

ARGONNE NATIONAL LABORATORY  
9700 South Cass Avenue  
Argonne, IL 60439

## Collective I/O tuning using analytical and machine learning models<sup>0</sup>

**Florin Isaila,<sup>1,3</sup> Prasanna Balaprakash,<sup>1,2</sup> Stefan M. Wild,<sup>1</sup> Dries Kimpe,<sup>1</sup>  
Rob Latham,<sup>1</sup> Rob Ross,<sup>1</sup> and Paul Hovland<sup>1</sup>**

Mathematics and Computer Science Division

Preprint ANL/MCS-P5264-1214

December 2014

---

<sup>0</sup>This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research program, under contract number DE-AC02-06CH11357.

<sup>1</sup>Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

<sup>2</sup>Leadership Computing Facility, Argonne National Laboratory, Argonne, IL, USA

<sup>3</sup>Department of Computer Engineering, University Carlos III, Spain

# Collective I/O tuning using analytical and machine learning models

## ABSTRACT

The optimization of parallel I/O has become challenging because of the increasing storage hierarchy, performance variability of shared storage systems, and the number of factors in the hardware and software stacks that impact performance. In this paper, we perform an in-depth study of the complexity involved in I/O autotuning and performance modeling, including the architecture, software stack, and noise. We propose a novel hybrid model combining analytical models for communication and storage operations and black-box models for the performance of the individual operations. The experimental results show that the hybrid approach performs significantly better and shows a higher robustness to noise than state-of-the-art machine learning approaches, at the cost of a higher modeling complexity.

## 1. INTRODUCTION

The complexity of today’s large-scale supercomputers has substantially increased the challenges of optimizing the performance of parallel applications. Factors contributing to this evolution include the larger amount of concurrency, deepening memory/storage hierarchies, the distribution of system services (e.g., the separation of compute and I/O nodes), and the lack of global optimizations for the software stack. In this context, there is an increasing demand for novel methodologies and models to address these challenges.

On large-scale platforms many factors are involved in performance optimization. The architecture is an important factor, which can have an orders-of-magnitude effect on performance (e.g., caching in the memory hierarchy). The software stack offers tunable optimizations at different layers. Application factors such as access patterns also have a significant effect on performance. Moreover, external noise from application interference and system management activities can cause performance variability, which can cancel out the effect of an optimization.

Ideally, analytical models for all system components could be used for predicting the performance and automatically

choosing the best parameters for an optimization. However, analytical models for such complex systems are notoriously difficult to build and validate. At the other extreme, black-box modeling approaches, such as machine learning, leverage only the input and output parameters and therefore lack the flexibility of adapting to changing conditions (e.g., different levels of noise, modifications of a subsystem) and may require empirical evaluation of too many parameters to be computationally feasible. An intermediate approach is to use expert knowledge to build analytical models whenever the task is feasible and to complement the missing capabilities with data-driven approaches such as machine learning.

This paper presents a case study of a model-based approach for autotuning a popular collective I/O implementation, two-phase I/O from ROMIO [14], the most popular MPI-IO distribution. This collective I/O implementation is widely used in checkpoint and analysis output phases of computational science codes. Since its performance is known to be highly variable, it is a good target for optimization. The main contributions of the paper are the following. First, based on domain expertise, we build analytical models that drive the optimization process in combination with statistical performance models. Second, we identify several challenges inherent to this process and study the impact of architectural, software stack, and run time factors on the optimization process. Third, we present a comprehensive evaluation of our approach, including a comparison with the best-performing machine learning techniques in terms of speedup and model robustness.

The remainder of the paper is structured as follows. Section 2 introduces the problem and the experimental setup. Section 3 provides a high-level overview of our approach. Sections 4 and 5 respectively discuss black-box and hybrid modeling approaches of collective I/O. Section 6 presents the main modeling results. Section 7 discusses related work, and Sec. 8 presents conclusions about the hybrid approach and its benefits for improving performance predictability.

## 2. THE PROBLEM AND SETUP

For a given application and target architecture, the problem of finding an optimal I/O parameter configuration can be formulated as the mathematical optimization problem

$$\min_x \{f(x) : x \in \mathcal{D} \subset \mathbb{R}^m\}, \quad (1)$$

where  $x \in \mathbb{R}^m$  is a vector parameterizing  $m$  I/O decisions and  $\mathcal{D}$  captures the set of allowable configurations. Mathematically,  $f$  is a function that takes the vector  $x$  as input and returns a scalar  $f(x)$ , corresponding to a specified per-

**Table 1: Parameter values used for autotuning.**

Parameter	Value
$n$	128, 256, 512
$s_p$ (MB)	1, 2, 4, 8, 16, 32, 64, 96, 128, 196, 256
$s_{cb}$ (MB)	8, 16 (default), 32
$n_{a\_pset}$	4, 16, 64 (default)

formance metric for the configuration defined by  $x$ . In this paper, we focus on run time as the performance metric.

Several approaches exist for approximately solving (1). These approaches can be coarsely classified into *model-based tuning* and *search-based tuning*. In the former, the focus is on developing a performance model (a surrogate of  $f$ ), which is then used to identify the best configuration. In the latter, a search algorithm is employed to systematically navigate a search space of promising configurations in order to identify the best configuration.

In this paper, we focus on model-based tuning for two-phase I/O, the algorithm employed in the collective I/O implementation of ROMIO. The main idea of two-phase I/O is aggregating file system accesses from  $n \times c$  processes ( $n$  nodes with  $c$  cores) each writing  $s_p$  bytes to a smaller number ( $n_a$ ) of processes called *aggregators* before sending the data to the file system. In particular, for the Blue Gene/Q (BG/Q) platform used in this study, the  $n_a$  parameter is set through  $n_{a\_pset}$  as explained below. Each aggregator uses a buffer of size  $s_{cb}$ . In ROMIO  $n_{a\_pset}$  and  $s_{cb}$  are MPI Info hints and are set to default values; in this paper we examine the value of model-based tuning of these parameters.

For the performance of collective I/O operations, we use IOR, one of the most popular parallel I/O benchmarks [13], which has been shown to accurately reproduce representative parallel I/O access patterns and predict the performance of a significant class of scientific parallel applications. In our experiments,  $p = n \times c$  processes concurrently write contiguous non-overlapping regions to a file through MPI collective I/O (of MPICH 3.1 distributions), a common access pattern. Even though the models from this paper are evaluated for one benchmark, our approach is more general, as we model the collective I/O operation and not the overall access pattern of IOR. We concentrate on the file system write operations. However, a similar modeling and analysis approach can be performed for read operations. The IOR parameter values considered in our experiments are shown in Table 1. All the other IOR parameters have the default values. We refer to each unique combination of  $n$  and  $s_p$  values as an *instance*. There are 33 ( $3 \times 11$ ) instances, and our goal is to find the best  $(n_{a\_pset}, s_{cb})$  combination for each instance.

