Towards Smarter Schedulers: Molding Jobs into the Right Shape via Monitoring and Modeling

Jean-Baptiste Besnard[1](✉), Ahmad Tarraf[2], Clément Barthélemy[3], Alberto Cascajo[4], Emmanuel Jeannot[3], Sameer Shende[1], and Felix Wolf[2]

[1] ParaTools SAS, Bruyères-le-Châtel, France
jbbesnard@paratools.fr

[2] Department of Computer Science, Technical University of Darmstadt, Darmstadt, Germany

[3] INRIA Bordeaux, Bordeaux, France

[4] Computer Science Department, University Carlos III of Madrid, Madrid, Spain

Abstract. High-performance computing is not only a race towards the fastest supercomputers but also the science of using such massive machines productively to acquire valuable results – outlining the importance of performance modelling and optimization. However, it appears that more than punctual optimization is required for current architectures, with users having to choose between multiple intertwined parallelism possibilities, dedicated accelerators, and I/O solutions. Witnessing this challenging context, our paper establishes an automatic feedback loop between how applications run and how they are launched, with a specific focus on I/O. One goal is to optimize how applications are launched through moldability (launch-time malleability). As a first step in this direction, we propose a new, always-on measurement infrastructure based on state-of-the-art cloud technologies adapted for HPC. In this paper, we present the measurement infrastructure and associated design choices. Moreover, we leverage an existing performance modelling tool to generate I/O performance models. We outline sample modelling capabilities, as derived from our measurement chain showing the critical importance of the measurement in future HPC systems, especially concerning resource configurations. Thanks to this precise performance model infrastructure, we can improve moldability and malleability on HPC systems.

Keywords: Monitoring · Performance Modeling · MPI · IO · Malleability

1 Introduction

A few decades ago, programmers did not have to change their code to gain efficiency, as the benefits of the sequential Moore's law were not depleted, and it was possible to run the same code faster without any effort. However, as outlined in the well-known quotation "free lunch is over", power dissipation constraints have forced us to rely on multiple cores per socket and thus transitively led to compulsory shared-memory parallelism. With applications running in MPI+X (with X typically being OpenMP) to address inter- and intra-node parallelism, the execution space already grew substantially. Indeed, if MPI is suitable for memory locality as it works in distributed address space by nature, it is not the case for OpenMP which then required careful handling at a time OpenMP places were not devised. As a consequence, one of the practical ways was to run one MPI process per NUMA node and then rely on shared-memory parallelism inside each memory region. If we now continue our journey towards current hardware, the pressure on energy consumption has advocated for simpler (or specialized) cores, leading to the democratization of GPUs or, more commonly, accelerators. It means that from now on, a non-negligible part of the computation has to run on specialized hardware, which leads to data movements inside the node, from and to the accelerator. And again, these data movements are typically associated with at least affinity preferences (NUIOA) [15] or capacity constraints (stacked HBM, memory-mapped GPU memory, split address space, etc.). We are now at a point where the layering of runtime configurations and constraints are difficult to untangle – generally leading to inefficient use of parallel systems simply because of the difficulty of exploring the execution space.

Starting from this convoluted scenario, we can add a layer of complexity now unfolding in the HPC field: the horizontalization of computing [13,32]. Indeed, HPC software tends to be relatively monolithic, often due to the bulk-synchronous bias of MPI. It means features end up stacked in the same binary as shared libraries and the program alternates between the various functions over time. However, due to complexity constraints, this is probably about to evolve towards a specialization of the software components [35], if not simply as a mirror of the underlying hardware specialization. As a result, the program is likely to become more composite, creating several pieces communicating in workflows [1] or collocated in situ during the run. Similarly to what has been advocated around power constraints in PowerStack [2,6,32], we consider this process cannot be manual and should be generalized. The programmer should define the workflow (i.e., dependencies), and computational needs (memory, device) but addressing how to run shall be automated.

2 From Ad-Hoc to Always-On Monitoring

Understanding and examining an application's performance is crucial not only for efficient system utilization but also for identifying performance bottlenecks at an early stage, including during the execution. Given the complexity of these systems, performance limitations at any level can significantly impact the application's performance. E.g., Parallel file systems (PFSs) like Lustre and GPFS, form the storage backbone of HPC clusters and have been developed for over two decades. As they have been extensively optimized to support traditional compute-bound HPC applications, which sequentially read and write large data files [7], PFS cannot handle all types of workloads effectively (e.g., Deep learning workloads). This aspect can have such an impact that the I/O performance for identical workloads can differ by more than 200 times depending on the time when the workloads are executed [23]. In general, performance variability, which is the difference between execution times across repeated runs in the same execution environment, is far from being eliminated and will remain an active research area as several studies have suggested [26,27].

