Automating and Scaling Task-Level Parallelism of Tightly Coupled Models via Code Generation

Mehdi Roozmeh[0000−0002−1086−7578] and Ivan Kondov[0000−0002−8794−616X]

Steinbuch Centre for Computing, Karlsruhe Institute of Technology,
Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany
{mehdi.roozmeh, ivan.kondov}@kit.edu

Abstract. Tightly coupled task-based multiscale models do not scale when implemented using a traditional workflow management system. This is because the fine-grained task parallelism of such applications cannot be exploited efficiently due to scheduling and communication overheads. Existing tools and frameworks allow implementing efficient task-level parallelism, however with high programming effort. On the other hand, Dask and Parsl are Python libraries for low-effort up-scaling of task-parallel applications but still require considerable programming effort and do not equally provide functions for optimal task scheduling. By extending the wfGenes tool with new generators and a static task graph scheduler, we enhance Dask and Parsl to tackle these deficiencies and to generate optimized input for these systems from a simple application description and enable rapid design of scalable task-parallel multiscale applications relying on thorough graph analysis and automatic code generation. The performance of the generated code has been analyzed by using random task graphs with up to 10,000 nodes and executed on thousands of CPU cores. The approach implemented in wfGenes is beneficial for improving the usability and increasing the exploitation of existing tools, and for increasing productivity of multiscale modeling scientists.

Keywords: scientific workflow · tightly coupled model · task-based parallelism · code generation · scalability · productivity

1 Introduction

Exascale computing has a high potential for multiscale simulation in computational nanoscience. Due to the limited physical scalability and the large number of instances of sub-models, as well as the complexity of the couplings between sub-models of different scales, exploiting exascale computing in this domain is still a challenge. Particularly difficult is the rapid and scalable design of novel *tightly coupled* multiscale applications for which the domain scientists urgently need tools that facilitate the rapid integration of sub-models while maintaining high scalability and efficiency of the produced applications.

For *loosely coupled* multiscale applications, scientific workflows and workflow management systems (WMSs) have been established solutions. Thereby, the underlying models are usually wrapped by scripts or Python functions and so integrated as nodes and tasks into a workflow, while the couplings are represented

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

by dataflow links. The workflow nodes of such applications can be executed as separate jobs on one or more high performance computing (HPC) clusters.

Recently, we have demonstrated [16] that workflows for different WMSs can be automatically generated from a single abstract description, WConfig, and have provided a proof-of-concept implementation in the wfGenes tool [11]. The WConfig input file, written in JSON or YAML format, is a simple description of a scientific workflow, that specifies in an arbitrary order all functions with their input parameters and returned objects by unique global names. The wfGenes tool has been first employed in the settings of a collaborative project [1] in which the participants use two different WMSs, FireWorks [10] and SimStack [2], to perform multiscale simulation in nanoscience. This often requires defining a set of simulation workflows in two different languages simultaneously. Previously, we have shown how to generate a workflow to compute the adsorption free energy in catalysis for these two WMSs by using wfGenes [16]. Such a workflow is prototypical for *loosely coupled* applications, where the number of data dependencies is small in relation to the average execution time of single tasks. However, there is another class of multiscale models [6,8] in which the ratio between the number of data dependencies and the average task execution time is very high. These *tightly coupled* models are usually implemented in a program running as a single job on an HPC cluster in order to minimize the task scheduling overheads and the times for data transfers between dependent tasks.

In previous work [8], we implemented and optimized a tightly coupled multiscale model describing charge and exciton transfer in organic electronics. Thereby, we used the Python language to integrate electronic structure codes, such as Turbomole and NWChem, and the mpi4py package [7] to schedule the tasks and data transfers. We found that the use of Python greatly facilitated the adoption of the Message Passing Interface (MPI) through mpi4py as well as the integration of simulation codes in the domain of computational materials science that are provided as Python application programming interfaces. Nevertheless, we found that this approach had several disadvantages: i) The development effort was high due to lack of specific semantics for modeling application's parallelism. ii) The task graph and the task execution order had to be produced manually before starting the application. iii) The lack of domain-specific semantics prevented code reuse in other applications. The developer had to repeat the cumbersome procedure "from scratch" to design new task-parallel applications using that approach. More recently, Dask [15] and Parsl [3,4] have provided semantics for writing implicitly parallel applications in Python by decorating particular objects. While being powerful for rapid design of parallel applications in Python, Parsl and Dask have not been designed as typical WMSs from their outset.

