In §VI, we provide performance results, and in §VIII, we summarize our results.

II. SCIENTIFIC OVERVIEW: HIGH ENERGY DIFFRACTION MICROSCOPY

We review here the scientific application that motivates our work in I/O optimization for many-task computing. We note that the physical parameters of the scientific experiment have a direct impact on various computing requirements (data size, available concurrency, task granularity, etc.).

High-energy diffraction microscopy (HEDM) [7], [16], [17] is an important method for determining the grain structure of metals. It is performed at specialized light sources such as the APS. In the typical scientific workflow, the scientist applies for a week of beam time and spends that time gathering data. Over the next several months, the data is processed by using custom-built tools. In current practice, HPC is rarely if ever applied during the week of beam time.

Our work is intended to allow the use of HPC to analyze data quickly: as it is produced by the experiment detectors or immediately after. This integration of HPC into the scientific workflow has many potential benefits. As we show below, it can permit rapid detection of and recovery from errors. It may provide feedback to the scientist during the run, to improve the

quality of results and the utility of precious beam time. The overall result is to speed the process of scientific discovery, even if the HPC is applied after beam time.

We focus here on the application of HPC to an HEDM application, in an environment that encompasses an APS beam line, a small compute cluster, and the Argonne BG/Q HPC system. We use up to 64K cores of the BG/Q to provide near-real-time feedback to APS beam users.

HEDM is a diffraction-based imaging technique that can non-destructively determine the grain-level properties of polycrystalline materials. It yields unique in situ 3D information which has previously only been available through destructive, but more widely-available microscopy techniques, such as electron backscattering diffraction (EBSD). Hence, it is a valuable technique for analyzing grain defects in advanced alloy materials, such as those used in turbine blades for both energy (e.g., wind turbines) and engine applications (e.g., jet engine turbines). The technique allows a material sample, or even a manufactured part, to be studied in situ at an APS beamline across a range of applied thermal and mechanical loading conditions. A polycrystalline material sample (typically a metal alloy) is positioned within a high-energy ($E > 50$ keV) X-ray beam, producing forward-scattered X-rays that are collected by a range of detectors to yield 2D material information. A series of diffraction patterns is then collected as the sample rotated about a single axis perpendicular to the incident X-ray beam. The diffraction patterns are analyzed by software tools to reconstruct the 3D structure of the material, in order to determine the granular structure of material defects that can cause component failure in fabricated parts.

Depending on the detector type and its placement with respect to the sample, HEDM has two variants. In the *near-field* (NF) variant, a high-resolution detector (approx. $1.5 \mu\text{m}$ pixel size) is placed in close proximity to the sample (approx. 10 mm). In the *far-field* (FF) variant, a medium resolution detector (approx. $200 \mu\text{m}$ pixel size) is placed at a relatively larger distance from the sample (up to 1 m).

The near-field HEDM technique works as follows. A line-focused X-ray beam and a detector are used to collect diffraction data from a 2D cross-section (“layer”) of a rotating polycrystalline sample. 2D TIFF images, each 8 MB in size, are collected at each angle of rotation, typically 360 to 1,440 angles per layer, and for multiple detector distances. The computational analysis is split into two stages. In Stage 1, the diffraction images are binarized to detect pixels with diffraction signal. In Stage 2, a grid is simulated on the virtual 2D cross-section of the sample, and diffraction signals at each point on the grid are calculated and compared with the diffraction images. This computation is parallelized at the grid point level for a total of $\sim 10^5$ points per layer, enabling the concurrent use of tens of thousands of processor cores.

The NF-HEDM sample in Figure 2 shows the cross section of a roughly round gold wire. Each point in the hexagonal grid is displayed as a colored dot (the grid is a hexagonal prism in 3D). The four colors correspond to the four distinct grains identified in this sample, providing rich information

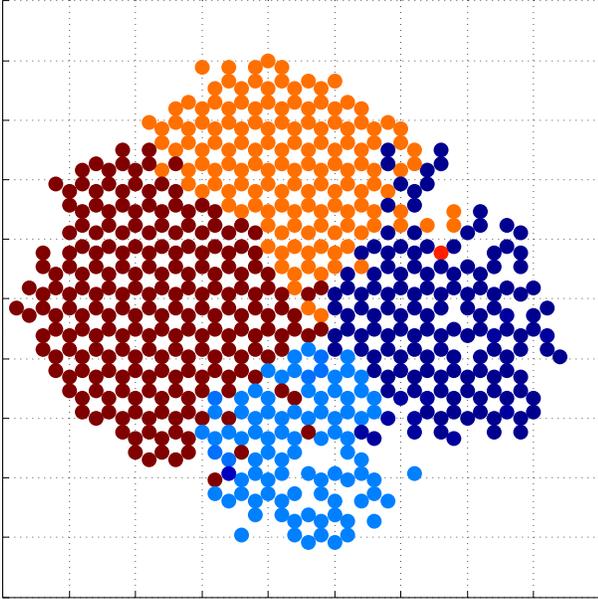


Figure 2: NF-HEDM grain identification for cross section of gold wire. Each grid square is $5\ \mu\text{m}$. Sample points of the same color have the same crystallographic orientation and are thus the same grain.

