Parsl - Parallel Scripting Library for Python: Benchmarking, Analysis, Expedition & Improvement

Ismail Elomari Alaoui[1]*, Reda Chaguer[1]‘

Abstract
Parsl is a parallel scripting library for Python. It facilitates and improves using multi-threading and parallel scripting in Python. Despite having other competitor libraries Parsl has proven to be one of the fastest, but it is still slow in comparison with other low level libraries. We will benchmark Parsl in numerous use cases and examine the situations where Parsl excels, but also study other cases, such as fine-grained parallelism, where it fails to create a fast, highly-scalable parallel script.
Research Project Github: https://github.com/RChaguer/Parsl-Benchmarking
Original Research Papers: Parsl [1, 2]

Keywords
Distributed systems — Parallel Scripting — High-Scalability — Fine-grained parallelism

[1] *Department of Computer Science, AI Major, IIT, Chicago, USA*
***Corresponding author**: ielomarialaoui@hawk.iit.edu
‘**Corresponding author**: rchaguer@hawk.iit.edu

Contents

1. Introduction

To introduce our work, we will need to firstly introduce and give background information of the library related to our work: Parsl [1, 2].

Parsl can scale analysis from a single node or machine to numerous nodes or a distributed system [3]. It can annotate functions which can be executed in parallel and asynchronously. Thus, it provides easy ways, for developers and programmers, to implement parallelism in Python.

Furthermore, using Parsl to parallelize big tasks is proven to be functional and efficient. However, when dealing with small tasks or what could be expressed as fine-grained parallelism, Parsl's performance drops down immensely.

Thus, we will examine the cases on which these performance drops happen and try to pin the problem which creates this costly drop. Then, we will try to improve Parsl in these particular situations.

As stated in Parsl's original research paper [1], many figures showed that Parsl requires long duration tasks for it to be faster than other libraries, as long tasks keep cores busy thus improving Parsl's performance (see Figure 1).

This clearly illustrates the overhead of task submission Parsl experiences for a more efficient higher scaling. Figure 1 presents the results of a simple benchmark where we compare sequential execution to Parsl's parallel scripts. Just like expected, in when tasks controlled by each thread or process are long, Parsl show a huge boost in performance. However, when these tasks are short/light the sequential code performs extremely better than Parsl's. Configuring some parameters, such as tile size, helps reduce Parsl's slowness in this benchmark, but it is not enough to outperform a simple sequential execution.

Thus, Parsl has a huge lack of performance when run under some precise circumstances, such as fine-grained parallelism (when tasks are small and have short durations).

2. Related work

Parsl is a parallel programming library and an open source project that provides simple, scalable, and flexible constructs for encoding parallelism via scripts that can be run on multiple platforms and in a variety of use cases in Python.

Parsl provides a simple model that can be easily integrated into science gateways to support the management and execution of workflows composed of Python functions and external applications. To manage the execution of potentially complex workflows with arbitrary computational resources, science gateways benefit from the Parsl model's extensibility, scalability,

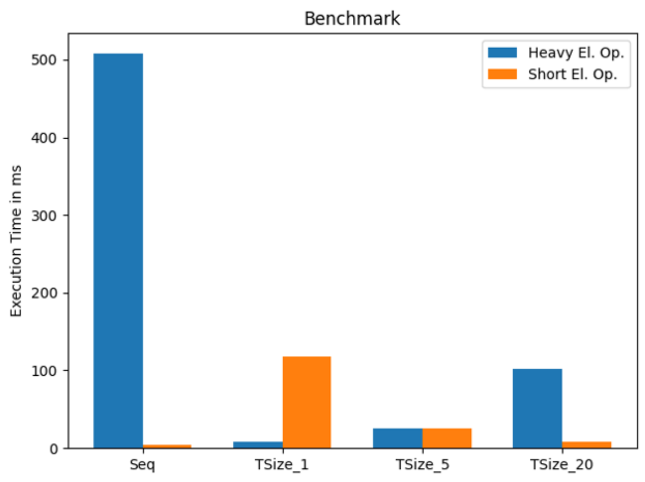


Figure 1. Parsl is weak in fine-grained parallelism

and robustness. Moreover, Parsl was created specifically to address new workflow modalities, such as interactive computing in Jupyter notebooks, and it provides a seamless and transparent way to scale these analyses from within the notebook.

