Improving Parallel I/O Autotuning with Performance Modeling

Babak Behzad
Univ. of Illinois at Urbana-Champaign

Surendra Byna
Lawrence Berkeley Laboratory

Stefan M. Wild
Argonne National Laboratory

Prabhat
Lawrence Berkeley Laboratory

Marc Snir
Argonne National Laboratory

ABSTRACT

Various layers of the parallel I/O subsystem offer tunable parameters for improving I/O performance on large-scale computers. However, searching through a large parameter space is challenging. We are working towards an autotuning framework for determining the parallel I/O parameters that can achieve good I/O performance for different data write patterns. In this paper, we characterize parallel I/O and discuss the development of predictive models for use in effectively reducing the parameter space. Applying our technique on tuning an I/O kernel derived from a large-scale simulation code shows that the search time can be reduced from 12 hours to 2 hours, while achieving 54X I/O performance speedup.

Keywords

Parallel I/O, Autotuning, Performance Optimization, Performance Modeling

1. INTRODUCTION

Achieving efficient parallel I/O in high-performance computing (HPC) applications is a nontrivial task because of the complex interdependencies between the multiple layers of I/O middleware and storage hardware. Each I/O middleware layer offers a set of tunable parameters. However, the configuration of these parameters to obtain the best possible I/O performance depends on diverse factors, such as the I/O application, storage hardware, problem size, and number of processors. HPC application developers, typically experts in their scientific domains, do not have the time or expertise to explore the intricacies of I/O systems. Their resorting to using default I/O parameters often results in poor performance. As the complexity and concurrency of future HPC systems grow, we expect that so too will obstacles to achieving high-performance I/O.

We have recently developed a parallel I/O autotuning framework [2] with the ambitious goal of hiding the complexity of the I/O stack from scientific application developers. The autotuning framework uses a genetic algorithm (GA) to search through a large set of possible parameters. After constructing a random initial population, the GA produces new generations of populations by applying mutation and crossover operations. The GA thus determines values for the tunable parameters that result in good I/O performance. The dynamic library of our autotuner applies the selected parameter values by intercepting the data write calls. Our current implementation of the tuner, called "H5Tuner," is capable of intercepting the write calls of the HDF5 library and applying tunable parameters from HDF5, MPI-IO, and parallel-file systems (i.e., Lustre and GPFS).

While we consistently demonstrated I/O write speeds between 2X and 100X in our previous work [2], the overhead of the GA approach was significant. For example, running the GA for fifteen generations with a population of forty members typically takes about twelve hours. This overhead is considerable; it severely limits the general-purpose applicability of such an autotuning framework.

In this paper, we significantly reduce the search time by using empirical models of the I/O performance. We characterize performance of a typical parallel I/O subsystem with multiple levels of data movement and develop performance prediction models. Existing models for predicting parallel I/O performance (see, e.g., [8, 6, 7]) often aim for highly accurate predictions of I/O performance and are relatively complex. Many of these models have limited applicability, being restricted to specific systems or I/O kernels. We take a two-step approach: the first step crafts an empirical model that effectively reduces the search space of interest and the second step searches in this small parameter space.

Our paper makes the following technical contributions: We develop an approach to construct automatically an I/O performance model. We then use the model to reduce the search space for good I/O configurations and we demonstrate the applicability of the autotuning framework to scientific I/O kernels with various problem sizes.

2. EMPIRICAL PERFORMANCE MODELS

We have tested our autotuning framework on Hopper system located at the National Energy Research Scientific Computing Center (NERSC). Hopper is a Cray XE6 system containing 6,384 twenty-four core nodes with 32GB of memory per node. We used a Lustre file system of Hopper with 156 OSTs and a peak bandwidth of about 35GB/s for storing data. We used Cray's MPI library v6.0.1, HDF5 v1.8.11,

HPDC'14, June 23–27, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2600212.2600708.

and H5Part v1.6.6 for compiling the I/O kernels.

We examined the VPIC-IO I/O kernel in our study. This kernel is extracted from a particle physics simulation (VPIC [3, 4]). The kernel mimics exact I/O operations of the real application configuration.