The experiments for our study are run on the Vesta BG/Q supercomputer at Argonne National Laboratory. Vesta has 2,048 compute nodes (4 racks of 512 compute nodes each) with PowerPC A2 cores (1.6 GHz, 16 cores/node, and 16 GB RAM). The compute nodes are interconnected in a 5D torus network and do not have persistent storage. Each compute node has 11 network links of 2 GB/s and can concurrently receive/send an aggregate bandwidth of 44 GB/s. While 10 of these links are used by the torus interconnect, the 11th link provides connection to the I/O nodes. On Vesta, a set of 32 compute nodes (known as a pset) has one I/O node acting as an I/O proxy. For every I/O node there are two network links of 2 GB/s toward two distinct compute nodes acting as bridge. Therefore, for every 128-node partition, there are  $n_b = 4 \times 2 = 8$  bridges. The I/O traffic from compute nodes passes through these bridge nodes on the

way to the I/O node. The I/O nodes are connected to the storage servers through Quad-data-rate (QDR) InfiniBand links. On BG/Q the programmer can set the number of aggregators per pset  $n_{a\_pset}$  (the hint on BG/Q is known as `bg_nodes_pset`). The total number of aggregators of an application  $n_a$  is computed as a function of  $n_{a\_pset}$ ,  $c$ ,  $n$ ,  $n_b$ :

$$n_a = (n_{a\_pset} + 1) \times n_b \times \frac{n}{128}. \quad (2)$$

In our study,  $c$  and  $n_b$  have constant values:  $c = 16$  (one process per core) and  $n_b = 8$  (architectural constant).

The file system on Vesta is GPFS 3.5. The data are stored on 40 NSD SATA drives with a 250 MB/s maximum throughput per disk; the block size is 8 MB. The file system blocks are distributed by GPFS in a round-robin fashion over several NSDs, with the goal of balancing the space utilization of all system NSDs. The I/O nodes are file system clients, and the size of the client cache on each I/O node is 4 GB.

### 3. APPROACH OVERVIEW

The major challenge in model-based I/O tuning stems from the fact that developing exact analytical expressions for function  $f$  in Eq. (1) is difficult, or even impossible. To address this, we investigate two approaches: *black-box* and *hybrid*.

The *black-box* approach to model the function  $f$  consists of developing a statistical surrogate model  $h(x) \approx f(x)$ , by fitting the model to *application run times* obtained from the evaluation of a small number of I/O parameter configurations on the *target machine*.

The *hybrid* approach that we propose is based on the fact that  $f$  depends on the number of communication and storage operations involved in I/O and their corresponding run times on the target machine. Therefore we have

$$f(x) = g(q(x, \hat{\theta}), \hat{\beta}, \oplus), \quad (3)$$

where  $q(x, \hat{\theta})$  is a function that computes the number of elemental operations for a configuration  $x$  using some system- and algorithm-specific hyperparameters  $\hat{\theta}$  and  $g(q(x, \hat{\theta}), \hat{\beta}, \oplus)$  is a function that uses  $q(x, \hat{\theta})$  to compute the run time of each elemental operation based on some system- and algorithm-level hyperparameters  $\hat{\beta}$  and combines  $\oplus$  these run times to predict the overall run time  $f(x)$ . Note that  $\hat{\beta}$  includes parameters that can have an impact on  $f(x)$  but remain unchanged and/or uncontrolled. The function  $g$  may also include the effects of network congestion, disk contention, and shared resources.

The development of the *hybrid* approach consists of three stages. First, by analyzing the ROMIO implementation, we derive an analytical model,  $m(x)$ , for  $q(x, \hat{\theta})$ . Second, because of the challenges involved in modeling the run time of I/O operations analytically (as outlined in Sec. 5.3), we develop a statistical surrogate model  $\tilde{g}(m(x), \hat{\beta})$  to estimate the run times of the elemental operations. Third, we use an analytical model  $\oplus$  to combine the run times of the elemental operations. We then have

$$\tilde{g}(m(x), \hat{\beta}, \oplus) \approx g(q(x, \hat{\theta}), \hat{\beta}, \oplus) = f(x). \quad (4)$$

The data for building the surrogate model  $\tilde{g}$  come from elemental-operation benchmark runs, which are independent of the application and are hence a one-time expense.

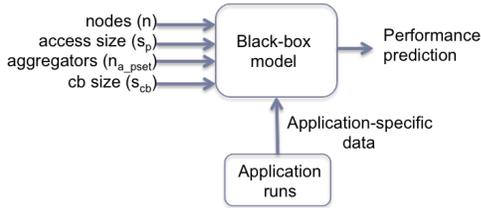


Figure 1: A purely black-box model.

In the following two sections, we detail the development of the models based on black-box and hybrid approaches.

## 4. BLACK-BOX MODELING

A purely black-box approach for I/O modeling (illustrated in Fig. 1) does not require detailed expert knowledge of the underlying algorithms used for communication, interconnect, or storage array operations. Typical black-box approaches are data-driven and based on supervised statistical modeling/machine learning. Formally, given a set of training points  $\{(x_1, y_1), \dots, (x_l, y_l)\}$ , where  $x_i \in \mathcal{D}$  and  $y_i = f(x_i)$  are I/O parameter configurations and the corresponding application run times, respectively, the supervised machine learning approach tries to find a surrogate function  $h$  for the expensive  $f$  such that the difference between  $f(x)$  and  $h(x)$  is minimal for all  $x \in \{x_1, \dots, x_l\}$  (and, ideally, all  $x \in \mathcal{D}$ ). The function  $h$ , which is called an empirical performance model, can be used to predict the run times of all  $x \in \mathcal{D}$  (not just those  $x_i$  for which  $f(x_i)$  has been evaluated) and can be used to identify configurations  $x^*$  that minimize the model  $h$ . We refer to this approach as **mL**.

The success of any such supervised-learning approach depends on rigorous model selection: identifying appropriate data-preprocessing techniques, selecting a supervised-learning algorithm that effectively models the relationship between the inputs and the output, and tuning the algorithm’s internal hyperparameters. A disadvantage of this approach stems from the fact that it is application-specific: a model obtained from training points for one application may not be useful for another application with different I/O characteristics. Moreover, a large number of training points—requiring many computationally expensive application runs or specialized methods for training-point selection—may be needed in order to obtain a sufficiently accurate model.

## 5. HYBRID MODELING

The hybrid modeling approach that we propose leverages the strengths of both analytical and data-driven models. In particular, our approach decouples the models for I/O operations from the models for estimating the performance (run times) of those operations on a given architecture. Figure 2 illustrates the hybrid approach. The first step is building performance-agnostic analytical models for I/O operation counts and data chunk sizes by using a priori knowledge of I/O algorithmic implementations. The second step is building data-driven performance models for the elemental I/O operations on the target architecture. The modeling effort at this step may include other performance-affecting external factors such as system noise, states of the operations (e.g., caching versus noncaching), and sensitivity to various architectural details (e.g., topology). The operation count models and data-driven performance models then are composed in a global model, which provides performance prediction.

