Towards a Scale Invariant Syntax for Dynamic Job-Level Workflows

Jean-Baptiste Besnard[1], Martin Schreiber[2], and Allen D. Malony[3]

[1] ParaTools SAS, Bruyères-le-Châtel, France
[2] Université Grenoble Alpes / Laboratoire Jean-Kuntzmann / Inria, Grenoble, France
[3] ParaTools Inc., Eugene, USA

Abstract. High-performance computing (HPC) is undergoing a significant transformation as workloads become increasingly complex. The proliferation of nested parallelism and collocated functionalities in scientific software has naturally increased software complexity, leading to challenges in expressing launch configurations in a portable manner. This paper proposes the concept of self-unfolding dynamic workflows, which aims to alleviate this bottleneck by defining a compact runtime and mapping syntax that enables compute locality and resource composition between multiple jobs. Our scale-agnostic syntax for resource composition allows for job expressivity and dynamic resource utilization, making it easier to deploy and dynamically manage resources for parallel applications. We present a prototype implementation of this syntax over Slurm and an online visualization tool, demonstrating the advantages of this approach in terms of job layout compactness and usability.

Keywords: Resource Mapping · Scheduling · Modeling · MPI.

1 Introduction

Advanced HPC workflows organize themselves into data pipelines using input/output operations to store intermediate results. The end-user typically defines their work as a succession of individual jobs chained by their inputs and outputs [7]. With such configuration, the corresponding workflow graphs are often realized as a linear successions of jobs. Looking at job scheduling itself, these jobs are static in terms of resources, being defined at launch time.

Over the past two decades, various efforts have addressed this static resource allocation problem, by proposing particular ways of requesting resources [10,4] during runtime and how to handle them. Different terminology is being used to express the dynamics of the problem, such *malleability* [3], *moldability* [11], *system-* or *application-driven*, and *invasion* [5,16]). In this paper we will generally describe dynamic resource utilization with the term *dynamism*.

Given current hardware trends encouraging larger nodes with heterogeneous computing units, mapping parallel workloads effectively is becoming increasingly important. This mapping is not only a matter of memory affinity [12] but also affects computation itself, for example, by leveraging GPUs more suitable for

regular computational parts of the program. In this paper, we propose to integrate mapping into the program, deviating from current models that rely on either the scheduler or runtimes to perform this task. This leads us to the notion of *job-defined workflows* – jobs capable of differentiating their components as part of their internal programming logic; a self-building workflow. In our framework, which we justify as desirable, we focus on the particular point of resource negotiation, defining and implementing a resource negotiation syntax that allows multiple components to map topologically in a portable manner. In addition, it is of low scheduling complexity to realize.

2 Jobs and Workflows

In this section, we introduce the definition of *job-defined workflows*. We begin with regular bulk synchronous jobs and, after observing them as potential workflows, acknowledge current trends in job horizontalization up to a point where we consider the job itself should be able to reshape itself over time, differentiate, and map its various components autonomously.

2.1 Regular job submission

On current HPC systems, job execution primarily relies on a batch manager, executing jobs in sequence. Desired configuration and sequencing of jobs can be expressed with a given command line and rely on system abstractions supported by the batch manager to perform their computation. For instance, in Slurm, each job capacity is given by an upper bound, and this limit remains fixed during the whole job execution. Such a static resource allocation is susceptible to situations where 1) too many resources are allocated *a priori*, potentially making the execution less efficient, or 2) where not enough resources are allocated, making the job vulnerable to timeout. Both of these are limitations in the expressiveness of the job command syntax and the operation of the batch manager.

From a scheduling perspective, the entire job can be appropriately viewed as a single operation on the global data flow, see Fig. 1. The arrows show the flow of activity between *compute* and *storage* operations, indicating that data must move through most of the memory hierarchy between each job as they proceed. There is no information in the command that tells about how that data should be managed. In the case of *Datum B*, it may be only a reduced subset of the previous stages where *Datum A0* and *Datum A1* are merely serving as temporary storage. Such scenarios can result in significant performance deterioration and storage waste over time. One way for circumventing these data movements is called *in-situ* [7], which explores how jobs may collaborate in place instead of relying on input/output operations, while still utilizing the same resources. Looking closer at Fig. 1, one can see that data-movement issues might also be addressed by having a way to specify runtime job interactions more explicitly.