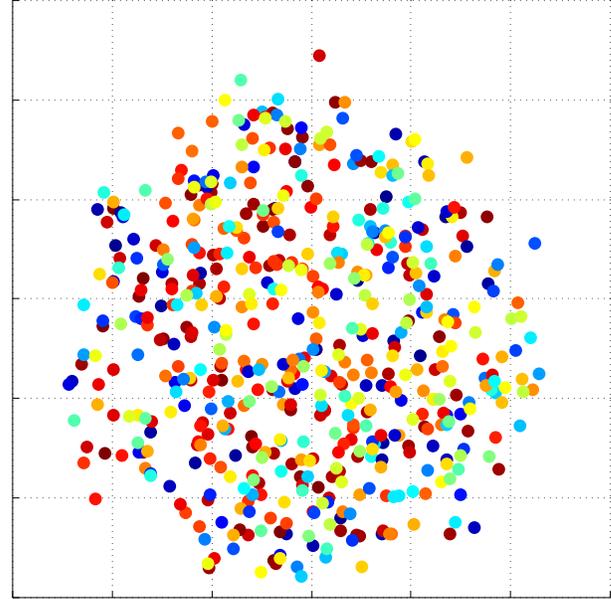


Figure 3: FF-HEDM grain identification for cross section of experimental material. Each grid square is $200\ \mu\text{m}$. In this technique, only grain centers are identified, and grain sizes vary widely, producing the apparently unstructured image shown.

about grain characteristics. The sample shown has 601 points corresponding to that many tasks; each task runs for about 10 minutes.

In far-field HEDM, a “box” beam with a rectangular cross-section illuminates a *volume* in the polycrystalline sample. The sample is rotated to capture multiple diffraction spots from each grain (contiguous region with the same crystal orientation) in the sample. A diffraction image, 8 MB in size, is recorded for 360 to 1,440 positions during the sample rotation. In the first step of analysis, the diffraction images are segmented, and properties of the diffraction spots are calculated. In the second step, the diffraction spots are assigned (called “indexing”) as belonging to grains, and properties of the grains are calculated. By imaging a volume instead of a cross section, FF-HEDM can image more material per unit time, but at a lower information level than NF-HEDM.

An FF-HEDM sample is shown in Figure 3. This shows the cross-section of a roughly round wire of an experimental material. In contrast to the NF-HEDM diagram, each dot here represents a grain center; only one point is shown per grain. This level of information is sufficient to measure material response to stress and deformation and provide feedback to material manufacturing techniques (e.g., annealing). This diagram has 572 grid points. The 4,109 tasks for this case are described further in §VI-D.

III. OVERVIEW OF THE SWIFT LANGUAGE

Swift [24] is a dataflow language for scientific computing, commonly used in cluster, cloud, and grid environments. Its

recent re-implementation, Swift/T [26], is designed for use on HPC systems. It operates by translating the Swift program into a format for execution on an MPI-based runtime [27], providing performance of up to 1.5 billion tasks/s on 512K cores of a Cray XE6 system. This performance is accomplished with a combination of compiler optimizations [2], an enhanced workflow enactment technology called Turbine [25], and a high-performance load balancer called ADLB [8].

Swift is an *implicitly parallel* language—all expressions may be evaluated concurrently, limited only by dataflow ordering and available processor. This allows for a natural parallel programming style, while still allowing for conventional constructs such as `for` loops and `if` conditionals, as well as other features that allow good use of practical machines [28]. Users may link to user code in compiled (C, C++) or scripting languages (e.g., Python, Julia) in *leaf functions*, external code that consumes and produces Swift data (numbers, strings, byte arrays, etc.). Data is moved through the Swift workflow over MPI but does not require the user to write MPI code.

