Towards a Standard for Requesting Resources in a Dynamic Environment

Jean-Baptiste Besnard[1] and Martin Schreiber[2]

[1] ParaTools SAS, Bruyères-le-Châtel, France
[2] Université Grenoble Alpes, Grenoble, France

Abstract. High-Performance Computing is undergoing a significant transformation as workloads become increasingly complex. Nested parallelism and the proliferation of collocated functionalities in scientific software have naturally led to increased software complexity. Consequently, launch configurations have become more difficult to express in a portable manner, requiring deep knowledge of both the application and target system to run a program successfully. In this paper, we explore the concept of a scale-agnostic syntax for running parallel jobs, possibly embedded inside the target binary. We expect this information to enable programs to clearly express their launch configuration to the underlying scheduler, making it easier to deploy and dynamically manage parallel applications. We present a prototype implementation of this syntax in an online visualization tool and discuss the advantages and disadvantages of this approach in terms of job expressivity and malleability.

Keywords: Resource Mapping · Scheduling · Modeling · MPI.

1 Introduction

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

Over the previous two decades, various work has been done to solve this static resource allocation where different nomenclature was used (malleability [1], moldability [13], system- or application-driven, invasion [3, 16]) with each one of them targeting just a particular way of requesting resources [12, 2] over runtime and handling them. We do not want to restrict ourselves to one of these particular cases and refer to the dynamic resource utilization as *dynamism* in the present work.

The present paper will investigate a way of requesting particular resources under dynamism while maintaining both data locality and portability between machines. We will motivate it briefly with workflows [14, 4] in the following sections.

1.1 Regular job submission

On current HPC systems, job execution primarily relies on a batch manager which allows the execution of jobs in sequence. Jobs can then be run with a given command line and they rely on system abstractions to perform their computation. As presented in Listing 1.1 using Slurm, each job capacity is given by an upper bound and this limit remains fixed during the whole job execution. This leads to the first limitation, namely, with a static resource allocation leading to situations where too many resources are allocated a priori, making the execution less efficient, or where not enough resources are allocated, resulting in a potential timeout of the job.

Listing 1.1: Sample sequence of command generating the flow shown in Fig. 1.

```
srun -p mypartition -n 512 ./A0 &
srun -p mypartition -n 512 ./A1 &
wait
srun -p mypartition -n 512 ./B
```

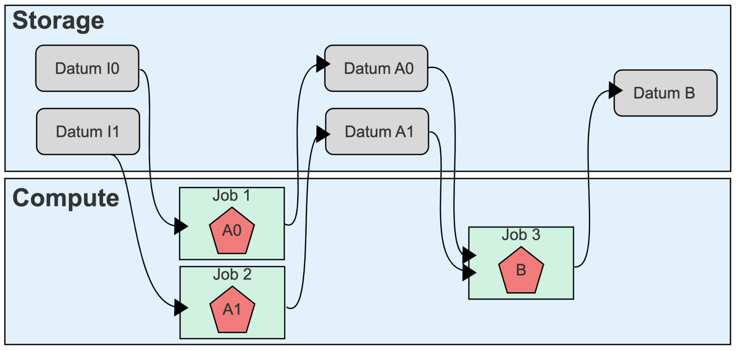


Fig. 1: Illustration of a feed-forward workflow composed of bulk-synchronous applications. This chart depicts the data flow from and to the storage systems to run the computations.

From the scheduling perspective, the whole job is seen as a single operation on the global data flow, see Fig. 1. The arrows going back and forth between *compute* and *storage* show that data has to move through most of the memory hierarchy between each job. If we look at the data itself, the last entry, here *Datum B* which is the final valorized data may be only a reduced subset of the previous stages, resulting in significant performance deterioration and storage wastes over time. One standard process of circumventing the aforementioned data movements is called in-situ [7] exploring how jobs may collaborate in place, instead of relying on I/O, however relying on the same resources.

Looking closer at Fig. 1, one can see that data-movements issues could be also solved by having a runtime alternative discussed in the next section.