To construct these models, we analyze the phases involved in the collective I/O implementation of ROMIO (see Fig. 3).

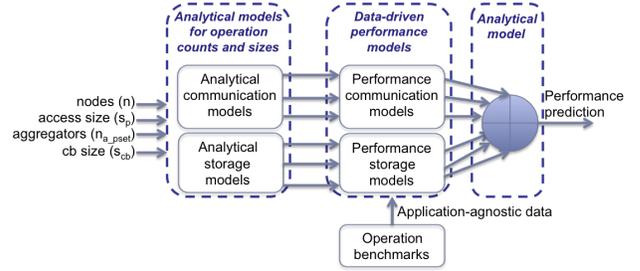


Figure 2: Hybrid model of two-phase I/O.

First, the total file range to be accessed is computed and partitioned among aggregators (file “partitioning”). Second, the data are shuffled into collective buffers from application processes to aggregators (“shuffle”). Third, the collective buffers are written to the file systems (storage “I/O”). These three steps involve network, CPU, and storage resources. Since the total time is dominated by network and storage time, we consider the CPU time negligible. Thus, the total run time  $f(x)$  of a collective I/O operation can be expressed as

$$\begin{aligned} f(x) &= g(q(x, \hat{\theta}), \hat{\beta}, \oplus) \\ &= t_{\text{partitioning}}(n, s_p, n_{a\_pset}, s_{cb}) \\ &\quad + t_{\text{shuffle}}(n, s_p, n_{a\_pset}, s_{cb}) + t_s(n, s_p, n_{a\_pset}, s_{cb}) \\ &= t_c(n, s_p, n_{a\_pset}, s_{cb}) + t_s(n, s_p, n_{a\_pset}, s_{cb}). \end{aligned}$$

In our use of IOR each process writes  $s_p$  bytes, and thus the total collective access size is given by  $s_t = n \times c \times s_p$ .

If the total data to be written ( $s_t$ ) is larger than the sum of all collective buffers, the second and third steps above are repeated until all the data are written. Each such repetition is called a *round*, and we can derive the number of rounds and the bytes transferred in the collective I/O operation. The maximum total size to be written in a round,  $s_r$ , is the total capacity of the aggregators’ collective buffers:  $s_r = n_a \times s_{cb}$ . The number of complete rounds is thus  $r_c = \lfloor \frac{s_t}{s_r} \rfloor$ . The last round may be incomplete because the remainder of the data to be transferred may be smaller than the total capacity of the aggregators’ collective buffers. The formula  $r_i = \lfloor (s_t \bmod s_r) > 0 \rfloor$  thus yields  $r_i = 1$  (TRUE) if the last round is incomplete, and  $r_i = 0$  (FALSE) if the last round is complete.

For an incomplete last round, the number of bytes transferred is calculated by subtracting the bytes transferred in complete rounds from the total access size:  $s_i = s_t - s_r \times r_c$ .

Using the preceding equations, we derive analytical models for communication and storage activity from ROMIO’s collective I/O implementation.

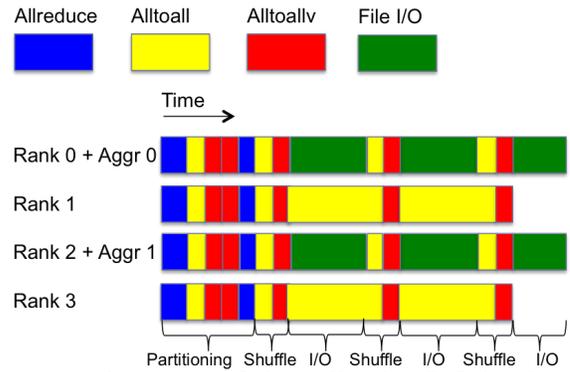


Figure 3: Anatomy of the collective I/O implementation from ROMIO. In this example, processes 0 and 2 act as aggregators.

## 5.1 Communication modeling

A collective I/O implementation analysis shows that on the target platform the following collective communication operations are involved: `MPI_Allreduce`, `MPI_Alltoall`, and `MPI_Alltoallv`. The communication time,  $t_c(n, s_p, n_{a\_pset}, s_{cb})$ , is given by the sum of communication operation times performed by the storage I/O operations `alltoallv` ( $t_{aav}$ ), `alltoall` ( $t_{aa}$ ), and `allreduce` ( $t_{ar}$ ):

$$t_c(n, s_p, n_{a\_pset}, s_{cb}) = t_{ar}(n, s_p, n_{a\_pset}, s_{cb}) \\ + t_{aa}(n, s_p, n_{a\_pset}, s_{cb}) + t_{aav}(n, s_p, n_{a\_pset}, s_{cb}).$$

Two-phase I/O uses two `MPI_Allreduce` operations in the file partitioning phase: one operation in which each node sends  $2 \times n \times c$  file offsets of 8 bytes and one operation in which each node sends one counter of 4 bytes, representing the size of data to be transferred. Given an `MPI_Allreduce` performance model  $\mathcal{M}_{ar}(n, c, s_{ar})$ , the allreduce operation time can be predicted by

$$t_{ar}(n, s_p, n_{a\_pset}, s_{cb}) = \mathcal{M}_{ar}(n, c, 2 \times n \times c \times 8) + \mathcal{M}_{ar}(n, c, 4).$$

Two-phase I/O calls `MPI_Alltoall`  $1 + r_c + r_i$  times in both the file partitioning and shuffle phases, each node sending to all other nodes one counter of 4 bytes. Given an `MPI_Alltoall` performance model  $\mathcal{M}_{aa}(n, c, s_{ar})$ , the alltoall operation time can be predicted by

$$t_{aa}(n, s_p, n_{a\_pset}, s_{cb}) = (1 + r_c + r_i) \times \mathcal{M}_{aa}(n, c, 4).$$

The collective communication operation `MPI_Alltoallv` is used in both the file partitioning and shuffle phases. In the shuffle phase, the communication pattern of `MPI_Alltoallv` depends on the file access pattern of the parallel application. In our use of the IOR benchmark, the access of each individual process is contiguous, causing each node to send one message to a single aggregator. The message size,  $s_{aav}$ , used for communication between a compute node and an aggregator is the minimum between the access size and the collective buffer size,  $s_{aav} = \min\{s_t, s_{cb}\}$ . The number of compute nodes sending a message to an aggregator is limited by the buffering capacity of the aggregator. For complete rounds, each of  $n_{aav\_send} = \min\{s_r/s_{aav}, n \times c\}$  processes sends one message to one of  $n_{aav\_recv} = \min\{n_{aav\_send}, n_a\}$  aggregators. For incomplete rounds, each of  $n_{aav\_send_{r_i}} = \min\{s_i/s_{aav}, n \times c\}$  processes sends one message to one of  $n_{aav\_recv_{r_i}} = \min\{n_{aav\_send_{r_i}}, n_a\}$  aggregators.

