# Improving the MPI-IO Performance of Applications with Genetic Algorithm based Auto-tuning

Ayşe Bağbaba, Xuan Wang
The High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart
Stuttgart, Germany
Email: bagbaba@hlrs.de, xuan.wang.51@gmail.com

***Abstract*—Parallel I/O is an essential part of scientific applications running on high performance computing systems. Understanding an application's parallel I/O behaviour and identifying sources of performance bottlenecks require a multi-layer view of the I/O. Typical parallel I/O stack layers offer many tunable parameters that can achieve the best possible I/O performance. However, scientific users do often not have the time nor the experience for investigating the proper combination of these parameters for each application use-case. Auto-tuning can help users by automatically tuning I/O parameters at various layers transparently. In auto-tuning, using naïve strategy, running an application by trying all possible combinations of tunable parameters for all layers of the I/O stack to find the best settings is an exhaustive search through the huge parameter space. This strategy is infeasible because of the long execution times of trial runs. In this paper, we propose a genetic algorithm based parallel I/O auto-tuning approach that can hide the complexity of the I/O stack from users and auto-tune a set of parameter values for an application on a given system to improve the I/O performance. In particular, our approach tests a set of parameters and then, modifies the combination of these parameters for further testing based on the I/O performance. We have validated our model using two I/O benchmarks, namely IOR and MPI-Tile-IO. We achieved an increase in I/O bandwidth of up to 7.74×over the default parameters for IOR and 5.59×over the default parameters for MPI-Tile-IO.**

***Index Terms*—Parallel I/O; Auto-tuning; Genetic Algorithm; MPI-IO; Lustre**

## I. Introduction

Today's engineering applications running on high-performance computing (HPC) systems are correspondingly bottlenecked by the time it takes to perform input and output (I/O) of data from/to the file system. Since some applications spend most of their total execution times in I/O, this becomes time and resource consuming task [1]. Understanding an application's I/O behaviour and identifying sources of performance bottlenecks require a multi-layer view of the I/O. This is challenging due to the complex inter-dependencies between the layers of I/O stack [2].

Figure 1 indicates a typical parallel I/O stack of many current HPC systems, through which an I/O request normally goes: user application, high-level I/O library, Message Passing Interface I/O (MPI-IO), Portable Operating System Interface POSIX-I/O, parallel file system and storage system [3]. Computer scientists have developed various algorithms, I/O libraries and file management softwares to approach the theoretical bandwidth limit within each layer of the I/O stack [3].

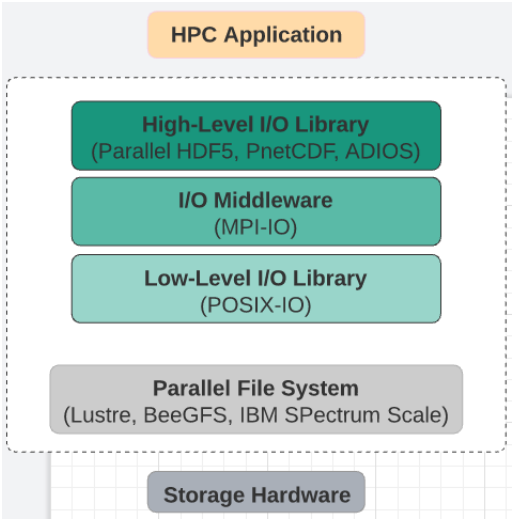



Fig. 1: Typical I/O Stack of an HPC System.

Each layer in the I/O stack offers a set of tunable parameters to adjust I/O performance of parallel applications to the optimum level such as Lustre file system stripe size and count, whether or not to perform collective I/O, number of data aggregators for collective I/O etc. [2]. However, the best combination of these parameters changes between systems and from one application use-case to the other.

Application related parameters, storage hardware properties, problem size and concurrency have an important effect on I/O parameters [4]. For example, choosing a larger stripe size and stripe count in the Lustre file system is normally recommended. However, for a small data transfer size (32 KB), the writing performance of striping over 4 OSTs with 1 MB stripe size was about %55 better than the performance of striping over 16 OSTs with 16 MB stripe size. On the other hand, the writing performance by disabling collective buffering optimization was about %173 better than the performance by enabling collective buffering optimization for a large data transfer size (64 MB). Disabling the collective buffering accelerated large scale I/O operations for writing large data collectively although it is an algorithm to accelerate the I/O operations,

Since scientists are experts in their own professional fields, they may have no time or experience for investigating the proper combination of tunable I/O parameters for each application use-case. Sometimes they might even drop the I/O performance by mistake [3]. They mostly implement the default settings which often causes poor I/O performance. Moreover, as the configuration space gets larger with today's supercomputer systems being complex, it brings further difficulties for users and developers to monitor the interactions between the configuration parameters and I/O performance [5]. Thus, there is an increasing demand for new studies to address these challenges.