We denote the independent variables/parameters (e.g., the stripe count of Lustre) in our model by $\mathbf{x} = [x_1, \cdots, x_{n_x}]$ and the scalar-valued output/dependent variable (e.g., the write time) associated with the configuration $\mathbf{x}$ by $y(\mathbf{x})$. In our setting, this output depends on the state of the system and can be viewed as stochastic. By y^j we denote a particular measurement of the output at a specific $\mathbf{x}^j$. Hence, collected data is of the form $\left\{(\mathbf{x}^j, y^j) : j = 1, \ldots, n_y\right\}$, where the $\mathbf{x}^j$ need not be distinct (which occurs if replicated measurements are conducted at a particular $\mathbf{x}^j$).

We consider smooth, nonlinear models, which can be written as linear combinations of n_b nonlinear basis functions ϕ,

$$m(\mathbf{x}; \beta) = \sum_{k=1}^{n_b} \beta_k \phi_k(\mathbf{x}). \tag{1}$$

Once a basis ϕ has been selected, the hyperparameters β can be selected by standard regression-/optimization-based approaches. For example, since these models are linear in β, a common approach is to employ

$$\hat{\beta} = \arg\min_{\beta} \sum_{j=1}^{n_y} \left(m(\mathbf{x}^j; \beta) - y^j \right)^2, \tag{2}$$

which corresponds to the maximum likelihood estimator for β under the assumption that y is Gaussian.

There can be many choices of basis functions; for simplicity, we focus on terms that are low-degree polynomials in either the parameter, x_i, or the inverse of the parameter, $\frac{1}{x_i}$. In particular, we consider terms of the form

$$\left\{ \prod_{i=1}^{n_x} (x_i)^{p_i} \,:\, p_i \in \{-1, 0, 1\}, i = 1, \ldots, n_x \right\}. \tag{3}$$

We could have expanded our set to include terms that could better account for differences in scale (e.g., $x_1 \log(x_2)$) or higher degree polynomials (e.g., $\frac{x_1^2 x_2}{x_3^2}$), but found that the set (3) was sufficiently rich for our purposes.

Since one of our goals in building a model of the form (2) was simplicity of the model, we desired to incorporate only a handful of basis terms, n_b, from the set (3). Each term in (3) can be defined by the integer vector $\mathbf{p} \in \{-1, 0, 1\}^{n_x}$. We let $\hat{m}(\mathbf{x}; \mathbf{P})$ denote the model prediction at $\mathbf{x}$ resulting from selecting a basis defined by $\mathbf{P} = \{\mathbf{p}^1, \cdots, \mathbf{p}^{n_b}\}$ and using the coefficients defined by (2). Given an initially empty set $\mathbf{P}$, we follow a greedy procedure (also known as a *forward* model selection approach) of adding to $\mathbf{P}$ the $\mathbf{p}$ that most reduces the prediction error. Formally, this means we determine the $\mathbf{p}$ that solves

$$\min_{\mathbf{p} \in \{-1,0,1\}^{n_x}} \sum_{j=1}^{n_y} \left(\frac{\hat{m}(\mathbf{x}^j; \mathbf{P} \cup \mathbf{p}) - y^j}{y^j} \right)^2. \tag{4}$$

After updating $\mathbf{P}$, this procedure can be repeated until: (i) we have reached a desired limit on the number of terms to include, (ii) we have exhausted the set in (3), or (iii) additional terms lead to negligible reductions of the prediction error (which, under certain regularity assumptions can be interpreted as the terms not being statistically significant). In our experiments, we always terminated the approach based on (i), reaching an upper limit to the number of model terms.

Parameter	Tested Values	# of Values
c, stripe count	1,2,4,8,16,32,64,96,128,156	10
s, stripe size (MB)	1,2,4,8,16,32,64,96,128	9

Table 1: Training configurations (90 in total) tested as part of the single-node experiment.

Before proceeding, we note that in (4) we are using a relative error metric that is slightly different from the usual least-squares error criterion (e.g., as used in (2)). We made this choice in order to bias our model terms toward smaller values of the output y. In the context of I/O models for optimization, we are less interested in accurately predicting large times than we are small times. An alternative approach to building models based on a bias toward high-performing configurations is discussed in [1].