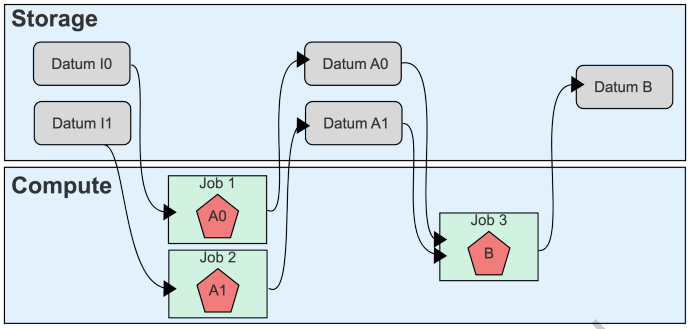


Fig. 1: A feed-forward workflow with bulk-synchronous applications. This chart depicts the data flow from and to the storage systems to run the computations.

2.2 Workflows with Job- and I/O-dependencies

The concept of *workflows* has gained interest in the HPC community and elsewhere as a way to develop applications with multiple jobs (programs) working cooperatively. We can envision the same computation above as a workflow (see Fig. 2 that is self-supported, meaning that components responsible for managing data movements are spawned alongside the job [17,14], as would be accomplished in a workflow runtime environment. When considering a workflow perspective for expressing runtime dynamics, it is possible to outline more complex dependency relationships, sequencing job inputs and outputs over time. A new syntax could enable users to describe job dependencies in the workflow manner more explicitly. The workflow concept for dynamic jobs could be captured with external definitions that include job sizes, job sequences, and conditional statements. Given such information, a job management system would be able to optimize I/O resource allocation for its specific needs, such as desired in our simple case study. In general, the greater expressivity of the jobs dynamics, interactions, and data flow, the greater the opportunity for the job management environment to more efficiently configure and allocate computing and memory resources.

In the context of horizontally scaled computing, our objective in this paper is to develop a scale-invariant resource negotiation syntax that enables an application program to adapt its structure over time based on the resources it has been allocated. An OpenMP program that alternates between serial and parallel regions is a good illustration of our vision. In the case of an application that could be composed of multiple parts interacting in workflow manner, our aim is to provide support for expressing this dynamic behavior at the scale of a distributed system, featuring *Nodes*, *NUMA* domains and *Slots* – terms that will be further described in the remainder of this paper. Our goal is then to allow a program to adapt its structure in any system while preserving a notion of locality, which is crucial for performance in larger HPC nodes.

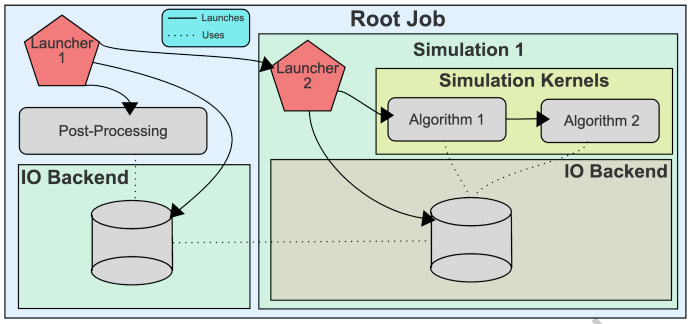


Fig. 2: Logical relationship in a nested self-configuring job.

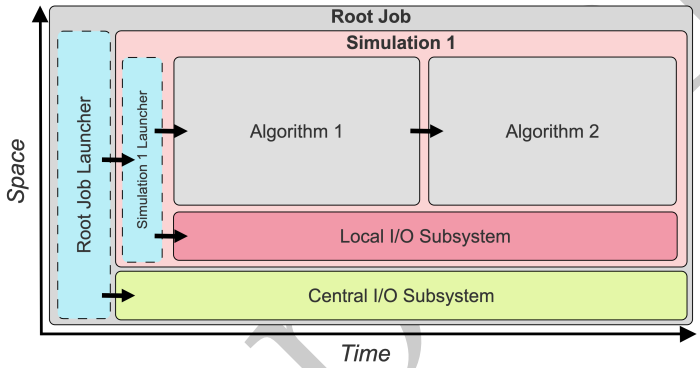


Fig. 3: Illustration of a job unfolding in both space and time as per Fig. 2.