Still, monitoring and modeling the performance of an application is an aspect that future systems require for optimal throughput. In light of heterogeneous computing and the current trends toward using programming models focusing on job malleability, characterizing an application's performance is crucial for application developers, as well as for system optimization. Considering malleability, e.g., a balance between the compute and I/O resources is required to utilize the different underlying components of a system effectively. Without detailed knowledge in both aspects, resource management of malleable jobs would be closer to random guessing. Once the monitoring data is available, performance models can be systematically generated to drive scheduling.

In this context, we present a monitoring infrastructure developed in the ADMIRE project, with a particular focus on the trade-offs the infrastructure relies on to provide always-on low overhead measurement capabilities. In the second part, we generate performance models with *Extra-P* [5], which exceed the computational aspect and consider also I/O. Then, we apply these monitoring and modeling capabilities to actual applications with the goal of defining dynamically what is the best configuration to launch them, either for efficiency or finding the best I/O configuration. Finally, we discuss how this work sets the basis for systems only featuring self-configured jobs and approaches like malleability.

3 On Performance and Execution Spaces

The execution space for applications can be complex due to current hardware. Indeed, what used to be mono-variadic (i.e., only the number of MPI processes) is now becoming a much more intricate space [3]. With compulsory shared-memory parallelism and accelerators, programs have to obey several constraints to run in their optimal configuration. From the user perspective, supercomputers are means for producing results, and therefore, end-users often optimize for *time-to-result*. This leads to running at the largest scale possible to minimize this *time-to-result*. However, looking closely at existing parallel execution descriptors, such a heuristic doesn't necessarily lead to efficient execution.

Overall, there are two main ways of running a parallel program on an HPC system. On one hand, one wants to accelerate computing, leading to **strong scaling** "If I double the dedicated resources I want my execution time to be divided by two". Strong scaling is limited by the sequential part of the program and facilitated by larger problem sizes. On the other hand, **weak scaling** "If I double both the problem size and resources, I want to run for the same time";

leads to payloads where the problem size increases with the number of cores [4]. Understanding how a program behaves is thus correlated with the ability to track down multiple executions of the same program, and more precisely the same test case in several configurations. Indeed, in current HPC machines, a program is bound to run in a non-linear space; The same problem can be solved on MPI, MPI+OpenMP, and even with accelerators such as GPUs. This leads to a combinatorial space where previously mentioned scalability rules are still valid. Exploring this space empirically is now impractical and therefore automated measurement and modeling capabilities are needed to obtain the correct configuration for its target problem – we foresee the process in the context of *smart schedulers*.

4 The Need for Smarter Schedulers

As mentioned, the job execution space is getting increasingly larger [3]. In addition, programs are getting more horizontal, running either as workflows or coupled computation (in-situ [13] or services oriented [35]), which increases further the combinatorial aspect. Consequently, we are convinced that manually launching programs is not realistic anymore as the target space is too large to be empirically explored. As programming models are facing difficulties hiding hardware complexity, requiring changes in programs (i.e. hybridization), the same is true for launch configurations which now suffer from this lack of abstraction.

As outlined in the second part of the paper, thanks to a monitoring infrastructure, it should be possible to build models of running applications to drive launch configurations over time. This major shift in how programs are launched naturally requires a change in habits – efficient resource usage not being an option anymore. It is important to note that our approach is not exclusive to our particular implementation and that by design it has to fit in a larger shared effort between standards and runtimes [19].

So, taking all of this into account, a smarter scheduler would enable a large set of optimizations [29] over several system components, including:

- **improved backfilling:** by leveraging model projection, jobs can be molded to improve their chance of being backfilled, improving platform utilization.
- **automated job configuration:** choose the configuration for jobs too maximize efficiency. This is elaborated in Sect. 7.1.
- **reconfiguration:** if a job can run on both CPU and GPU, the scheduler may choose depending on availability while being aware of the efficiency difference.
- **job horizontalization:** as the service island in machines probably get smaller, the scheduler could be the pivot to *service* side components such as I/O back-end [8] on demand. This is a core aspect in the ADMIRE project, which features ad-hoc filesystems [35].