Here, we consider models that could be employed in tuning for multiple file sizes simultaneously. Consequently we will have $n_x = 4$ independent variables, $\mathbf{x} = (c, s, a, f)$, and there are $3^{n_x} = 81$ possible terms in the set (3).

Experimentally, we ran tests using VPIC-IO and different file sizes (i.e., different core counts). The training set for each of the VPIC-IO experiments and their file sizes are shown in Table 2. We have chosen to decrease the size of the training set as the core counts (and hence file sizes) increase because of the corresponding increase in computational resources required. The way that these training sets are chosen is done in a systematic and automatic manner: For example, for the 2048-core experiments for stripe count, out of the 10 values shown in Table 1, 3 were chosen to cover the space: [16, 32, 256]. We chose 4 values (in MB) for stripe size, [1, 4, 16, 64], and 5 values for the number of aggregators, [16, 32, 48, 64, 80]. This leads to 60 configurations used for training our model. Since 2048 cores on at least needs 85 nodes on Hopper, and we follow the one-aggregator-per-node rule, 80 is the maximum value of the aggregators.

# of cores	file size (GB)	training set size
128	32	216
256	64	120
512	128	72
1024	256	60
2048	512	60

Table 2: Breakdown of training set for the parallel I/O model.

Using the above approach on the entire training data set, we obtain a six-term basis of $\{1, f, \frac{f}{a}, \frac{a}{c}, \frac{cs}{a}, \frac{cf}{a}\}$. However, inspection of this basis shows that any resulting model is necessarily monotone in s: if the coefficient for $\frac{cs}{a}$ is positive, the write times are increasing in s, otherwise the write times are non-increasing in s. Consequently, we made the decision to include a seventh term. The term with a factor $\frac{1}{s}$ that best solved (4) given the other six terms was determined to be $\frac{a}{s}$. Therefore, our seven-term model is of the form

$$m(\mathbf{x}) = \beta_1 + \beta_2 f + \beta_3 \frac{f}{a} + \beta_4 \frac{a}{c} + \beta_5 \frac{a}{s} + \beta_6 \frac{cs}{a} + \beta_7 \frac{cf}{a}, \tag{5}$$

with a fit to the data yielding $\hat{\beta} = [-20.65,\ 0.11,\ 4.17,\ 27.13,\ 4.50,\ 0.0038,\ 0.01]$.

In the next section we will analyze this model's ability

to perform space reduction and optimization for a variety of I/O tuning tasks. Before proceeding to this study, we note that the model (5) includes both actionable parameters (c, s, a) as well as an ancillary parameter (f) determined from an input. In the context of model-based optimization, we could use this new model in a minimization for any file size for which the model is deemed reliable,

$$m^*(f) \equiv \min_{(c,s,a)\in\Omega} m(c, s, a, f). \quad (6)$$

We also note that the application considered here is a weak-scaling application (i.e., the number of processors used to run the application is directly proportional to the file size). Therefore, there was no need to use the number of processors (p) as another parameter in the model. If instead, the file size is fixed as we scale the number of processors, p, should also be an independent variable in the model.

3. APPLYING PERFORMANCE MODELS

We show the process of using the empirical model in our I/O autotuning framework in Figure 1. The three steps of the autotuning process are: *pruning, exploration,* and *refitting*. In the pruning step, for a given I/O kernel and problem size, the framework predicts the I/O cost for all combinations of tunable parameters and selects the top twenty configurations with the least I/O cost. In the exploration step, the framework executes the I/O kernel with the selected twenty configurations to determine their empirical (rather than predicted) performance. The framework then refits the model with the newly collected write time data included. In the simplest case, which we use in this paper, the autotuning system runs the top ten configurations with the refitted model and returns the best-performing configuration to the user. One can use this configuration for future executions of the application at varying levels of concurrency.

The selection of the best performing configurations from the model-predicted write times and the number of iterations of refitting are controllable by the user of our framework. While we used the top twenty configurations, which proved to be effective in our tests, if a user prefers to select a different number of best-predicted configurations or wishes to refit the model iteratively, the user can configure the framework with simple settings.