Additionally, the complexity of interacting with various resource fabrics and execution models is abstracted by Parsl. It instead facilitates the creation of resource-independent Python scripts. Furthermore, it also has advanced capabilities such as automated elasticity, multi-site execution support, fault tolerance, and automated direct and wide area data management.

Compared to other solutions like FireWorks [4] and Dask [5], Parsl delivers a more flexible executor model with less restrictions on applications and hardware used. For those reasons, we chose to investigate more on how we can profit from a performance boost and be compared to available solutions in other languages like Swift/T [6] and XTask [7] and explore the nature of performance bottlenecks and overheads that prevents Parsl from running faster.

3. Proposed solution

As noted in the previous paragraphs, Parsl shows bad performance in the case of parallelism on small tasks. In fact, all of the python parallelism libraries have the exact same problem to a certain extent [6, 8, 3], when compared to other languages' similar libraries.

All in all, we know Python is slow in data-intensive computing. However, we will try to create numerous cases and situations and try to focus certain parts of Parsl's source code, in order to figure out the bottlenecks which influence on this library's performance in fine-grained parallelism. If these benchmarks lead us to the core problem, we will then focus on the conception of a different approach which would most definitely improve Parsl's performance in these particular use cases.

In this paper, we will be evaluating our benchmarking by the conception and programming of the most extreme cases which make Parsl very slow, the slower the better. The scenarios and the algorithms we create will target certain parts of Parsl's source code. This would help us thin out other bugs and pin the problem which creates these overheads and bottlenecks. So, we will start by implementing the most basic, simple algorithms: Fibonnacci, primes, sparse matrix multiplication, Planar Convex-Hull. These algorithms are just numerous simple, small tasks done in recursion and would target Parsl's problem in the face of extremely fine-grained parallelism.

By monitoring the function execution time for those applications, we will try to identify the library modules slowing down the execution process and consuming resources. By figuring out the nature of those problems, we can propose some solutions to try minimizing the loss of performance by changing the current architecture or the implementation solution provided by the original researchers.

4. Evaluation

This section presents our work and ideas for benchmarking and analyzing Parsl, a novel parallel scripting library for python, in fine-grained parallelism.

All experiments are performed on multiple devices having different specifications. We mainly used a 6 cores AMD Ryzen 5 4500U and a 16 cores Intel(R) Xeon(R) CPU @ 2.20GHz provided by Google Cloud Services on Fedora and Ubuntu Linux Distributions.

Moreover, we will discuss in section 4.1, how exactly we profiled the programs and benchmarks we conceived and created in order to figure out potential bottlenecks which could be halting Parsl's performance. Additionally, subsection 4.2 presents some tools we used to visualize program profiles and have a clear idea about Parsl's problems.

Subsection 4.3 focuses on our first test to firstly understand parsl's functionality (Python apps and bash apps, futures, ...) It helped us expand, to a certain extent, our understanding of Parsl's terminology and get a firm grasp of Parsl's slow execution when parallelizing small and light-computational tasks.

Additionally, section 4.4 presents numerous benchmarks we conceived and implemented to target certain specific areas of Parsl's code and to validate some of the theories we had on why is Parsl slow in fine-grained parallelism.

Furthermore, paragraph 4.5 summarizes our findings and presents some hypotheses and theories we had about possible bottlenecks which are influencing negatively on Parsl.

Finally, we propose numerous potential solutions, in section 4.6, to some of the problems we discovered.

4.1 Profiling programs

A profile is a set of statistics that describes how often and for how long various parts of the program executed. These statistics will help us deduce areas which could be affecting Parsl's performance in fine-grained parallelism. We used 2 of the most famous python program profilers.