In summary, the following `MPI_Alltoallv` operations take place in two-phase I/O: one operation in which each node sends one file offset of 8 bytes to  $n_a$  aggregators, one operation in which each node sends one counter of 4 bytes to  $n_a$  aggregators,  $r_c$  operations in which  $n_{aav\_send}$  nodes send one message of  $s_{aav}$  bytes to  $n_{aav\_recv}$  aggregators, and  $r_i$  operations in which  $n_{aav\_send_{r_i}}$  nodes send one message of  $s_{aav}$  bytes to  $n_{aav\_recv_{r_i}}$  aggregators. Given an `MPI_Alltoallv` performance model  $\mathcal{M}_{aav}(n, c, n_{aav\_send}, n_{aav\_recv}, s_{aav})$ , the time can be predicted by

$$t_{aav}(n, s_p, n_{a\_pset}, s_{cb}) = \mathcal{M}_{aav}(n, c, n \times c, n_a, 4) \\ + \mathcal{M}_{aav}(n, c, n \times c, n_a, 8) \\ + r_c \times \mathcal{M}_{aav}(n, c, n_{aav\_s}, n_{aav\_recv}, s_{aav}) \\ + r_i \times \mathcal{M}_{aav}(n, c, n_{aav\_send_{r_i}}, n_{aav\_recv_{r_i}}, s_{aav}).$$

## 5.2 Storage modeling

In each round, once the data are collected in the collective buffers, aggregators issue storage accesses. If a round is

complete, each aggregator issues one file access of  $s_{cb}$  bytes. If a round is incomplete, only a fraction  $\frac{s_i}{s_r}$  of the  $n_a$  aggregators are involved in the storage access. Given a storage performance model  $\mathcal{M}_s(n, c, s_p)$ , the storage time is

$$t_s(n, s_p, n_{a\_pset}, s_{cb}) = r_c \times \mathcal{M}_s(n, \frac{n_a}{n}, s_{cb}) \\ + r_i \times \mathcal{M}_s(n, \frac{s_i}{s_r} \times \frac{n_a}{n}, s_{cb}).$$

## 5.3 Challenges of analytically modeling the performance of operations

Given a perfectly predictable system and an analytical model that calculates the count and the size of storage and network operations for a parametric configuration, one can exactly predict the performance of the system for various parameters and pick their optimal values. However, several factors present challenges to predicting the performance of computer subsystems such as the storage and network. In this section we discuss these factors and the construction of data-driven performance models for network and storage operations on BG/Q.

### 5.3.1 Communication

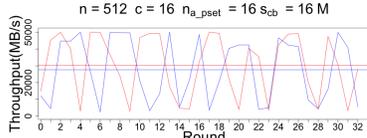
The performance of the 5D torus network on the BG/Q architecture is known to have small noise, since applications do not share network partitions. In all our evaluations of 5D network performance, the estimated standard deviation is within 0.1% of the mean of the operation latency. Performance interference is more likely to occur on other network links traversed by storage I/O traffic. The InfiniBand network and storage network are shared among all the applications.

### 5.3.2 Storage

Shared storage performance is notoriously difficult to model and predict [10, 15]. Several factors complicate the modeling, including interference, locking, network topology, caching effects, and disk architecture. We now illustrate some of these factors on the BG/Q platform used in this study.

**Interference:** The storage performance of a particular application on a BG/Q platform is subject to interference from other nodes concurrently accessing the shared file system. These nodes include I/O nodes serving the compute nodes of other applications, login nodes, and service nodes. The interference occurs at various levels of the storage hierarchy, including file system caches on the I/O nodes, interconnects (i.e., compute to I/O node, InfiniBand, and storage network), and storage. The interference is difficult to measure and model, since it can be caused by unexpected variations in user activities, applications, or system administration tasks, typically appears in short bursts, and may depend on the stateful services such as caching.

Figure 4 shows the performance of two different runs representative for the storage I/O phase of collective I/O. The writing processes are selected based on the topology-aware algorithm of the collective I/O implementation, and the concurrent accesses are repeated 32 times in which 544 processes on 512 nodes (in all three cases  $n_{a\_pset} = 16$ ) are writing 16 MB each ( $s_{cb} = 16$  MB). Even though the same parameters are used, the throughput in each round differs significantly. The variation in performance from one round to the next is most likely due to file system caching effects. The performance pattern appears to be out of phase, which may indicate that the states of the caches at the beginning of the



**Figure 4: Concurrent POSIX file write throughput for two different runs. Each process writes 16 MB, for 32 consecutive times. The horizontal lines represent the mean throughput for all rounds.**

two runs are different. Furthermore, the unexpected drops in performance can probably be attributed to other applications, users, or administrative tasks sharing the file system.

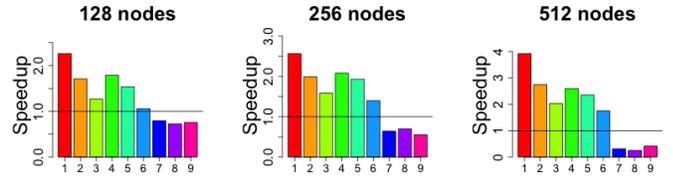
**Locking:** GPFS guarantees POSIX read/write atomicity semantics for file system operations [12]. The GPFS locking protocol dynamically partitions a shared file into byte-range locking domains. The partitioning into lock domains is performed on demand. The first client is granted a lock for the whole file. Subsequent accesses from other clients gradually partition the file into locking domains, so the first access of any client for a new non-overlapping domain is slower because it involves the partitioning of the locking domain. The locking overhead increases with lock domain fragmentation.

In order to efficiently address this behavior in the collective I/O implementation for GPFS, the file is partitioned into a number of domains equal to the number of aggregators, and the domains are aligned to the file system block size (first phase in Fig. 3). With this approach, even if the collective I/O operation consists of several rounds, the overhead of locking is perceived only in the first round. For instance, in Fig. 4 the first operation is significantly slower than the average because it incurs the locking overhead.

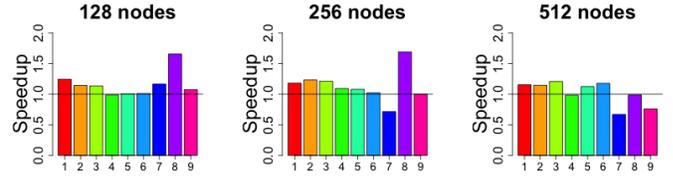
Figure 5 shows a comparison between the aggregate throughput in the first round and the average of the aggregate throughputs of subsequent rounds. The first round access was repeated 10 times on a newly created file, while the number of rounds for writing 8 MB/process, 16 MB/process, and 32 MB/process were 128, 64, and 32, respectively. For the number of writing processes we use the number of aggregators per pset ( $n_{a\_pset}$ ) from collective I/O listed in Table 1.