2.3 Job-defined workflows

To this purpose, we propose the concept of *a job-defined workflow*, situated at the intersection of in-situ [7] and workflows [13]. Instead of dependencies specified by a workflow graph, the workflow would be dynamically set up during a job's runtime, as shown in Fig. 3. An entire workflow for an application would then be embedded within a single job. The job could then establish its ancestral dependencies (child, grandchild, and so on) during runtime. A clear disadvantage is that the scheduler cannot anticipate the entire workflow before the job begins, leading to potential delays due to resource allocation as the job runs. The counter argument is that otherwise all job interactions and resource allocation would have to be statically defined. Rather, a significant advantage is the ability to adjust resources depending on the current state of the workflow (e.g., whether GPUs or only CPUs are available). Indeed, conversely reshaping a job over time (i.e., dynamic resource orchestration) is inherently linked with complex data management issues, resulting from the need to relocate all the datasets. In contrast, self-unfolding workflows embed inside their programming logic the idea

of the job's reconfiguration, making the job more adaptable to reshape itself at the most opportune moment.

3 HPC Integration

We have outlined the idea of replacing job workflows with jobs that can dynamically unfold their own workflows. This approach moves away from traditional batch scripts to more sophisticated orchestrators, elevating this feature to first-class citizenship in the application development model. Of course, enabling a system with such dynamism require changes throughout the entire HPC stack. In this work, we focus on one of these aspects, namely how to describe required resources concretely. Indeed, as presented in Figs. 2 and 3, enabling self-unfolding programs requires job requests for resources over time. What was initially the concern of the user through manual mapping now becomes part of the program.

Consequently, there is a need for a new syntax that allows specifying how to request and compose particular resources in a *scale-agnostic* manner. Below, we present our first steps towards a system-independent description where the program can restructure itself over available resources without having to hard-code specific values. An initial reaction one might have is that MPI can already launch collocated jobs in MPMD mode. Unfortunately, it is not fully portable. Indeed, the '`mpirun`' command is not standardized in MPI, and each MPI implementation has its own flags. While the meaning of the '`-np`' flag in this command is generally known, other flags, such as binding flags, are not predictable at all. As a result, when it comes to MPMD programs, one quickly becomes overwhelmed when trying to map processes accordingly. Therefore, it is far from trivial to map a given process image once per node while allocating other cores for another purpose.

To enable scenarios like the one depicted in Fig. 3, there is a need for new approaches to job scheduling in HPC, particularly the enabling of nested scheduling techniques — allocating a subset of the machine and allowing various components to allocate their mapping recursively. Certainly, this concept is not novel. For example, the Flux [1] scheduler is based on the principle of a nested scheduler. Given the hierarchical nature of the hardware itself, our approach naturally allows for the separation of concerns between different layers of the scheduler. A key aspect in such an approach is the isomorphism between layers. Firstly, how can a hierarchy of jobs populate a set of resources in a scale-invariant manner by recursively sharing computing elements? Secondly, is there a compact way to describe such nesting? Our proposed solution to these two questions is the syntax for resource composition.

4 A Syntax for Resource Composition

A resource mapping syntax requires a description of the resources as well a mapping jobs to these resources. Consequently, we proceed with an abstract

machine model, followed by a resource composition syntax that is tightly coupled to the machine model.

4.1 Hierarchical abstraction of HPC hardware into levels

Choosing hardware-related descriptors is never trivial, as HPC hardware evolves quickly, rendering previous choices obsolete in future versions. Therefore, we propose working on an overly simplified machine model. We consider a machine comprising a set of **Nodes**, each running separate memory region with instances of lower splitting levels. Inside a Node, one can find Non-Unified Memory Access (**NUMA**) regions, which are privileged memory affinity regions. These may not be actual NUMA nodes but could be sockets or caches depending on runtime (or machine) configuration. Each NUMA region contains **Slots**. Slots do not always correspond to cores; instead, they practically represent individual MPI processes spanning their affinity. Thus, if there is one MPI process per node, all these levels are equivalent. Overall, this machine description is similar to what Slurm does with Node, Process, and Core as a hierarchy, with the addition of the NUMA level, which has become increasingly important as nodes continue to grow in size. Besides, we consider these levels to be embedded in each other, which means all Slots are part of various NUMAs which themselves belong to different Nodes, hence $slot \subseteq numa \subseteq node$ making Slots the smallest granularity of resource allocation. For the development of a grammar, we will introduce formally the derivation $R \rightarrow (node|numa|slot)$ to refer to each of the levels.