- **cProfile** [9, 10] provides deterministic profiling of Python programs.
- **Yappi: Yet Another Python Profiler** [11] profiles multithreaded Python applications transparently.

cProfile

cProfile is a C extension with reasonable overhead that makes it suitable for profiling long-running python programs.

Multi-threaded program profiling using Yappi

Yappi supports multithread/CPU time profiling, analyze per-thread information, has the ability to hook underlying threading model events/properties and provides decorators to profile individual functions easily.

These tools provide statistics and create profiles (.prof files). These statistics can be formatted into reports via the *pstats* module, and can be visualized and analyzed to figure out which functions and methods are the longest to execute.

4.2 Visualizing program profiles using Snakeviz

SnakeViz [12, 13] is a browser based graphical viewer for the output of Python's cProfile module and an alternative to using the standard library *pstats* module.

During our first test, we profiled a Fibonacci of 5 execution using cProfile and visualized it with SnakeViz. Figure 2 presents the results which gave us clues on potential bottlenecks.

4.3 Understanding Parsl and initial benchmark: Fibonacci

In order to understand how Parsl really works, what happens when we execute a python script parallelized using Parsl, we had to choose a simple function (complex in its overall execution but can be split into numerous small light-computational tasks). We chose Fibonacci. Because we wanted to compare a parallel and a sequential execution, we chose a very low number, specifically Fibonacci of 5.

In figure 2 (see Appendix), we noticed that the total execution time is 7.56 seconds, and one called method named "acquire", of "_thread_lock" objects, takes about 5.008 seconds which is basically more than 70% of the execution time!

All in all, with profilers and SnakeViz, we had access to numerous interesting informations such as the number of calls of each function or method, the total execution time of each function and the execution time per call.

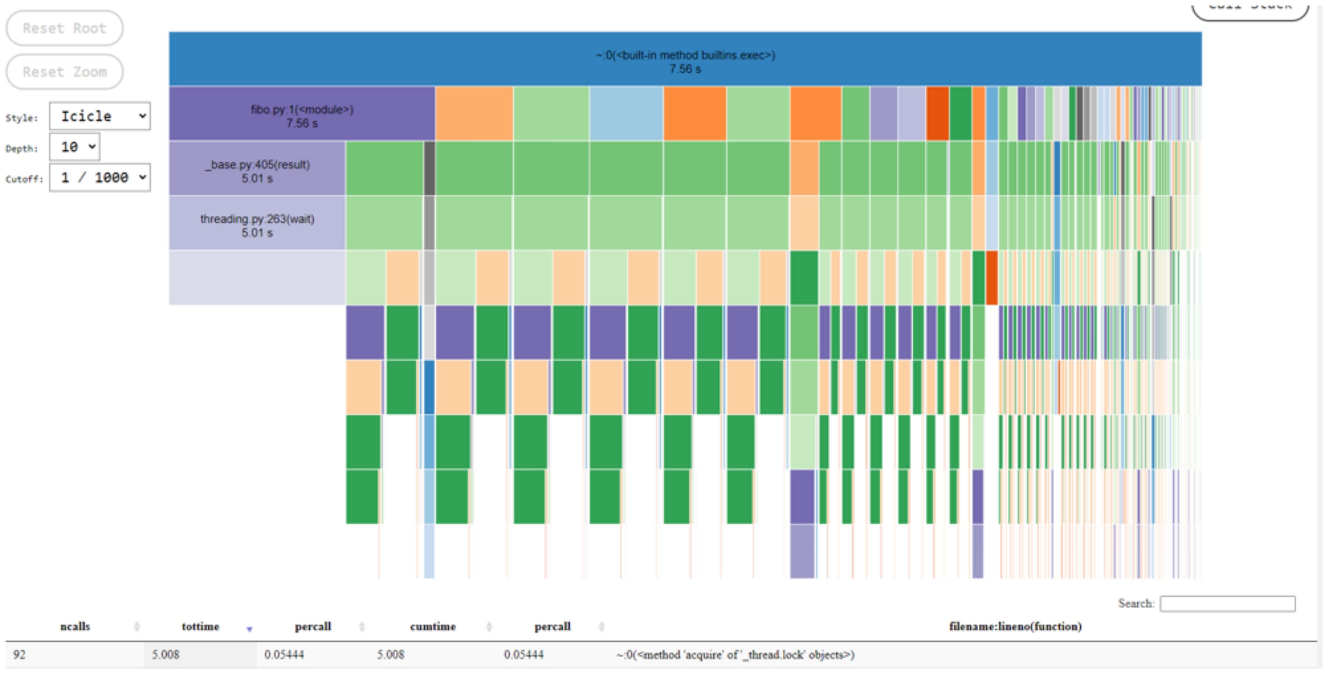
Upon profiling and visualizing numerous tests we realized, when the tasks are small enough, a method named ***acquire*** of ***_thread_lock*** objects takes the longest execution time almost every benchmark. This could be one of the problems halting Parsl's success when dealing with parallelizing light tasks.

