Characterization and Identification of HPC Applications at Leadership Computing Facility

Zhengchun Liu
Argonne National Laboratory
Lemont, IL, USA
zhengchun.liu@anl.gov

Ryan Lewis
Northern Illinois University
DeKalb, IL, USA
rlewis5@niu.edu

Rajkumar Kettimuthu
Argonne National Laboratory
Lemont, IL, USA
kettimut@anl.gov

Kevin Harms
Argonne National Laboratory
Lemont, IL, USA
harms@alcf.anl.gov

Philip Carns
Argonne National Laboratory
Lemont, IL, USA
carns@mcs.anl.gov

Nageswara Rao
Oak Ridge National Laboratory
Oak Ridge, TN, USA
raons@ornl.gov

Ian Foster
Argonne National Laboratory
Lemont, IL, USA
foster@anl.gov

Michael E. Papka
Argonne National Laboratory and
Northern Illinois University
Lemont, IL, USA
papka@anl.gov

ABSTRACT

High Performance Computing (HPC) is an important method for scientific discovery via large–scale simulation, data analysis, or artificial intelligence. Leadership-class supercomputers are expensive, but essential to run large HPC applications. The Petascale era of supercomputers began in 2008, with the first machines achieving performance in excess of one petaflops, and with the advent of new supercomputers in 2021 (e.g., Aurora, Frontier), the Exascale era will soon begin. However, the high theoretical computing capability (i.e., peak FLOPS) of a machine is not the only meaningful target when designing a supercomputer, as the resources demand of applications varies. A deep understanding of the characterization of applications that run on a leadership supercomputer is one of the most important ways for planning its design, development and operation.

In order to improve our understanding of HPC applications, user demands and resource usage characteristics, we perform correlative analysis of various logs for different subsystems of a leadership supercomputer. This analysis reveals surprising, sometimes counter-intuitive patterns, which, in some cases, conflicts with existing assumptions, and have important implications for future system designs as well as supercomputer operations. For example, our analysis shows that while the applications spend significant time on MPI, most applications spend very little time on file I/O. Combined analysis of hardware event logs and task failure logs show that the probability of a hardware FATAL event causing task failure is low. Combined analysis of control system logs and file I/O logs reveals that pure POSIX I/O is used more widely than higher level parallel I/O.

Based on holistic insights of the application gained through combined and co-analysis of multiple logs from different perspectives and general intuition, we engineer features to "fingerprint" HPC applications. We use t-SNE (a machine learning technique for dimensionality reduction) to validate the explainability of our features and finally train machine learning models to identify HPC applications or group those with similar characteristic. To the best of our knowledge, this is the first work that combines logs on file I/O, computing, and inter-node communication for insightful analysis of HPC applications in production.

CCS CONCEPTS

• **Computing methodologies** → *Parallel computing methodologies*; • **General and reference** → **Performance**; • **Software and its engineering** → **Software creation and management**.

KEYWORDS

High Performance Computing; Logs data mining; Application Identification; Characterization

ACM Reference Format:
Zhengchun Liu, Ryan Lewis, Rajkumar Kettimuthu, Kevin Harms, Philip Carns, Nageswara Rao, Ian Foster, and Michael E. Papka. 2020. Characterization and Identification of HPC Applications at Leadership Computing Facility. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3392717.3392774

1 INTRODUCTION

Advanced computing capabilities are used to tackle a rapidly growing range of challenging science and engineering problems, many of which are compute-, communications-, and data-intensive as well [41]. A similar term, *leadership computing*, is widely used in the United States that refers to the advanced technical capabilities, including supercomputers, software and expert staff, that support a wide range of science and engineering research. It is a large enough scale and cost that is typically shared among multiple researchers, institutions, and applications. Thus, teams of technicians, performance engineers, domain scientists, and computational scientists are needed to prepare these behemoths to effectively and efficiently execute the massive computations and drive high-end modeling and simulation science [17]. A large fraction of leadership-class investments have been driven by the mission-critical requirements of the U.S. Department of Energy (DOE) and the U.S. Department of Defense (DOD) [41]. The pressure on Leadership Computing Facilities (LCFs) to align leadership systems with the needs and goals of breakthrough science projects means that priority is given to jobs that require a large fraction of the entire machine, need to run for long periods of time, or cannot be accomplished without LCF resources [16]. As listed in the TOP 500 project [46], similar systems are available in Europe, China, and Japan, although not referred to as leadership computers.

To address the dearth of the characterization of applications and workloads on leadership computers and similar systems, researchers have resorted to the analysis of logs that have been collected intentionally either for application characterization purposes (e.g., by tools like Lustre Monitoring Tool (LMT) [42] and Darshan [33]) or for operational purposes (e.g., file system metadata snapshots [26]). That is to say, the characterization does not rely on explicit benchmarks, but instead on logs of debugging, operation, or tracing. This has led to a prolific line of publications that shed light on supercomputer scheduling [1, 20, 44], parallel file systems [26, 33, 42], user behavior [2, 43], networking [31] and data movement [28, 32].

Here we continue this line of research by combining, correlating, and analyzing multi-dimensional logs of supercomputing resources operated by the Argonne Leadership Computing Facility (ALCF), and using data mining and machine learning techniques in order to provide systematic insights into the applications and systems without adding any extra instrumenting burden to the precious supercomputer. This study distills insights by observing trends and characteristics using statistical methods. In particular, we use cumulative density function (CDF), probability density function (PDF) and complementary cumulative distribution function(CCDF). CDF, at x, gives the probability that the random variable X takes a value less than or equal to x; PDF specifies the probability that the random variable X takes on a specific value x; and CCDF is used to find the probability of a variable taking a value greater than x. The resulting insights may help the following entities, groups, and efforts:

(1) Resource providers, e.g., LCFs, to optimize their resources and operations;
(2) Researchers and tool developers to build new (or optimize existing) HPC frameworks and runtime systems;
(3) End users to optimize their application and resource requests to get better performance;
(4) Funding agencies to plan investments for upgrading existing systems and building new supercomputers;
(5) To serve as a crucial first step towards hardware-software co-design for the next generation machine.

The rest of this paper is organized as follows. We introduce the Mira supercomputer studied in this work, and its storage system, in §2. Next, we report job level analysis in §3; file I/O related observations in §4; computing demand and memory behavior of jobs in §5; Finally, we explore the use of a model-based classifier to group applications with similar behavior in terms of hardware resource demand and usage in §6. We review related work in §7, and summarize our conclusions and discuss future work in §8.

2 SYSTEM ARCHITECTURE AND LOGS

We briefly introduce the Mira supercomputer and the datasets used in this study, define some terms, and describe our analysis platform.

2.1 The Mira supercomputer

Mira is a 10 petaFLOPS (10^{16} floating point operations per second: FLOPS) IBM Blue Gene/Q system at ALCF. It consists of 49 152 computing nodes in 48 racks totaling 786 432 processors interconnected using a proprietary 5D torus network, and 768 terabytes of total memory. Mira debuted in June 2012 and was retired at the end of 2019.

Mira is equipped with two General Parallel File Systems (GPFS) with 20 PB and 7 PB capacity, and a block size of 8 MB, capable of 240 GB/s and 90 GB/s peak throughput respectively. The home folder was mounted to a purpose-built 1 PB GPFS with a much smaller block size (256 KiB). For data backups, ALCF has three 10,000-slot libraries using Linear-Tape Open (LTO) 6 tape technology. The LTO tape drives have built-in hardware compression for an effective capacity of 36-60 PB.

2.2 Datasets

We combined and co-analyzed five datasets in this study. These datasets are primarily collected for use by the facility operations team and/or to assist users in troubleshooting, debugging, bottleneck detection, and optimization purposes.

Cobalt logs. Cobalt, denotes Component-Based Lightweight Toolkit, is the queuing system used at ALCF. Users submit their requests for resource and time allocations to Cobalt and Cobalt treats each submission as a **job**. For each job, Cobalt records a great deal of metadata information including: a unique job ID; timestamp of the submission, execution and termination; number of compute nodes requested and actually used; and the location of the allocated nodes, among other data. This dataset contains records for all 300 023 jobs run on Mira from 2015 to 2019.

Control system logs. A single ALCF resource request (i.e., a single job) may run multiple applications, either in sequence or in parallel; each such application execution is called a **task**. Tasks are managed by the control system, which assigns each task a unique control system ID, and records task-level information such as executable ID, start time, and end time. This dataset contains records for all 2 592 361 tasks executed on Mira from 2015 to 2019.