4.2 Mapping Syntax

We aim to develop a resource allocation syntax that can allocate resources from any node configuration without relying heavily on specific resources, such as cores from a specific NUMA domain, etc. Instead, it should be largely independent of specific resources and allow for some degree of resource mapping. This syntax should also enable preferences for process group layouts. Our motivation for creating this syntax stems from the observation that locality constraints are critical performance factors in modern HPC payloads [12,6]. Once specified, the syntax can remain unchanged and enable the program to map resources onto the target system in a portable way, based on the previously introduced machine model. The per-job proposed grammar is then given by a regular expression of the form $J \rightarrow (A|E|[0-9]+)R$ with terminals A and E explained below. To share resources between jobs that should be started, we also support allocating resources for multiple jobs, as comma-separated lists: $L \rightarrow (J, L|J)$. As illustrated in our online simulator, `https://dynamic-resource.github.io/project/grammar/`, job specifiers in J have the following meanings:

Specifier	Meaning	Example
`A`	Equal sharing among all jobs	`Anode`,`Anode` 2 jobs sharing nodes
`E`	One slot from each resource	`Enode` one slot per node
`0-9+`	Fixed number of resources	`4slot` allocate 4 slots

4.3 Mapping Logic

Now that we have defined the various specifiers (`A`, `E`, `[0-9]+`) and levels (`Node`, `Numa`, `Slot`), this section defines the mapping algorithm. This logic is straightforward and simply consists of performing the following operations:

- First, the "each" specifier allocates one slot per dedicated level, starting from the highest level to the lower ones (node, numa, slot).
- Second, the "fixed" specifier allocates a given number of slots iterating at the granularity of the given resource level.
- Third, the "all" specifier splits resources between the remaining processes as evenly as possible using a scatter policy in function of the target level.

Overall, the logic is straightforward: resources are linearly allocated for the given target level, and no complex computation is involved. The advantage of this model is that resources can be dynamically split between jobs while ensuring locality. Despite being mostly agnostic to resources, this syntax enforces several constraints. There must always be sufficient resources to provide at least one slot to each program using the "all" specifier. Similarly, a program using "each" should have sufficient resources on each instance of the given level. Eventually, fixed allocations should also have their resources fulfilled.

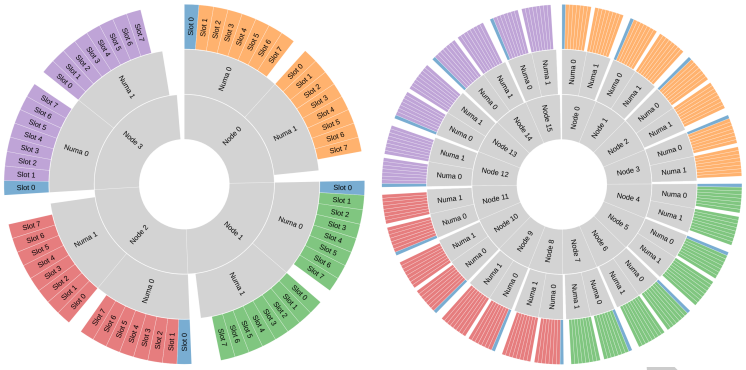
Due to its similarity, we would like to highlight a feature in OpenMP that allows resource mapping on shared-memory systems. Using the environment variable `OMP_PLACES` [[8]] enables referring to threads, cores, sockets, and other memory-hierarchy-related resources, analogous to what is referred to in this work as levels. Then, setting `OMP_PROC_BIND` in the environment allows either spreading the threads across places or keeping them close within one place. We would like to note that in addition of crossing nodes' boundaries, the way we can allocate resources is significantly more flexible than OpenMP's implementation.

4.4 Summary

In this section, we introduced the mapping syntax that we intend to use for job-defined workflows. This syntax, although simple, enables (1) resource differentiation by splitting slots and (2) a straightforward means of mapping arbitrary processing while maintaining locality – as illustrated in Fig. 4. Furthermore, this syntax being resource-agnostic, it also supports transparent scaling as shown by the example of scaling up the initial mapping from 4 nodes to 16 nodes (Fig. 4a to Fig. 4b) without modifying the resource specification.