For smaller numbers of aggregators, the average performance of all rounds excluding the first round is substantially better than the performance of the first round. The reason is that the first round includes the locking overhead. For the largest number of aggregators (the default configuration of collective I/O on BG/Q), the performance of the first round is better. This may appear surprising but is easily explained: For a high number of aggregators, the total bytes written in each round represent a significant fraction of the aggregated file system client cache. Consequently, in the first round the chances are higher that there is enough available cache space and the performance is better than in the subsequent rounds, in spite of the cost of locking. For example, for 128 nodes and 520 aggregators writing 32 MB each, each round writes out  $520 \times 32 = 16,640$  MB, whereas the total file cache capacity of the four I/O nodes is  $4 \times 4,096 = 16,384$  MB. In this case, one round exceeds the cache capacity.

**Topology:** On BG/Q platforms, file system performance is topology-dependent. Bridge nodes are closer than non-bridge nodes to the I/O nodes and will thus see a higher file system throughput. The collective I/O implementation for BG/Q partially leverages topology information for choosing the process ranks that perform data aggregation. First, bridge nodes are selected and, subsequently, the remainder



**Figure 5: Non-locking versus locking speedup. The x-axis represents the 9 parameter configurations from Table 1 corresponding to the cross-product of**



**Figure 6: Topology-aware versus uniform aggregator placement speedup. The x-axis represents the 9 parameter configurations from Table 1 corresponding to the cross-product of  $s_{cb}$  and  $n_{a\_pset}$ .**

number of aggregators are distributed evenly into the pset of each bridge.

Figure 6 shows the speedup of topology-aware aggregator placement over the uniform aggregator placement. As expected, in most cases the topology-aware aggregator placement outperforms the uniform aggregator placement. The performance improvement can be up to 69%. In some cases, however, the uniform distribution places approximately 4 aggregators on each node, and the topology-aware placement does not appear to bring any advantage.

**Other factors:** Besides interference, locking, and topology we identified other factors that pose additional challenges to building an analytical performance model for the shared storage of BG/Q architectures. For example, the concurrent storage I/O operations coming from cores of the same nodes are serialized in the I/O forwarding implementation. Furthermore, the cache write-back policy directly influences the rate at which cache space becomes available and can have significant impact on performance. The performance of storage arrays also strongly depends on various factors such as caching, seeks, and rotational delays.

### 5.3.3 Communication and storage surrogate models

Building a model that incorporates all the factors affecting performance presents a challenge because many of these factors are difficult to observe and directly control. Therefore, we augment our model-based approach with a supervised-learning approach to model the performance.

For simplicity, the analytical model presented in Sec. 5.2 treats the write operations from all rounds equally. In reality, however, the performance varies from round to round because of the factors mentioned above. Building a performance model for each round is not feasible, however, because the number of rounds varies and each particular round can show highly variable performance in repeated runs.

We address this problem in the following way. First, we consider the first round independently, because it includes the file locking; we estimate the storage performance of the first round by averaging several runs. Second, for all subsequent rounds we average the performance of a number of accesses that is large enough to exceed the cache capacity. With this approach we account for caching effects, minimize

the impact of noise coming from other applications (by distributing the impact of access bursts of other applications over several rounds), and distribute as evenly as possible the impact of flushing over several rounds. Third, for estimating the storage performance of all runs we use performance data for the topology-aware case, which is representative for the POSIX operations of collective I/O. Based on these observations, we reformulate the storage model as follows:

$$\begin{aligned}
t_s(n, s_p, n_{a\_pset}, s_{cb}) &= \mathcal{M}_{s\_lock}(n, \frac{n_a}{n}, s_{cb}) \\
&+ \max\{0, r_c - 1\} \times \mathcal{M}_{s\_nl\_full}(n, \frac{n_a}{n}, s_{cb}) \\
&+ r_i \times \mathcal{M}_{s\_nl\_full}(n, \frac{s_i}{s_r} \times \frac{n_a}{n}, s_{cb}).
\end{aligned}$$

The data for storage performance are obtained based on a custom benchmark that concurrently writes data to a file from a subset of processes of an application. The benchmark allows the selection of the number of MPI processes performing POSIX storage I/O, and a process selection module chooses the nodes involved. Currently, two process selection techniques are supported for an MPI program with  $n \times c$  processes, of which  $n_a$  processes are performing storage I/O: uniform selection and topology-aware selection. In uniform selection, every  $\lceil n \times c / n_a \rceil$ -th process rank performs writes to the file system. The topology-aware selection represents the choice of aggregators from the collective I/O implementation: first the bridge nodes are chosen as aggregators, and then all the remaining processes are evenly distributed among the psets. The input for the data-driven storage performance model consists of three sets of data corresponding to the three cases discussed above. The first set, “s\_lock,” corresponds to the full and partial rounds including the file locking and consists of 270 points representing the latency of concurrently writing to a file 1, 2, and 4 file blocks (a file block is 8 MB) for various subsets of processes out of 2,048, 4,096, and 8,192 ranks. The second set, “s\_nl\_full,” measures full rounds not including locking and consists of 27 points. Each point represents the average latency of repeating a sequence of concurrent writes 1, 2, and 4 file blocks to the file system by three different subsets of processes out of 2,048, 4,096, and 8,192 ranks. In all measurements, the dataset is at least twice the size of the file system cache. The third set, “s\_nl\_partial,” corresponds to partial rounds not including locking and consists of 270 points representing the latency of writing 1, 2, and 4 file blocks for various subsets of processes out of 2,048, 4,096, and 8,192 ranks.

For building surrogate performance models, we measured the performance of collective communication operations involved in the collective I/O implementation based on an extended version of the ALCF MPI benchmark [11]. The original version of this benchmark reports the latency of three collective communication operations: Barrier, Broadcast, and Allreduce. We extended the benchmark for measuring the latency of two other collective operations: Alltoall and Alltoallv. The model input data for Allreduce consists of 57 points representing the mean latency of 100 repetitions for 2,048, 4,096, and 8,192 ranks and for message sizes between 4 bytes (one integer) and 1 MB (in powers of 2). The model input data for Alltoall consists of 51 points representing the mean latency of 100 repetitions for 2,048, 4,096, and 8,192 ranks and for message sizes between 1 byte and 256 kB (in powers of 2). The model input data for Alltoallv consists of 1,044 points for distributing message sizes between 1 byte and 64 MB (in powers of 2) for sets of 2,048, 4,096, and 8,192 ranks.