Darshan [5] is a lightweight I/O instrumentation library that can be used to investigate the I/O behavior of production applications. It records statistics such as the number of files opened, time spent on performing read, write, and metadata operations separately, and the amount of data accessed by an application. Darshan is enabled by default at ALCF but can be disabled by users. This dataset contains records for 385 019 tasks executed on Mira from 2016 to 2019: that is, for about 23% of all tasks executed on Mira during that period.

RAS Event [25]. The IBM BG/Q's reliability, availability, and serviceability (RAS) infrastructure enables the reporting of hardware and software events. RAS messages contain basic information related to the event, such as message ID, severity, location, and text. The severity may be FATAL, designating a severe error event that presumably leads the application to fail or abort; WARN, designating potentially harmful situations, such as exceeding a soft error threshold or failure of a redundant component; and INFO, for informational messages. This dataset contains 20M `RAS-INFO` messages, 50M `RAS-WARN` messages, and 774K `RAS-FATAL` messages collected on Mira from 2015 to 2019.

Autoperf [7] is a library for automatically collecting hardware performance counter and MPI information. It is enabled by default on Mira, but can be disabled by the user. Autoperf transparently collects performance data from running jobs and saves it into files upon completion of a job. Specifically, for each MPI process, it records the number of calls to each MPI routine, time spent in each routine, and total bytes communicated by each routine. In order to minimize overhead and save storage space, for each counter across all MPI processes of a given execution, Autoperf records only the maximum, minimum, and average values as well as the value from MPI rank 0. Autoperf records are missing for applications that do not use MPI or that failed to call `MPI_Finalize`, and for executables that were built with other conflicting profiling libraries (e.g., BGPM, TAU, HPCTW) or linked with a compiler other than IBM XL or GCC. This dataset contains records for 377 968 tasks run on Mira from 2016 to 2019, i.e., about 22% of all tasks run on Mira during that period.

All datasets have been parsed and saved as data frames in CSV files. The CSV files, with username and project name anonymized, are publicly available at https://reports.alcf.anl.gov/data. We note that none of these datasets were purposely collected for this study. The logs analysis workflow code and algorithm implementations are available at https://github.com/ramsesproject/alcf-logs.

2.3 Term definitions

For ease of reference, we define the following terms that are used throughout this paper.

- **Application**: A program (i.e., executable) that runs at the leadership computing facility.
- **Job**: The computation performed on the computing resources allocated in response to a user request to the HPC system scheduler. A job may comprise one or more tasks, depending on how the resources allocated to the job are used.
- **Task**: An application execution within a job: on Mira, via a `mpirun`, `aprun`, or `jobrun` command. A task will often comprise more than one process.
- **Process**: An MPI process/rank.
- **Communication**: Inter-process communication via MPI, whether inter-node or intra-node.

2.4 Analysis Platform

We first parsed all raw data on the Cooley cluster at ALCF, as it had direct access to log files on Mira's GPFS, and saved useful counters to CSV files. We then performed our analysis on a 12-node cluster with 128 GB of RAM per node, using the Python Pandas library [39] to load and manage the parsed logs and the Apache Spark [50] cluster-computing framework to efficiently process the large (85 million records) collection of log files.

3 CHARACTERIZING JOBS RUN ON MIRA

Each job run on Mira comprises one or more tasks that execute in parallel or sequentially. Different tasks of the same job may belong to the same or different application/executable. The same application may be run many times with different resource settings (e.g., degree of parallelism) and arguments (e.g., simulation scale and input data). Most of the application characterization-based analyses presented in this paper are task based, because one task is a particular execution/run (statistically, one sample) of an application.

3.1 Task run time breakdown

Here, we partition an application's runtime into three major components, *file I/O*, *MPI communication* (including idle time due to load imbalances), and *computing* on CPU, and measure what portion of a task's total execution time is spent on each of these three major components. The results of this analysis can guide future design or upgrade decisions. We used three sets of logs for this purpose: (1) for each given task, Darshan [45] logs capture file I/O behavior, including the time taken and size for each file I/O operation (i.e., open, read, write, seek, close, etc.). (2) Autoperf [7] logs record some hardware performance counters, as well as the time spent on each MPI routine for each task. (3) The Cobalt scheduler logs have job properties such as wall-time, run-time, job size in nodes, and number of tasks per job.

MPI processes that belong to the same task may perform inter process communication (IPC, both inter-node and intra-node) at the same time, and their file operations will likely overlap as well. Unfortunately, the logs collected do not have enough information to figure out the wall-time spent on file I/O, communication, and computation. Thus, for each file read/write by a given task, we use the time spent on file operations and the number of cores used (recorded in Darshan) to determine the "file I/O core seconds" (time in seconds × number of cores used). We then add up the file I/O core seconds of all files that belong to the given task to determine the "cumulative file I/O core seconds" for the task: T^{cul}_{fio}. Likewise, we determine the "cumulative IPC core seconds" for the task: T^{cul}_{comm}, and the "cumulative overall core seconds" (by adding up the "run time × number of cores" for each process): T^{cul}_{rt}. Then, for each task, we compute the fractional core seconds on file I/O:

$$F_{fio} = T^{cul}_{fio} / T^{cul}_{rt}, \tag{1}$$

and fractional core seconds on communication:

$$F_{comm} = T^{cul}_{comm} / T^{cul}_{rt}, \tag{2}$$

where both F_{fio} and F_{comm} range from 0 to 1, with a value closer to 1 indicating more intensive use of the corresponding subsystem. We consider that all remaining core seconds are spent on computation:

$$F_{comp} = 1 - \left(F_{comm} + F_{fio}\right). \tag{3}$$

Thus, F_{comp}, F_{comm}, and F_{fio} can be used to classify an application as computation-, communication- or file I/O-intensive. Figure 1 presents the complementary cumulative distribution function of F_{fio}, F_{comm} and F_{comp} for all tasks from 2015 to 2019. The CCDF

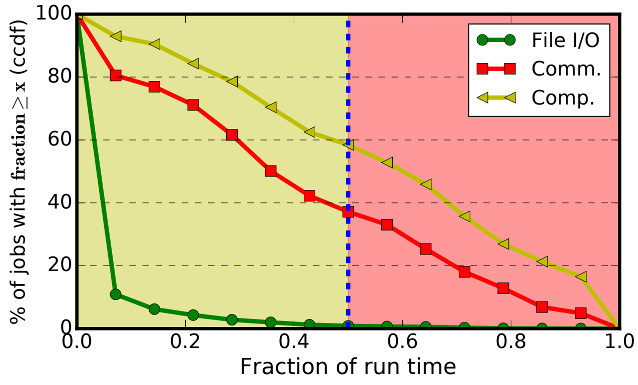


Figure 1: Complementary cumulative distribution functions of time spent on file I/O, MPI communication, and computation, separately, expressed as a fraction of the total run time. Any cumulative that exceeds 0.5 is considered "intensive-" and appears in the red-colored portion.

in Figure 1 shows that most tasks are computation-intensive: about 60% spent more than half of their total machine time on computation. MPI communication also consumed significant amounts of time, with MPI calls occupying more than half of total machine time for about 40% of all tasks. Surprisingly, file I/O operations are not very intensive; for nearly 95% of tasks, file I/O took less than 20% of total machine time.

Observation 1. Overall, nearly 40% of the total machine time was spent within MPI communication. But, about 30% of the tasks spent little time in MPI communication, and nearly half spent less than 2% of their total machine time on file I/O. These findings reveal that computation is still the most time consuming operation of applications. Jobs spend a significant amount of time on MPI, suggesting it is important to optimize MPI communication, either through equipping better network hardware for future designs or by optimizing frequently used MPI routines where bottlenecks occur.

3.2 Tasks per Job

As noted in §2.3, a job can be comprised of one or more tasks. We term a job that consists of a single task as a *single-task* job, and a job that launches more than one task over the course of its run, whether in sequence (one after the other) and/or concurrently (two or more at once), as a *multi-task* job. These two job types could also be referred to as traditional or SPMD (single program, multiple data), and workflow or MPMD (multiple program, multiple data), respectively. Figure 2 presents the statistics of single- and multi-task jobs from different perspectives for each of 2016–2019: Figure 2a shows the average number of tasks per job; Figure 2b presents the relative numbers of single-task versus multi-task jobs; and Figure 2c compares the machine time consumption by job type.