In this work, we extend wfGenes to generate Python input code for Parsl and Dask starting from an existing workflow description. This is extremely beneficial for the use cases where the workflow model is not available as a graph and/or the developer is familiar neither with these tools nor with Python. We find that the newly integrated task graph scheduler works with two different executor strategies of Dask and Parsl, lazy evaluation and immediate execution, respectively,

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

and enables maximum level of parallelism while preserving the functionality of generated code. In the next Section 2, we provide an overview of related work. In Section 3, we provide some implementation insights into the new features, the generators for Dask and Parsl, and the static task graph scheduler. We employ the thus extended wfGenes tool to generate executable Python code from task graphs representative for a tightly coupled application and measure the parallel performance in Section 4. In Section 5 we summarize the paper.

2 Related Work

Numerous frameworks, tools and WMSs can be employed in task-based multiscale computing. In principle, dedicated environments, such as the domain-specific Multiscale Modeling and Simulation Language (MMSL) and the Multiscale Coupling Library and Environment (MUSCLE) can be adopted for any applications of multiscale modeling and computing applications, in particular for solving tightly coupled problems (see Ref. 18 and the references therein). In this work, we pursue a more general concept allowing to use the same tools also in a high-throughput computing context [17] and to address additional domain-specific requirements from computational nanoscience and virtual materials design. Therefore, we identify task-based parallel computing as a common concept in this more general context of usage.

Task-based parallel applications can be implemented in many different ways. The most preferred approach to do this in HPC is based on established standards such as MPI and OpenMP. The MPI standard [12] is only available for C and Fortran, and for other languages available via third-party libraries, e.g. Boost [5] for C++ and mpi4py [7] for Python. Although MPI provides a powerful interface allowing to implement any kind of parallelism, there are no specific definitions to support task-based parallelism directly. Starting from version 3, OpenMP [14] provides support for task-based parallelism based on compiler directives. However, OpenMP only supports shared-memory platforms and C, C++ and Fortran languages and requires support by the corresponding compiler.

There are domain-specific languages for writing task-based parallel computing applications. Swift [19] is a domain-specific language enabling concurrent programming to exploit task and data parallelism implicitly rather than describing the workflow as a static directed acyclic graph. A Swift compiler translates the workflow for different target execution backends. For instance, Swift/T [20] provides a Swift compiler to translate code for the dataflow engine Turbine [21] that uses the asynchronous dynamic load balancer (ADLB) based essentially on MPI. Another domain-specific language, Skywriting [13] also provides semantics for task parallelism and for handling dataflow.

Starting from version 3.2, the Python standard library provides the `concurrent` package to facilitate task parallelism through an abstract interface allowing asynchronous execution of the same Python code on different backends, e.g. implemented in the `multiprocessing` and `mpi4py` [7] packages. However,

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

these can be regarded as execution engines for tasks that have to be scheduled according to their data dependencies and resource requirements.

3 Implementation

3.1 wfGenes architecture

The wfGenes implementation has been described in detail in previous work [16]. For a better understanding of the new features introduced below, we will here briefly outline the basic stages that are shown in Figure 1. First, the model designer authors an abstract workflow description (WConfig) in that all details about the concrete backend system implementation are left out. After that, the WConfig is validated and analyzed, and a task graph of the application is created including the dependencies describing the control flow and the dataflow. In addition, wfGenes extracts various properties of a directed acyclic graph relying on multi-level analysis in the following two steps. First, a thorough dependency analysis is performed using join operations over the input/output lists of the nodes. Second, the depth-level–breadth structure of the graph, discussed in Section 4, is measured by counting the dependent partitions of the graph, the depth levels, and total number of parallel tasks in each depth level, called breadth. Finally, a workflow in the language of the target backend system is automatically generated and validated against a schema provided from the target backend system. In our previous work [16], generators for two different backend systems, FireWorks [10] and SimStack [2], have been implemented as a proof of concept.

3.2 Task-level parallelism

The graph analysis stage, shown in Figure 1, includes a node-level global dependency analysis and a task-level local dependency analysis that enable graph-aware code generation in wfGenes [16]. Combining the results of these two analyses allows automated optimization of the granularity of the workload towards increasing the degree of parallelism of the generated input code. This is carried out through a transformation of the workflow graph as depicted in Figure 2. The original WConfig describes two logical nodes, A and B, that are strictly sequential, each containing a group of tasks that can be scheduled in parallel, as shown in Figure 2a. Now, if there is no need to schedule Node A and Node B on different resources, for example on different computing clusters or in different batch jobs, then Tasks 4 and 5 can be scheduled much earlier, as shown in Figure 2b. In this way, the degree of parallelism can be increased by automatically replacing the node-level with task-level granularity without having to modify the original user input in WConfig. The maximum degree of parallelism that can be realized in Figure 2a is two while it is three in Figure 2b. It is noted that the order of execution, shown in Figure 2b as rows from top to bottom, is for the maximum degree of parallelism that can be achieved if sufficient computing resources are provided.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