This paper presents a genetic algorithm based I/O auto-tuning approach for MPI-IO ROMIO library and Lustre parallel file system. This study tries to auto-tune I/O configuration parameters between I/O stack layers transparently to the user in order to improve the I/O performance of parallel applications running on HPC systems. In particular, our auto-tuning approach generates a model using genetic algorithm (GA) and searches good combinations of tunable I/O parameters by using the model for higher I/O rate. The GA model can determine the optimal set of parameter values with a small number of tests rather than executing a number of application runs with all possible parameter combinations. This study makes the following contributions:

- It develops a GA based I/O auto-tuning approach that can hide the complexity of multiple I/O stack layers from users and increase I/O efficiency.
- It builds a model based on GA to search the I/O parameter space.
- It uses the GA model to reduce the search space for good configurations.
- It saves resources and core hours which are spent for trial runs unnecessarily in a naïve strategy.
- It considers the dynamic run-time conditions of a parallel I/O system.
- It evaluates and demonstrates the proposed approach with two main HPC I/O benchmarks.

The remainder of the paper is structured as follows: Section II motivates this work and gives related work regarding I/O research. Section III describes the background of the study, I/O performance factors and general auto-tuning approach which we propose for solving HPC I/O tuning problems. All the experimental setup and results are presented in IV. Section V concludes the paper and discusses some possible future research steps.

## II. Related Work

Among various optimizing potentialities, I/O request is one of the most requested parts. There are various approaches to find good configuration parameters in a large search space through auto-tuning to improve I/O performance of scientific applications.

The challenge of optimizing parallel I/O is that configuration parameters need to be sensibly evaluated and tuned. Tools such as Vampir [6] and Darshan [7] can be used for I/O monitoring and analysis, however; they cannot determine and tune I/O configuration parameter values [8].

Scalable I/O for extreme performance [9] offers a real-time parallel I/O optimization by assigning an I/O tracing thread running on each compute node. However; overhead produced by the MPI instrumentation is too high in a production environment for this study. Pattern-driven parallel I/O tuning for HDF5 applications is developed to optimize I/O performance of HDF5 applications across platforms and applications automatically [10]. Behzad et al. implemented genetic algorithm in I/O auto-tuning to traverse the search space systematically [11], [12]. A solution based on MPI-IO library could be more widely used and supports parallel HDF5. [3] presents an I/O auto-tuning framework for MPI-IO library, with a naïve strategy based auto-tuning that tries all possible combinations of configuration parameters to find the best. Improving parameter search strategy in such a framework would save resources. Analytical models are often inadequate and time consuming for expected predictive accuracy due to complexity of the state-of-the-art file systems [13].

At this stage, there is a need for an auto-tuning solution to search I/O parameter space effectively and auto-tune good configurations transparently to the users for MPI-IO and Lustre parallel file system which are widely used in scientific applications. It would also offer optimization possibilities for the other file systems such as IBM Spectrum Scale and BeeGFS. The parameters discussed in this paper are system dependent, but new parameters could be easily integrated to the configuration files.

## III. Auto-tuning Parallel I/O

In this section, we describe the background of the study, I/O performance factors and a GA based I/O auto-tuning approach which we propose for solving HPC I/O tuning problems. We show how our auto-tuning approach determines and auto-tunes I/O parameters by using the GA model.

A naïve strategy based auto-tuning requires running application with all possible combinations of I/O parameters to detect the best performing parameter set. This approach is an exhaustive search through the huge parameter space due to the long execution times of trial runs. It consumes time and resources even for unsuccessful parameter sets for the given application and system.

Figure 2 shows the overall architecture of our I/O auto-tuning approach that has two modules: *IO_Tuner* and *IO_Optimizer*. *IO_Tuner* module automatically tunes a good I/O parameter set proposed by a GA model *IO_Search* included in the *IO_Optimizer* module. The *IO_Search* executes the application or benchmark with a preselected random initial set of tunable parameters. I/O parameter search space is also given as input. It searches the parameter space to find the best performing parameter set iteratively. *IO_Tuner* takes the found configuration from *IO_Optimizer* and dynamically links to MPI-IO calls of the application. Then, I/O performance results can be used to refit *IO_Search* with the dynamic conditions of a parallel I/O system adaptively.