We see that, on average, the total number of tasks is several times (6x–10x) more than that of jobs. If we consider multi-task and single-task jobs simply by their percentage share of the total number of jobs, as shown in Figure 2b, single-task is consistently the majority, with 76% in 2015, up to 89% in 2017 and falling back to 72% in 2019. However, considering them based on their consumption of machine time (e.g., core-hours), extracted by cross-referencing with scheduler logs, single-task jobs steadily account for only slightly more than 60% of total machine time every year. In both cases, no clear year-over-year trend emerges.

Observation 2. Multi-task jobs are common and account for a considerable share of machine time. This observation suggests that better system-level support for such jobs (e.g., via support for MPI task management features [49] and for workflow languages such as Parsl [3] and Swift [48]) may be desirable. So, tool may be improved to support for dependency-aware task level scheduling, with the goal of increased backfilling.

3.3 Job size

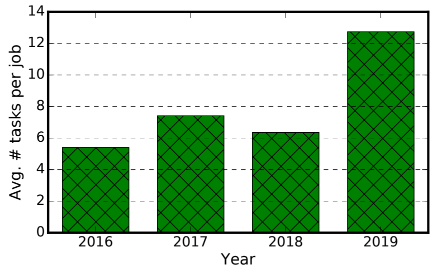
A major motivation for building a leadership supercomputer is that some applications (e.g., HACC [15], Quantum Monte Carlo method such as QMCPACK [23]) require numbers of nodes that cannot be obtained on smaller systems [18], i.e., leadership computers enable new capabilities. Here we examine the job size in terms of the number of nodes used, in order to determine how many nodes and how much machine time was used by computations of different sizes so as to observe the ubiquity of capability applications. We note that Mira computing nodes are divided into partitions and that the number of nodes allocated for a given job is a ceiling-rounding of the number requested. For example, the minimum partition size is 512 nodes, and thus 512 nodes are allocated to a job even if it requests just a single node. Figure 3 plots the distribution of job size by the percentage of total jobs (Figure 3a) and by the percentage of total machine time (Figure 3b) that they consumed per year.

We see from Figure 3a that most jobs are small: just 1% of all jobs used more than half of Mira's nodes. Nearly half of all jobs run with 512 nodes, which is just 1/96 of Mira's capacity. As far as machine time is concerned, most core-hours are consumed by jobs with 8192 nodes (1/6 of Mira), a number that may have been biased by Mira's scheduling policy. Jobs that used half or more of Mira's nodes consumed less than 10% of the total core-hours each year, except in 2017, in which they consumed 14% of total core-hours. Jobs using the entire machine consumed less than 5% of the total machine time, except in 2015, in which jobs using all of Mira consumed 6.2% of total machine time.

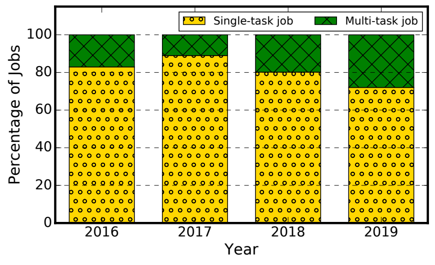
Observation 3. Although LCF encourages users to run large jobs by implementing queuing policies that prioritize jobs that use many nodes, most jobs are surprisingly small in terms of the number of nodes used. LCFs may need to devise strategies to increase the number of large jobs that take advantage of the scale of leadership resources.

3.4 Failure

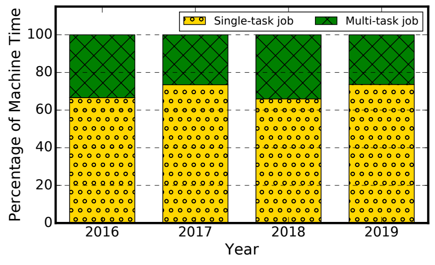
The exponentially increasing transistor count on a single chip and the large number of nodes (e.g., Mira has nearly 50 000 nodes) present in a supercomputer creates challenges when providing



(a) Average tasks per job.

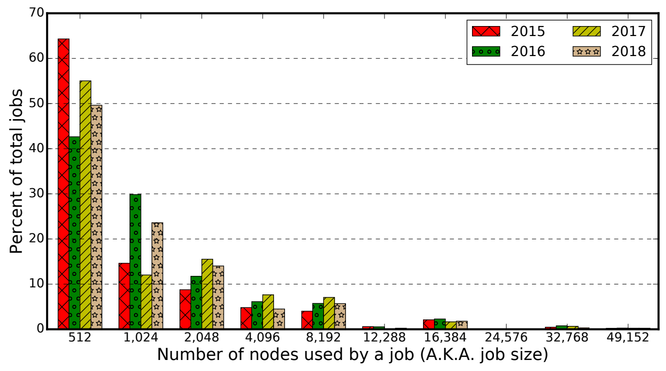


(b) Percentage by number of jobs.

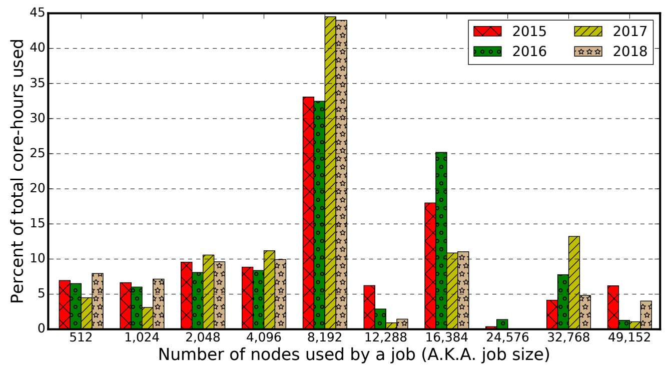


(c) Percentage by core hours.

Figure 2: Traditional single-task HPC jobs versus multi-task workflow jobs by year.



(a) By total number of jobs.



(b) By total core-hours.

Figure 3: Annual job size distribution. The sharp increase at 8192 in (b) is likely biased by the scheduling policy [10].

reliable hardware resources to run users' applications. Hardware failure is hard to avoid, for example, there are hundreds of thousands of FATAL messages annually on Mira. A fault-tolerant design can enable a system to continue its intended operation, possibly at a reduced level, rather than failing completely, when some part of the system fails. For example, much work [8, 21] has been done to predict hardware errors so that appropriate actions (e.g., scheduling, checkpointing) can be taken ahead to minimize their impact on applications. However, there has been little study of how likely a hardware failure is to cause application failure.

To investigate this issue, we used the Control System log and IBM RAS message datasets to study the probability that an RAS-FATAL event causes an execution failure. The Control System log provides the exit code for each application run (task). If a job fails to start or exits because of any signal (from the resource manager because maximum execution time is exceeded, by user request, or due to hardware failure: i.e., RAS-FATAL), this exit code will be between 1 and 255, inclusive. An RAS-FATAL message indicates a severe error event that presumably leads the application to fail or abort. We quantify here the likelihood of such a message leading to application failure or abort. If there is an application running on the node when the RAS-FATAL occurs, the control system ID will be recorded in the RAS message, otherwise N/A is shown. This mechanism allows us to correlate RAS messages and task behaviors.

Let us define event A as a failure of an application run, i.e., an exit code between 1 and 255 inclusive, and event B as an RAS-FATAL on any node where an application is running. Thus, based on Bayes' theorem we can calculate the conditional probability of an application failure caused by an RAS-FATAL event by:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}, \tag{4}$$

where $P(A)$ and $P(B)$ are the probabilities of observing A and B respectively; they are known as the marginal probability. We calculate $P(A)$ by dividing "total number of tasks failed" by "total number of tasks"; and $P(B)$ is the probability that an RAS-FATAL occurs during task execution, which can be calculated by dividing "number of task in all RAS-FATAL message" by "total number of tasks". We note that multiple RAS-FATAL messages may be associated with the same task, either at a different time or on different nodes, moreover, we perform a unique operation on the task ID list and consider duplicated IDs as one task. We also ignore RAS-FATAL messages where there is no application running on the compute node, e.g., during maintenance or when nodes sit idle. The conditional probability $P(B|A)$ denotes the probability of an RAS-FATAL occurring when we see a failure in task execution.

Let X denote the set of task IDs (duplicates removed) in RAS-FATAL events. These tasks either exit normally or fail. A failure may or may not be due to the RAS-FATAL. Let Y represent the set of failed task IDs. Then we can calculate $P(B|A)$ by:

$$P(B|A) = |X \cap Y|/|Y|, \tag{5}$$

where $|Y|$ is the size of set Y, similarly $|X \cap Y|$ is the size of the intersection set between X and Y. Table 1 lists the results of these calculations for the years 2015–2019.