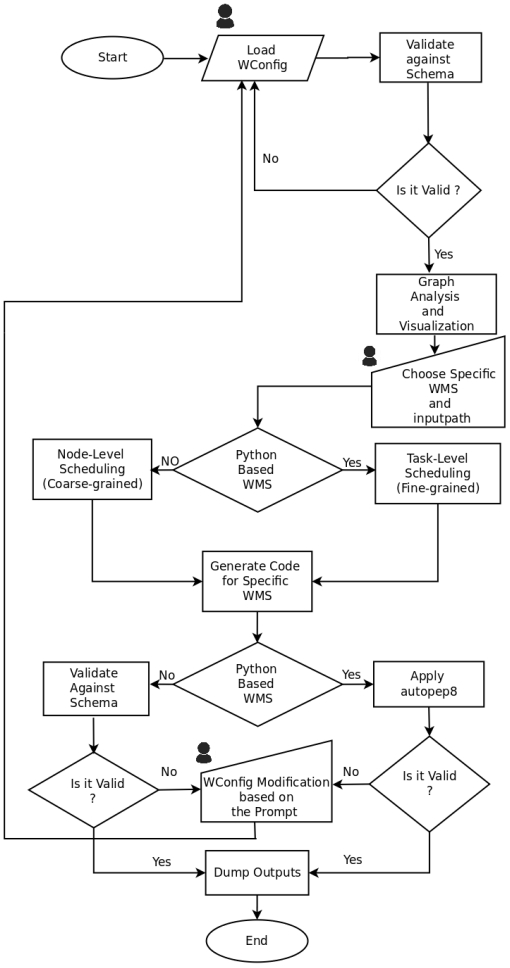
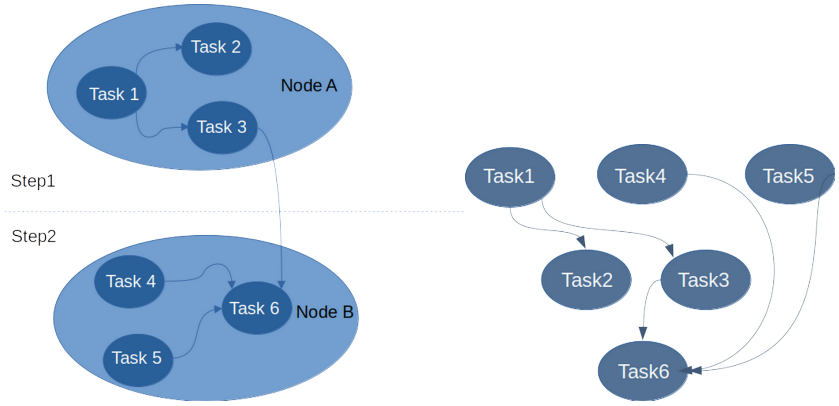


Fig. 1: Flowchart of wfGenes process stages

3.3 Task scheduling

Parsl and Dask resemble each other in many different aspects, however, the differences become more apparent with the task scheduling. A proper scheduling strategy for any task-parallel application relies on the task graph. Both Dask and Parsl are equipped with mechanisms to extract the task graph of the application model from decorated Python code. Although these decorators have different syntax, they both enable an underlying scheduler to perform static analysis and to direct the execution toward parallel computing. While the stages of task graph generation and the task scheduling are logically distinct, they are not well separated in time. In particular, Parsl schedules tasks *on the fly*, i.e. during the

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6



(a) Node Level Parallelism (b) Task Level Parallelism

Fig. 2: Transforming the workflow granularity in order to exploit the maximum possible parallelism.

scheduling and execution stages. In other words, Parsl's executor performs immediate evaluation of the delayed execution objects (futures) when the `result()` method is called and blocks further computation until all tasks within the call are completed before it proceeds with tasks defined further in the python code after this call. This behavior may lead to unwanted idle workers since the scheduler cannot launch any tasks after this barrier before synchronization. On the other hand, Dask performs lazy evaluation of deferred execution objects after constructing the relevant portion of the task graph by applying the `compute()` method to these objects. This strategy is problematic for computations with task graphs that evolve at run time, i.e. dynamic workflows. In particular, Dask lazy evaluation objects cannot be used in loop boundaries or in conditional statements. These problems are resolved in wfGenes by ordering the task calls in a way i) to avoid unnecessary barriers when using Parsl and ii) to obtain results of delayed execution objects when they are needed to dynamically branch the data flow when using Dask. This is done in the generation phase using the task graph produced by the newly integrated task graph scheduler.