After, contextualizing our approach with related work, the following sections illustrate a possible implementation of such a smarter scheduler. After introducing the monitoring and modeling architecture (Sect. 6), we show how it can be applied to two use cases (Sect. 7).

5 Related Work

The concept of running parallel programs in the right configuration is not new, and several attempts have been made to provide such a feature, commonly known as *auto-tuning* [3,16,17,37], which involves exploring the execution space of a program to find optimal configurations. Our contribution follows the same approach but with a unique focus on systematizing it at the scheduler level. This systematization introduces specific monitoring requirements, such as always-on monitoring and data management challenges. A similar effort to our ADMIRE project is PowerStack [38], which focuses on power optimization rather than I/O. In the PowerStack framework, similar needs for measurement and performance models are identified, with the ultimate goal of optimizing overall resource usage. However, our approach differs in its generalization to all launch parameters including I/O backend, combining moldability and at-term malleability to leverage performance models. We believe that moldability gains have not been depleted yet by current approaches, despite being simpler to implement compared to malleability. In terms of the monitoring approach, a closely related contribution is the Data-Center Data-Base (DCDB) [25], which shares many aspects with our model. In our approach, as we will further outline, Prometheus manages this role and aggregation is done directly on the node, including from the target application, Overall, our methodology is similar in that it systematically manages application metrics by design.

In HPC systems, the resources are managed by the Resource Management System (RMS), which is responsible for multiplexing resources between multiple users and jobs. This component becomes more crucial in dynamic environments running moldable and malleable applications. Typical RMSs can configure the resources for a job before the execution, whereas dynamic applications require an RMS capable of reshaping the resource allocations at runtime [10,34]. Several efforts are looking into this, including the PMIX standard [19] but such support for malleability is not mainstream. Authors in [20] proposed CooRMv2, an RMS that is able to give more/fewer resources to the executing jobs based on their requests. This approach relies on pre-allocated resources estimated from peak usage, which can result from underutilization. D'Amico et Al. [11] proposed a new job scheduling policy for malleable applications to increase the response of the jobs. This approach differs by using shared computing nodes for all possible jobs, instead of exclusive node allocations. While this reduces job response time, interference between jobs in the same node could affect the results (CPU, memory, I/O, etc.). Developing smart schedulers for dynamic environments in HPC can yield many benefits for all (researchers and developers). The executions could be more efficient, reduce job completion times, and improve the global system performance.

6 ADMIRE Monitoring Infrastructure

Our proposal aims to measure how applications run to fine-tune them before or during their execution. With a particular focus on I/O, the project features several ad-hoc file systems [14,22,35], each with its specificities. By building this feedback loop between the expression of a job and its parameters, the project aims at defining a new way of using parallel machines due to a generalized auto-tuning approach crossing all layers of the parallel machine. These measurements can then be leveraged to reconfigure the run when it starts (i.e. moldability), either choosing the right scale to run or the right ad-hoc file system. This can also be done at runtime (i.e. malleability). However, we focus on moldability in this paper since it is already benefiting from this infrastructure, which fulfils the malleability constraints. Among these constraints, we have first the ability to model all runs, enforcing *always-on* measurements. Transitively, the measure has to be *low-overhead* not to impact performance. A second aspect related to programmability is that it should be non-intrusive by default as malleability supposes a holistic view of the system. Consequently, the ADMIRE monitoring infrastructure focused on the interception of the parallel programming interfaces, voluntarily leaving the applicative side apart. We developed instrumentation layers for MPI, and for the ad-hoc file systems part of the project. Similarly, to capture the I/O syscalls, we have modified *strace* to attach and detach from running programs to provide on-demand data. The goal in maintaining this data variety in the system is to provide a large set of information to feed potential models guiding the dynamicity decisions. To do so, the time series for each node and profiles per job are generated by a dedicated component relying on the *TAU metric proxy*.

6.1 TAU Metric Proxy

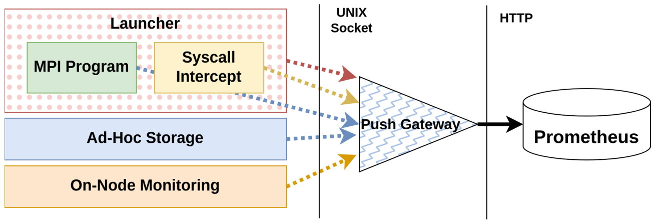


Fig. 1. Interconnection with the aggregating push gateway to implement on-node always on counter tracking.

