Executorlib – Up-scaling Python workflows for hierarchical heterogenous high-performance computing

Jan Janssen [1,2], **Michael Gilbert Taylor** [2], **Ping Yang** [2], **Joerg Neugebauer** [1], **and Danny Perez** [2]

1 Max Planck Institute for Sustainable Materials, Düsseldorf, Germany **2** Los Alamos National Laboratory, Los Alamos, NM, United States of America

Summary

Executorlib enables the execution of hierarchical Python workflows on heterogenous computing resources of high-performance computing (HPC) clusters. This is achieved by extending the Executor class of the Python standard library for asynchronously executing callables with an interface to HPC job schedulers. The initial release of Executorlib supports the Simple Linux Utility for Resource Management (SLURM) and the flux framework as HPC job schedulers to start Python processes with dedicated computing resources such as CPU cores, memory, or accelerators like GPUs. For heterogenous workflows, Executorlib enables the use of parallel computing frameworks like the message passing interface (MPI) or of dedicated GPU libraries on a per workflow step basis. Python workflows can be up-scaled with Executorlib from a laptop up to the latest Exascale HPC clusters with minimal code changes including support for hierarchical workflows.

Statement of Need

The convergence of artificial intelligence (AI) and high-performance computing (HPC) workflows (Silva et al., 2021) is one of the key drivers for the rise of Python workflows for HPC. To avoid intrusive code changes, interfaces to performance critical scientific software packages were traditionally implemented using file-based communication and control shell scripts, leading to poor maintainability, portability, and scalability. This approach is however losing ground to more efficient alternatives, such as the use of direct Python bindings, as their support is now increasingly common in scientific software packages and especially machine learning packages and AI frameworks. This enables the programmer to easily express complex workloads that require the orchestration of multiple codes. Still, Python workflows for HPC also come with challenges, like (1) safely terminating Python processes, (2) controlling the resources of Python processes and (3) the management of Python environments (Straßel et al., 2020). The first two of these challenges can be addressed by developing strategies and tools to interface HPC job schedulers such as SLURM (Jette et al., 2002) with Python in order to control the execution and manage the computational resources required to execute heterogenous HPC workflows. A number of Python workflow frameworks have been developed for both types of interfaces, ranging from domain-specific solutions for fields like high-throughput screening in computational materials science, e.g., fireworks (Jain et al., 2015), pyiron (Janssen et al., 2019), and aiida (Huber et al., 2020), to generalized Python interfaces for job schedulers, e.g., myqueue (Mortensen et al., 2020) and PSI/j (Hategan-Marandiuc et al., 2023), and task scheduling frameworks that implement their own task scheduling on top of the HPC job scheduler, e.g., dask (Rocklin, 2015), parsl (Babuji et al., 2019), and jobflow (Rosen et al., 2024). While these tools can be powerful, they introduce new constructs unfamiliar to most Python developers, adding complexity and creating a barrier to entry.

Features and Implementation

To address this limitation while at the same time leveraging the powerful and novel hierarchical HPC resource managers like flux framework ([Ahn et al., 2014](#)), we introduce Executorlib, which instead leverages and naturally extends the familiar Executor interface defined by the Python standard library from single-node shared-memory operation to multi-node distributed operation on HPC platforms. [Figure 1](#) illustrates the internal functionality of Executorlib.
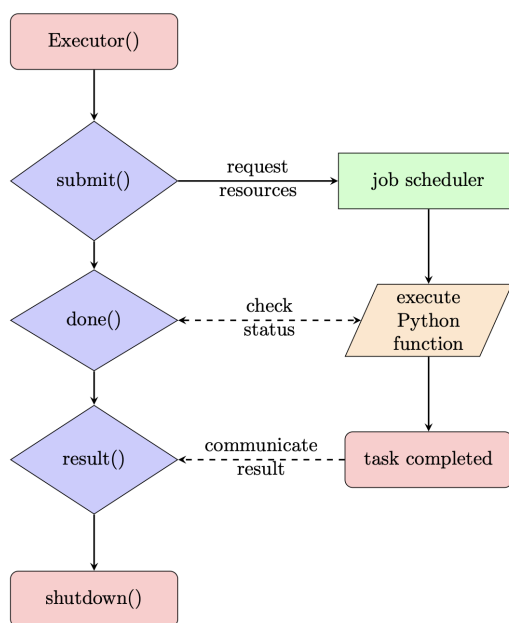

Figure 1: Illustration of the communication between the Executorlib Executor, the job scheduler and the Python process to asynchronously execute the submitted Python function (on the right).

Currently, Executorlib supports five different job schedulers implemented as different Executor classes. The first is the `SingleNodeExecutor` for rapid prototyping on a laptop or local workstation in a way that is functionally similar to the standard `ProcessPoolExecutor`. The second, `SlurmClusterExecutor`, submits Python functions as individual jobs to a SLURM job scheduler using the `sbatch` command, which can be useful for long-running tasks, e.g., that call a compute intensive legacy code. The third is the `SlurmJobExecutor`, which distributes Python functions in an existing SLURM job using the `srun` command. Analogously, the `FluxClusterExecutor` submits Python functions as individual jobs to a flux job scheduler and the `FluxJobExecutor` distributes Python functions in a flux job. Given the hierarchial approach of the flux scheduler there is no limit to the number of `FluxJobExecutor` instances which can be nested inside each other to construct hierarchical workflows.