1.2 Workflows with Job- and I/O-dependencies

In Fig. 2, we present the same computation seen as self-supported, in the sense that components for managing data movements were spawned alongside the job [17]. When considering workflows, it is possible to outline more complex dependency graphs, sequencing job inputs and outputs over time. This enables users to describe job dependencies in the workflows explicitly. In general, these workflows are external definitions, including job sizes, the job sequence, and some conditionals. The system would now be able to optimize, e.g., I/O resource allocation for these particular needs. In addition, generalizing such jobs would lessen the requirements on the service island which currently occupies a non-negligible part of supercomputers mostly to implement I/O – leaving more space for computing while keeping transitional data in the higher layers of the memory hierarchy.

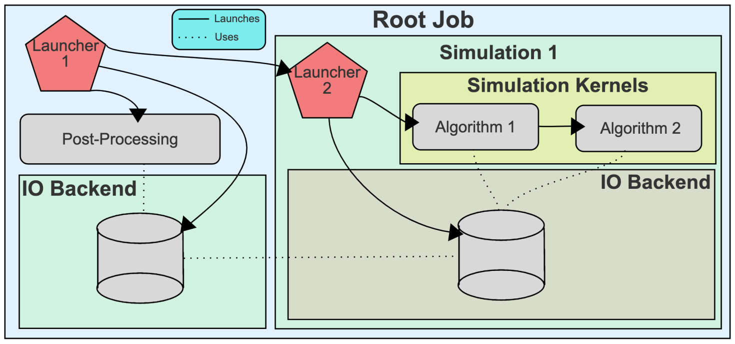


Fig. 2: Logical relationship in a nested self-configuring job.

Besides applying in-situ ideas also here (but with varying resources) one can see that it would make the workflow of Fig. 2 much simpler. However, this supposes the definition of a distributed cross-job data description [5, 4] which is yet to be fully defined for HPC. This paper looks at the resource allocation aspect with the idea of embedding it inside the job itself.

1.3 Job-defined workflows

One of the motivations for the contribution of this paper is the idea of *a job-defined workflow*, standing somewhere at the crossroads of in-situ [7] and workflows [14]. Rather than having the dependencies given in terms of a workflow, the workflow could be dynamically set up over a job's runtime, see Fig. 3. An entire workflow would then be embedded into a single job. The job could then set up its child dependencies during its runtime (and the children's grandchildren's, etc.). A clear disadvantage is that the scheduler can't foresee the entire workflow

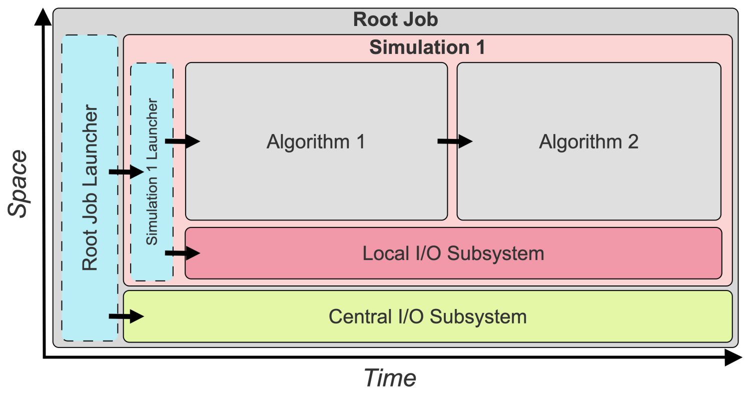


Fig. 3: Illustration of a job unfolding in space and time as per Fig. 2.