As time-based metrics are critical when making dynamic decisions (i.e., as such decisions have to be done over time), we decided early in the project to store them in a dedicated Time-Series Database (TSDB). One of the most prominent ones is Prometheus, which has a significant record of usage in production while remaining simple and heavily interfaceable thanks to its HTTP-based interface. Prometheus *pulls* data from a variety of *exporters*, which are practically HTTP servers embedded in the various components of interest. The database then regularly accumulates values over time to store as time series. While this scheme is excellent for composability, it faces challenges in the HPC context as it is not affordable to open an HTTP listening socket in each MPI process and to dynamically track all running processes. To solve this problem, we had to design a *push gateway* which allows components to push data inside Prometheus. Note that there were existing push gateways but none of them was designed for HPC as they rely on HTTP. As shown in Fig. 1, the *TAU metric proxy* is an aggregating push gateway based on UNIX sockets, capable of handling hundreds of local clients. The client-side library is a single C file that allows forwarding the counters in an opt-in manner. The performance counters are stored in memory at the client level, and polling threads send them periodically over the UNIX socket to the push gateway for decoupling. When reaching the server, two arrays accumulate the values, one for the given job and another for the whole node. In addition, a global summative system view at high frequency is provided by a Tree-Based overlay Network (TBON) thanks to the LIMITLESS [9] monitoring infrastructure, which is coupled to the proxy. This architecture allows several distinct components to contribute to node-level counters in a scalable manner, in Fig. 1, we have the application, its launcher, the strace wrapper, the ad-hoc file system and node-level monitoring pushing data concurrently. Conjointly, the Prometheus time-series database polls the metric-proxy HTTP server which holds the aggregated state for the node, inserting new points and time series inside the database for data persistence. At a given moment, the metric proxy only holds a single value for all the counters and exposes them dynamically when receiving an HTTP request using the OpenMetric Format. In addition, the measurement chain can only handle counters due to the summative nature of the measurements, a common design approach when creating Prometheus collectors as storage relies on delta encoding. In addition, PromQL, the language allowing arbitrarily complex queries to the performance database includes means of computing derivative (`rate` function) to infer dynamic behaviours over time.

To complement this node-level view with a per-job one, when all participants of a given job have disconnected, the per-job array is released and stored in the file system according to the *Slurm* job-id. We run a metric proxy on each node, meaning that we maintain performance counters on a per-node basis. Per-job profiles are also generated on a per-node basis and lazily aggregated from the file system to create a single summed-up profile. Such profiles are generated for all jobs and stored in a dedicated *profile storage directory*. Conjointly, we designed a Python library to read and compare values from these profiles. Besides, each profile contains meta-data describing the associated job, including time span, allocation parameters, and spanning nodes.

6.2 Performance Modeling

Performance modeling has a long research history [5,17,28,31,36]. These models were usually used to generate scalability models that show how the runtime scales in accordance with one or more execution parameters, one of them often being the number of processors. Extra-P, for example, is an automatic performance-modeling tool that generates empirical performance models. A performance model is a mathematical formula that expresses a performance metric of interest (e.g., execution time or energy consumption) as a function of one or more execution parameters (e.g., size of the input problem or the number of processors). The tool has a long research history, with recent updates adding noise-resilient empirical performance modeling capabilities to use cases such as Deep Neural Networks [31] or statistical meaningfulness [30].

To generate performance models, Extra-P requires repeated performance measurements. It is suggested that at least five measuring points per parameter should be performed. By profiling an application, the required data for model generation can be collected. Moreover, by continuously passing data from the Prometheus database, it is possible to continuously improve the models. So far, Extra-P has been used to model the call path of the application, focusing on computational and communication aspects but excluding I/O. Though recent terms, such as the storage wall [18], try to quantify the I/O performance bottleneck from the application scalability perspective. Indeed, due to the fact that I/O subsystems have not kept up with the rapid enhancement of the remaining resource on an HPC Cluster, I/O bottlenecks are often encountered due to various aspects (I/O contention, hardware restrictions, etc.). Thus, there is a need to analyze the scalability behaviour in regard to I/O as well. Thus, instead of developing a new tool, we used Extra-P to generate performance models for various I/O metrics. This is done by providing the I/O data in an Extra-P compatible format. Moreover, when using the JSON Lines format, the continuously collected I/O monitoring data can be appended to such a file, allowing us to refine the performance models whenever more data is available. This becomes especially interesting if several I/O metrics are captured alongside significant computational metrics. By generating several performance models, we can judge the computational I/O intensity of an application regarding the number of processors. Moreover, if we model, e.g. the write bandwidth over the number of nodes or MPI ranks, we can depict which roofline in Fig. 2a is encountered and can hence decide on using burst buffers to counter such a scenario.