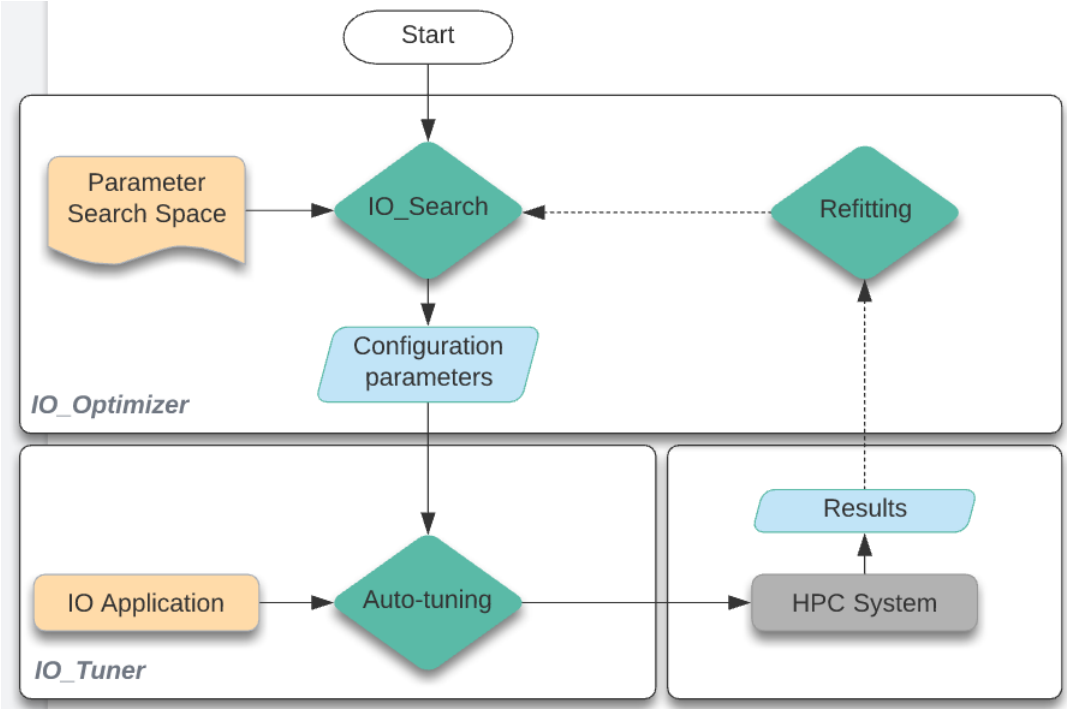



Fig. 2: Overall Architecture of the Auto-tuning Approach

### A. MPI-IO ROMIO Library

MPI-IO provides a high-level interface for applying parallel I/O algorithms, controlling file layout on file system, partitioning file data among processes logically, issuing collective and asynchronous/non-blocking data access etc. [3].

*MPI info* is an object that stores an unordered set of *(key, value)* string pairs,and they are passed as info parameter to MPI subroutines [3]. The MPI file info passes file access information from user applications to MPI-IO libraries or even the underlying distributed parallel file systems, so that the parallel I/O performance can be improved.

Except MPI reserved file hints, different MPI implementations and MPI-IO libraries can define their own file hints/info. For example, ROMIO defines *romio_cb_read* and *romio_cb_write* to enable/disable the collective buffering for MPI collective reading/writing subroutines. In addition, it delivers the MPI reserved file hints *striping_factor*(=*stripe_count* in Lustre) and *striping_unit*(=*stripe_size* in Lustre) to the underlying Lustre file system, which affect the Lustre file striping configurations.

ROMIO is one of the most used MPI-IO libraries on the market [3]. It consists of a large part of portable codes, a small part of file system and running machine optimized codes. To conquer the performance barrier of standard Unix I/O and the portable barrier of POSIX I/O, ROMIO has designed and implemented a component named Abstract-Device Interface of I/O (ADIO). Various parallel I/O APIs for standard UNIX and POSIX as well as specified file systems are implemented in ADIO [3].

Two parallel I/O algorithms, data sieving and collective I/O are integrated to achieve higher performance for small data accesses as well as non-contiguous I/O requests. Another optimizing possibility for MPI-IO is passing MPI hints to ROMIO [3].

### B. Lustre Parallel File System

The Lustre file system is an open-source, parallel file system supporting many requirements of leadership class HPC simulation environments. It is an object-based file system composed of three components: Metadata Servers (MDSs), Object Storage Servers (OSSs), and clients. Lustre clients are installed in the compute nodes or I/O nodes of an HPC system, which are connected with MDSs and OSSs via high speed connecting networks such as InfiniBand [3].

Each MDS manages one or multiple Metadata Targets (MDTs), which store the metadata information such as file name, path, permissions etc. The OSSs provide file I/O services and manage Object Storage Targets (OSTs), where the application data are stored. The OSTs are like multiple disks connecting to OSSs. Users can decide how many OSTs they want to stripe their files over. This approach enables concurrent accesses to multiple OSTs and eventually accelerates the I/O requests. Besides the number of OSTs, users can also set the stripe size, which indicates how many bytes can be stored in one OST stripe before moving to the next OST or next stripe. Different settings of these two factors lead to a huge difference of I/O performance. One target of this study is to find out the optimal combination for each I/O request and to set them at run-time [3].