4. EXPERIMENTAL RESULTS

In this section, we present the I/O write time results for the VPIC-IO kernel at different scales. We first compare the performance of our autotuning framework using the empirical models with that of the previous framework using GAs [2]. We then evaluate the effectiveness of the model-based framework on a variety of problem settings.

To develop the model, we ran various training configurations. The number of configurations for each scale is shown in Table 2. The total time to run all the configurations of VPIC-IO at the specified number of processors was 16.5 hours. Note that this training cost is a *one-time expense* for the performance model. The resulting model is used for predicting write times across different concurrencies. Once the model is formed, the incremental time spent in the pruning, exploration, and refitting steps is minimal. For example, the exploration step of the VPIC-IO kernel using 2048 cores took 31 minutes. In contrast, our GA-based tuning process,

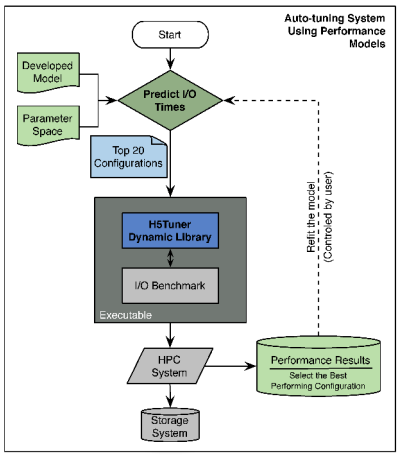

Figure 1: Design of our new autotuning system making use of performance models.

which tested roughly 400 configurations for the VPIC-IO kernel (running at 2048 cores), ran for 12 hours.

To summarize, the GA-based approach has a high runtime overhead associated with every kernel and scale level. The empirical-model-based approach has a one-time cost associated with fitting a model for a specific kernel, but can thereafter be used to predict times for any number of processors, with a fractional cost for refitting.

We now evaluate the framework with a large space of 640 configurations for the VPIC-IO kernel running on 512 cores. Figure 2 shows the twenty selected configurations with the least predicted write times for the original space along with the actual time it took to run them on the platform. The autotuning framework found configurations that achieve an approximately 1.4X speedup of write time performance by using the larger configuration/search space. The new configurations use larger stripe counts, stripe sizes, and numbers of aggregators. In this 512-core VPIC-IO experiment, the number of nodes used is equal to 22 (i.e., 512 divided by 24 cores per node). It has been suggested by some studies that using one aggregator per node achieves the best write times. We observe that all of the top-ten configurations use more aggregators than the number of nodes. Further analysis is needed to characterize the reasons for such behavior.

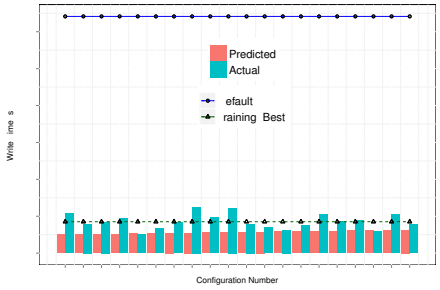

Figure 2: Comparison of the write times of the top-twenty configurations for VPIC-IO on 512 cores.

We tested the autotuning framework to tune the VPIC-IO kernel on a different number of processors (2048 cores). We compare the performance of the selected top-twenty config-

urations (from a space of 640 input configurations) and the best ten configurations (after the refitting process), respectively, in Figure 3. In this test, we observe that the tuned parameters values differ for I/O kernels running at different number of processors. Among the configurations, the number of aggregators is again larger than the number of nodes (85 for the 2048-core test) in many cases. Although the accuracy of predicted write times is lower than the 512-core experiment, the best configuration achieves a 27X speedup over the default configuration.

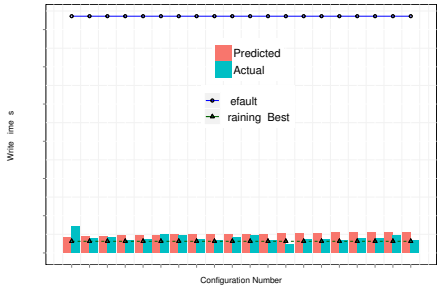

Figure 3: Comparison of the write times of the top-twenty configurations for VPIC-IO on 2048 cores.