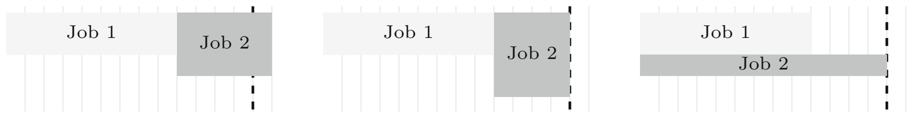
In the context of a *smart scheduler*, Extra-P is used to generate either offline or online models corresponding to all executions on a system. These models are then leveraged to guide decisions with respect to optimization criteria as mentioned in Sect. 4.

7 Use-Cases

This section shows two examples of heuristics driving job configuration. We start with an auto-tuning job configuration to maintain running jobs within a given time frame. As a second example, we show how job requirements in terms of I/O can be extracted from relatively compact metrics linked to bandwidth and I/O operations per second (IOPS).

7.1 Deadline Scheduling of Moldable Jobs

In this section we focus on moldability. However, this approach can be extended for malleability, i.e. changing configuration at runtime. The reason we consider this use case as relevant is that we are convinced that the moldability gains have



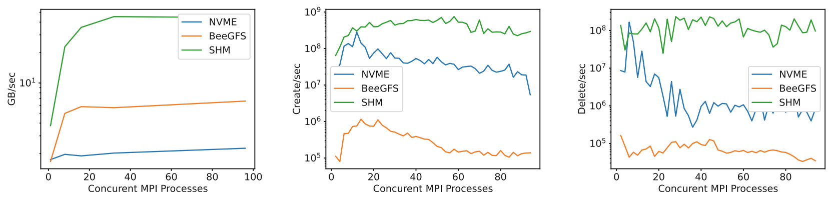
(a) Job 2 takes too long (b) Use *more* resources (c) Use *less* resources

Fig. 2. Molding job 2 to fit a deadline. Job 1 is fixed. There are two possibilities: either use more resources to accelerate job 2, i.e. scale up; or use fewer resources to take advantage of the scheduler backfilling policy. Solution (c) is better because it reaches the deadline using less resources, which is more efficient.

not been depleted. To validate and implement our moldability goals, we base our first implementation of a *smart scheduler* on a simple shell script wrapping the Slurm command line. More precisely, let us assume that the user wants to run a job with a given *deadline* (e.g. the job needs to be done by Monday morning or the end of the night). The tool will (1) extract this deadline parameter, (2) compute the number of cores required to reach the deadline given the Extra-P scaling model and (3) launch the job with this configuration. The monitoring infrastructure will then generate a profile of this run that will, in turn, be used to refine the scaling model and improve prediction for subsequent runs. As of now, this solution is incomplete because it does not take into account the time spent in the Slurm scheduler queue. Ideally, the tool would balance the expected performance of the application with the wait time incurred when asking for a larger set of resources, allowing it to finish on time, but not earlier, improving the job efficiency as described in Fig. 2. Some schedulers such as Slurm and OAR provide interfaces to query the expected wait time of an application in the queue, but can only rely on estimates provided by application users, which are usually inaccurate [24]. Several approaches have been proposed to obtain better estimates of the queuing time, using e.g. statistical analysis [21] or based on simulating the scheduler behavior [33]. Finally, note that this approach is not restricted to deadline scheduling, any launch parameter could be extracted and redefined before the run.

7.2 Characterising I/O Applicative Requirements

Typically, the file system (FS) is a shared resource in HPC, which makes it subject to contention. To outline the potential performance effect of I/O, we have implemented a dedicated I/O benchmark to measure peak performance in terms of bandwidth and I/O operations per second (IOPS). As shown in Fig. 3, FSs can have very different responses in the function of both their nature and of the contention level. Measurements were run on the same two bi-socket nodes connected in Infiniband (100 Gb/s ConnectX-4) for the three FSs. As mentioned in Sect. 6.2, characterizing the scalability behavior of an application in terms of I/O can bring various advantages, especially for choosing the appropriate configuration that uses the different resources on an HPC cluster.



(a) Write bandwidth (b) Rate of create IOPS (c) Rate of delete IOPS

Fig. 3. Experimental FS peak performance in function of MPI processes.