Table 1: Quantitative results of the probability relationship between RAS-FATAL events and job failure.

Year	$P(A)$	$P(B)$	$P(B\|A)$	$P(A\|B)$
2015	0.213	1.49e-04	1.86e-04	0.265
2016	0.209	5.68e-04	7.19e-04	0.265
2017	0.183	3.43e-04	8.38e-04	0.447
2018	0.284	5.40e-04	4.94e-04	0.260
2019	0.214	2.17e-04	2.45e-04	0.242

Counter-intuitively, $P(A|B)$ is low, indicating that RAS-FATAL is not the most likely cause of an application failure or abort. The frequency of RAS-FATAL (i.e., $P(B)$) slightly increases with the age of the machine.

Observation 4. Surprisingly, the likelihood of a hardware FATAL event causing an application failure is not very high. Although how the vendor classifies the events as FATAL plays a significant role in this; it is quite possible that the applications that run on leadership computers and their dependent libraries are designed to have reasonably high fault tolerant capability.

4 CHARACTERIZING I/O BEHAVIOR

Applications use I/O to store output for further analysis, to checkpoint application memory for guarding against system failure, to exercise out-of-core techniques for data that do not fit in processor memory, and so on. Parallel file systems stripe data over multiple storage servers for high performance. Parallel I/O middleware libraries, such as MPI-IO, HDF5 [11] and PnetCDF [19] enable applications to store their data efficiently on parallel file systems. Much effort is invested in such I/O libraries to reduce or completely hide I/O costs from applications. Here, we observe the actual usage and behavior of parallel file I/O libraries.

4.1 Use of parallel I/O

As for how applications read/write data to/from parallel file systems in production, we studied the number of applications that use basic POSIX vs. higher level MPI-IO middleware (directly or indirectly via parallel file libraries such as HDF5). Since MPI-IO uses POSIX and Darshan captures POSIX without distinguishing raw POSIX from MPI-IO, we mark tasks that perform MPI-IO-based I/O operations (reads and/or writes), for at least one file, as MPI-IO aware. If no MPI-IO operations are used to read/write files, we mark those tasks/applications as basic POSIX only.

Figure 4 presents the file I/O library usage in applications. We can see that MPI-IO is less widely used than basic POSIX, accounting for only about 30% of total tasks on average. Luu et al. [35] saw similar behavior in 2015 in a study of the file I/O behavior of about a million jobs at another LCF, finding that nearly three quarters of applications only used POSIX. Mira data suggests that this result remains true in 2019.

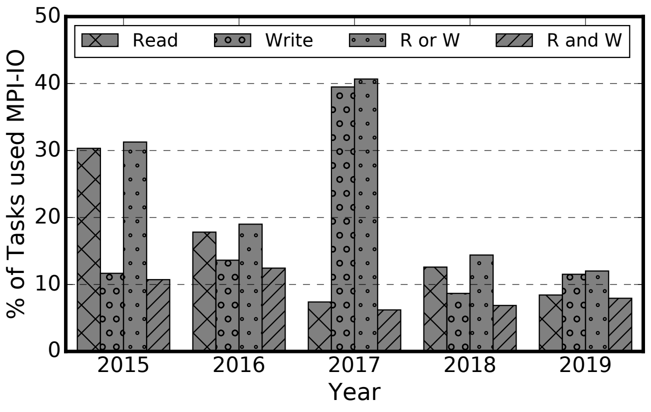


Figure 4: Percentage (to the total number of tasks) of tasks that used MPI-IO in its application.

Observation 5. In terms of file I/O library usage, basic POSIX is much more widely used than the high-level parallel MPI-IO. Although not all POSIX-using applications achieve poor I/O throughput, this observation motivates investigations of why higher-level parallel I/O libraries are not more widely used.

4.2 I/O sizes

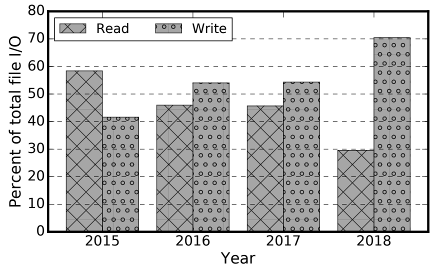
Figure 5a compares the bytes read from, and written to, the GPFS storage system over time. We see that from 2016 onwards the storage system is dominated by write workloads. In 2018, more than twice as many bytes were written than read. In contrast, a recent study [42] of I/O patterns at the National Energy Research Scientific Computing Center (NERSC) reported that 1.75× more data was read than written in 2018. NERSC is not an LCF; further investigation is required to determine if the pattern that we observe at ALCF is also seen at other LCFs.

Figure 5b and Figure 5c present the cumulative distribution of file sizes for files read and written on Mira from 2015 to 2018. File sizes have increased steadily over the four years studied, but most files are still smaller than one gigabyte. Indeed, we see from Figure 5b that the median size for files read is only about 1 MB and from Figure 5c that the median size for files written is only slightly larger. Considering that the GPFS block size for Mira is 8 MB, files that are smaller than 8 MB (more than half according to Figure 5b-c) cannot benefit from parallel I/O.

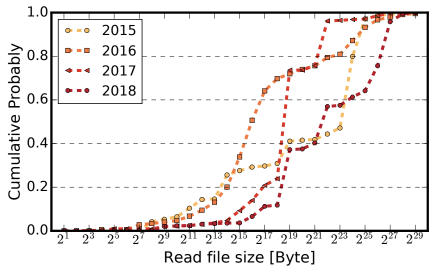
Observation 6. The average size of files read and written on Mira is small, hindering efficient parallel I/O. This observation suggests more work is needed to educate users as to the benefits of larger files and/or to adapt storage and I/O systems to perform better for smaller files.

4.3 Metadata time

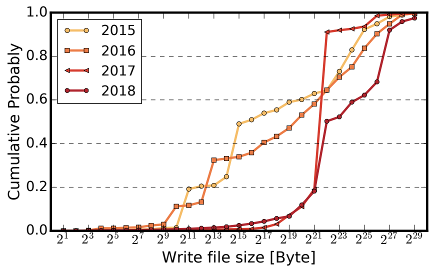
Considering that most files read/written on Mira were small in size, we may expect metadata operations to occupy a significant fraction of the time taken by file read/write operations. To investigate this supposition, we quantitatively studied the overhead associated with metadata operations for files read or written by applications. Specifically, for each file read or written, we extracted the total



(a) Annual total read/write bytes.



(b) Read file size.



(c) Write file size.

Figure 5: Annual total read/write size and cumulative distribution of tasks' file I/O size.

time as well as the time spent on operations related to metadata from Darshan logs. We then computed the fraction of time (with reference to the total file I/O time) spent on metadata operations for each file and studied their distribution for files read/written from/to the GPFS over four years. Figure 6 shows the distribution of the fraction of time spent on metadata operations. Metadata operations account for a surprisingly large fraction of total I/O time. For example, the median in 2018 for both MPI-IO and POSIX is about 0.38, which means that at least 38% of file I/O time was spent on metadata operations for half of the files read or written. Overall, the fraction of time consumed by metadata operation is less when MPI-IO was used, perhaps because MPI-IO is mostly used for large files.

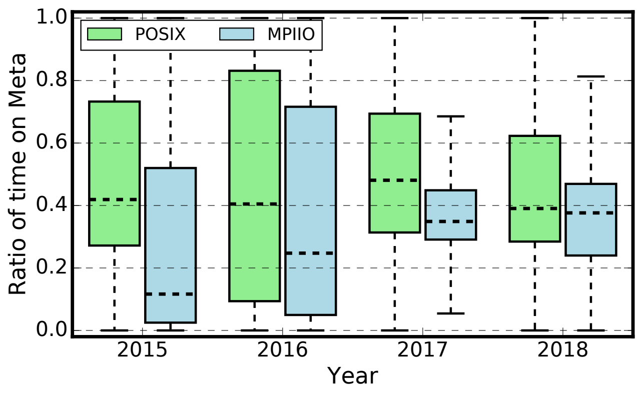


Figure 6: Per-file fraction of I/O time consumed by metadata operations.