4.4 Other Benchmarks

Several Benchmarks were made on two different function approaches. We used as an example for the recursive one Fibonacci and for the iterative one Square Array.

In figure 3, the sequential version performed better than the parallel version which is expected not only in Parsl but also in other libraries and languages. This is due to the additional thread creation cost and the exponential scale of the number of threads and processes related to the input in recursive functions which explains the bad performance of parallel algorithms compared to the sequential versions.

In figure 4, the sequential version, for $10^{-8}s$ tasks, still performs better than the parallel ones for the same reasons we discussed earlier in the Fibonacci version. Moreover, the thread function is better than the multiprocess versions because of the cost of context swapping between workers. This cost is superior



Style: Icicle
Depth: 10
Cutoff: 1 / 1000

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
92	5.008	0.05444	5.008	0.05444	~:0(<method 'acquire' of '_thread.lock' objects>)

Figure 2. Visualizing Fibonacci of 5 profile with SnakeViz

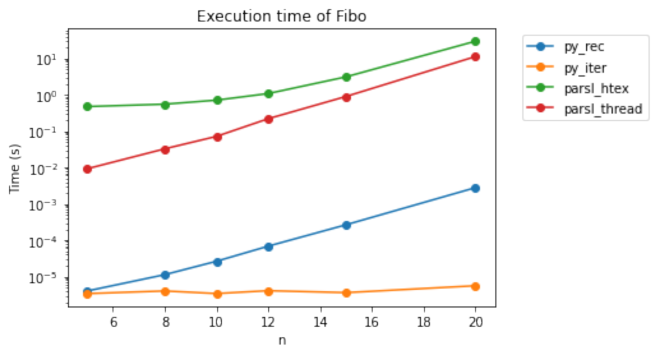


Figure 3. Benchmark : Fibonacci

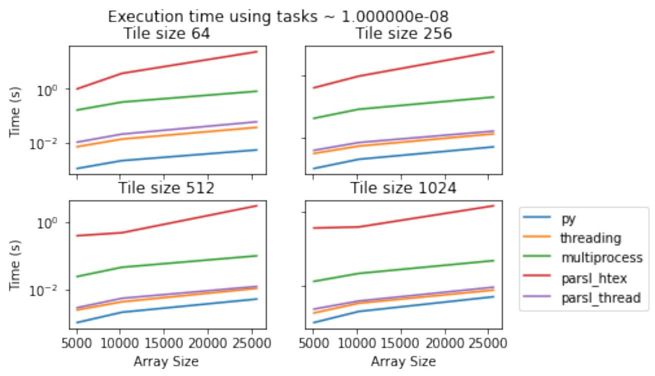


Figure 4. Benchmark : Square Array ($10^{-8}s$ tasks)

than the lost time in thread creation.