3.4 Task graph scheduler

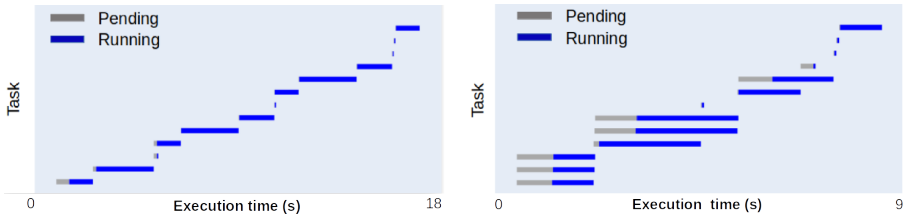
To circumvent these scheduling and dependency handling issues, we have extended wfGenes with an independent task graph scheduler that is switched before the code generator, as is shown in Figure 1. The task ordering and grouping technique implemented in the task scheduler enables Parsl's executor to launch the maximum number of independent tasks in parallel and avoids unnecessary barriers during execution.

Building the task graph prior to code scheduler allows exploiting fully the task parallelism. In the case of Python code generated for Dask, the wfGenes'

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

task scheduler is not necessary since Dask extracts the dependency information from the Python code with no additional effort from developer's side. Here, we use Dask's first generation scheduler which is based on lazy evaluation of the task graph prior to execution. The second generation of Dask executors uses immediate execution of callables that resembles the strategy for Parsl used in this work. Nevertheless, for execution of dynamic workflows, the two tools offer no low-effort solution since the user must implement explicit barriers to assure synchronization.

Figure 3 shows the execution patterns of two different versions of Python code produced by wfGenes for Parsl with disabled and enabled wfGenes' task graph scheduler. In the case of disabled scheduler (Figure 3a) almost all tasks are executed sequentially. With enabling the scheduler (Figure 3b) independent tasks are run in parallel. Although there is some overhead leading to increased pending times, the total running time is about twice shorter. This demonstrates the benefit from wfGenes' task graph scheduler in terms of improved parallel speedup when code for Parsl is generated.



(a) wfGenes scheduler disabled (b) wfGenes scheduler enabled

Fig. 3: Parallel execution with Parsl of code generated from a task graph including 13 tasks with maximum concurrency of three tasks with wfGenes' task graph scheduler turned off and on. For better visualization, Figure 3b is zoomed by a factor of two.

4 Use case

In this section, we address a use case of a tightly coupled multiscale model that is realized as a fine-granular task-parallel application. The up-scaling of such a multiscale application on HPC clusters is a highly non-trivial scheduling problem as it has been shown for computing charge carrier transport properties in organic semiconductors [8]. The scheduling strategy in Ref. 8 has been implemented with huge effort using the state-of-the-art techniques and tools. Although this application can now be implemented with significantly less effort using Dask and Parsl, that have become available in the mean time [3,4,15], we are showing here an approach with a much better usability, in that the Python code for Dask and Parsl does not have to be manually written but can be generated from a simple workflow description.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

Here we are not going to repeat the specific application in Ref. 8. Instead, we use several randomly generated task graphs that have use-case specific characteristics relevant for the performance and scalability of the task-based multiscale application. These characteristics are the total number of tasks, the number of parallel tasks and the task execution times. Such an approach allows us to generalize the results of our experiments beyond the domains of the available application use cases by varying these parameters. To this end, a random graph generator [9] is used to produce random task graphs that are then converted into valid WConfig workflow descriptions, that in turn are used to generate input for FireWorks, Dask and Parsl.

In order to simulate the execution time, we call a sleep function in every task. As we use random graphs, the ideal completion time depends on the structure of the graph and the maximum number of independent nodes at each depth level. The depth levels are the points in time when one or more nodes are scheduled for execution. The breadth at a depth level is the maximum number of nodes that can be scheduled in parallel. wfGenes inspects the WConfig and reports the graph structure and the breadth at each depth level. For example, the graph used in the measurements has the following depth-level–breadth structure:

```
depth_level-breadth: {'1': 2952, '2': 3890, '3': 1881,
                      '4': 769, '5': 300', '6': 124, '7': 50,
                      '8': 23, '9': 9, '10': 2, '11': 1}
```

This analysis provides the necessary information to estimate the resource requirements and to calculate the ideal completion time of the application

$$T_{\text{total}} = T_{\text{task}} \sum_{d=1}^{D} \lceil \frac{B_d}{P} \rceil \tag{1}$$

where D is the maximum depth level of the graph, B_d is the breadth, i.e. the maximum number of parallel tasks at depth level d, P is the number of available processing elements, that is here the number of CPU cores, and T_{task} is the average task execution time. For calculating the parallel speedup we need the time for running the same task graph sequentially. To this end, we have used T_{task} multiplied by the total number of tasks. This estimated time is in a good agreement with the measured sequential execution time.