As a language, Swift has many features that promote big data programming. First, Swift has a rich feature set for typical filesystem tasks: it supports simple calls to common shell programs, provides `glob` and other typical operations, and includes a *mapper* concept to *map* Swift variables to filesystem objects (or URLs). Second, Swift provides locality and work type features to *send work to data* and reduce data movement. Third, the dataflow programming model can express a wide range of data analytics algorithms elegantly and make them

```

1  main {
2    file d[];
3    int N = string2int(argv("N"));
4    // Map phase
5    foreach i in [0:N-1] {
6      file a = find_file(i);
7      d[i] = map_function(a);
8    }
9    // Reduce phase
10   file final <"final.data"> = merge(d, 0, tasks-1);
11 }
12
13 (file o) merge(file d[], int start, int stop) {
14   if (stop-start == 1) {
15     // Base case: merge pair
16     o = merge_pair(d[start], d[stop]);
17   } else {
18     // Merge pair of recursive calls
19     n = stop-start;
20     s = n % 2;
21     o = merge_pair(merge(d, start, start+s),
22                   merge(d, start+s+1, stop));
23   }
24 }

```

Figure 4: MapReduce-like application expressed in Swift.

extensible through the addition of Swift code.

For example, consider the popular MapReduce [5] framework. It would be interesting to extend this framework to support loops, conditionals, or subworkflows, but such extensions would require significant changes to the whole MapReduce implementation. A Swift implementation of MapReduce allows such changes to be made and tested piecemeal by adding lines to the script. We show in Figure 4 a simplified Swift implementation of MapReduce, with a single merge bucket. (A complete MapReduce implementation is the subject of work under preparation for submission elsewhere.) In this code, `find_file()`, `map_function()`, and `merge_pair()` are user-written leaf functions that consume and produce ordinary files. These functions can be implemented in any language and presented to Swift.

A dataflow diagram of the MapReduce implementation is shown in Figure 5. The application proceeds as follows. First, an integer N is obtained from the user command line (line 3). The *map phase* operates simply: for each value of i from 0 to $N-1$, a file a is obtained and processed with `map_function()` (lines 6–7). The result is stored in array d . The map functions operate concurrently and are automatically load balanced, limited only by available processors. The *reduce phase* (line 10) begins as soon as mergeable data is available. This is implemented here as a recursive (line 21) pairwise reduction on the contents of d . `merge_pair()` is called on consecutive entries in d (line 16), then recursively reduced pairwise up to the top call to Swift function `merge()`. The resulting file (variable `final`) is stored in physical file `final.data` (line 10).

Note that this dataflow expression of simplified MapReduce does not have a barrier between the map and reduce phases (see [23]).

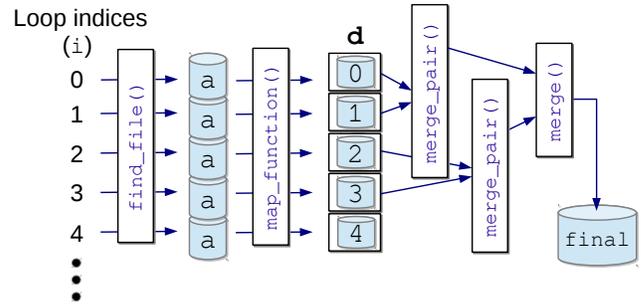


Figure 5: Dataflow diagram of MapReduce-like application.

IV. THE SWIFT I/O HOOK

As described in §III, Swift may be used to express complex data analytics workflows, make use of data locations, and operate with high performance. In typical HPC settings, however, data is stored in a shared parallel filesystem and is not resident on compute nodes (as in HDFS). Thus, data must be quickly *staged* to compute nodes for processing. Then, Swift execution can proceed, operating on node-local data as well as shared data (if desired).

This feature can be used to address filesystem congestion problems due to bandwidth *or* many-small-file congestion. For example, on an HPC system such as the BG/Q, plain executables are already broadcast efficiently to all compute nodes prior to execution by the standard BG/Q operating system. Swift scripts, however, may desire to make use of a large collection of small Python scripts or C++ shared libraries. The operating system cannot automatically load these on compute nodes efficiently. The Swift I/O hook can be used to pre-stage any such data: scripts, shared objects, configuration files, and so on, thus improving application start time and reducing impact on other system users. Bandwidth saving is a simpler benefit; we eliminate unnecessary duplicate bandwidth consumption. The Swift I/O hook is thus designed to reduce I/O traffic for both small and large file operations, including its operations used in its implementation.

The Swift I/O hook is implemented in the following way. First, a *leader communicator* is automatically constructed by the runtime. This MPI communicator consists of exactly one ADLB worker process per node. If a *leader hook* (e.g., the I/O hook) is provided in an environment variable, it is executed by each process in the the leader communicator. More complex scripts can be distributed by using the leader hook environment variable and then entered by the hook script.

We show in Figure 6 the code for an example Swift I/O hook. This fragment, which is evaluated by the interpreter in the Swift runtime, defines a list of broadcast definitions, each of which targets (broadcast to) a node-local directory location. The file list can contain glob patterns.