Observation 7. Metadata operations account for a significant portion of total I/O time, due to the fact that the majority of files are small. Where feasible, system administrators should optimize their system configuration for smaller files. For example, the Lustre file system can be configured to store the data for small files on the metadata server, thus avoiding the overhead of creating an object storage target (OST) object, and initiating new remote procedure calls to the OST [34]. It will be valuable to encourage the use of parallel I/O, which is optimized for large files (at least a few MB in size), to achieve better per-file throughput.

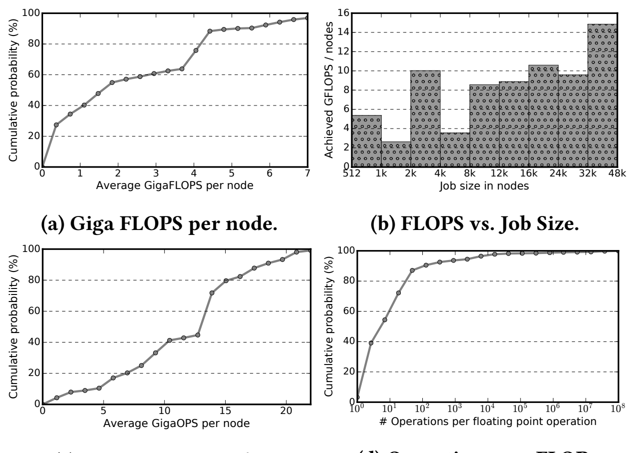
5 HARDWARE PERFORMANCE MONITORS

IBM hardware performance monitoring (HPM) [47] provides non-intrusive, very accurate, and low overhead counters to instrument the execution of an application at the *process* level (recorded in our Autoperf logs), node level, and/or job level. HPM gathers fine-grained performance metrics, revealing how many hardware resources the code has used. Thus, it can be used to identify and eliminate performance bottlenecks. In our analysis, we use HPM counters of many different applications at scale to study the characteristics of applications run on Mira. We mostly focus on the arithmetic floating point operations and main memory access behavior of applications.

5.1 Computing

Autoperf aggregates and records the total number of floating point operations (the PEVT_INST_**QFPU**_ALL field in its records) performed by each MPI process during an application's execution. Thus, given the number of MPI processes per node and the application's total runtime, we can compute, for each application, the achieved average FLOPS per node. Autoperf also records the total number of non-floating point operations (integer, load, store, and branch operations) as PEVT_INST_**XU**_ALL, and thus we can also compute the achieved average operations per seconds (OPS) per node by adding the QFPU and XU counters.

Figure 7 shows the cumulative distribution of OPS and FLOPS for all tasks for which Autoperf recorded data during our study period. The peak floating point performance of the PowerPC A2 processor is (1.66 GHz) × (16 cores) × (4 vector lanes) × (2 operations per FMA) = 212.48 GFLOPS/node. However, as shown in Figure 7a, most tasks for which data are recorded achieve only a small fraction of this number. For example, the 80th percentile achieved less than 4 GFLOPS, or ~2% of peak. Fortunately for Mira, but unfortunately for system analysts, these numbers are unlikely to be representative of total system behavior. As previously noted, Autoperf records are available for only 22% of all tasks during the period 2016 to 2019. There is anecdotal evidence that power users (e.g., HACC [14] which consumed about 4% of Mira's annual hours, achieved 7.02 PFlops on Mira, i.e., 70% of peak FLOPS) disable Autoperf, due to concerns about its overheads [7]. The 22% of tasks for which data are available likely achieve poor FLOPS numbers either because they have a low flop/byte ratio and are thus memory-bound, as presented in §5.2,



(a) Giga FLOPS per node. **(b) FLOPS vs. Job Size.**

(c) Giga OPS per node. **(d) Operations per FLOP.**

Figure 7: Cumulative distribution of the achieved per-node OPS, FLOPS, and Operations per FLOP for the 22% of all Mira tasks from 2016 to 2019 for which Autoperf data are available.

or because they do not perform many floating point operations, as we discuss in the following section.

As introduced in §1, the primary goal of leadership computers is to enable the execution of large applications that cannot be run anywhere else. Both resource allocation and the scheduling policy of Mira are biased to favor those large jobs [10]. Figure 7b shows the relationship between jobs size and FLOPS achieved. The trend in Figure 7b is counter intuitive, as one would expect large applications to achieve less FLOPS/node because of scaling problems. One possible explanation is that users with large applications put more effort into optimizing their code for Mira (and perhaps only for Mira), so that they can run their application at scale. For example, Lattice QCD, which consumes more than 10% of Mira's core–hours annually, achieved 1 PFLOPS using 32k nodes (about 30 GFLOPS per node) by carefully tuning their assembly code for Mira [37]. Thus, the average GFLOPS/nodes shown in Figure 7b is not representative of all tasks that run on Mira. It captures only 20% of the machine time and is also likely skewed by tasks that are small and/or not optimized for Mira.

As not all applications perform intensive floating point operations, we also characterize an application by the number of arithmetic operations (including floating point operations) performed per second: OPS. Figure 7c shows the cumulative distribution of achieved OPS. We see that achieved OPS are much higher than FLOPS. In order to further study the floating-point intensity of different applications, we divide the total number of operations by the total number of floating point operations (OPS/FLOPS), with results shown in Figure 7d. If we assume that the core operation of the application is calculation (e.g., solving partial differential equations), i.e., all other operation are auxiliary, then for more than 60% of the tasks for which Autoperf data are available, at least 10 auxiliary operations are performed for each floating point operation. Therefore, although the TOP500 project [46] uses FLOPS achieved by the LINPACK benchmark (which has a high percentage of floating-point arithmetic operations [9]) as the metric to rank supercomputers, the balance of OPS and FLOPS can better quantify the capability of a supercomputer for many applications.

Observation 8. Most of the applications that utilize the resources more efficiently and achieve high FLOPS/node (as high as 70% of peak FLOPS) have Autoperf instrumentation turned off. The fact that a significant percentage of the applications that achieved low FLOPS did not turn off the Autoperf instrumentation provides an opportunity for the operations team to proactively identify and notify the users of inefficient utilization of system resources, and to potentially provide appropriate training and support to the users to optimize their code. LCFs need to work with their power users and their applications to understand the impact of Autoperf- like instrumentation on the performance of their code. It will help reduce the overhead of the instrumentation and pave the way for increasing the coverage of instrumentation to more applications, which will help us get a holistic view of resource utilization and better understand the opportunities for optimization.

5.2 Memory

Each Mira compute node has 16 GB 1.333 GHz DDR3 memory with a peak throughput of 42.6 GB/s. Counters `PEVT_L2_FETCH_LINE` and `PEVT_L2_STORE_LINE` record the number of fetch and store operations, respectively, each of which corresponds to 128 bytes read or write. Much as we did for FLOPS, we calculated the average achieved RAM throughput. Figure 8 shows the cumulative distribution by application runs from 2015 to 2019.

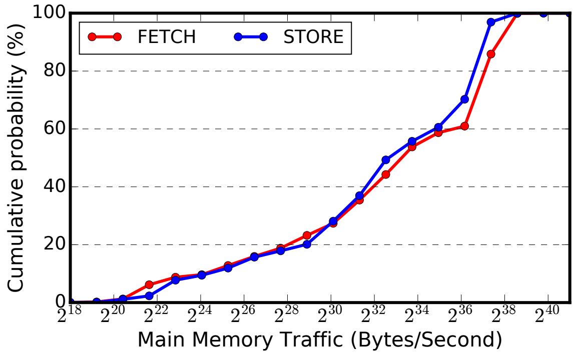
We see in Figure 8a that achieved main memory access throughput is high compared to peak arithmetic throughput. For example, about 40% of application runs achieved more than 32 GB/s, 75% of peak memory throughput. Some application runs achieved better than RAM peak throughput because of high cache hit ratio.

In order to investigate the balance between main memory load and arithmetic logic unit throughput, Figure 8b shows the average number of main memory accesses per arithmetic operation. The more main memory traffic, the more likely that an application is memory bound. We see that, on average, about 40% of all tasks performed more than three `FETCH`s from memory per arithmetic operation, and about 20% of tasks performed more than 2.5 `STOR`s to memory per arithmetic operation.