We evaluate the model developed using the training configurations at smaller scales (see Table 2) in order to tune the I/O kernel running at 8192 cores. Note that we did not use any configurations from the 8192-core runs in training the model. The 8192-core runs use 342 nodes of Hopper and produce roughly 2TB of data. We used a configuration/search space of 1080 configurations for the model.

We show the I/O cost of the selected top-twenty configurations after the pruning and exploration steps in Figure 4. We observe significant performance improvement. The speedups over the default I/O configurations on Hopper at a concurrency of 8192 cores are on the order of 54X for VPIC-IO.

Table 3 summarizes the achieved speedups for the VPIC-IO kernel running at different concurrencies. The table also shows the size of the data written to the file system and the I/O bandwidth achieved. Overall, the tuned configurations achieve speedups ranging from 3.5X to 50X, which is consistent with exploring the search space using GAs. The time to traverse the search space after training was reduced from 12 hours to a maximum of two hours. In most cases, exploring the top-twenty configurations took one hour, resulting in significant improvements to overall parallel I/O performance.

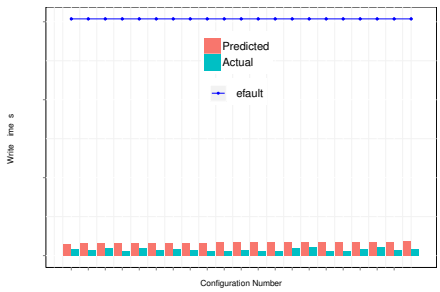

Figure 4: Comparison of the write times of the top-twenty configurations for VPIC-IO on 8192 cores.

# cores	I/O Kernel	File Size (GB)	Actual B.W. (MB/s)	Default B.W. (MB/s)	Speed-up
128	VPIC	32	2074.65	471.75	4.40
512	VPIC	128	5185.4	408.6	12.69
1024	VPIC	256	6181.75	336.6	18.37
2048	VPIC	512	11422.28	412.19	27.71
8192	VPIC	2048	18857.3	345.27	54.62

Table 3: Speedups of VPIC-IO with our autotuning framework.

5. CONCLUSION AND FUTURE WORK

This paper has presented an important development in our work on autotuning parallel I/O. We have dramatically reduced the run time for our framework from *12 hours to 2 hours* by incorporating an empirical performance model. The model accounts for major parameters pertaining to parallel I/O operations on a production supercomputing platform. We fit the model with a relatively small training set of application runs. The model was then used to predict configurations with high levels of I/O performance on two applications and at varying levels of concurrency.

Our current approach of determining a training set is based on a batch execution model. Namely, we precompute a training set with a space-filling design in advance, and evaluate the training set in a single batch job. We could have opted for an adaptive, "sequential design of experiments" approach (see, e.g., [5]), where each configuration is based on the results of the previous runs. This has the potential to further reduce the size of the training set.

Acknowledgment

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract numbers DE-AC02-05CH11231 and DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center.

6. REFERENCES

[1] P. Balaprakash, R. Gramacy, and S. M. Wild. Active-learning-based surrogate models for empirical performance tuning. CLUSTER '13, pages 1–8, 2013.

[2] B. Behzad, L. Huong Vu Thanh, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming parallel I/O complexity with auto-tuning. SC '13, 2013.

[3] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):7, 2008.

[4] S. Byna, A. Uselton, Prabhat, D. Knaak, , and Y. He. Trillion particles, 120,000 cores, and 350 TBs: Lessons learned from a hero I/O run on Hopper. In *CUG'13*, 2013.

[5] R. B. Gramacy and H. K. H. Lee. Adaptive design and analysis of supercomputer experiments. *Technometrics*, 51(2):130–145, 2009.

[6] H. Shan, J. Shalf, and K. Antypas. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *SC' 08*. ACM/IEEE, 2008.

[7] E. Smirni, C. L. Elford, D. A. Reed, and A. A. Chien. Performance modeling of a parallel I/O system: An application driven approach. In *PPSC*. SIAM, 1997.

[8] H. You, Q. Liu, Z. Li, and S. Moore. The design of an auto-tuning I/O framework on Cray XT5 system. In *CUG'11*, Fairbanks, Alaska, May 2011.