Generally speaking, I/O subsystems present a similar performance behaviour driven by two main parameters, bandwidth, and IOPS. The Bandwidth increases with the number of nodes, as does the bisection bandwidth between compute and storage (more network links), up to a point where the back-end storage capabilities are saturated, leading to peak bandwidth. As far as IOPS are concerned, we observe such an increase, a plateau, and often contention, as the POSIX coherency requirements do imply a form of locking on meta-data operations. As we further develop, we rely on a derivation of these two rooflines [12,36] to implement a multi-variadic saturation diagram, guiding our FS choices.

In Fig. 3, the peak performance measurements on a dual socket 64-core AMD Milan featuring multiple FSs (SHM, BeeGFs, local NVME) are shown. We can observe behaviours matching the roofline models. In particular, each file system parametrizes a given roofline, and thus, such compact representation can be leveraged to characterize I/O trade-offs between FS.

We summarize our I/O parametrization implementation in Fig. 4, based on the peak values shown in Fig. 3. Metadata operations in BeeGFS, like in most HPC-oriented file systems, have lower efficiency compared to bandwidth. On the other hand, the single local NVME has better metadata performance but cannot match the performance of a whole storage array. In comparison, SHM is significantly faster. This diagram provides a practical way to quantify the performance differences between file systems and see what limits the I/O performance for a given program, whether it is IOPS or bandwidth. We also overlaid the execution coordinates of multiple applications using average bandwidth and average IOPS, creating a combined resource saturation diagram that can be used to measure the sensitivity of a program to I/O. The I/O benchmarks showed variable performance, whereas LULESH (with visualization activated) and BT-IO (class C) mainly remained fixed in this diagram. We are currently using models backed up by Extra-P to project the total dataset size and execution times to compute this mapping and anticipate saturation for a given file system, which can guide moldability. In particular, we anticipate that machine learning payloads may lead to higher IOPS, leading to patterns diverging from HPC applications.

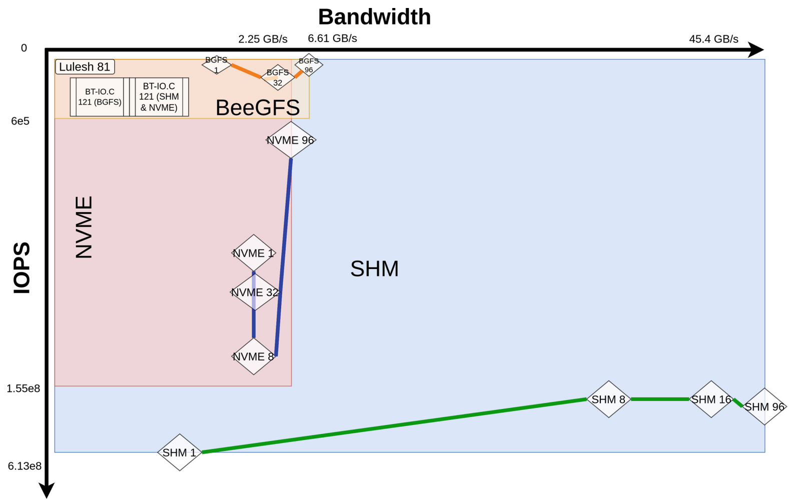


Fig. 4. Resource saturation diagram over bandwidth and IOPS. All scales are logarithmic. Applications are mapped as per average bandwidth and IOPS.

8 Conclusion and Future Work

To enable productivity and ease of use of HPC platforms, and in light of the increasingly complex launch configuration space of applications, there is a need for potential abstractions and automation. The concept of the *smart scheduler* has been discussed in this paper as a means of abstracting the use of HPC systems through monitoring and dynamic job configuration (moldability and malleability), it fits in a larger effort transversal to the whole execution chain. We briefly described the ADMIRE monitoring infrastructure and highlighted its capabilities for always-on monitoring, real-time performance tracking, and job profile generation. Additionally, we presented two use cases demonstrating dynamic job configuration and I/O tuning at launch time. Currently, we are integrating the ADMIRE infrastructure to leverage the concept of *smart scheduler*. This integration is expected to provide practical results shortly. Overall, the main contribution of the paper is offering a new approach to managing the complexity of HPC systems to enable their efficient use.

While we focused on moldability in this paper, the monitoring approach is also especially suited for malleability, as it provides key information (e.g., compute and I/O loads) that can be utilized. Hence, our future work focuses on using the described approach scheduling algorithms that consider malleable jobs, to effectively utilizes the different components of the HPC cluster and enhance the system throughput. Moreover, since in such a context, it can be valuable to know some key aspects like the periodicity of I/O phases (in case they are periodic), future work also centers on adding predictive capabilities to this infrastructure to boost the malleable decisions.