In order to avoid filesystem metadata contention, the file list is *also* broadcast by using `MPI_Bcast()`, with the result that only one process performs any globs. (A naive implementation would simply run the glob on each process to obtain the list

```

1 | broadcast to /tmp files {
2 |   ~/dataset-1/*.cfg
3 | }
4 |
5 | broadcast to /tmp/bulk files {
6 |   ~/dataset-1/bulk/file1.index
7 |   ~/dataset-1/bulk/file2.index
8 |   ~/dataset-1/bulk/*.bin
9 | }

```

Figure 6: Example Swift I/O hook specification.

of transfers to perform, congesting the shared filesystem.)

The hook is then used from the command line as follows:

```
1 | SWIFT_IO_HOOK=$(cat hook) swift-t options program ...
```

At execution time, the Swift/T implementation processes the I/O hook shortly after initializing its communicators. Swift/T performs the globs on one rank, broadcasts the list of files to transfer, then uses `MPI_File_read_all()` to make read-only replicas of each file on each node-local file system.

Future directions: The leader hook is a generic mechanism that may be generalized for more complex functionality in the near future. The leader communicator, a new feature, is derived from the Swift/T *hostmap* functionality, which maps host names (nodes) to MPI ranks, allowing for location-specific programming at the workflow level [28].

The leader hook is actually a Tcl fragment and thus in essence provides an extension language for Swift/T. In principle, the user can use this language to access the leader communicator and program arbitrary operations. Since this is an error-prone process, however, we provide the high-level wrapper syntax shown in Figure 6 (which is still Tcl code).

V. APPLICATION DESCRIPTION

Having discussed the scientific application of interest in §II and our software system in §III – §IV, we now describe the motivating scientific workflow and practical details that motivate our solution.

A. The scattering science workflow

Scattering science is a complex process that involves instruments, computers, and humans in the loop. The APS is a major scientific investment, hosting 5,000 users per year [1]. Users typically visit the laboratory for one week, during which they have access to beam resources 24 hours a day. Typically, data is collected on portable hard drives that are carried to the user’s home institution for processing and analysis. High-resolution detectors can produce 15 TB per week, but typical users fill less than one hard drive.

Our high-level goal is to improve this human-in-the-loop workflow by delivering enough computing power to the application so that analysis can be performed *during beam time*, that is, while the visitor is present at ANL. The HEDM application considered in this paper could use ~22 M CPU core hours per week, a quantity that can be obtained only on extreme-scale supercomputers. However, the process is additionally constrained by the size of data to be moved to the supercomputer and the amount of data to be transferred to

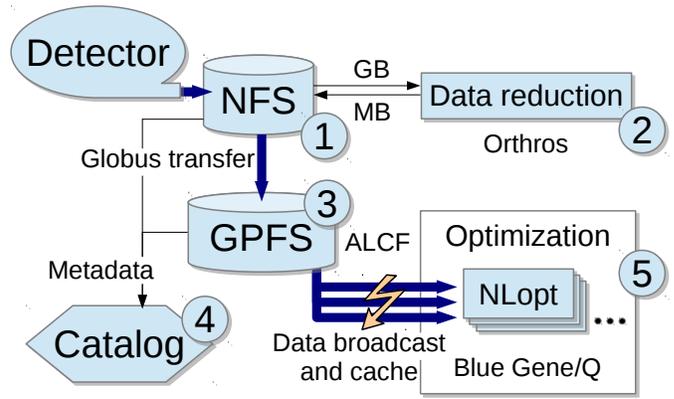


Figure 7: Cross-lab APS to ALCF workflow.

compute nodes—these are the Big Data problems. In the most recent run, considered here, 2 TB of data were collected in two days. Without our optimizations, roughly half of the time is spent in I/O and half in computing.

Additionally, the human-in-the-loop nature of the runs motivates I/O advancements. It is desirable for the investigator to be able to visualize detector data and detect anomalies or make decisions as soon as detector data is available. In order to keep up with the detector data generation rate, the entire workflow must complete in five minutes. Thus, I/O overheads cannot be amortized into long running jobs, but must operate at near-interactive rates. Our methods have been able to reduce run times for experimental datasets to below that rate, given enough compute cores. (Reservations on BG/Q systems have been allocated to users on the beamline, allowing interactive use without queues.)

B. Workflow practicalities

The raw TIFF files are reduced to binary files containing only information about the diffraction signal of the detector. Because of the sparse nature of the data, each 8 MB raw file can be reduced to an ~1 MB binary file. This reduction is performed by using 320 cores on Orthros, each using about 2 min of CPU core time. These binary files are fed into an orientation detection program to map the orientation of each point in a grid in the 2D cross-section of the sample. Two approaches are possible for this step. Using 320 cores on Orthros, at about 30 s for each grid point (a total of 100K grid points), we can obtain results in three hours. Using 10,000 cores on the Mira BG/Q or a similarly parallel resource, we can scale this workflow to obtain results in less than five minutes.