### C. Performance Factors

There are many performance factors of I/O stack involved in the I/O efficiency. By researching the application characteristics, Lustre file system and the MPI-IO ROMIO library, it can be seen that the following parameters can affect I/O performance significantly:

- *number of cores*: Concurrency has a significant impact on the I/O rate.
- *problem size*: Optimal configurations change depending different problem sizes.
- *MPI-IO subroutine*: Tuning of different MPI-IO subroutines (collective vs. individual) are different.
- *romio_cb_read*: The collective buffering optimization for reading operations can be enabled or disabled.
- *romio_cb_write*: The collective buffering optimization for writing operations can be enabled or disabled.
- *striping_factor*: It specifies the number of Lustre OSTs (stripe_count) to stripe new files.
- *striping_unit*: It specifies the size (in bytes) of each Lustre file system OST stripe unit (stripe_size) used for new files.
- *cb_nodes*: It specifies the number of target nodes to be used for collective buffering.

Search scope of the parameters that we worked on in this study is given in Table I. Our auto-tuning system can help users and developers to explore how these parameters interact with each other. It can traverse parameter space to determine a good parameter set which can achieve high I/O performance.

TABLE I: Training Set Configurations' Scope

| Name | Value |
|---|---|
| number of cores | 64 - 1200 |
| number of bytes | 256 KB - 64 MB |
| number of aggregators | 1 - 16 |
| striping count | 1 - 16 |
| striping unit | 1 MB - 16MB |
| collective I/O | automatic; disable; enable |
| I/O pattern | collective, individual |

### D. IO_Optimizer: Configuration Search

Heuristic search algorithms such as genetic evolution algorithms and simulated annealing can traverse a search space in a reasonable amount of time [11]. Selecting less combinations of parameters and running a small number of tests are reasonable to search a huge and complex parameter space. We developed a GA based model *IO_Search* in *IO_Optimizer* module to search I/O parameter space.

*IO_Search* randomly selects individuals of initial parameter set. Then, it modifies the combination of parameters for further testing based on the I/O performance. Over consecutive generations, the population can approach an optimal parameter set which gives better I/O performance than the default performance of the default settings.

A GA is a randomized search algorithm that mimics the process of natural evolution by modifying a population of individual solutions [14]. It randomly selects individuals of initial population from the current population as parents. Then, it uses these individuals to generate the children for the next generation. Through iterative generations, the bad individuals are eliminated, the good individuals are saved. Good children are produced by good parents.

A set of operators such as reproduction, crossover, and mutation is used to adjust the initial population to generate successive populations with time [15]. Reproduction is a process based on the objective function (fitness function) of each individual to determine how "good" the individual is [15]. Thus, individuals with higher fitness value can contribute to the next generation. Crossover is a process in which members of the last population are mated at random in the mating pool [15]. Mutation is the random change portions of the individual with small probability [15]. Random mutations provide a sampling of the remainder of the space [11]. A GA is expected to converge to an optimal or near-optimal solution [11].

Figure 3 shows the workflow of our GA model *IO_Search*. The *IO_Search* starts the GA for a given concurrency and problem size with the I/O benchmark or application. A predefined parameter space is given to the *IO_Search* as an input that includes all possible values of tunable parameters. At first, the *IO_Search* generates an initial population of I/O parameters and creates a configuration file containing the selected parameters to be used by the *IO_Tuner* to auto-tune the I/O application or benchmark.

In our experiments, population size is selected 10 by the *IO_Search*, it can be configured. The fitness value of an individual is defined as the I/O bandwidth. As the *IO_Search* passes through a new generation, it calculates the fitness of individuals; namely I/O bandwidth. The best individuals named as the "elite members" which give high I/O performance are transferred to the next generation. The rest of the population in the next generation is generated by applying crossovers and mutations on the current population. These steps are repeated for each generation until stop criteria is reached. The *IO_Search* defines the number of generations as 30 so that maximum of 300 runs of the given I/O application or benchmark can be executed by the *IO_Search*.

Mutation rate is defined as 15% in the *IO_Search* that means mutation is applied on 15% of the current population for each generation. Finally, the best-performing I/O parameters for the given application and scale are stored in the configuration file.

### E. IO_Tuner: Setting I/O Parameters

*IO_Tuner* is our tuning module which takes the best-performing I/O parameters found by the *IO_Search* included in the *IO_Optimizer* and then dynamically set these parameters at different layers of the I/O stack. In this study, the *IO_Tuner* works on MPI-IO ROMIO library and Lustre parallel file system parameters.