Acknowledgment. This work has been partially funded by the European Union's Horizon 2020 under the ADMIRE project, grant Agreement number: 956748-ADMIRE-H2020-JTI-EuroHPC-2019-1.

References

1. Ahn, D.H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., Schulz, M.: Flux: a next-generation resource management framework for large HPC centers. In: 2014 43rd International Conference on Parallel Processing Workshops, pp. 9–17. IEEE (2014)
2. Arima, E., Comprés, A.I., Schulz, M.: On the convergence of malleability and the HPC PowerStack: exploiting dynamism in over-provisioned and power-constrained HPC systems. In: Anzt, H., Bienz, A., Luszczek, P., Baboulin, M. (eds.) ISC High Performance 2022. LNCS, vol. 13387, pp. 206–217. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-23220-6_14
3. Balaprakash, P., et al.: Autotuning in high-performance computing applications. Proc. IEEE **106**(11), 2068–2083 (2018)
4. Besnard, J.B., Malony, A.D., Shende, S., Pérache, M., Carribault, P., Jaeger, J.: Towards a better expressiveness of the speedup metric in MPI context. In: 2017 46th International Conference on Parallel Processing Workshops (ICPPW), pp. 251–260. IEEE (2017)
5. Calotoiu, A., Hoefler, T., Poke, M., Wolf, F.: Using automated performance modeling to find scalability bugs in complex codes. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 45 (2013). tex.organization: ACM Citation Key: CA13
6. Cantalupo, C., et al.: A strawman for an HPC PowerStack. Technical report, Intel Corporation, United States; Lawrence Livermore National Lab. (LLNL) (2018)
7. Carns, P.H., et al.: Understanding and improving computational science storage access through continuous characterization. ACM Trans. Storage **7**(3), 8:1–8:26 (2011). https://doi.org/10.1145/2027066.2027068
8. Carretero, J., Jeannot, E., Pallez, G., Singh, D.E., Vidal, N.: Mapping and scheduling HPC applications for optimizing I/O. In: Proceedings of the 34th ACM International Conference on Supercomputing, pp. 1–12 (2020)
9. Cascajo, A., Singh, D.E., Carretero, J.: LIMITLESS-light-weight monitoring tool for large scale systems. Microprocess. Microsyst. **93**, 104586 (2022)
10. Cera, M.C., Georgiou, Y., Richard, O., Maillard, N., Navaux, P.O.A.: Supporting malleability in parallel architectures with dynamic CPUSETs mapping and dynamic MPI. In: Kant, K., Pemmaraju, S.V., Sivalingam, K.M., Wu, J. (eds.) ICDCN 2010. LNCS, vol. 5935, pp. 242–257. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11322-2_26
11. D'Amico, M., Jokanovic, A., Corbalan, J.: Holistic slowdown driven scheduling and resource management for malleable jobs. In: ACM International Conference Proceeding Series (2019). https://doi.org/10.1145/3337821.3337909
12. Denoyelle, N., Goglin, B., Ilic, A., Jeannot, E., Sousa, L.: Modeling large compute nodes with heterogeneous memories with cache-aware roofline model. In: Jarvis, S., Wright, S., Hammond, S. (eds.) PMBS 2017. LNCS, vol. 10724, pp. 91–113. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-72971-8_5