The ~10 MB output file contains information about the orientation of each point in a grid in the 2D cross-section of the sample. All analysis software has been developed in-house in Sector 1, implemented in C. Parallel processing of the analysis routines is implemented by using the Swift parallel scripting language.

Figure 7 shows the NF-HEDM workflow in detail. The detector produces data on an NFS installation at the APS (1)

```

1  main {
2    parameterFile = argv("p");
3    microstructureFile = argv("m");
4    start = toint(argv(1));
5    end = toint(argv(2));
6    foreach row in [start:end] {
7      FitOrientation(parameterFile, row,
8                    microstructureFile);
9    }
10 }

```

Figure 8: Swift/T fragment for NF-HEDM stage 2.

and large numbers of data reduction jobs are run on the local cluster, Orthros (2). The resulting data is moved via the Globus transfer service [6] to Argonne Leadership Computing Facility (ALCF) storage (3), and recorded in a metadata catalog (4) [9]. Finally, the HPC component runs (5), in which a large batch of hundreds of thousands of optimization operations are performed rapidly across tens of thousands of CPU cores of a BG/Q.

In this model, C analysis code developed for this work is linked to the NLOpt optimizer library and the GNU scientific library. These C code tasks are grouped into a large Swift script, which compiles into an MPI program for execution on a large HPC resource.

C. Code fragments

As a concrete example, we show in Figure 8 the Swift code for stage 2 of the NF-HEDM application. This program takes as arguments input parameter file and output microstructure file names, as well as a range of grid points to analyze, allowing for variable-sized runs. Then, each grid point (row) is analyzed in parallel with `FitOrientation()`, a C function that uses NLOpt to fit the grain. These tasks are eligible to run concurrently with automatic load balancing due to the Swift `foreach` loop. The C function requires all input data specified by the parameter file and distributed by the Swift I/O hook; the output is inserted into the microstructure file.

VI. PERFORMANCE

We next present performance data for the HEDM applications when running under our system. Cluster timings were obtained on *Orthros*, a 320-core x86 cluster at the APS. An *Orthros* node has 64 AMD cores running at 2.2 GHz. HPC results were obtained on the BG/Q systems at the ALCF. Each BG/Q node has 16 cores (64 threads) in the PowerPC A2 architecture running at 1.6 GHz. Runs at or below 8,192 cores were performed on the ALCF’s smaller BG/Q, *Cetus*; runs above that were performed on the larger *Mira* system. Both BG/Q systems run GPFS [14]; the installation supports a peak I/O performance of 240 GB/s [4].

A. NF-HEDM: Data reduction step (cluster)

The data reduction step involves carrying out a median calculation on each pixel of the detector using all images. Then, independently on each image, a median filter is applied followed by a Laplacian of Gaussian filter to determine the

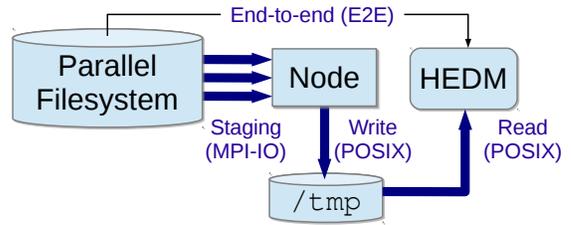


Figure 9: Definitions of data movement operations in the Swift I/O hook. HEDM is the unmodified scientific application.

edges of the diffraction spots; a connected components labeling step; and a flood fill operation to get information regarding all useful pixels in the image. When run on *Orthros* at our maximum allocation size of 320 cores, this stage required 106 s to process a total of 736 images from two detector distances.

B. NF-HEDM: Analysis step (HPC)

The analysis step highlights the key contribution of this paper: the Swift I/O hook. We profile the use of the I/O hook as well as the overall application.

1) *I/O profiling*: We first isolate the input operations in order to study their performance independently of the rest of the application. As shown in Figure 9, we distinguish three I/O steps: *Staging*, *Write*, and *Read*. The Swift I/O hook performs the *Staging* and *Write* steps, replicating data from the filesystem to node-local storage. (Note that on the BG/Q the `/tmp` RAM disk is actually an I/O node service.) The *Read* phase is performed by the application task itself, by simply reading from the appropriate directory specification (e.g., `/tmp`).

In our first experiment, we configured NF-HEDM to process a 577 MB data set from GPFS. Each node requires a full replica of the data set for use by tasks on that node. Figure 10 shows the performance measured for the Swift I/O hook as it reads data from GPFS using MPI-IO and writes the resulting data to node-local storage. At our highest reported node count, 8,192 nodes, the system delivers data at an aggregate rate of 134 GB/s.