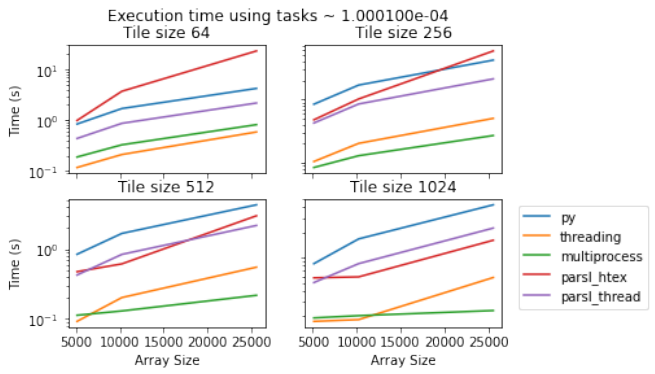


Figure 5. Benchmark : Square Array ($10^{-4}s$ tasks)

By increasing the duration of the elementary task duration, the parallel versions start performing better, especially the threading one as we can see in the figure 5. But as we reach a total duration of $10^{-3}s$ in figure 6, the multiprocess versions outperform threads because the duration of switching context began to be negligible compared to the time lost because of thread locks caused by the GIL in CPython. We also have to consider the lack of performance of Parsl version compared to the basic versions using builtin python parallel libraries. This performance gap can be explained by the extra layers contained in Parsl to add more features such as logging and scalability options using multiple nodes in the `htex` configuration.

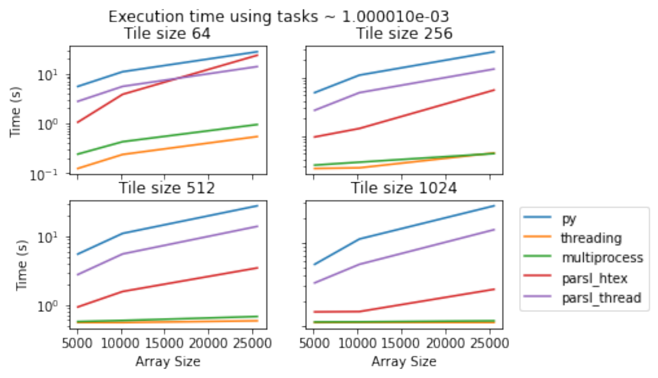


Figure 6. Benchmark : Square Array ($10^{-3}s$ tasks)

4.5 Potential problems and bottlenecks

In this section, we discuss, theorize and validate some potential bottlenecks and problems, which could be halting Parsl's performance. These theories are inspired from the tests, benchmarks and analysis we presented in previous sections. We will try now to give these hypotheses solid validating grounds by targeting actual parts of Parsl's code.

Threading library and GIL [14, 15]

The first problem which caught our attention immensely is the unusually long execution time of the method ***acquire*** of ***_thread_lock*** objects.

Following an extensive research of this method, we figured out it belongs to the ***threading*** library, a library Parsl uses to wrap the program to be parallelized. A quick follow up research gave us additional clues.

threading is a module which constructs higher-level threading interfaces on top of the lower level _thread module. However, according to python's official documentation, there are some very important cpython implementation details, which could be enhancing Parsl's slowness in fine-grained parallelism. The documentation says: *"In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use multiprocessing or concurrent.futures.ProcessPoolExecutor. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously."*

This means the interpretation of the parallelized code is actually sequential. This is exactly why, in heavy-computational tasks, the parallel code can compensate the time lost in this sequential interpretation. However, in fine-grained parallelism, Parsl can not compensate the lost time because tasks are very small, thus each thread finishes the execution very fast.

This is one of our most validated theories throughout our experiments and benchmarks. In order to further understand this bottleneck we needed to continue our research on GIL [14, 15].

All in all, the Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time. The impact of the GIL is not visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code, such as a python or bash script parallelized using Parsl.

4.6 Possible solutions and innovative ideas

This section presents some potential solution which could fix or at least minimize Parsl's bottlenecks in fine-grained parallelism.