At first, the *IO_Tuner* reads the configuration file including optimal configurations generated by the *IO_Search*. As soon as I/O applications or benchmarks call MPI-IO subroutines, *IO_Tuner* is triggered to apply these optimal configurations before executing the I/O operation transparently. It passes the

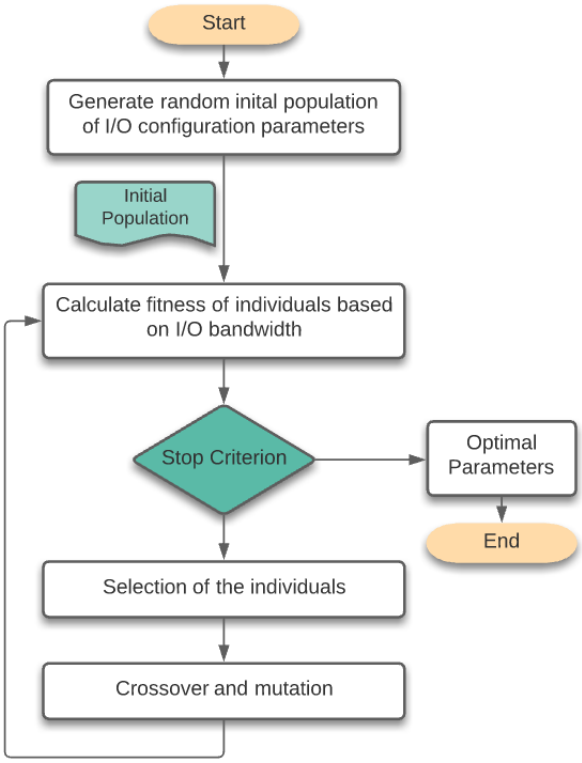



Fig. 3: Overall Architecture of the IO_Search

intercepted MPI-IO functions of the application or benchmark in the PMPI wrapper. At this stage, the optimal parameters are set and then the original MPI-IO function is called. In this way, auto-tuning can be done transparently to the users without source code modification.

The previously found optimal configurations could be out of date and no longer optimal. Sometimes users have achieved a better I/O performance with some ”brand new” configurations. After executing I/O operations, performance results can be used to refit *IO_Search* with the dynamic conditions of a parallel I/O system adaptively for scientists and engineers to find out the latest optimal configurations.

## IV. Experimental Results

In this section, we present experimental results of our proposed auto-tuning approach, I/O benchmarks used in this study, supercomputer on which the experiments were conducted.

### A. Benchmarks

We chose two I/O benchmarks to evaluate our approach: Interleaved or Random (IOR) and MPI-Tile-IO. These represent different I/O write motifs with different data sizes.

- IOR—I/O benchmark [16]: IOR (LLNL 2015) is an I/O benchmark developed at Lawrence Livermore National Laboratory (LLNL). It is one of the main HPC I/O benchmarks because it is highly configurable and contains different I/O interfaces. We have configured IOR in the range from 32 MB to 64 MB block sizes to write in the shared files collectively.
- MPI-Tile-IO [17]: The MPI-Tile-IO benchmark tests the IO-performance in a real world scenario. It test how it performs when its challenged with a dense 2D data layout. We have configured MPI-Tile-IO number of tiles as much as number of cores, and in the range from 8096 KB to 16184 KB element sizes.

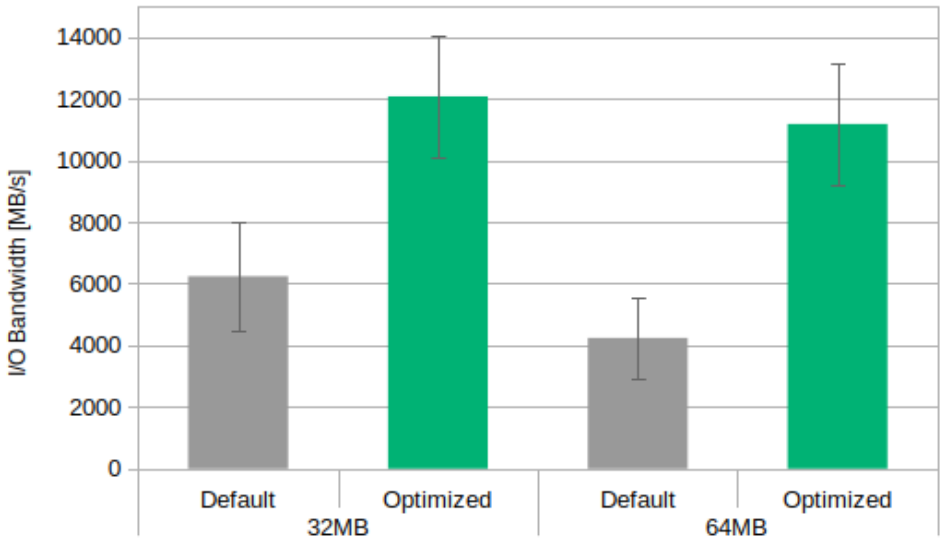
### B. System setup

The experiments were conducted on the NEC Cluster platform (Vulcan) at HLRS (High Performance Computing Center Stuttgart). Vulcan consists of several front end nodes for interactive access and several compute nodes of different types for execution of parallel programs. It has 761 nodes with total 24 cores, Centos 7 operating system, PBSPro Batchsystem, Infiniband + GigE node-node interconnect, 500 TB (Lustre) for vulcan global disk [18], the bandwidth is about 3 GB/sec. The system consists of 54 OST storage targets, each of them one RAID6 lun, 8+PQ, 2 TB disks.