The *Read* phase consistently takes 10.8 ± 0.1 s regardless of allocation size, for a per-process read bandwidth of 53.4 MB/s—comparable to a conventional node-local disk with ideal scalability.

To determine the aggregate end-to-end input bandwidth achieved by our approach, we add together the time taken for the joint *Staging/Write* and *Read* phases. The upper line in Figure 11 shows the result, which reaches 101 GB/s on 8,192 nodes. In contrast, the original I/O approach, in which each task reads input data independently from GPFS, without the use of collectives achieves only 21 GB/s on 8,192 nodes.

We modified NF-HEDM to cache all inputs in application memory (for each variable, tasks first check to see if it has already been read, if not, they perform read operations to instantiate it). Since Swift/T reuses the same processes for

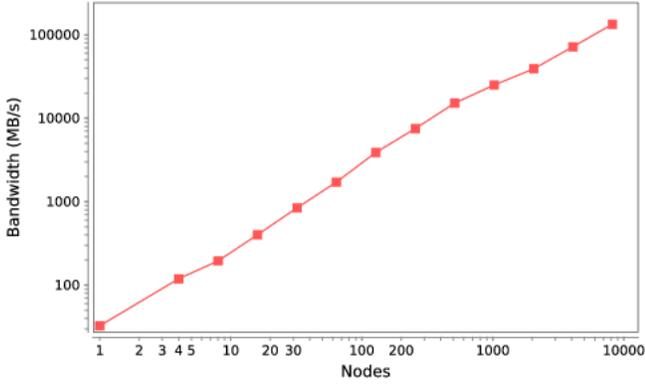


Figure 10: Staging+Write performance for NF-HEDM data set.

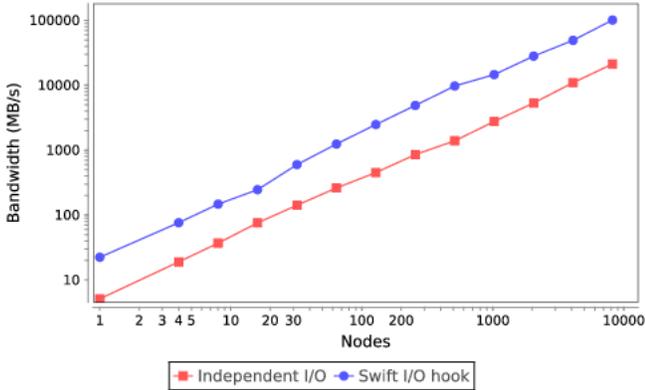


Figure 11: End-to-end performance for NF-HEDM data set.

subsequent tasks, HEDM tasks after the first do not need to perform *Read* operations at all. This approach reduces input time to effectively zero for subsequent tasks.

In terms of wall time, the Swift I/O hook reduces the input time from 210 s to 46.75 s, improvement by a factor of 4.7, making data available to 8,192 nodes (524,288 hardware threads).

C. FF-HEDM: Stage 1 (cluster)

In FF-HEDM stage 1, each process loads a diffraction image (8 MB) and characterizes all peaks in the image. The output is saved as a text file (~ 50 KB). We performed runs on 720 images, with each image being processed in parallel. Depending on the number of diffraction spots in each image, the processing time per image can vary from 5 s to 160 s. Figure 12 shows the scaling results for the 720 jobs run on Orthros.

D. FF-HEDM: Stage 2 (cluster)

The number of tasks in this case is data-dependent, varying with the number of grains within the sample volume illuminated by the diffraction beam. For our experiments, we work with a sample that comprises 4,109 grains and thus tasks, with

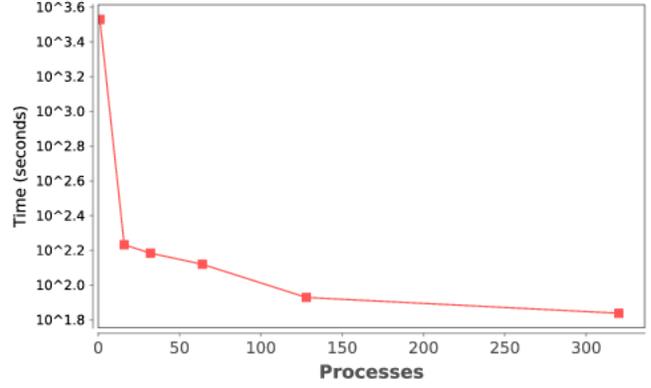


Figure 12: Makespan scaling result for FF-HEDM stage 1.