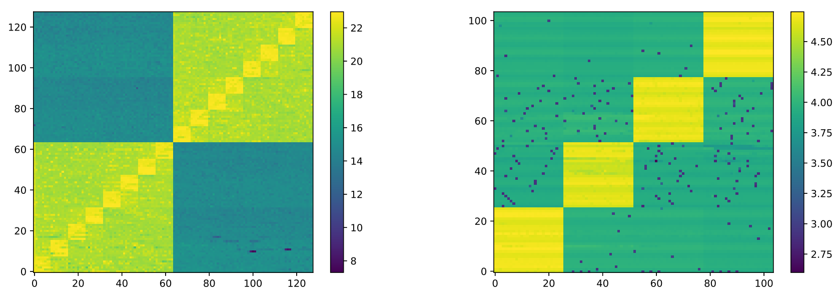
before the job starts, leading to potential delays due to resource allocation as the job runs. However, we also see strong advantages by, e.g., changing the resources depending on the current state of the workflow (e.g., whether there are GPUs or just CPUs available). Indeed, conversely reshaping a job over time is necessarily linked with complex data management issues, due to the need to move all the data-set. Whereas in the case we outlined here, self-unfolding workflows embed programmatically the idea of the job's reconfiguration, making the job more prompt to reshape itself at the most suitable moment.

1.4 Intermediate summary

We have outlined the idea of replacing job workflows with jobs having the potential to dynamically unfold workflows. This leads away from traditional batch scripts to more elaborated orchestrators to put this feature as the first class citizen in the programming model. Enabling a system with such dynamism would require changes throughout the entire HPC stack. We'll focus in the present work on one of these peculiarities, namely how to describe the required resources very concretely. Indeed, as presented in Figs. 2 and 3, enabling self-unfolding programs supposes the job requests resources over time. In addition, this process, which is initially the concern of the user as shown in Listing 1.1 now becomes part of the program. Consequently, there is a need for a new syntax allowing to specify how to request and compose particular resources in a *scale-agnostic* manner. As we will show in the next sections, we take the first steps with our syntax towards a system-independent description where the program can restructure itself over the available resources without having to hardcode specific values. this provides just one particular way of resource allocation which where we see our approach somewhere between very explicit allocations (e.g., 5 CPUs) and implicit allocations (e.g., give me the number of resources fitting to some policies).

2 Composing with Memory Affinity

Current architectures present increasingly complex memory hierarchies. Contemporary HPC hardware generally hosts several processors each attached to a different NUMA socket, notwithstanding the case for accelerators, leading to potentially detached memories or partial overlap, for example using High-Bandwidth Memory (HBM). A direct consequence of this evolution is the need for the locality to become central in most HPC runtimes; OpenMP devised *places* [8] and MPI added support for topological discovery using new communicator splitting values [9]. However, on the allocation side, the end-user is mostly tributary to the choices of the underlying scheduler in terms of process mapping. Most of the time, nodes are allocated linearly, and defining a more complex mapping syntax can become cumbersome due to system-specific descriptions. This usually leads users to resort to the default behavior – ignoring locality or more precisely leaving it to the underlying runtimes.



(a) 2 sockets 64c Epyc Milan (b) 4 sockets 26c Xeon Platinum 8367HC

Fig. 4: Core to core bandwidth variation on two different HPC-class machines (all scales are in GB/sec). For transfers of 256 MB averaged 10 times.

In order to validate the machine structure we retain in the following sections – structure driven by data locality – we have designed a simple memory benchmark[3] computing memory bandwidth from core to core on a whole machine. As presented in Fig. 4, memory locality can lead to well-known important performance discrepancies when accessing data outside of the current NUMA node. This is particularly visible in Fig. 4b, featuring 4 Sockets and 4 NUMA nodes, in this case, the adjacency matrix clearly shows more optimal areas. Similar performance effects are still present in Fig. 4a which features more recent processors (128 cores over two AMD Milan sockets, running on two NUMAs). Observing these performance effects and associated bandwidth variation clearly testifies that locality is a key element in current HPC, which is of course well known, see

[3] `https://github.com/dynamic-resource/mapper`

[10, 6]. However, our take in the rest of this paper is that scheduling syntaxes are still failing at capturing locality requirements in a compact and actionable manner.

3 A Syntax for Resource Composition

A resource mapping syntax requires some description of the resources as well as mapping jobs to these resources. Hence, we continue with an abstract machine model followed by a resource composition syntax which is tightly bound to the machine model.