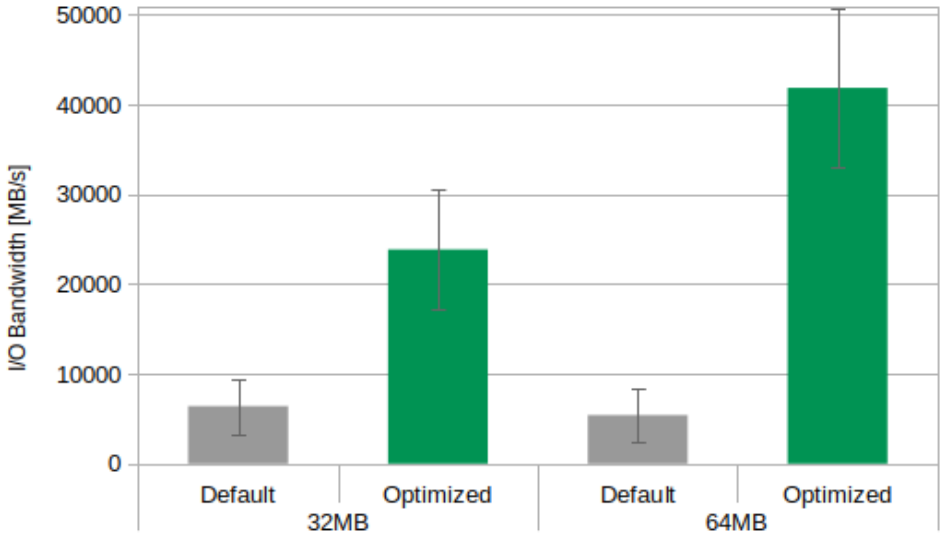
The default setup of Lustre striping configuration on the experimental file system is striping_factor=4 and striping_unit=1048576. OpenMPI version is from version 4.0.3.

### C. Experimental Results

We conducted our experiments for two benchmarks in different file sizes on Vulcan. In this section, we show the results of our approach. The experiments have been repeated multiple times both default and tuned configurations, and




(a) 240 cores




(b) 1200 cores

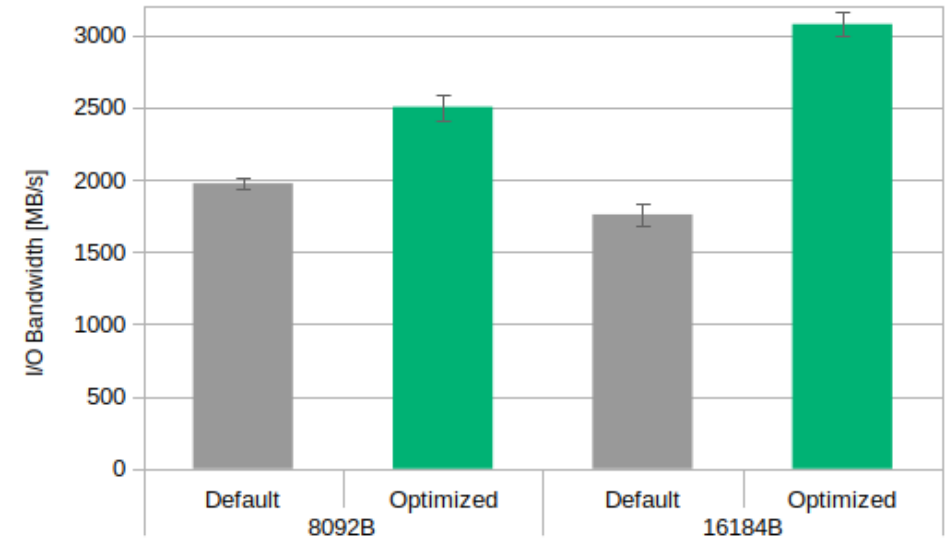
Fig. 4: Default vs. optimized write bandwidth on IOR for various transfer sizes running on 240 cores and 1200 cores of Vulcan. Y-axis represents I/O bandwidth in MBps and x-axis represents transfer sizes (in MB). The scales of the I/O bandwidth axes are different in the plots.

average values have been plotted. The default experiments are measured by applying the system default settings that are mostly defined by system administrators for HPC platform users.