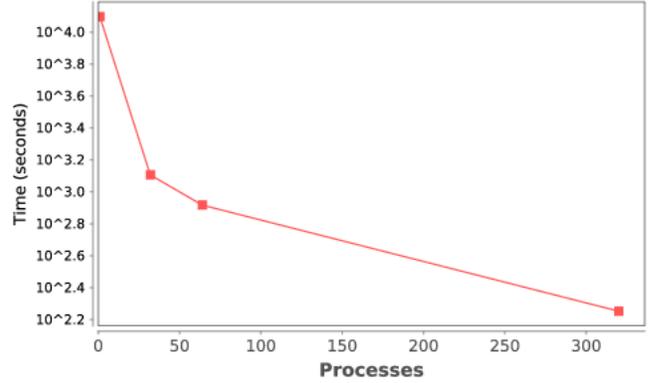


Figure 13: Makespan scaling result for FF-HEDM stage 2.

the run-time per task varying between 5 and 25 s, depending on the optimization landscape. Figure 13 shows the scaling results for the 4,109 jobs run on Orthros.

VII. RELATED WORK

We aim in this work to make Big Data analysis feasible on HPC systems such as the BG/Q. Our approach is to rapidly fill node-local storage with data for in-place processing, and then launch analytics tasks using a flexible, general-purpose dataflow programming model. While this approach is unique, much related work has been done.

The term scratch has multiple meanings but can be applied to the use of a specialized, high-performance filesystem that trades reliability for performance. However, while scratch space may be intended to be used as cache, it is not typically used as such for a variety of reasons, leaving users with just another filesystem. The Scratch as a Cache system [11] acts on user-provided hints in the job submission script, loading inputs into the fast filesystem automatically, improving on workflow-oblivious least-recently used (LRU) and other common techniques. The interface to Scratch as a Cache is similar to ours in that files are specified by the user and moved to a high-performance store, improving on automated techniques. However, we focus on the use of node-local storage.

Workflow-aware storage/scheduling systems, including the Batch-Aware Distributed Filesystem (BAD-FS) [3] and the Workflow-Aware File System (WASS) [20]), use workflow information to make caching decisions, including an ad hoc broadcast mechanism in WASS. BAD-FS does not use collective I/O operations. WASS performance peaked at eight replicas (our results show performance gains up to and beyond 8K replicas). BAD-FS and Freeloader [22] use *cooperative caches* that focus on write-once, read-many workloads (like ours) but access *nearby* cached copies, instead of using collectives. Other uses of node-local storage in HPC typically focus on *write* caching and aggregation, not reads, as in our work.

Data Diffusion [13] uses a distributed cache with a distributed index to support many-task, data-intensive Falkon [12] workloads for up to 128 processors; FusionFS employs a similar architecture reporting larger results up to 1,024 nodes. The Chirp [19] and AMFS [29] systems feature broadcast operations in the cache storage system but use ad hoc data distribution libraries and do not report performance results at the scale of interest here.

Extract-Transform-Load (ETL) is a collection of methodologies and technologies used in data warehouses to move data from sources to processors. It is responsible for “(i) the extraction of the appropriate data from the sources, (ii) their transportation to a special-purpose area of the data warehouse where they will be processed,” and other operations [21]. ETL is conventionally connected strongly to database applications and does not emphasize collective operations or concurrency. MapReduce has been considered as an ETL system [18] but not with a highly concurrent preliminary staging operation. MapReduce typically operates on in situ (i.e., compute node resident) data, as does Dremel [10].

MRAP [15] explored the idea of copying and converting data from a parallel filesystem and scientific data format to a Hadoop system in a MapReduce-friendly format. However, it does not consider the use of collective I/O and focused on formatting issues.

VIII. SUMMARY

X-ray scattering is a broad and complex application area. We have focused here on a key part of the X-ray scattering analysis problem—moving data to processors for computational analysis. We described how the Swift language can be used to express data analysis tasks in an elegant way, supporting common data analytics patterns such as MapReduce as well as the scientific pattern for HEDM. We reviewed the scientific and practical details of X-ray science and the HEDM application, and we provided a detailed presentation of our key contribution: the use of MPI-IO for data staging for Big Data analytics. We described the implementation of this technique and presented performance results. We showed that our HPC-oriented approach can be used to complete the whole analysis phase of the HEDM workflow extremely quickly (~5 minutes), which is suitable for interactive scientific use.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. DOE Office of Science under contract DE-AC02-06CH11357 and by NSF award ACI 1148443. Computing resources were provided in part by NIH through the Computation Institute and the Biological Sciences Division of the University of Chicago and Argonne National Laboratory, under grant S10 RR029030-01, and by NSF award ACI 1238993. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