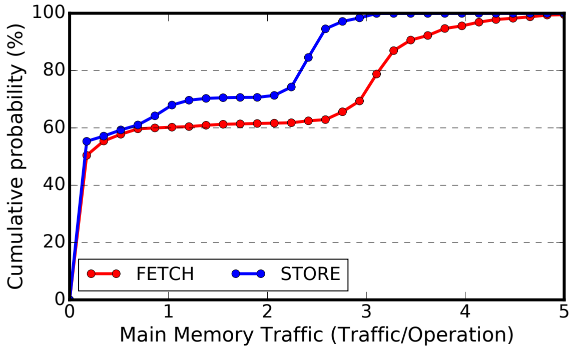
Observation 9. Hardware performance monitors provide a low-level signature for each task. When compared with FLOPS achieved, we find that main memory throughput is much closer to its peak. This observation reveals how worrisome the "memory wall" [38] (i.e., the growing disparity of speed between CPU and memory outside the CPU chip) is for applications on HPC systems.

5.3 Bytes per floating point operation

We next use the `bytes-per-FLOP` metric, which denotes the amount of communication in bytes (either to local main memory or to remote memory via MPI) for each floating point operation, to characterize each application. We then characterize trends in this metric over time and compare with the supercomputer hardware specification in the Top500 list [46].

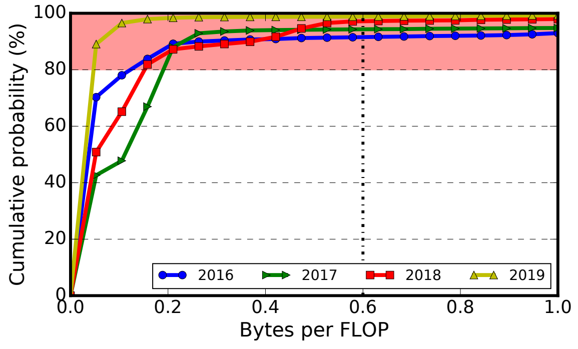


(a) Main memory throughput in bytes per second.

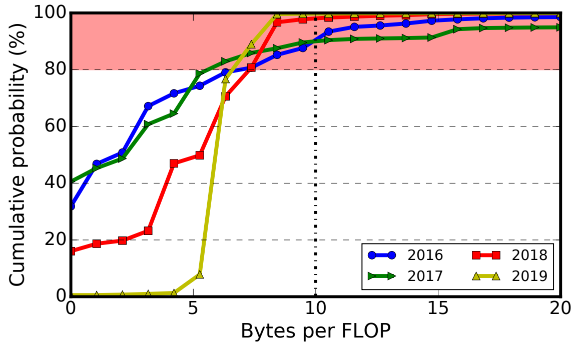


(b) Main memory load in accesses per arithmetic operation.

Figure 8: Cumulative distribution of average per-node main memory traffic on Mira for the period 2015–2019.



(a) Remote memory.



(b) Local memory.

Figure 9: Cumulative distribution of `bytes-per-FLOP`, to (a) remote and (b) local memory.

As `bytes-per-FLOP` ratios of supercomputers continue to decline, communication is supposed to become a bottleneck for performance scaling [40]. During the period 2010–2018, the TOP 500 supercomputer systems increased their computational throughput by an average of 65×, but their inter-node communication bandwidth by only 4.8× [4, 46]: a decrease by 93% in available `bytes-per-FLOP`.

Figure 9a shows the inter-node communication `bytes-per-FLOP` of applications run on Mira from 2016 to 2019. For communication-intensive applications, i.e., those with `bytes-per-FLOP` greater than the 80th percentile (red region) in annual statistics, there is a clear slightly increasing trend. For example, the percentage of applications with MPI `bytes-per-FLOP` greater than 0.6 decreased from about 10% in 2016 to less than 1% in 2019. This observation is consistent with that of §3.1 concerning trends in application computing demands.

Figure 9b shows the local memory (i.e., RAM) `bytes-per-FLOP`. While we do not see a clear trend year over year, the local memory `bytes-per-FLOP` of I/O-intensive applications have clearly decreased from 2017 through 2019. For example, the number of applications demanding 10 bytes or more per floating point operation has decreased from about 10% in 2017 to only around 1% in 2019.

Comparing Figure 9a with Figure 9b, we see that applications perform 40× more local than remote memory `bytes-per-FLOP`. These numbers are informative to the design of supercomputer as the applications' `bytes-per-FLOP` characterization reflects the actual demand.

Observation 10. The bytes exchanged with other processes per FLOP, and the bytes accessed from local memory per FLOP, decrease year over year. This trend is in line with that of supercomputers' hardware `bytes-per-FLOP` capability.

6 TASK REPRESENTATION

Based on our analysis and understanding, we next engineer and extract features from logs to represent applications in order to study the commonality of HPC applications. We use the following seven features to represent a task; these capture file I/O, computation, communication, and main memory access.

- Fraction of runtime used for inter-process communication (i.e., time spent on MPI routines);
- Fraction of runtime used for `MPI-IO`;
- Achieved operations per second;
- Achieved floating point operations per second;
- Number of processes per node;
- Average RAM fetch per CPU cycle; and
- Average RAM store per CPU cycle.

T-distributed stochastic neighbor embedding (t-SNE) [36] is a technique developed for the visualization of high-dimensional data. It converts similarities between data points to joint probabilities and tries to minimize the Kullback-Leibler divergence [24] between the joint probabilities of the low-dimensional embedding and the high-dimensional data [36]. Here, we start with using t-SNE, with a perplexity of 30 and 1000 iterations, to reduce our seven-dimensional feature representation to two dimensions for visualization. Figure 10a plots the two-dimensional t-SNE embedding of the feature representation. As one can see, there are multiple clear clusters. Theoretically, we can always visually see clusters using t-SNE because it can embed with any given high dimensional data, even random noise. Thus, we further label the plot in Figure 10a with a known property (e.g., the executable name) of the task to explore if the shown cluster is aligned with the task property. As an example, we label points (i.e., tasks) in Figure 10b with the same color/marker (zoom-in encouraged) if the executable names of the tasks are the same. To simplify visualization and reading, we only labeled the 10 most common executable names, and colored the rest red. Similarly, in Figure 10c, we label tasks by the name of the user who owns the application execution.

As we can see, either the username or the executable align with the t-SNE embeddings fairly accurately. This finding suggests that those tasks in the same visual cluster share some similarity and that the task representation features we use are capable of distinguishing and grouping tasks. We note that the same executable name may be run with different configurations (e.g., level of parallelization) or parameters (e.g., problem size). Thus, the same user may also run different applications. Thus, tasks with same executable names can be broken into multiple clusters (e.g., green in Figure 10b) and similarly, tasks in more than one clusters may belong to the same user. This analysis, at least, shows that HPC tasks can be represented by the presented features accurately. With this information, we could, for example, detect anomalous tasks of a given project.

While t-SNE provides compelling images and clear evidence of clustering, they are not easily used for the automatic assignment of tasks to clusters. (Moreover, because t-SNE is based on non-convex optimization, different runs with different initial states can lead to quite different results.) We thus built a machine learning model, based on the eXtreme Gradient Boosting (XGB) algorithm, to classify the application using the proposed features. XGB [6] is a high-performing gradient boosting optimized software framework and implementation, which is used widely by data scientists to achieve state-of-the-art results in machine learning challenges such as Kaggle and KDDCup; XGB is similar to Gradient boosting trees [12], but the former adopts a more regularized model formalization than the latter which gives it better performance. Researchers have used XGB to provide accurate performance predictions for computer systems and applications [29]. Moreover, XGB uses both novel algorithmic optimizations (a novel approximate tree learning algorithm, an efficient procedure to handle training point weights in approximate tree learning) and system level optimizations (out-of-core computation to process datasets that are too large to fit in main memory, and cache-aware learning to improve speed and scalability).

We build the classifier and train it as follows. We take the data from 2018 (totaling 127 585 Autoperf sampled tasks) and assign each item a label as follows: for the top 20 executables (covering 94.9% of all tasks), the executable name; for the remaining task, "other". Thus, it totals 21 labels for this classification (i.e., application identification) problem. We then use 70% of the labeled data, chosen at random, to train the classifier, and the remaining 30% to evaluate the trained classifier. We did a 5-fold cross-validation on the training dataset; once the model is trained, we used the test dataset to evaluate the trained model. Our testing accuracy is 99.5% (the distribution of 21 labels are relatively balanced), suggesting that our seven features can accurately distinguish applications, and indeed even distinguish different runs of the same application with different hardware resource settings or arguments. That is to say, the seven features, taken as a whole, can serve as a digital representation (called "fingerprint" in this study) of the application.