3.1 Abstracting HPC Hardware into levels

Choosing hardware-related descriptors is never trivial as HPC hardware is evolving so quickly that it may become irrelevant in a future version. Therefore, we propose to work on an overly simplified machine model. We consider a machine built with a set of **Nodes** such as each node is running a 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 are not necessarily actual NUMA nodes, they could be sockets or caches depending on runtime (or machine) configuration. Eventually, each **NUMA** contains **Slots**. **Slots** are not always cores, they practically correspond to individual MPI processes spanning on their affinity. It means that if there is one MPI process per node all these levels are equivalent. Overall, this machine description is not far from what slurm does with Node, Process, and Core as a hierarchy with the only addendum of the NUMA level which is of increasing importance since nodes are becoming larger. 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

$$\text{slot} \subseteq \text{numa} \subseteq \text{node}$$

making Slots the smallest granularity of resource allocation. For the development of a grammar, we will introduce formally the derivation

$$R \rightarrow (\text{node}|\text{numa}|\text{slot})$$

to refer to each of the levels.

3.2 Mapping Syntax

We aim to create a resource allocation syntax that can allocate resources from any node configuration. This syntax should be largely independent of specific resources and avoid relying using only resource specifiers such as “10 cores from NUMA domain 12, 10 cores from NUMA domain 13, etc.”. It should also allow for some degree of resource mapping, allowing preferences for process group layouts.

Our motivation for creating this syntax comes from the observation that locality constraints are critical performance factors in modern HPC payloads, as outlined in Section 2. Once the syntax is specified, it 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 later since we first need to introduce job lists.

To share resources between jobs that should be started, we also support allocating resources for multiple jobs which can be expressed by a list of jobs

$$L \rightarrow (J, L|J).$$

Before explaining the grammar which can be still rather complicated to be understood, we already like to point out our visualizer using HTML + Javascript available at `https://dynamic-resource.github.io/project/grammar/`. All figures used in this section to further exemplify the mapping syntax are generated using this simulator. It also allows exploring the possibilities with this grammar. The job specifiers in J have the following meanings:

- `A`: For a list L of jobs, this refers to all resources to be equally shared among all of them specifying this qualifier.
  E.g., `Anode,Anode` would share all nodes equally between 2 jobs.

- `E`: refers to one slot from "each" resource.
  E.g., `Enode` allocates one slot from each node.

- `[0-9]+` stands for a fixed number of resources, hence the rather traditional way of resource allocation.
  E.g., `4slot` would allocate 4 slots.

3.3 Mapping Logic

After presenting the syntax and explaining its meaning, its behavior is still not determinate due to various possibilities to implement it. Hence, what follows is a discussion of one of many potential realizations. We do not want to claim this to be the best realization but provide this in particular for the reproducibility of our results. In particular, it sheds light on the logic of the simulator.

- 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 works in descending level order while accounting for resources taken by each modifier (solely at Numa and Node level). Resources are allocated linearly.
- Third, the "all" specifier splits resources between the remaining processes as evenly as possible.

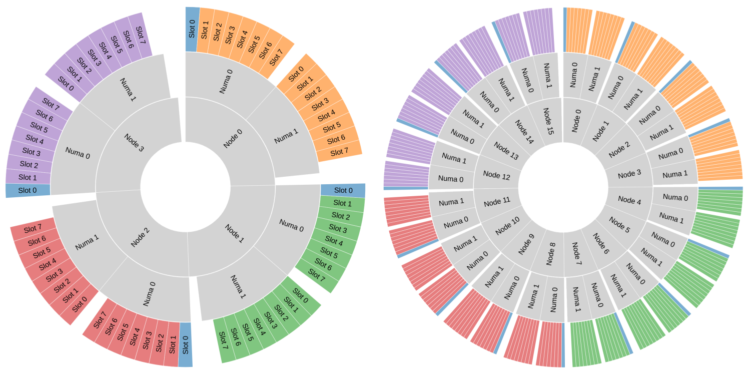
Then, slots are allocated again from the highest level policy to the lower one. The level is then associated with locality, Node will allocate free Numa linearly on nodes whereas Numa will do it over Numas. Eventually, the slot level will proceed to the linear allocation over free slots.