**Table 2: Cross-validation results for ml and hybrid**

Approach	component	$R^2$	RMSE
ml	-	0.85	1.79e+01
	$\tilde{\mathcal{M}}_{aa}$	0.89	3.58e-01
hybrid	$\tilde{\mathcal{M}}_{aav}$	0.88	6.73e+01
	$\tilde{\mathcal{M}}_{aar}$	0.84	4.58e-04
	$\tilde{\mathcal{M}}_{s\_nl\_full}$	0.57	1.89e+00
	$\tilde{\mathcal{M}}_{s\_lock}$	0.79	2.85e-01
	$\tilde{\mathcal{M}}_{s\_nl\_partial}$	0.83	2.49e-01

We deploy supervised learning to build surrogate performance models  $\tilde{\mathcal{M}}_{aav}$ ,  $\tilde{\mathcal{M}}_{aa}$ ,  $\tilde{\mathcal{M}}_{aar}$ ,  $\tilde{\mathcal{M}}_{s\_lock}$ ,  $\tilde{\mathcal{M}}_{s\_nl\_full}$ , and  $\tilde{\mathcal{M}}_{s\_nl\_partial}$  for  $\mathcal{M}_{aav}$ ,  $\mathcal{M}_{aa}$ ,  $\mathcal{M}_{aar}$ ,  $\mathcal{M}_{s\_lock}$ ,  $\mathcal{M}_{s\_nl\_full}$ , and  $\mathcal{M}_{s\_nl\_partial}$ , respectively. Substituting all these terms in Eq. (5), we obtain the final model for **hybrid**:

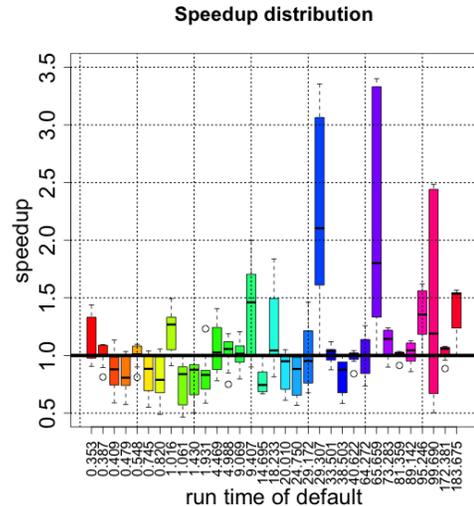
$$\begin{aligned}
f(x) &= g(g(x, \hat{\theta}), \hat{\beta}, \oplus) \\
&\approx \tilde{g}(m(x), \hat{\beta}, \oplus) \\
&\approx \tilde{t}_c(n, s_p, n_{a\_pset}, s_{cb}) + \tilde{t}_s(n, s_p, n_{a\_pset}, s_{cb}) \\
&\approx \tilde{\mathcal{M}}_{ar}(n, c, 2 \times n \times c \times 8) + \tilde{\mathcal{M}}_{ar}(n, c, 4) \\
&+ (1 + r_c + r_i) \times \tilde{\mathcal{M}}_{aa}(n, c, 4) \\
&+ \tilde{\mathcal{M}}_{aav}(n, c, n \times c, n_a, 4) + \tilde{\mathcal{M}}_{aav}(n, c, n \times c, n_a, 8) \\
&+ r_c \times \tilde{\mathcal{M}}_{aav}(n, c, n_{aav\_s}, n_{aav\_recv}, s_{aav}) \\
&+ r_i \times \tilde{\mathcal{M}}_{aav}(n, c, n_{aav\_send_{r_i}}, n_{aav\_recv_{r_i}}, s_{aav}) \\
&+ \tilde{\mathcal{M}}_{s\_lock}(n, \frac{n_a}{n}, s_{cb}) \\
&+ \max\{0, r_c - 1\} \times \tilde{\mathcal{M}}_{s\_nl\_full}(n, \frac{n_a}{n}, s_{cb}) \\
&+ r_i \times \tilde{\mathcal{M}}_{s\_nl\_partial}(n, \frac{s_i}{s_r} \times \frac{n_a}{n}, s_{cb}).
\end{aligned}$$

## 6. EXPERIMENTAL RESULTS

In this section, we present the model selection experiments in which we identify the suitable learning algorithms for **ml** and for the components of **hybrid**. We then compare the speedups obtained by **ml** and **hybrid** over **default** (i.e., using all default settings) and **optimal** (i.e., the parameter configuration providing the best performance). We also assess the impact of the number of training points on the effectiveness of **ml**, and we discuss the robustness of the two approaches with respect to noise.

### 6.1 Model selection

For a given supervised learning task, model selection con-



**Figure 7: Speedup distribution for 33 instances of IOR benchmark.**

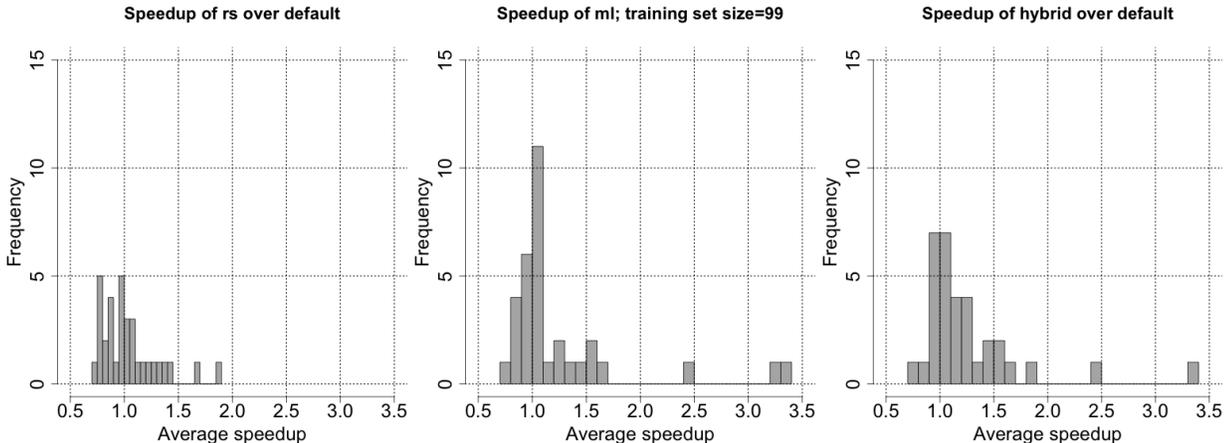


Figure 8: Average speedup of rs, ml, and hybrid.

sists of finding the right combination of data-preprocessing techniques, supervised-learning algorithm, and the algorithm’s hyperparameter values. For data preprocessing, we consider Box-Cox transformation, centering, scaling, and principal component analysis [3]. For the learning algorithm, we test a wide range of algorithms: linear regression, neural networks, support vector machines, random forest, and cubist [3, 6]. All the algorithms are implemented in the R “caret” package [7], which provides a framework for the model selection. Within this framework, for each learning algorithm, we sampled 30 candidate parameter configurations prescribed by the package and identified the best parameter setting using 10-fold cross-validation. We then compared each algorithm with the best parameter setting and selected the one based on Root Mean Squared Error (RMSE) and coefficient of determination ( $R^2$ ) values [3]. When a model perfectly fits the data, RMSE and  $R^2$  will be 0 and 1, respectively. For the ml approach, we generate the training set as follows. For each instance (a unique combination of  $n$  and  $s_p$ ), we sample 3 points using fractional factorial design. Consequently, 99 points are available for training the model. For hybrid, the training points are obtained independently of the IOR benchmark, as discussed in Sec. 5.3.3.