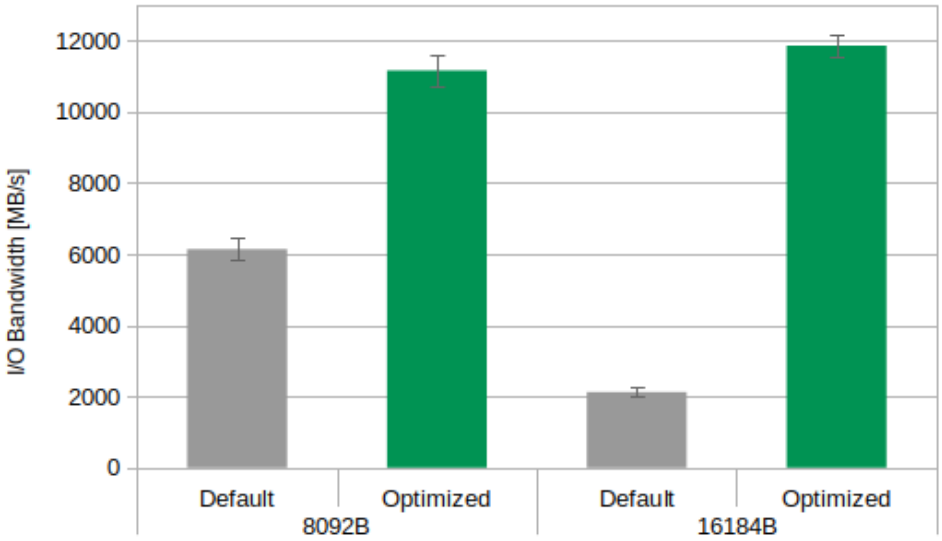
Summary of performance improvements we have obtained by using the tuned parameters that our auto-tuning system detected for IOR benchmark are summarized in Figure 4 on Vulcan at 240 and 1200 cores concurrencies and for MPI-Tile-IO benchmark in Figure 5 on Vulcan at 64 and 256 cores concurrencies. Y-axis represents I/O bandwidth in megabytes per second and x-axis represents data sizes. The scales of the I/O bandwidth axes are different in the plots. Note that only a subset of the combinations were run due to limited access to the platform. For all experimental tests, the I/O bandwidth is calculated as the ratio of the amount of data to be written into a file to the time taken for writing the data. In the measured I/O time, opening, writing, and closing the file overhead is included.

*IO_Search* ran for $\sim 2.5$ hours and $\sim 4$ hours for the two concurrencies in IOR, $\sim 1.0$ hours and $\sim 1.5$ hours for the two concurrencies in MPI-Tile-IO to search through the parameter space. In most cases, GA passed through 10 to 30 generations.

Using our auto-tuning approach, we achieve an increase in I/O bandwidth of up to 7.74×over the default parameters for IOR benchmark and 5.59×over the default parameters bandwidth for MPI-Tile-IO benchmark as shown in Figure 4




(a) 64 cores




(b) 256 cores

Fig. 5: Default vs. optimized write bandwidth on MPI-Tile-IO for various transfer sizes running on 64 cores and 256 cores of Vulcan. Y-axis represents I/O bandwidth in MBps and x-axis represents element sizes of core times core number of tiles (in KB). The scales of the I/O bandwidth axes are different in the plots.

and Figure 5.

Table 2 shows the I/O performances of the default and the optimized experiments for two use-cases are shown in Figure 4 and Figure 5. We also show the speedup that the auto-tuned settings achieved over the default settings for each experiment.

## V. Conclusion and Future Work

In this study, we presented a GA based I/O auto-tuning approach to tune the parallel I/O stack parameters. It achieves improvements for write bandwidths in main HPC I/O benchmarks on supercomputer Vulcan. Thereby, the overhead to find the best parameters is drastically reduced from long execution times (application-dependent) for a naïve strategy to few hours(data-dependent). It is able to improve write performance 7.74×over the default parameters for IOR benchmark. Moreover, our approach achieves an average bandwidth improvement of 5.59×for non-contiguous writes in MPI-Tile-IO. For any benchmark or I/O application, our model can be applied with negligible effort.

This approach can be understood by users with little knowledge of parallel I/O without any post-processing step. It is implemented upon the MPI-IO library to be compatible with MPI based engineering applications, and be portable to different HPC platforms as well. The parameters discussed in this paper are system dependent, but new parameters can be easily integrated to the configuration files. Future efforts will further

TABLE II: I/O Speedups of Applications with Optimized Parameters over Default Parameters

| Application | | IOR (MB/s) | | MPI-Tile-IO (MB/s) | |
|---|---|---|---|---|---|
| #Cores | | 240 | 1200 | 64 | 256 |
| Use case 1 | Default | 6238.56 | 6402.08 | 1974.462 | 6138.917 |
| | Tuned | 12075.01 | 23859.15 | 2503.22 | 11257.42 |
| | Speedup | 1.93 | 3.73 | 1.27 | 1.83 |
| Use case 2 | Default | 4238.57 | 5412.91 | 1759.326 | 2122.027 |
| | Tuned | 11186.23 | 41859.44 | 3075.17 | 11859.23 |
| | Speedup | 2.64 | 7.74 | 1.75 | 5.59 |

explore more accurate representations of the configuration parameters and statistical methods. As future work, the auto-tuning solution will be tested on engineering applications in different professional areas to show the usability.