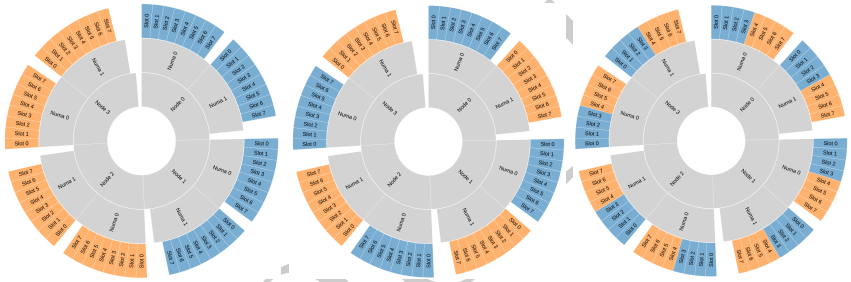
5 Use Cases

To illustrate some use cases for our mapping syntax and more generally for self-unfolding tasks, we now present some practical examples. First, we discuss MPMD applications, then focus on the requirements for parallel I/Os and eventually generalize to self-unfolding jobs.



(a) 4 Nodes, 32 slots (b) 16 Nodes, 128 slots

Fig. 4: Usage of the `Enode,A,A,A,A` syntax (four jobs plus one job once on each node) for scaling with 32 slots in 4a and 128 slots in 4b



(a) `A,A` (b) `Anode,Anode` (c) `Anuma,Anuma`

Fig. 5: Comparison of MPMD mapping syntaxes over Slots, Nodes, and NUMAs.

5.1 Elastic MPMD

Fig. 5 presents the resulting process layout for two applications, first globally, then over nodes, and eventually over NUMAs. These layouts split resources between the two jobs using various levels of the abstract machine model hierarchy. As of now, obtaining such control from Slurm, for example, is not trivial and requires precise handling of the topology and command-line parameters. In contrast, in these three simple examples, we were able to quickly explore three different mappings for the two applications by simply changing the mapping syntax while keeping the same number of processes.

5.2 I/O Collocation

I/O backends such as ad-hoc storage are well-known candidates for collocation. In general, the service needs to be spawned once per node to enable local processes to communicate efficiently through data channels with the I/O backend.

Fig. 4 illustrates how one may allocate one slot per node using the `Enode` syntax for I/O while providing the rest to the application. In summary, our mapping syntax simplifies the deployment of ad-hoc services, directly expressing from a machine hierarchy what needs to run where and avoiding potential complexities in the Slurm command-line interface for similar mappings.

5.3 Job-defined Workflows

The two previous examples are simple ones derived from relatively classical jobs, particularly the MPMD one. However, the focus of this contribution is on the notion of *self-defined workflow*, which defines a job that deploys itself on resources and actively interacts with the scheduler. In such a scenario, there is first a need for resource allocation and deallocation, which can be handled through the PMI(x) or MPI session [9] interfaces. There is also a need for remapping the program over the newly allocated resources, either to occupy new cores or to alternate between configurations as the workflow unfolds. As a consequence, the ability to shrink or grow the allocation combined with our resource mapping syntax would allow a program to *reshape* itself over time in a portable manner – opening the way for even more dynamism in traditional HPC workloads.

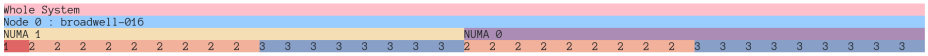
6 Implementation

In order to show a practical realization of our approach, we have implemented our scale-invariant mapping syntax over Slurm by using a wrapper written in Rust (`https://github.com/dynamic-resource/lmap`). The wrapper relies on a two-phase allocation within a pre-allocated set of resources (using `salloc`). In the first phase, it runs a job on all allocated slots (`srun`) to gather information through standard output, including the rank (`PMI_RANK`) and bindings using hwloc. Collected data are then used to compute the final mapping for the requested jobs, which are specified in a YAML job file (as per Listing 1.1) utilizing the previously described mapping syntax.