For hybrid, no single algorithm performs best for all the components. While  $k$  nearest neighbor was the best for  $\tilde{\mathcal{M}}_{s\_lock}$ ,  $\tilde{\mathcal{M}}_{s\_nl\_full}$ , and  $\tilde{\mathcal{M}}_{s\_nl\_partial}$ , support vector machines using radial basis functions outperformed other algorithms for  $\mathcal{M}_{aa}$  and  $\tilde{\mathcal{M}}_{aa}$ , and the cubist algorithm was the best for  $\tilde{\mathcal{M}}_{aar}$ . For the ml approach, support vector machines using radial basis functions performed best. The cross-validation results from the best learning algorithms for ml and hybrid are shown in Table 2. For ml, even though the  $R^2$  value is 0.85, we found that the model is less effective for shorter run times. For hybrid, the best algorithms obtain  $R^2$  values greater than 0.80, except on  $\tilde{\mathcal{M}}_{s\_nl\_full}$  (which we attribute to the paucity of training data).

## 6.2 Speedup comparison

We now examine the effectiveness of the ml and hybrid approaches over default. We also include random search (rs) as a baseline, in which the parameter configuration for each IOR instance is chosen uniformly at random. To reduce the impact of randomness, we repeat ml, hybrid, and rs 10 times. As a measure of effectiveness, we consider the average speedup obtained by a given approach with respect to the default. This is computed as follows. For a repetition  $r$  and an instance, speedup is given by the ratio of run time of the default and the best configuration obtained by a given

approach, respectively; the average speedup is given by the arithmetic mean of 10 speedup values from 10 repetitions.

First, we assess the effectiveness of the default configuration and speedup potential in the IOR benchmark. For each instance, we compute the speedup/slowdown of 9 configurations with respect to the default. Figure 7 shows the speedup/slowdown distribution on the 33 IOR instances. We observe that the default configuration is optimal or close to optimal for 12 instances. For the remaining 21 instances, however, other configurations show run times shorter than that of the default: more than 3x (2 instances), between 1.5x and 2.5x (6 instances), and up to 1.5x (13 instances). From these plots, we can also assess the difficulty of finding a better parameter configuration by analyzing the quantiles of the distribution. For 15 instances, the 75% quantile of the distribution is above 1, which suggests that 7 out of 9 configurations are better than the default. Consequently, on such instances, we conjecture that rs can find configurations that are better than the default.

Figure 8 shows the average speedup for rs, ml, and hybrid. As expected, rs finds better configurations than the default for 15 instances. However, the obtained speedups are less than 1.5x (the exceptions being 2 configurations for which rs obtains 1.7x and 1.9x). On 18 configurations, the run times of configurations obtained by rs are worse than the default. The ml and hybrid clearly outperform rs, and hybrid is significantly better than ml. The ml and hybrid approaches obtain speedups of more than 1.5x on 6 instances and between 1.1x and 1.5x on 5 and 11 instances, respectively. The hybrid approach is worse than the default on 2 instances, whereas the ml is worse on 5 instances.

Figure 9 shows the speedup distribution for rs, ml, hy-

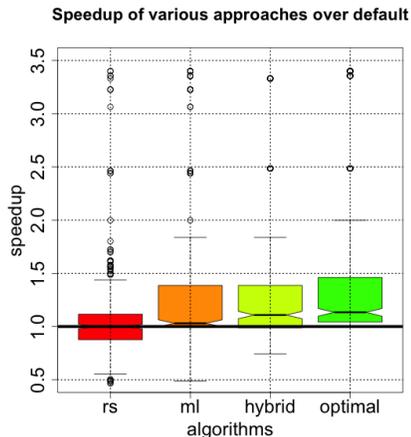


Figure 9: Comparison of rs, ml, hybrid, optimal, and default.

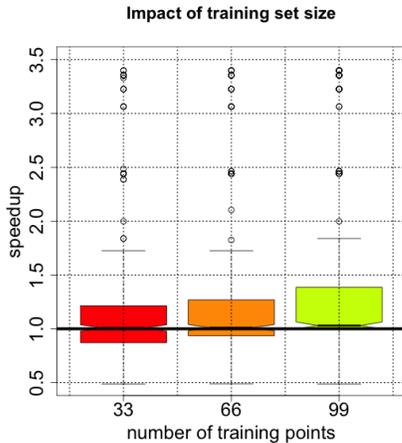


Figure 10: Impact of different numbers of training points on ml.

brid, and optimal with respect to the default over 10 replications. The distributions show a trend similar to that of the average speedups as shown in Fig. 8. The hybrid is close to optimal. Moreover, the 5% quantile of hybrid is at 0.25, suggesting that the potential relative slowdown is better than that of ml.

### 6.3 Impact of training points in ml

Crucial to the effectiveness of the ml approach is a sufficiently large number of points used for training the model. Therefore, we studied the impact of the number of training points on the speedup. In addition to the default training set size of 99 points, we used training set sizes of 66 and 33 points, chosen at random from the 99 training points for each replication. Figure 10 shows the speedup distributions obtained over 33 instances and 10 replications. We observe that the number of training points has a significant impact on the speedups. With the decrease in the training set size, the speedups become progressively worse. With 33 training points, only 50% of the configurations are better than the default, and the distribution is comparable to that of rs.

We expect that the number of training points used for building the surrogate storage models also affect the hybrid. Nevertheless, it will be a one-time application-independent cost and time to generate the training points from the benchmarks is rather tiny.

### 6.4 Impact of noise

We next assess the robustness of ml and hybrid with respect to the noise. Although previous research acknowledges the impact of noise in tuning [1], the robustness of tuning approaches has not been investigated thoroughly. While a detailed noise characterization in I/O tuning is beyond the scope of this work, we present a preliminary analysis of the impact of noise on the effectiveness of the ml and hybrid approaches. We assume that noise is random and can be characterized by a parameterized distribution. For our study, we consider Gaussian noise with mean  $\mu = 0$  and various standard deviations  $\sigma \in \{0.001, 0.01, 0.1, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0\}$ . For each run time  $t_i$  in the training set of ml and hybrid, we set  $t_i = t_i \times (1 + \mathcal{N}(\mu, \sigma^2))$ : for hybrid, we inject noise to the run times of the training set in each component (Alltoall, Alltoally, Allreduce, full round time, lock, and partial run times); for ml, the noise is injected for the run times of parameter configurations.

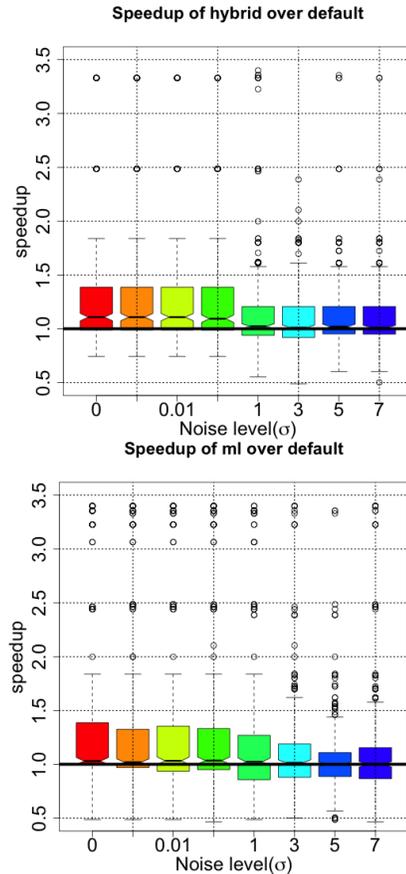


Figure 11: Impact of noise on ml and hybrid.