Overall, the logic is simple: resources are linearly allocated, 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 enough 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 like to point out such a feature in OpenMP which allows similar resource allocations, but on shared-memory systems. Using the environment variable `OMP_PLACES`[8] allows referring to threads, cores, sockets, and other memory-hierarchy-related resources, similar to what refers in the present work as levels. Then, setting `OMP_PROC_BIND` in the environment allows to either "spread" the threads across places or keep them "close" within one place. We like to point out that the way how we can allocate resources is significantly more flexible than OpenMP's implementation.

3.4 Summary

In this section, we have introduced the mapping syntax we intend to use for job-defined workflows. This syntax despite being very simple enables (1) resource differentiation splitting slots and (2) simple means of mapping arbitrary processing when maintaining locality.



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

Fig. 5: Usage of the `Enode,A,A,A,A` syntax (four jobs plus one job once on each node) for scaling with 32 slots in 5a and 128 slots in 5b

We show in Fig. 5 the compact and efficient syntax to describe the intended resource mapping. Moreover, this syntax being resource agnostic, it also supports transparent scaling as it scales up the initial mapping of Fig. 5a from 4 nodes to 16 in Fig. 5b without changing the resource specification which is possible due to runtime resource allocation.

4 Use Cases

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

4.1 I/O Collocation

I/O backends such as ad-hoc storage are well-known candidates for collocation. In general, the service has to be spawned once per node to allow local processes to communicate through efficient data channels with the I/O backend. Fig. 6 presents how one may want to allocate one slot per node for I/0 while providing the rest to the application. The same syntax is also shown in conjunction with more jobs in Fig. 5. In summary, our mapping syntax makes the deployment of ad-hoc services much more simple, directly expressing from a machine hierarchy what has to run where and dodging potential complexities in the slurm command line to do similar mappings.

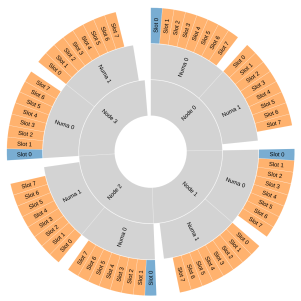


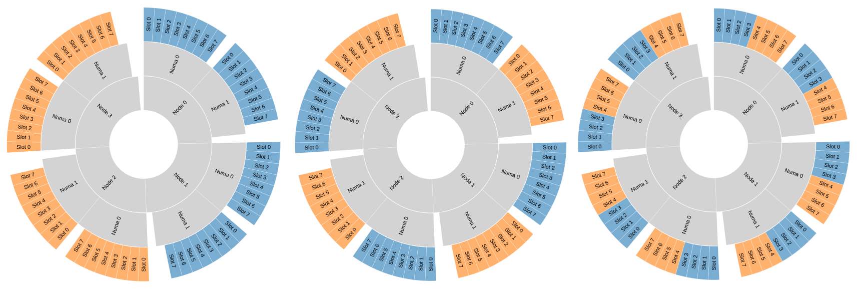
Fig. 6: `Enode,A`

4.2 Elastic MPMD

Fig. 7 presents the resulting process layout for two applications, first globally, then over Nodes, and eventually over NUMAs. These layouts are splitting resources between the two jobs using various levels of the abstract machine model hierarchy. As of now, getting such control from Slurm, for example, is not trivial and requires precise handling of the topology and command line parameters. Whereas, in these three simple examples we were able to quickly explore three different mapping for the two applications by simply changing the mapping syntax while keeping the same number of processes.

4.3 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 the notion



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

Fig. 7: Comparison of MPMD mapping syntaxes over Slots, Nodes, and NUMAs.

of *self-defined workflow* which defines a job deploying itself on resources, and therefore actively interacting with the scheduler. In such a scenario, there first is a need for resource allocation and deallocation, which can be handled through the PMI(x), see [11] interface. 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 even more for a success of dynamism in traditional HPC workloads.