4.1 Measurement results

A task graph with 10,000 nodes has been produced and transformed to WConfig as described in the previous section. Afterwards, separate inputs for FireWorks, Parsl and Dask have been generated. The workflows have been executed on different number of processor cores on a node of the HPC systems bwUniCluster (with two Intel Xeon Gold 6230 processors) and HoreKa (with two Intel Xeon Platinum 8368 processors). For every run, the parallel speedup has been calculated by dividing the measured total running time by the total time of sequential execution, i.e. on one worker.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

In Figure 4 the speedup on all 80 hardware threads of a bwUniCluster compute node is shown for different task execution times. Dask shows an almost constant speedup of about 60 for all task execution times due to minimum monitoring effort on parallel workers. In contrast to Dask, FireWorks has a very low speedup of around 7 for the task graph with execution time of 1 s. The speedup is improved with increasing the execution time. This can be explained with the communication overheads due to queries to a remote MongoDB database performed by the FireWorks executor. Very short task execution times become comparable to the times of these queries and the latter limit the scalability of parallel execution with the FireWorks executor. With sufficiently long task execution times, FireWorks' speedup is comparable to that of Dask and Parsl, as shown in Figure 4.

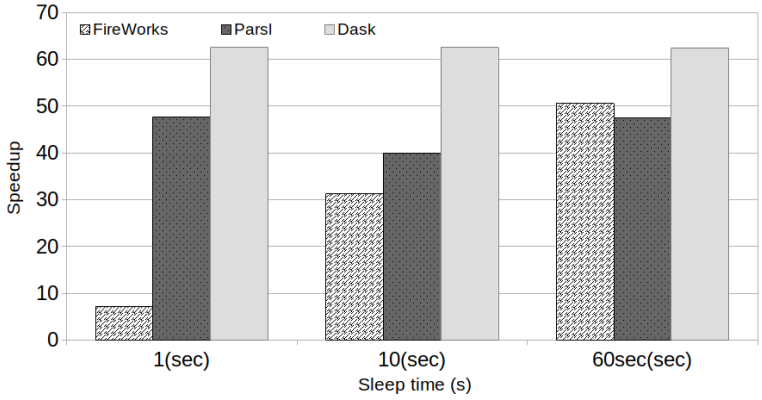


Fig. 4: Parallel speedup of FireWorks, Parsl and Dask for varying task execution times using 80 workers

Furthermore, in Figure 5 we consider the speedup with varying the number of workers. In addition to *balanced* task graphs, in which the task execution times of all tasks are either 1 or 10 s, we consider *unbalanced* task graphs in which 10% of the tasks have long execution times and 90% have short execution times. Mixing two types of tasks with different execution times in similar ratios is prototypical for multiscale task-parallel applications for computing charge carrier transport properties in organic semiconductors [8]. In Figures 5a and 5b, we show the performance for a balanced task graph with 10,000 tasks for 1 and 10 s task execution times, respectively, by measuring the total running times on 1 up to 32 HoreKa compute nodes using 128 workers per compute node. In addition, we measure the performance for unbalanced task graphs: one in which 10% and 90% of the tasks run 10 s and 1 s each, respectively (Figure 5c) and one in which 10% and 90% of the tasks run 100 s and 10 s each, respectively (Figure 5d).

In all measurements, the running time decreases with the number of workers due to parallel execution of independent tasks. After a certain number of

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

workers, e.g. 1000 in Figure 5a, no more time gain can be observed because no more tasks are available for parallel execution. The same limit holds also for the ideal time shown in Figure 5 calculated using Eq. (1). In the cases of balanced workloads, Dask exhibits a slightly better performance than Parsl and a very similar scaling with the number of workers running on up to 32 nodes. In the case of a balanced workload with long task execution time shown in Figure 5b, the measured running times with both Dask and Parsl are virtually the same as the ideal time.