The speedup distribution of ml and hybrid for various  $\sigma$  levels is plotted in Fig. 11. The results show that despite relative noise added to the six components, hybrid is more robust than ml. For  $\sigma$  values up to 0.5, the speedup distributions of hybrid remain the same, but the speedup distribution of ml deteriorates starting at  $\sigma = 0.001$ . This result indicates that the magnitude of noise that is injected in ml significantly affects the relative rank/order of the configurations for a given instance. On the other hand, hybrid appears to be more robust to noise and preserves the relative ordering of the configurations. This behavior can be attributed to the separation of the analytical operation count models from the surrogate performance models, with the noise only affecting the latter.

## 7. RELATED WORK

Autotuning the performance of software I/O stack on increasingly complex platforms is extremely hard. The high numbers of parameters at various levels of the stack generate a huge search space. Genetic algorithms [2] and simulated annealing [5] have been used to address this problem. However, these approaches involve a huge overhead in estimating a high number of parameter combinations. Performance models have been used to reduce the parameter space of autotuning tuning [1]. However, system noise and other applications can interfere in the optimization process and pose additional challenges. You et al. [17] use queuing theory to model the performance of the Lustre file system on Cray XT5 systems and used a search-based approach to optimize access parameters. They reduce the search overhead using

simulation instead of real runs. Our work concentrates on model-based and not on search-based autotuning.

The optimization process has been also approached by using expert knowledge of the I/O system in run-time optimizations such as dynamically adapting middleware file domain partitioning to parallel file system locking protocols [9] and dynamically tuning parameters of collective I/O such as number of aggregators and buffer sizes [4, 16]. Expert knowledge remains valuable; but the number of components, availability of information on how these components operate, and complexity of interactions between components limit how well experts can guide applications to optimal solutions. In this paper we use expert knowledge for building analytical models for counts and sizes of network and storage operations involved in collective I/O.

Another set of works concentrates on using machine learning for parallel I/O autotuning. Kumar et al. [8] use several linear and nonlinear regression techniques to characterize and model the performance of the PIDX parallel I/O library and select tunable parameters. Yu et al. [18] leverage neural networks for building a platform-independent performance metamodel for tuning a set of parameters of the Panda I/O library. In our work we leverage various state-of-the-art machine learning techniques for building performance models of network and storage operations. However, we use machine learning in a hybrid model with the goal of exploring intermediate approaches that make practical use of expert knowledge of a collective I/O implementation.

## 8. CONCLUSION

This paper presents a case study of the effectiveness of model-based autotuning in the context of ROMIO collective I/O implementation. We compare a black-box approach and a hybrid approach leveraging analytical modeling and machine learning. In the hybrid approach analytical models are used for deriving the count and sizes of communication and storage operations. We address the challenges of evaluating the performance of storage operations by building a custom benchmark that synthesizes various factors affecting the modeling accuracy including architecture, system software, and noise. The results show that the two approaches outperform the currently adopted default values, and that the hybrid one is significantly better than the black-box approach. The hybrid approach shows a higher robustness to noise than does the black-box approach. Applying the hybrid methodology to a new platform does not require any applications runs, but only the evaluation of elementary network and storage operations through an arbitrary benchmark. This approach overcomes the limitations of the previously proposed search-based tuning methods that have significant runtime overhead.

While these results look promising, our approach is only a small step toward the ultimate objective of improving the autotuning of large-scale software stacks. Our findings could be used not only for improving the efficiency of autotuning but also for increasing the performance predictability of individual subsystems. Without performance predictability the increasing complexity of the storage I/O data path could make global performance optimization intractable.

## 9. REFERENCES

[1] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir. Improving parallel I/O autotuning with

- performance modeling. In *Proc. of HPDC '14*, pages 253–256, 2014.
- [2] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming parallel I/O complexity with auto-tuning. In *Proc. of SC '13*, pages 68:1–68:12, 2013.
- [3] C. M. Bishop. *Pattern Recognition and Machine Learning*, volume 1. Springer, New York, 2006.
- [4] M. Chaarawi and E. Gabriel. Automatically selecting the number of aggregators for collective I/O operations. In *CLUSTER (2011)*, pages 428–437, 2011.
- [5] Y. Chen. Automated tuning of parallel I/O systems: An approach to portable I/O performance for scientific applications. *IEEE Trans. Softw. Eng.*, 26(4):362–383, Apr. 2000.
- [6] G. Holmes, M. Hall, and E. Prank. Generating rule sets from model trees. In N. Foo, editor, *Advanced Topics in Artificial Intelligence*, volume 1747, pages 1–12. 1999.
- [7] M. Kuhn. Building predictive models in R using the caret package. *J. Stat. Soft.*, 28(5):1–26, 2008.
- [8] S. Kumar, A. Saha, V. Vishwanath, P. Carns, J. A. Schmidt, G. Scorzelli, H. Kolla, R. Grout, R. Latham, R. Ross, M. E. Papka, J. Chen, and V. Pascucci. Characterization and modeling of PIDX parallel I/O for performance optimization. In *Proc. of SC '13*, pages 67:1–67:12, 2013.
- [9] W.-k. Liao and A. Choudhary. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *Proc. of SC '08*, pages 3:1–3:12, 2008.
- [10] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. Managing variability in the IO performance of petascale storage systems. In *Proc. of SC '10*, pages 1–12, 2010.
- [11] V. Morozov, J. Meng, V. Vishwanath, J. R. Hammond, K. Kumaran, and M. E. Papka. ALCF MPI benchmarks: Understanding machine-specific communication behavior. In *Proc. of ICPPW '12*, pages 19–28, 2012.
- [12] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of FAST '02.*, 2002.
- [13] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proc. Conf. Supercomputing*, pages 42:1–42:12, 2008.
- [14] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proc. Front. Mass. Parallel Comput.*, FRONTIERS '99, pages 182–189, 1999.
- [15] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *Proc. of FAST '07.*, pages 5–5, 2007.
- [16] J. Worrigen. Self-adaptive hints for collective I/O. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 202–211, 2006.
- [17] H. You, Q. Liu, Z. Li, and S. Moore. The design of an auto-tuning I/O framework on Cray XT5 system. In *Cray Users Group Conf.*, CUG '11, May 2011.
- [18] S. Yu, M. Winslett, J. Lee, and X. Ma. Automatic and

portable performance modeling for parallel I/O: A machine-learning approach. *SIGMETRICS Perform. Eval. Rev.*, 30(3):3–5, Dec. 2002.

The submitted manuscript has been 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 No. 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.