## ACKNOWLEDGMENT

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the POP project, grant agreement No. 824080.

## REFERENCES

[1] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of petascale I/O workloads", in 2009 IEEE International Conference on Cluster Computing and Workshops, 2009.

[2] H. Luu, B. Behzad, R. Aydt and M. Winslett, "A multi-level approach for understanding I/O activity in HPC applications," 2013 IEEE International Conference on Cluster Computing (CLUSTER), Indianapolis, IN, 2013, pp. 1-5.

[3] X. Wang, (2017). A light weighted semi-automatically I/O-tuning solution for engineering applications (Doctoral dissertation). Retrieved from OPUS - Publication Server of the University of Stuttgart, http://dx.doi.org/10.18419/opus-9763. 1145/2834976.2834977.

[4] M. Agarwal, D. Singhvi, P. Malakar and S. Byna, "Active Learning-based Automatic Tuning and Prediction of Parallel I/O Performance", 2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW), Denver, CO, USA, 2019, pp. 20-29.

[5] F. Isaila, P. Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross,and P. Hovland, "Collective I/O tuning using analytical and machine learning models", in International Conference on Cluster Computing. IEEE, 2015, pp. 128–137.

[6] A. Knupfer, H.Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M.S. Muller, W.E. Nagel, "The vampir performance analysis toolset", In: M. Resch, R. Keller, V. Himmler, B. Krammer, A. Schulz, (eds.) Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools,pp. 139–155. Springer, Heidelberg, 2008.

[7] Argonne National Laboratory: Darshan. http://www.mcs.anl.gov/project/darshan-hpc-io-characterization-tool

[8] J. Kunkel, M. Zimmer, E. Betke, "Predicting Performance of Non-contiguous I/O with Machine Learning", in: Kunkel J., Ludwig T. (eds) High Performance Computing. ISC High Performance 2015. Lecture Notes in Computer Science, vol 9137. Springer, Cham.

[9] J. M. Kunkel, M. Zimmer, N. Hübbe, A. Aguilera, H. Mickler, X. Wang, A. Chut, T. Bönisch, J. Lüttgau, R. Michel, and J. Weging, "The SIOX Architecture — Coupling Automatic Monitoring and Optimization of Parallel I/O", in Proceedings of the 29th International Conference on Supercomputing - Volume 8488, ser. ISC 2014, Leipzig, Germany: Springer-Verlag New York, Inc., 2014, pp. 245–260, ISBN: 978-3-319-07517-4. DOI: 10.1007/978-3-319-07518-1_16. [Online]. Available: http://dx.doi.org/10.1007/978-3-319- 07518-1_16.

[10] B. Behzad, S. Byna, Prabhat, and M. Snir, "Pattern-driven Parallel I/O Tuning", in Proceedings of the 10th Parallel Data Storage Workshop, ser. PDSW '15, Austin, Texas: ACM, 2015, pp. 43–48, ISBN: 978-1-4503-4008-3. DOI: 10.1145/2834976 . 2834977. [Online]. Available: http : / / doi . acm . org / 10 .

[11] B. Behzad, S. Byna, Prabhat, and M. Snir: Optimizing I/O Performance of HPC Applications with Autotuning. ACM Trans. Parallel Comput. 5, 4, Article 15, 27 pages, (2019).

[12] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, R. Aydt, Q. Koziol,M. Sniret al., "Taming parallel I/O complexity with auto-tuning", in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. ACM, 2013, p. 68.

[13] S., Madireddy, "Machine Learning Based Parallel I/O Predictive Modeling: A Case Study on Lustre File Systems", in: ISC'18: International Conference on High Performance Computing, 2018.

[14] Son H., Lee G., Kang K., Kang Y.-J., Youn B.D., Lee I., Noh Y. "Industrial issues and solutions to statistical model improvement: a case study of an automobile steering column", Structural and Multidisciplinary Optimization, Volume 61, 2020. G. Saeed, 16 - Structural Optimization for Frequency Constraints, Editor(s): Amir Hossein Gandomi, Xin-She Yang, Siamak Talatahari, Amir Hossein Alavi, Metaheuristic Applications in Structures and Infrastructures, Elsevier, 2013, Pages 389-417.

[15] W. Roetzel, X. Luo, D. Chen, Chapter 6 - Optimal design of heat exchanger networks, Editor(s): Wilfried Roetzel, Xing Luo, Dezhen Chen, Design and Operation of Heat Exchangers and their Networks, Academic Press, 2020, Pages 231-317.

[16] Parallel file system I/O benchmark, http://github.com/LLNL/ior. Last accessed 20 Feb 2021.

[17] MPI-Tile-IO benchmark, https://www.mcs.anl.gov/research/projects/pio-benchmark/. Last accessed 20 Feb 2021.

[18] The NEC Cluster (Vulcan), https://kb.hlrs.de/platforms/index.php/Vulcan. Last accessed 20 Feb 2021.