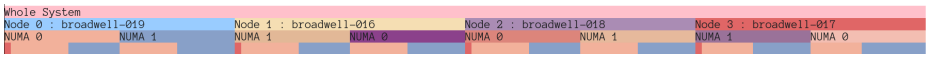
Listing 1.1: Sample job file for `lmap`

```
- map: "Anuma"                                                    1
  command: ["./ocean", "-i", "./atlantic.dat"]                    2
- map: "Anuma"                                                    3
  command: ["./atmosphere", "-i", "./cloudy.dat"]                 4
- map: "Enode"                                                    5
  command: ["ad_hoc_storage"]                                     6
```

Fig. 6 presents a sample console output from `lmap` displaying the layout associated with the jobfile listed in Listing 1.1. This output represents the hierarchy of various levels using colors to distinguish between different nodes, NUMAs, and slots. In this particular case, on the last line, slots are colored and numbered according to the job running on them. It can be seen that our wrapper successfully mapped one `ad_hoc_storage` per node (red) automatically in a scale-invariant



(a) Single node with 36 cores on 2 NUMAs



(b) 4 nodes with 36 cores on 2 NUMAs

Fig. 6: Console output screenshots of `lmap` for `Enode`, `Anuma`, and `Anuma`. Running `lmap -d ./map.yaml`. With `map.yaml` shown in Listing 1.1.

manner, implementing the use-case described in Section 5.2. The layout is forwarded to Slurm using the `--multi-prog`[4] option, which allows mapping arbitrary binaries to individual processes. Overall, `lmap` provides a convenient way for Slurm to map MPMD programs without depending on platform-dependent flags. It is clear that the ability to correctly map programs that are tightly bound has direct performance consequences as what would previously be inter-node communications can become shared-memory (SHM or CMA) exchanges. This is, of course, of particular interest, but not limited to, ad-hoc storage.

7 Conclusion

In this paper, we initially proposed the concept of *self-unfolding workflows*. From traditional batch executions leading to workflows reliant on the file system between bulk synchronous applications, we posed the question of how to alleviate this bottleneck. Building upon existing work in in-situ processing and workflow management, we asked how to define a compact runtime and more precisely map syntax to ease the expression of more horizontal HPC payloads — collocating multiple programs within an allocation. To this end, we presented a new *scale-agnostic* syntax for resource composition enabling (1) compute locality and (2) resource composition between multiple jobs. This syntax has been implemented both in a dedicated simulator and over Slurm. In addition, we discussed multiple use cases for this syntax, illustrating the corresponding job layout and the syntax's compactness. This work represents an initial step towards scaling HPC payloads under dynamics with all the aforementioned benefits, and much remains to be done to eventually enable ideas long conceived in other fields such as in-situ processing and workflow management.

8 Future Work

The heterogeneous computing landscape is poised for a paradigmatic shift as the increasing complexity of parallel hardware with resource specialization demands symmetrical software specialization. The convergence of malleability [15],

[4] `https://slurm.schedmd.com/srun.html#SECTION_MULTIPLE-PROGRAM-CONFIGURATION`.

moldability [3], workflows [13], and in-situ [7] computing concepts has long been recognized as a key to unlocking the full potential of HPC payloads. Despite their theoretical advantages, these ideas have yet to be fully integrated into mainstream HPC practice. However, this evolution is not limited to a specific part of the HPC stack; it fundamentally changes how programs run and unfold, requiring new practices or even a revolution in HPC. In this paper, we only scratch the surface of the overall issue by defining potentially *job-defined workflows* and associated mapping capabilities. To enable the dynamism required by these new workflows, we identify essential requirements for both resource allocation (PMIx and Wiring) and code-coupling (asynchronous messaging such as RPCs). Furthermore, we emphasize the need for a transversal effort across all relevant HPC standards (PMIx, MPI, OpenMP) and tools to facilitate this shift towards a more horizontal view of HPC payloads.

9 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. This project has received funding from the Federal Ministry of Education and Research and the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955701, Time-X. The JU receives support from the European Union's Horizon 2020 research and innovation program and Belgium, France, Germany, Switzerland. The authors gratefully acknowledge the computing time on the high-performance computer at the University of Turin from the laboratory on High-Performance Computing for Artificial Intelligence [2].