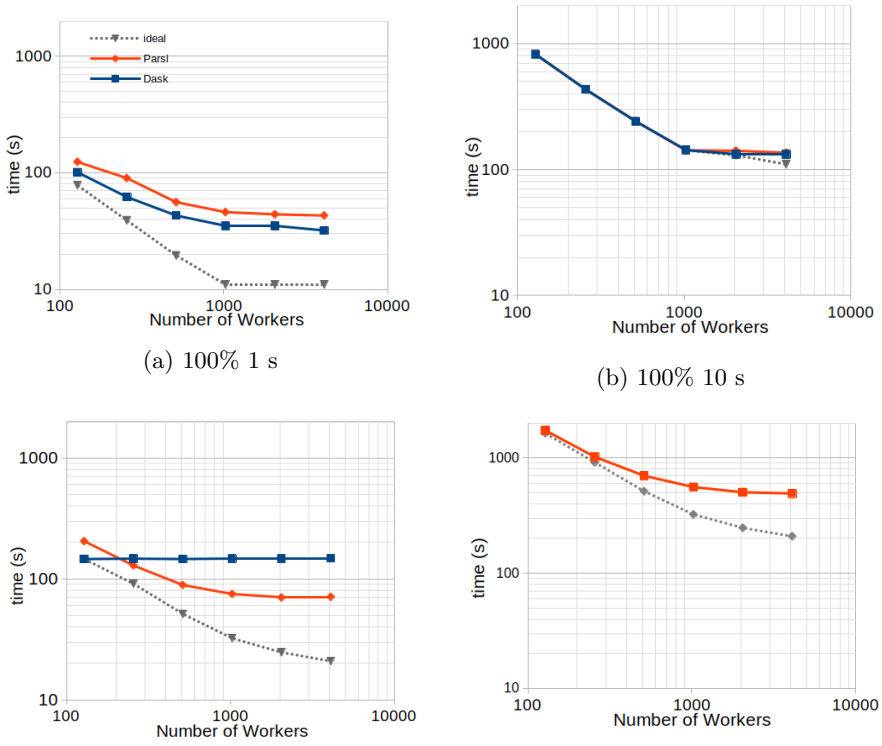
With unbalanced workloads, Parsl shows better performance and overall speedup with respect to Dask. The measured time with Dask for the unbalanced workload in Figure 5c does not improve any more already with 100 workers and gives rise to a flat plateau-like dependence that is due to Dask's executor implementation. Dask's executor cannot scale as good as Parsl due to a local limit of the SLURM workload manager not allowing more than 64 concurrently queued jobs. Therefore, no measurement with Dask is available in Figure 5d. This problem is circumvented in the case of Parsl by submitting a single job allocating several nodes that can be efficiently utilized by Parsl's executor, as depicted in Figure 5d. The workers in Figure 5d are uniformly distributed over up to 256 HoreKa compute nodes while in Figures 5a, 5b and 5c the scaling is performed on up to 64 nodes.

Figure 6 depicts the efficiency (%) of parallelization that is here defined as the ratio of the measured total running time with the ideal time defined in Eq. (1). The best parallel efficiency with both Dask and Parsl is achieved for balanced workloads with long task execution times (10 s) since the communication latency has less impact on the overall performance. Furthermore, Dask has overall better efficiency than Parsl that is more pronounced for the workload with short task execution times (1 s).

With unbalanced workloads, the efficiency is generally reduced presumably due to a load imbalance that cannot be mitigated at run time due to the lack of dynamic load balancing. Communication latency times and scheduling overheads do not seem to be the only reasons for the reduced performance. This can be seen in the measurement of the same unbalanced task graph with ten times larger task execution times (in Figures 6, blue dotted line) for which the parallel efficiency is improved by up to 25% but still 30% lower than the efficiency of the balanced case with the long task execution time. Strikingly, the code for the Parsl executor generated with wfGenes task graph scheduler shows higher efficiency with unbalanced workloads. In contrast, Dask shows poor scalability with the number of workers for the unbalanced workload.

A similar measurement comparing FireWorks, Dask and Parsl has been performed in a recent study [3]. However, every workload used in Ref. 3 includes independent tasks with equal execution times. Here, we investigate the effect of dependencies, using a random task graph, and of the imbalance on the performance and scalability. In addition, the task execution ordering in the generated code for Parsl has been automatically optimized employing the wfGenes task graph scheduler. In the most simple case of a balanced workload our results are

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6



(a) 100% 1 s

(b) 100% 10 s

(c) 90% 1 s, 10% 10 s

(d) 90% 10 s and 10% 100 s

Fig. 5: Completion time of a balanced (a, b) and an unbalanced workload (c, d). The ideal time is calculated by Eq. (1).

in agreement with Ref. 3. In the case of Parsl, the `HighThroughputExecutor` over SLURM has been used offering the high throughput execution model for up to 4000 nodes. In the case of Dask, the SLURM scheduler shows the best performance characteristics for these use cases.