5 Conclusion

In this paper, we first presented the idea of self-unfolding jobs. From regular batch executions leading to workflows relying on the file system and thus moving data through most of the memory hierarchy, we posed the question of how to alleviate this bottleneck. Acknowledging existing work in in-situ and workflows, we pose the questions of defining a compact runtime and more precisely mapping syntax to ease the expression of more horizontal HPC payloads – collocating multiple programs in the allocation. In particular, we motivated this approach with a memory bandwidth benchmark clearly outlining that locality requirements can lead to important performance impacts. Hence, 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 in a dedicated simulator, illustrating its use at `https://dynamic-resource.github.io/project/grammar/`. Eventually, we discussed some use cases for this syntax, illustrating the corresponding job layout and syntax's compactness. This work is an initial step towards scaling of HPC payloads under dynamism with all the aforementioned benefits, and much is to be done to eventually enable ideas long devised in other fields such as in-situ and workflows.

6 Future Work

Malleability[1], moldability[13], workflows[14], and in-situ[7] are all known and active research topics in HPC. However, despite their advantages, they are not yet leveraged in HPC payloads. Meanwhile, the complexity of parallel hardware with resource specialization may vanish in a need for symmetrical software specialization – finally unfolding ideas explored in the aforementioned fields. Yet, this evolution is transversal to the whole HPC stack, changing how programs run, and how they unfold requires new practices, if not a revolution for HPC. In this paper we have only scratched the surface of the overall issue, only defining potentially self-unfolding workflows and associated mapping capabilities. In particular, the present paper only covers the cases of resource requests without providing any other information such as how the program would perform on the new resources in an implicit way[15, 18]. We see requirements for both resource allocations and also chances of them co-existing if designed in the right way. It appears clearly that an effort transversal to all the relevant HPC standards (PMIx, MPI, OpenMP) and tools are needed to make this shift. First to demonstrate, quantitatively the advantages of this model and then to progressively open users to new programming primitives enabling more horizontal programming.

7 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.

References

1. 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)
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: High Performance Computing. ISC High Performance 2022 International Workshops: Hamburg, Germany, May 29–June 2, 2022, Revised Selected Papers. pp. 206–217. Springer (2023)
3. 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)

4. Colonnelli, I., Cantalupo, B., Merelli, I., Aldinucci, M.: Streamflow: cross-breeding cloud with hpc. IEEE Transactions on Emerging Topics in Computing **9**(4), 1723–1737 (2020)
5. Crusoe, M.R., Abeln, S., Iosup, A., Amstutz, P., Chilton, J., Tijanić, N., Ménager, H., Soiland-Reyes, S., Gavrilović, B., Goble, C., et al.: Methods included: Standardizing computational reuse and portability with the common workflow language. Communications of the ACM **65**(6), 54–63 (2022)
6. Denoyelle, N., Goglin, B., Ilic, A., Jeannot, E., Sousa, L.: Modeling large compute nodes with heterogeneous memories with cache-aware roofline model. In: High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 8th International Workshop, PMBS 2017, Denver, CO, USA, November 13, 2017, Proceedings 8. pp. 91–113. Springer (2018)
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. Goglin, B., Jeannot, E., Mansouri, F., Mercier, G.: Hardware topology management in mpi applications through hierarchical communicators. Parallel Computing **76**, 70–90 (2018)
10. Hoefler, T., Jeannot, E., Mercier, G.: An overview of process mapping techniques and algorithms in high-performance computing. High Performance Computing on Complex Environments pp. 75–94 (2014)
11. 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). https://doi.org/10.1145/3555819.3555856, `https://dl.acm.org/doi/10.1145/3555819.3555856`
12. 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)
13. 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 **85**, 1–12 (2019)
14. 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)
15. Özden, T., Beringer, T., Mazaheri, A., Fard, H.M., Wolf, F.: Elastisim: A batch-system simulator for malleable workloads. In: Proceedings of the 51st International Conference on Parallel Processing. pp. 1–11 (2022)
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)

18. Wu, X., Marathe, A., Jana, S., Vysocky, O., John, J., Bartolini, A., Riha, L., Gerndt, M., Taylor, V., Bhalachandra, S.: 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)