REFERENCES

- [1] APS web site. <http://www.aps.anl.gov/About/Welcome>.
- [2] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Proc. SC*, 2014.
- [3] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *Proc. USENIX Symposium on Networked Systems Design and Implementation*, 2004.
- [4] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. Harms. Scalable parallel I/O on a Blue Gene/Q supercomputer using compression, topology-aware data aggregation, and subfiling. In *Proc. Euromicro PDP*, 2014.
- [5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Conference on Operating System Design and Implementation*, 2004.
- [6] I. Foster. Globus Online: Accelerating and democratizing science through cloud-based services. *IEEE Internet Computing*, (May/June):70–73, 2011.
- [7] U. Lienert, S. Li, C. Hefferan, J. Lind, R. Suter, J. Bernier, N. Barton, M. Brandes, M. Mills, B. Jakobsen, and W. Pantleon. High-energy diffraction microscopy at the Advanced Photon Source. *JOM*, 63(7):70–77, 2011.
- [8] E. L. Lusk, S. C. Pieper, and R. M. Butler. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review*, 17, 2010.
- [9] T. Malik, K. Chard, and I. Foster. Benchmarking cloud-based tagging services. In *Proc. CloudDB*, 2014.
- [10] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB*, 3(1-2):330–339, 2010.
- [11] H. M. Monti, A. R. Butt, and S. S. Vazhkudai. /Scratch as a cache: Rethinking HPC center scratch storage. In *International Conference on Supercomputing*, pages 350–359, 2009.
- [12] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Toward loosely coupled programming on petascale systems. In *Proc. SC*, 2008.
- [13] I. Raicu, Y. Zhao, I. T. Foster, and A. Szalay. Accelerating large-scale data exploration through data diffusion. In *Proc. Workshop on Data-aware Distributed Computing at HPDC*, 2008.
- [14] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. FAST*, 2002.
- [15] S. Sehrish, G. Mackey, J. Wang, and J. Bent. MRAP: A novel MapReduce-based framework to support HPC analytics applications with access patterns. In *Proc. HPDC*, 2010.
- [16] H. Sharma, R. M. Huizenga, and S. E. Offerman. A fast methodology to determine the characteristics of thousands of grains using three-dimensional X-ray diffraction, I: Overlapping diffraction peaks and parameters of the experimental setup. *Journal of Applied Crystallography*, 45:693–704, 2012.
- [17] H. Sharma, R. M. Huizenga, and S. E. Offerman. A fast methodology to determine the characteristics of thousands of grains using three-dimensional X-ray diffraction, II: Volume, centre-of-mass position, crystallographic orientation and strain state of grains. *Journal of Applied Crystallography*, 45:705–718, 2012.

- [18] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: Friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.
- [19] D. Thain, C. Moretti, and J. Hemmes. Chirp: A practical global filesystem for cluster and grid computing. *Journal of Grid Computing*, 7(1):51–72, 2009.
- [20] E. Vairavanathan, S. Al-Kiswany, L. Costa, Z. Zhang, D. Katz, M. Wilde, and M. Ripeanu. A workflow-aware storage system: An opportunity study. In *Proc. CCGrid*, 2012.
- [21] P. Vassiliadis. A survey of Extract–Transform–Load technology. *International Journal of Data Warehousing and Mining*, 5(3):1–27, 2009.
- [22] S. S. Vazhkudai, X. Ma, V. W. Freeh, J. W. Strickland, N. Tammineedi, T. Simon, and S. L. Scott. Constructing collaborative desktop storage caches for large scientific datasets. *ACM Transactions on Storage*, 2(3):221–254, 2006.
- [23] A. Verma, B. Cho, N. Zea, I. Gupta, and R. H. Campbell. Breaking the MapReduce stage barrier. *Cluster Computing*, 16(1):191–206, 2013.
- [24] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Par. Comp.*, 37:633–652, 2011.
- [25] J. M. Wozniak, T. G. Armstrong, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster. Turbine: A distributed memory data flow engine for many-task applications. In *Proc. Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET) at SIGMOD*, 2012.
- [26] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Large-scale application composition via distributed-memory data flow processing. In *Proc. CCGrid*, May 2013.
- [27] J. M. Wozniak, T. Peterka, T. G. Armstrong, J. Dinan, E. L. Lusk, M. Wilde, and I. T. Foster. Dataflow coordination of data-parallel tasks via MPI 3.0. In *Proc. EuroMPI*, 2013.
- [28] J. M. Wozniak, M. Wilde, and I. T. Foster. Language features for scalable distributed-memory dataflow computing. In *Proc. Data-Flow Execution Models for Extreme-Scale Computing at PACT*, 2014.
- [29] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. Foster. Design and analysis of data management in scalable parallel scripting. In *Proc. SC*, 2012.

(The following paragraph will be removed from the final version.)

This manuscript was created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.