5 Conclusion

Tightly coupled multiscale models often exhibit task-based parallelism with fine granularity. On the other hand, the complexity of the models and of the available tools limits developer's productivity in the design of novel applications and deployment on high performance computing resources. We have addressed these issues by extending the wfGenes tool. To eliminate the coding effort for application developers, we have added support for Parsl and Dask into the generator stage of wfGenes. In order to optimize the scheduling in both Dask and Parsl, we have integrated a task graph scheduler to wfGenes that allows optimal ordering

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

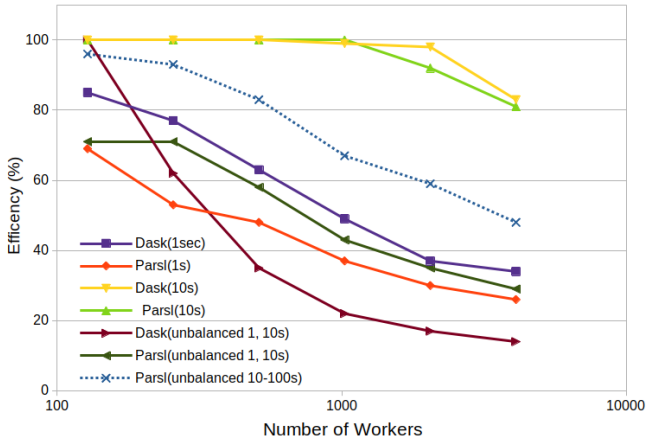


Fig. 6: Parallel efficiency with Dask and Parsl for task graphs with variations of the task execution times

of the Python function calls in the generated Python code, such that enables maximum task parallelism.

We demonstrate the benefits of our approach by generating valid and efficient Python code for Parsl and Dask and executed the produced code on up to 256 computing nodes on an HPC cluster. The performance measurements have shown that the application executed with Dask has better performance and parallel scaling for all balanced workloads whereby all tasks have the same duration. In contrast, Parsl outperforms for unbalanced workloads, in which 10% of the tasks have ten times longer duration than the rest 90% of the tasks. It must be noted that the code for Parsl has been generated employing the wfGenes' internal task graph scheduler. Additionally, it has been shown that the parallelism of the application cannot be sufficiently exploited without wfGenes' task graph scheduler. Our measurements with the FireWorks executor suggest that the performance penalties introduced by the MongoDB queries, necessary for the operation of the executor, are only acceptable for long task execution times (larger than 60 seconds). Therefore, scaling applications with task graphs including more than 10,000 nodes is not recommended for short running tasks using FireWorks. The measurements show that the right choice from Dask, Parsl and FireWorks largely depends on the resource requirements profile of the specific workload, even for the same application.

Our approach is not limited to the demonstrated use case. Rather it can be employed in any use case where a formal description of a tightly coupled simulation is available but the implementation in a concrete WMS or parallelization with Dask and Parsl is not otherwise feasible. For example, by combining the built-in functions `FOREACH` and `MERGE` in WConfig, common dataflow patterns

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

such as map-reduce can be formally described and through wfGenes translated to workflows for different target WMSs, e.g. FireWorks and SimStack, or into Parsl and Dask inputs. In use cases with extensive number of tightly coupled tasks, the end user can benefit from the high performance and scalability of the generated code without having the burden to code the application in the syntax and semantics of the target system. In future work, we will enable embedding of Parsl and Dask scripts into FireWorks workflows, and will exploit useful features, such as memoization and checkpointing in Parsl.

Acknowledgment

The authors gratefully acknowledge support by the GRK 2450. This work was partially performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