Regarding the applicability of the classifier (or task fingerprint), we note three potential opportunities:

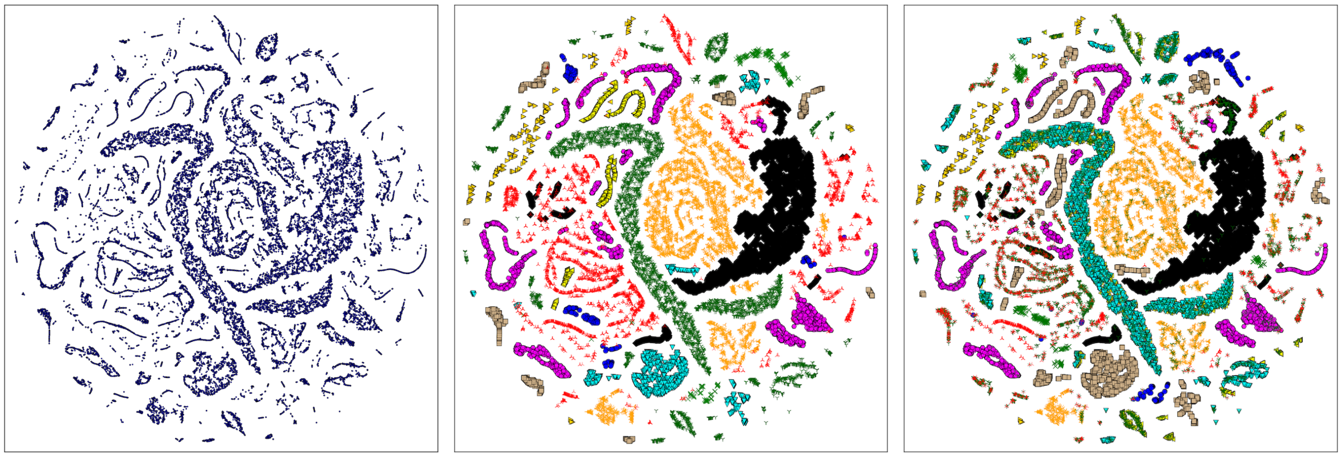
- Verify whether the users are using the system appropriately (actual application runs closely resemble the proposed usage) i.e., to ensure leadership computers are being used for intended scientific purpose;
- Since the fingerprint is built on performance counters, we can use it to categorize applications, for example as I/O intensive, communication intensive, computing intensive, or memory bound. Such categories can be used to target applications for optimization and other specialized services, particularly if they are to be executed repeatedly.
- Adding energy consumption dimension to the representation could help categorize applications/projects/users for better energy-cap control.

***In summary**: Tasks can easily be grouped and identified using our interpretable feature representation. Statistically, it means that the proposed features can explain the various behaviors and characteristics of different applications.*

7 RELATED WORK

The large scale supercomputer, high performance storage system, high speed network, and dedicated data transfer nodes are four major subsystem resources of a leadership computing facility. Much log analysis work has been done to characterize both subsystem performance and user behavior [7, 26, 27, 31–33, 42]. For example, Patel et al. [42] analyzed the Lustre storage system monitoring log of NERSC's Cori supercomputer to investigate the I/O behavior of large-scale applications. Chunduri et al. [7] characterized MPI usage on Mira at Argonne, obtaining useful insights for MPI developers, network hardware developers, and supercomputing center operators. Liu et al. analyzed production logs of file transfer to/from LCFs [28, 32] and proposed methods [22, 30] to optimize the file transfer performance based on the their observations.

Lockwood et al. [33] presented important considerations for HPC storage practitioners from a combination of facility level storage system logs and benchmark tests. Lim et al. [26] provide comprehensive insights on user behavior from multiple science domains through metadata analysis of a Petascale file system at a leadership computing facility. Gainaru et al. [13] analyze the effects of interference on application file I/O bandwidth and found that a significant percentage of the computing capacity of large-scale platforms is wasted because of interference incurred by multiple applications that access a shared parallel file system concurrently [13].



(a) Unlabeled 2D t-SNE embedding. (b) Labeled by executable name. (c) Labeled by username.

Figure 10: Two-dimensional t-SNE embedding of task representation. Dots (tasks) with the same color share the same information: executable name in (b) and username in (c).

Unfortunately, each of these studies only focused on analyzing logs of a single subsystem. In contrast, we present a more global view of HPC application characterization. To the best of our knowledge, there is not any work that combined logs of multiple subsystems and co-analyzes to characterize the leadership computing facility applications.

8 DISCUSSION AND CONCLUSION

In this study, we co-analyzed five different logs to gain insights into job characteristics, categorization, and system usage. To the best of our knowledge, this is the first study of its kind to systematically characterize the leadership computing applications from real logs. Our analyses provided useful insights concerning the file operation behavior, communication patterns, computational demand, and run time breakdown of applications. We believe that these insights can benefit researchers, tool developers, resource providers, end users, and funding agencies from different perspectives. They also have the potential to yield new insights that can help the co-design of hardware and system tools, optimize existing system tools, and improve the performance and experience for end users.

Our analysis results and observations are shaped by the specifics of the Mira supercomputer, the ALCF, and Mira's application workloads. To the extent that the ALCF is a typical leadership computing facility, they likely have broader applicability. We believe that this characterization is useful for optimizing facility management, improving energy efficiency, and optimizing scheduling policy. Moreover, log-based characterization does not introduce any extra instrumenting burden to the precious supercomputer and it also adds value to the logs that are already being collected for debugging, troubleshooting, and auditing purposes. We also hope that insights gained from our analysis can suggest directions for further analysis and encourage other HPC centers to undertake similar such efforts.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. The datasets used in this research were generated from resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We thank Doug Waldron and Sudheer Chunduri, both from Argonne Leadership Computing Facility, for providing dataset descriptions. We also would like to thank the four anonymous reviewers for their helpful comments.

REFERENCES

[1] William Allcock et al. 2017. Experience and Practice of Batch Scheduling on Leadership Supercomputers at Argonne. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1–24.
[2] Gonzalo Pedro Rodrigo Alvarez et al. 2016. Towards Understanding Job Heterogeneity in HPC: A NERSC Case Study. In *16th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Comp. (CCGrid)*. IEEE, 521–526.
[3] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M Wozniak, Ian Foster, et al. 2019. Parsl: Pervasive parallel programming in Python. In *28th International Symposium on High-Performance Parallel and Distributed Computing*. 25–36.
[4] K. Bergman. 2018. Empowering Flexible and Scalable High Performance Architectures with Embedded Photonics. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 378–378. https://doi.org/10.1109/IPDPS.2018.00047
[5] Philip Carns. 2014. Darshan. In *High Performance Parallel I/O*. Chapman and Hall/CRC, 351–358.
[6] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. *arXiv preprint arXiv:1603.02754* (2016).
[7] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. 2018. Characterization of MPI Usage on a Production Supercomputer. In *International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas). IEEE Press, Piscataway, NJ, USA, Article 30, 15 pages. http://dl.acm.org/citation.cfm?id=3291656.3291696
[8] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. 2018. Desh: Deep Learning for System Health Prediction of Lead Times to Failure in HPC. In *27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona). Association for Computing Machinery, New York, NY, USA, 40–51. https://doi.org/10.1145/3208040.3208051
[9] Jack J Dongarra et al. 1992. Performance of various computers using standard linear equations software. *ACM SIGARCH Computer Architecture News* 20, 3 (1992), 22–44.

[10] Argonne Leadership Computing Facility. [n.d.]. Job Scheduling Policy for Mira/Cetus/Vesta. https://www.alcf.anl.gov/support-center/miracetusvesta/job-scheduling-policy-miracetusvesta.

[11] Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A file format and I/O library for high performance computing applications. In *Supercomputing*, Vol. 99. 5–33.

[12] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[13] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. 2015. Scheduling the I/O of HPC Applications Under Congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 1013–1022. https://doi.org/10.1109/IPDPS.2015.116

[14] Salman Habib, Vitali Morozov, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Tom Peterka, Joe Insley, Venkat Vishwanath, Zarija Lukic, David Daniel, Patricia Fasel, and Nicholas Frontiere. 2013. Blasting Through the 10 Petaflops Barrier: HACC on the BG/Q. https://press3.mcs.anl.gov//salman-habib/files/2013/05/hacc_pflops.pdf.

[15] Salman Habib, Adrian Pope, Hal Finkel, Nicholas Frontiere, Katrin Heitmann, David Daniel, Patricia Fasel, Vitali Morozov, George Zagaris, Tom Peterka, Venkatram Vishwanath, Zarija Lukić, Saba Sehrish, and Wei-keng Liao. 2016. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy* 42 (2016), 49–65.