References

1. Ahn, D.H., Bass, N., Chu, A., Garlick, J., Grondona, M., Herbein, S., Ingólfsson, H.I., Koning, J., Patki, T., Scogland, T.R., et al.: Flux: Overcoming scheduling challenges for exascale workflows. Future Generation Computer Systems **110**, 202–213 (2020)
2. Aldinucci, M., Rabellino, S., Pironti, M., Spiga, F., Viviani, P., Drocco, M., Guerzoni, M., Boella, G., Mellia, M., Margara, P., Drago, I., Marturano, R., Marchetto, G., Piccolo, E., Bagnasco, S., Lusso, S., Vallero, S., Attardi, G., Barchiesi, A., Colla, A., Galeazzi, F.: HPC4AI, an AI-on-demand federated platform endeavour. In: ACM Computing Frontiers. Ischia, Italy (May 2018)
3. Aliaga, J.I., Castillo, M., Iserte, S., Martín-Álvarez, I., Mayo, R.: A survey on malleability solutions for high-performance distributed computing. Applied Sciences **12**(10), 5231 (2022)
4. 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: High Performance Computing. ISC High Performance 2022 International Workshops: Hamburg, Germany, May 29–June 2, 2022, Revised Selected Papers. pp. 206–217. Springer (2023)

5. Bungartz, H.J., Riesinger, C., Schreiber, M., Snelting, G., Zwinkau, A.: Invasive computing in hpc with x10. In: Proceedings of the third ACM SIGPLAN X10 Workshop. pp. 12–19 (2013)
6. Carretero, J., Jeannot, E., Pallez, G., Singh, D.E., Vidal, N.: Mapping and scheduling HPC applications for optimizing I/O. In: Ayguadé, E., Hwu, W.W., Badia, R.M., Hofstee, H.P. (eds.) ICS '20: 2020 International Conference on Supercomputing, Barcelona Spain, June, 2020. pp. 33:1–33:12. ACM (2020)
7. 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)
8. Eichenberger, A.E., Terboven, C., Wong, M., an Mey, D.: The design of openmp thread affinity. In: OpenMP in a Heterogeneous World: 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings 8. pp. 15–28. Springer (2012)
9. Huber, D., Streubel, M., Comprés, I., Schulz, M., Schreiber, M., Pritchard, H.: Towards dynamic resource management with MPI sessions and pmix. In: EuroMPI/USA'22: 29th European MPI Users' Group Meeting. p. 57–67. ACM, Chattanooga TN USA (Sep 2022)
10. Iserte, S., Mayo, R., Quintana-Orti, E.S., Pena, A.J.: Dmrlib: Easy-coding and efficient resource management for job malleability. IEEE Transactions on Computers **70**(9), 1443–1457 (2020)
11. Le Fèvre, V., Herault, T., Robert, Y., Bouteiller, A., Hori, A., Bosilca, G., Dongarra, J.: Comparing the performance of rigid, moldable and grid-shaped applications on failure-prone hpc platforms. Parallel Computing pp. 1–12 (2019)
12. León, E.A.: mpibind: A memory-centric affinity algorithm for hybrid applications. In: Proceedings of the International Symposium on Memory Systems. pp. 262–264 (2017)
13. Lüttgau, J., Snyder, S., Carns, P., Wozniak, J.M., Kunkel, J., Ludwig, T.: Toward understanding I/O behavior in hpc workflows. In: 2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS). pp. 64–75. IEEE (2018)
14. Martinelli, A.R., Torquati, M., Aldinucci, M., Colonnelli, I., Cantalupo, B.: Capio: a middleware for transparent I/O streaming in data-intensive workflows. In: 2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC). pp. 153–163. IEEE (2023)
15. Tarraf, A., Schreiber, M., Cascajo, A., Besnard, J.B., Vef, M.A., Huber, D., Happ, S., Brinkmann, A., Singh, D.E., Hoppe, H.C., Miranda, A., Peña, A.J., Machado, R., Gasulla, M.G., Schulz, M., Carpenter, P., Pickartz, S., Rotaru, T., Iserte, S., Lopez, V., Ejarque, J., Sirwani, H., Wolf, F.: Malleability in modern hpc systems: Current experiences, challenges, and future opportunities. IEEE Transactions on Parallel and Distributed Systems pp. 1–14 (2024)
16. Teich, J., Weichslgartner, A., Oechslein, B., Schröder-Preikschat, W.: Invasive computing-concepts and overheads. In: Proceeding of the 2012 Forum on Specification and Design Languages. pp. 217–224. IEEE (2012)
17. Vef, M.A., Moti, N., Süß, T., Tocci, T., Nou, R., Miranda, A., Cortes, T., Brinkmann, A.: Gekkofs-a temporary distributed file system for hpc applications. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). pp. 319–324. IEEE (2018)