References

1. Research Training Group 2450 — Tailored Scale-Bridging Approaches to Computational Nanoscience, https://www.compnano.kit.edu/
2. SimStack: The Boost for Computer Aided-Design of Advanced Materials, https://www.simstack.de/
3. Babuji, Y., Foster, I., Wilde, M., Chard, K., Woodard, A., Li, Z., Katz, D.S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J.M.: Parsl: Pervasive Parallel Programming in Python. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '19. pp. 25–36. ACM Press, Phoenix, AZ, USA (2019). https://doi.org/10.1145/3307681.3325400
4. Babuji, Y., Woodard, A., Li, Z., Katz, D.S., Clifford, B., Foster, I., Wilde, M., Chard, K.: Scalable Parallel Programming in Python with Parsl. In: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning) - PEARC '19. pp. 1–8. ACM Press, Chicago, IL, USA (2019). https://doi.org/10.1145/3332186.3332231
5. Boost: Boost C++ Libraries. http://www.boost.org/ (2020), accessed: 2020-12-18
6. Borgdorff, J., Bona-Casas, C., Mamonski, M., Kurowski, K., Piontek, T., Bosak, B., Rycerz, K., Ciepiela, E., Gubala, T., Harezlak, D., Bubak, M., Lorenz, E., Hoekstra, A.G.: A distributed multiscale computation of a tightly coupled model using the multiscale modeling language. Procedia Computer Science **9**, 596–605 (2012). https://doi.org/10.1016/j.procs.2012.04.064
7. Dalcín, L., Paz, R., Storti, M., D'Elía, J.: MPI for Python: Performance improvements and MPI-2 extensions. Journal of Parallel and Distributed Computing **68**(5), 655–662 (May 2008). https://doi.org/10.1016/j.jpdc.2007.09.005
8. Friederich, P., Strunk, T., Wenzel, W., Kondov, I.: Multiscale simulation of organic electronics via smart scheduling of quantum mechanics computations. Procedia Computer Science **80**, 1244–1254 (2016). https://doi.org/10.1016/j.procs.2016.05.495

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6

9. Haghighi, S.: Pyrgg: Python random graph generator. The Journal of Open Source Software **2**(17) (sep 2017). https://doi.org/10.21105/joss.00331
10. 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.: FireWorks: A dynamic workflow system designed for high-throughput applications. Concurrency and Computation: Practice and Experience **27**(17), 5037–5059 (2015). https://doi.org/10.1002/cpe.3505
11. Mehdi Roozmeh: wfGenes, https://git.scc.kit.edu/th7356/wfgenes, accessed: 2022-01-14
12. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 3.1. Tech. rep., High-Performance Computing Center Stuttgart, Stuttgart, Germany (2015), https://www.mpi-forum.org/docs/mpi-3.1/
13. Murray, D.G., Hand, S.: Scripting the cloud with Skywriting. In: HotCloud'10: Proc. of 2nd USENIX Workshop on Hot Topics in Cloud Computing. USENIX (2010), https://www.usenix.org/legacy/events/hotcloud10/tech/full_papers/Murray.pdf
14. OpenMP Architecture Review Board: OpenMP application program interface version 3.0 (May 2008), http://www.openmp.org/mp-documents/spec30.pdf, accessed: 2020-12-18
15. Rocklin, M.: Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In: Python in Science Conference. pp. 126–132. Austin, Texas (2015). https://doi.org/10.25080/Majora-7b98e3ed-013
16. Roozmeh, M., Kondov, I.: Workflow generation with wfGenes. In: IEEE/ACM Workflows in Support of Large-Scale Science (WORKS). pp. 9–16. Institute of Electrical and Electronics Engineers (IEEE) (2020). https://doi.org/10.1109/WORKS51914.2020.00007
17. Schaarschmidt, J., Yuan, J., Strunk, T., Kondov, I., Huber, S.P., Pizzi, G., Kahle, L., Bölle, F.T., Castelli, I.E., Vegge, T., Hanke, F., Hickel, T., Neugebauer, J., Rêgo, C.R.C., Wenzel, W.: Workflow Engineering in Materials Design within the BATTERY 2030 + Project. Advanced Energy Materials p. 2102638 (Dec 2021). https://doi.org/10.1002/aenm.202102638
18. Veen, L.E., Hoekstra, A.G.: Easing Multiscale Model Design and Coupling with MUSCLE 3. In: Krzhizhanovskaya, V.V., Závodszky, G., Lees, M.H., Dongarra, J.J., Sloot, P.M.A., Brissos, S., Teixeira, J. (eds.) Computational Science – ICCS 2020, vol. 12142, pp. 425–438. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-50433-5_33
19. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: A language for distributed parallel scripting. Parallel Computing **37**(9), 633–652 (2011). https://doi.org/10.1016/j.parco.2011.05.005
20. Wozniak, J.M., Armstrong, T.G., Wilde, M., Katz, D.S., Lusk, E., Foster, I.T.: Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. pp. 95–102. IEEE, Delft (May 2013). https://doi.org/10.1109/CCGrid.2013.99
21. Wozniak, J.M., Armstrong, T.G., Maheshwari, K., Lusk, E.L., Katz, D.S., Wilde, M., Foster, I.T.: Turbine: A Distributed-memory Dataflow Engine for High Performance Many-task Applications. Fundamenta Informaticae **128**(3), 337–366 (2013). https://doi.org/10.3233/FI-2013-949

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08760-8_6