[16] J. J. Hack and M. E. Papka. 2014. New Frontiers in Leadership Computing. *Computing in Science & Engineering* 16, 6 (Nov 2014), 10–12. https://doi.org/10.1109/MCSE.2014.125

[17] J. J. Hack and M. E. Papka. 2015. Big Data: Next-Generation Machines for Big Science. *Computing in Science & Engineering* 17, 4 (July 2015), 63–65. https://doi.org/10.1109/MCSE.2015.78

[18] Jim Collins Jared Sagoff. 2019 (accessed Dec 3, 2019). *A game changer for computational materials science*. https://www.alcf.anl.gov/news/argonne-s-mira-supercomputer-set-retire-after-years-enabling-groundbreaking-science.

[19] Jianwei Li, Wei-keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. 2003. Parallel netCDF: A High-Performance Scientific I/O Interface. In *ACM/IEEE Conference on Supercomputing*. 39–39. https://doi.org/10.1109/SC.2003.10053

[20] Wayne Joubert et al. 2012. An Analysis of Computational Workloads for the ORNL Jaguar System. In *26th ACM Int. Conf. on SuperComp.* ACM, 247–256.

[21] Qiao Kang, Ankit Agrawal, Alok N. Choudhary, Alex Sim, Kesheng Wu, Rajkumar Kettimuthu, Peter H. Beckman, Zhengchun Liu, and Wei-Keng Liao. 2019. Spatiotemporal Real-Time Anomaly Detection for Supercomputing Systems. In *Big Data Predictive Maintenance using Artificial Intelligence workshop*.

[22] Rajkumar Kettimuthu, Zhengchun Liu, David Wheeler, Ian Foster, Katrin Heitmann, and Franck Cappello. 2018. Transferring a petabyte in a day. *Future Generation Computer Systems* 88 (2018), 191–198. https://doi.org/10.1016/j.future.2018.05.051

[23] Jeongnim Kim, Andrew D Baczewski, Todd D Beaudet, Anouar Benali, M Chandler Bennett, Mark A Berrill, Nick S Blunt, Edgar Josué Landinez Borda, Michele Casula, David M Ceperley, et al. 2018. QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter* 30, 19 (2018), 195901.

[24] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79–86.

[25] Gary Lakner, Brant Knudson, et al. 2013. *IBM system Blue Gene solution: Blue Gene/Q system administration*. IBM Redbooks.

[26] Seung-Hwan Lim, Hyogi Sim, Raghul Gunasekaran, and Sudharshan S. Vazhkudai. 2017. Scientific User Behavior and Data-sharing Trends in a Petascale File System. In *International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado). ACM, New York, NY, USA, Article 46, 12 pages. https://doi.org/10.1145/3126908.3126924

[27] Yuanlai Liu, Zhengchun Liu, Rajkumar Kettimuthu, Nageswara Rao, Zizhong Chen, and Ian Foster. 2019. Data Transfer between Scientific Facilities – Bottleneck Analysis, Insights and Optimizations. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 122–131. https://doi.org/10.1109/CCGRID.2019.00023

[28] Zhengchun Liu, Prasanna Balaprakash, Rajkumar Kettimuthu, and Ian Foster. 2017. Explaining Wide Area Data Transfer Performance. In *26th International Symposium on High-Performance Parallel and Distributed Computing* (Washington, DC, USA). ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/3078597.3078605

[29] Zhengchun Liu, Rajkumar Kettimuthu, Prasanna Balaprakash, and Ian Foster. 2018. Building a Wide-Area Data Transfer Performance Predictor: An Empirical Study. In *1st International Conference on Machine Learning for Networking* (Paris, France). Springer, 20.

[30] Zhengchun Liu, Rajkumar Kettimuthu, Ian Foster, and Peter H. Beckman. 2018. Toward a smart data transfer node. *Future Generation Computer Systems* 89 (2018), 10–18. https://doi.org/10.1016/j.future.2018.06.033

[31] Zhengchun Liu, Rajkumar Kettimuthu, Ian Foster, and Yuanlai Liu. 2018. A comprehensive study of wide area data movement at a scientific computing facility. In *38th IEEE International Conference on Distributed Computing Systems* (Vienna, Austria). IEEE, 8.

[32] Zhengchun Liu, Rajkumar Kettimuthu, Ian Foster, and Nageswara S. V. Rao. 2018. Cross-geography Scientific Data Transferring Trends and Behavior. In *27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona). ACM, New York, NY, USA, 267–278. https://doi.org/10.1145/3208040.3208053

[33] Glenn K. Lockwood, Shane Snyder, Teng Wang, Suren Byna, Philip Carns, and Nicholas J. Wright. 2018. A Year in the Life of a Parallel File System. In *International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas). IEEE Press, Piscataway, NJ, USA, Article 74, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291755

[34] Lustre. 2019 (accessed Dec 3, 2019). *Data on MDT Solution Architecture*. http://wiki.lustre.org/Data_on_MDT_Solution_Architecture.

[35] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. 2015. A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In *24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA). ACM, New York, NY, USA, 33–44. https://doi.org/10.1145/2749246.2749269

[36] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, Nov (2008), 2579–2605.

[37] Robert Mawhinney. 2013. Lattice QCD from Mira or Probing Quarks at a Sustained Petaflops. https://www.alcf.anl.gov/files/Mawhinney_ESP_May_2013_0.pdf.

[38] Sally A McKee, Steven A Moyer, Wm A Wulf, and Charles Hitchcock. 1994. Increasing memory bandwidth for vector computations. In *Programming Languages and System Architectures*. Springer, 87–104.

[39] Wes McKinney et al. 2010. Data structures for statistical computing in Python. In *9th Python in Science Conference*, Vol. 445. SciPy Austin, TX, 51–56.

[40] George Michelogiannakis, Yiwen Shen, Min Yee Teh, Xiang Meng, Benjamin Aivazi, Taylor Groves, John Shalf, Madeleine Glick, Manya Ghobadi, Larry Dennison, and et al. 2019. Bandwidth Steering in HPC Using Silicon Nanophotonics. In *International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado). Association for Computing Machinery, New York, NY, USA, Article Article 41, 25 pages. https://doi.org/10.1145/3295500.3356145

[41] National Academies of Sciences, Engineering, and Medicine. 2016. *Future Directions for NSF Advanced Computing Infrastructure to Support U.S. Science and Engineering in 2017-2020*. The National Academies Press, Washington, DC. https://doi.org/10.17226/21886

[42] Tirthak Patel, Suren Byna, Glenn K. Lockwood, and Devesh Tiwari. 2019. Revisiting I/O Behavior in Large-scale Storage Systems: The Expected and the Unexpected. In *International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado). ACM, New York, NY, USA, Article 65, 13 pages. https://doi.org/10.1145/3295500.3356183

[43] Stephan Schlagkamp et al. 2016. Analyzing Users in Parallel Compiting: A User-Oriented Study. In *2016 Int. Conf. on High Performance Comp. & Simulation*. IEEE, 395–402.

[44] Stephan Schlagkamp et al. 2016. Consecutive Job Submission Behavior at Mira Supercomputer. In *25th ACM Int. Symposium on High-Performance Parallel and Dist. Comp.* ACM, 93–96.

[45] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright. 2016. Modular HPC I/O Characterization with Darshan. In *5th Workshop on Extreme-Scale Programming Tools (ESPT)*. 9–17. https://doi.org/10.1109/ESPT.2016.006

[46] TOP500. 2019 (accessed May 3, 2019). *TOP500 Supercomputer*. https://www.top500.org.

[47] Bob Walkup. 2019 (accessed Dec 3, 2019). *Application Performance Characterization and Analysis on Blue Gene/Q*. https://www.alcf.anl.gov/files/miracon_AppPerform_BobWalkup_1.pdf.

[48] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. 2011. Swift: A language for distributed parallel scripting. *Parallel Comput.* 37, 9 (2011), 633–652.

[49] Justin M Wozniak, Matthieu Dorier, Robert Ross, Tong Shu, Tahsin Kurc, Li Tang, Norbert Podhorszki, and Matthew Wolf. 2019. MPI jobs within MPI jobs: A practical way of enabling task-level fault-tolerance in HPC workflows. *Future Generation Computer Systems* 101 (2019), 576–589.

[50] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664