13. Dorier, M., Dreher, M., Peterka, T., Wozniak, J.M., Antoniu, G., Raffin, B.: Lessons learned from building in situ coupling frameworks. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, pp. 19–24 (2015)
14. Duro, F.R., Blas, J.G., Isaila, F., Carretero, J., Wozniak, J., Ross, R.: Exploiting data locality in Swift/T workflows using Hercules. In: Proceedings of NESUS Workshop (2014)
15. Goglin, B., Moreaud, S.: Dodging non-uniform I/O access in hierarchical collective operations for multicore clusters. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, pp. 788–794. IEEE (2011)
16. Gupta, R., Laguna, I., Ahn, D., Gamblin, T., Bagchi, S., Lin, F.: STATuner: efficient tuning of CUDA kernels parameters. In: Supercomputing Conference (SC 2015), Poster (2015)
17. Hoefler, T., Gropp, W., Kramer, W., Snir, M.: Performance modeling for systematic performance tuning. In: State of the Practice Reports, SC 2011, pp. 1–12. Association for Computing Machinery, New York (2011). https://doi.org/10.1145/2063348.2063356
18. Hu, W., Liu, G., Li, Q., Jiang, Y., Cai, G.: Storage wall for exascale supercomputing. Front. Inf. Technol. Electron. Eng. **17**(11), 1154–1175 (2016). https://doi.org/10.1631/FITEE.1601336
19. Huber, D., Streubel, M., Comprés, I., Schulz, M., Schreiber, M., Pritchard, H.: Towards dynamic resource management with MPI sessions and PMIx. In: Proceedings of the 29th European MPI Users' Group Meeting, pp. 57–67 (2022)
20. Klein, C., Pérez, C.: An RMS for non-predictably evolving applications. In: Proceedings - IEEE International Conference on Cluster Computing, ICCC, pp. 326–334 (2011). https://doi.org/10.1109/CLUSTER.2011.56
21. Kumar, R., Vadhiyar, S.: Identifying quick starters: towards an integrated framework for efficient predictions of queue waiting times of batch parallel jobs. In: Cirne, W., Desai, N., Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2012. LNCS, vol. 7698, pp. 196–215. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35867-8_11
22. Martí Fraiz, J.: dataClay: next generation object storage (2017)
23. Miranda, A., Jackson, A., Tocci, T., Panourgias, I., Nou, R.: NORNS: extending Slurm to support data-driven workflows through asynchronous data staging. In: 2019 IEEE International Conference on Cluster Computing (CLUSTER), USA, pp. 1–12. IEEE (2019). https://doi.org/10.1109/CLUSTER.2019.8891014
24. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parallel Distrib. Syst. **12**(6), 529–543 (2001). https://doi.org/10.1109/71.932708
25. Netti, A., et al.: DCDB wintermute: enabling online and holistic operational data analytics on HPC systems. In: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, pp. 101–112 (2020)
26. Nikitenko, D.A., et al.: Influence of noisy environments on behavior of HPC applications. Lobachevskii J. Math. **42**(7), 1560–1570 (2021). https://doi.org/10.1134/S1995080221070192
27. Patki, T., Thiagarajan, J.J., Ayala, A., Islam, T.Z.: Performance optimality or reproducibility: that is the question. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver Colorado, pp. 1–30. ACM (2019). https://doi.org/10.1145/3295500.3356217

28. Petrini, F., Kerbyson, D., Pakin, S.: The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In: SC 2003: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, p. 55 (2003). https://doi.org/10.1145/1048935.1050204
29. Prabhakaran, S., Neumann, M., Rinke, S., Wolf, F., Gupta, A., Kale, L.V.: A batch system with efficient adaptive scheduling for malleable and evolving applications. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 429–438. IEEE (2015)
30. Ritter, M., Calotoiu, A., Rinke, S., Reimann, T., Hoefler, T., Wolf, F.: Learning cost-effective sampling strategies for empirical performance modeling. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 884–895 (2020). https://doi.org/10.1109/IPDPS47924.2020.00095
31. Ritter, M., et al.: Noise-resilient empirical performance modeling with deep neural networks. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 23–34 (2021). https://doi.org/10.1109/IPDPS49936.2021.00012
32. Schulz, M., Kranzlmüller, D., Schulz, L.B., Trinitis, C., Weidendorfer, J.: On the inevitability of integrated HPC systems and how they will change HPC system operations. In: Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, pp. 1–6 (2021)
33. Smith, W., Taylor, V., Foster, I.: Using run-time predictions to estimate queue wait times and improve scheduler performance. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1999. LNCS, vol. 1659, pp. 202–219. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-47954-6_11
34. Sudarsan, R., Ribbens, C.J.: ReSHAPE: a framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. In: Proceedings of the International Conference on Parallel Processing (2007). https://doi.org/10.1109/ICPP.2007.73
35. Vef, M.A., et al.: GekkoFS-a temporary distributed file system for HPC applications. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 319–324. IEEE (2018)
36. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009). https://doi.org/10.1145/1498765.1498785
37. Wood, C., et al.: Artemis: automatic runtime tuning of parallel execution parameters using machine learning. In: Chamberlain, B.L., Varbanescu, A.-L., Ltaief, H., Luszczek, P. (eds.) ISC High Performance 2021. LNCS, vol. 12728, pp. 453–472. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78713-4_24
38. Wu, X., et al.: Toward an end-to-end auto-tuning framework in HPC PowerStack. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER), pp. 473–483. IEEE (2020)