To assign dedicated computing resources to individual Python functions, the Executorlib Executor classes extend the submission function `submit()` to support not only the Python function and its inputs, but also a Python dictionary specifying the requested computing resources, `resource_dict`. The resource dictionary can define the number of compute cores, number of threads, number of GPUs, as well as job scheduler specific parameters like the working directory, maximum run time or the maximum memory. With this hierarchical approach, Executorlib allows the user to finely control the execution of each individual Python function, using parallel communication libraries like the Message Passing Interface (MPI) for Python ([Dalcín et al., 2005](#)) or GPU-optimized libraries to aggressively optimize complex compute intensive tasks of heterogenous HPC that are best solved by tightly-coupled parallelization approaches, while offering a simple and easy to maintain approach to the orchestration of many

such weakly-coupled tasks. This ability to seamlessly combine different programming models again accelerates the rapid prototyping of heterogenous HPC workflows without sacrificing performance of critical code components.

Usage To-Date

While initially developed in the US DOE Exascale Computing Project's Exascale Atomistic Capability for Accuracy, Length and Time (EXAALT) to accelerate the development of computational materials science simulation workflows for the Exascale, Executorlib has since been generalized to support a wide-range of backends and HPC clusters at different scales. Based on this generalization, it is also been implemented in the pyiron workflow framework (Janssen et al., 2019) as the primary task scheduling interface.

Additional Details

The full documentation including a number of examples for the individual features is available at executorlib.readthedocs.io with the corresponding source code at github.com/pyiron/executorlib. Executorlib is developed an as open-source library with a focus on stability.

Acknowledgements

References

Ahn, D. H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., & Schulz, M. (2014). Flux: A next-generation resource management framework for large HPC centers. *2014 43rd International Conference on Parallel Processing Workshops*, 9–17. https://doi.org/10.1109/ICPPW.2014.15

Babuji, Y., Woodard, A., Li, Z., Katz, D. S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J. M., Foster, I., Wilde, M., & Chard, K. (2019). Parsl: Pervasive parallel programming in Python. *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 25–36. https://doi.org/10.1145/3307681.3325400

Dalcín, L., Paz, R., & Storti, M. (2005). MPI for Python. *Journal of Parallel and Distributed Computing*, *65*(9), 1108–1115. https://doi.org/10.1016/j.jpdc.2005.03.010

Hategan-Marandiuc, M., Merzky, A., Collier, N., Maheshwari, K., Ozik, J., Turilli, M., Wilke, A., Wozniak, J. M., Chard, K., Foster, I., Silva, R. F. da, Jha, S., & Laney, D. (2023). PSI/j: A portable interface for submitting, monitoring, and managing jobs. *2023 IEEE 19th International Conference on e-Science (e-Science)*, 1–10. https://doi.org/10.1109/e-Science58273.2023.10254912

Huber, S. P., Zoupanos, S., Uhrin, M., Talirz, L., Kahle, L., Häuselmann, R., Gresch, D., Müller, T., Yakutovich, A. V., Andersen, F. F., Casper W. Ramirez, Adorf, C. S., Gargiulo, F., Kumbhar, S., Passaro, E., Johnston, C., Merkys, A., Cepellotti, A., Mounet, N., Marzari, N., … Pizzi, G. (2020). AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance. *Scientific Data*, *7*(1). https://doi.org/10.1038/s41597-020-00638-4

Jain, A., Ong, S. P., Chen, W., Medasani, B., Qu, X., Kocher, M., Brafman, M., Petretto, G., Rignanese, G.-M., Hautier, G., Gunter, D., & Persson, K. A. (2015). FireWorks: A dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, *27*(17), 5037–5059. https://doi.org/10.1002/cpe.3505

Janssen, J., Surendralal, S., Lysogorskiy, Y., Todorova, M., Hickel, T., Drautz, R., & Neugebauer, J. (2019). Pyiron: An integrated development environment for computational materials science. *Computational Materials Science*, *163*, 24–36. https://doi.org/10.1016/j.commatsci.2018.07.043

Jette, M. A., Yoo, A. B., & Grondona, M. (2002). SLURM: Simple Linux utility for resource management. *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, 44–60. https://doi.org/10.1007/10968987_3

Mortensen, J. J., Gjerding, M., & Thygesen, K. S. (2020). MyQueue: Task and workflow scheduling system. *Journal of Open Source Software*, *5*(45), 1844. https://doi.org/10.21105/joss.01844

Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. In K. Huff & J. Bergstra (Eds.), *Proceedings of the 14th Python in science conference* (pp. 126–132). https://doi.org/10.25080/Majora-7b98e3ed-013

Rosen, A. S., Gallant, M., George, J., Riebesell, J., Sahasrabuddhe, H., Shen, J.-X., Wen, M., Evans, M. L., Petretto, G., Waroquiers, D., Rignanese, G.-M., Persson, K. A., Jain, A., & Ganose, A. M. (2024). Jobflow: Computational workflows made simple. *Journal of Open Source Software*, *9*(93), 5995. https://doi.org/10.21105/joss.05995

Silva, R. F. da, Casanova, H., Chard, K., Altintas, I., Badia, R. M., Balis, B., Coleman, T., Coppens, F., Di Natale, F., Enders, B., Fahringer, T., Filgueira, R., Fursin, G., Garijo, D., Goble, C., Howell, D., Jha, S., Katz, D. S., Laney, D., … Wolf, M. (2021). A community roadmap for scientific workflows research and development. *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 81–90. https://doi.org/10.1109/WORKS54523.2021.00016

Straßel, D., Reusch, P., & Keuper, J. (2020). Python workflows on HPC systems. *2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, 32–40. https://doi.org/10.1109/PyHPC51966.2020.00009