GIL

Firstly, if the GIL is Parsl's main problem, we could try using multi-processing instead of multi-threading. Though using processes instead of threads could be determined in Parsl's configurations, we noticed Parsl always wraps the portion of codes to be parallelized in a thread using the ***threading*** module. In order to avoid the GIL bottleneck, we could change Parsl's architecture to use the ***multiprocessing*** or ***concurrent.futures.ProcessPoolExecutor*** module where each Python process gets its own Python interpreter and memory space so the GIL will not be problematic.

However, as we already explained in previous sections, this will not improve Parsl's performance. We actually theorize it will make it worse, especially when tasks are small, because when using processes we waste time swapping contexts repetitively.

A very innovative solution is to conceive and implement a more advanced custom threading/multiprocessing python module, where each thread/process has its own interpreter but the memory is not forked.

PyPy app

Another solution we propose, is to add a PyPy [16, 17] application in Parsl. Parsl has a python app and a bash app. It is possible to add another implementation with other Python interpreters such as PyPy. PyPy [1] is a fast, compliant alternative implementation of Python. We chose it because it is already fast enough on its own, it can sequentially solve recursive Fibonacci of 36 in 0.26 seconds whilst the normal Python interpreter takes 5.86 seconds. Adding a PyPy app to Parsl will be a huge advancement in both fine-grained and more complex heavy-computational parallelism as well.

Real-time monitoring tool

The real difficulty of this project, debugging Parsl in fine-grained parallel computing, lies actually in the lack of real time monitoring tools or options for Parsl or even parallel and multicore programming in Python.

Parsl creates logs of execution and profilers were thoroughly used to create program and benchmark profiles. These profiles were then visualized to simplify their comprehension and understanding and this really helped us get a firm grasp on how Parsl works. Furthermore, this has led us to some potential problems such as GIL, especially when using Parsl's ThreadPool executor. However, these tools are not enough to discover most of Parsl's bottlenecks and problems.

A proper real-time monitoring tool seems benificial not only for this reasearch project but also for boosting Parsl's utility as it could be a great tool for developers as well as clients.

EasyPAP [18] would be a great example of real-time monitoring frameworks. EasyPAP provides an easy-to-use programming environment to learn parallel programming. The idea is to parallelize sequential computations on 2D matrices (which are images, most of the time) over multi-core and GPU platforms. At each iteration, the current matrix can be displayed, allowing to visually check the correctness of the computation method.

Furthermore, EasyPAP provides additional monitoring options where the user can see, in real-time, on what area of the matrix/image each CPU or GPU is working on, the usage percentage and idleness of each processing unit.

This real-time visual aspect would be extremely beneficial for Parsl [1, 2] as well as XTask [7], Xsearch or even Swift/T [6] as it would help developers understand the cause of many limitations and problems, such as load balancing, and conceive innovative ideas and incredible solutions for modern parallel computing and distributed systems' bottlenecks.

5. Conclusion and Future Work

Throughout this project, we gained a deeper understanding of the scientific process and how to be able to develop more advanced research questions and form and test our hypotheses especially in parallel computing and distributed systems fields.

This project can be considered as a success as we managed to find critical problems which affect heavily on the execution time performance of Parsl. Moreover, proposed numerous solutions capable, at least theoretically, of improving Parsl's performance not only in fine-grained parallelism but also in heavy-computational parallel executions. We are convinced some of these solutions, if implemented correctly, would help library users boost performance in their projects and applications.

Future work consists mainly of implementing some or all of the proposed solutions we came up with, thus contribute actively to the open source project. Of course, this would mean a huge change and addition to Parsl's architecture. This is mainly why we could not do this during this research project, especially with the limited time. Additionally, implementing such enormous changes would require a lot of testing in order to firstly validate our hypotheses and then verify that our update did not halt Parsl's excellent achievements in other areas such as heavy-computational parallelism.

Moreover, we could also work, in the future, on enabling ephemeral caching of data on nodes [2] as this would improve Parsl's data management impressively. We find this subject interesting as it will certainly expedite high and low-scaled parallelism.

[1]PyPy is a replacement for CPython. It is built using the RPython language that was co-developed with it. The main reason to use it instead of CPython is speed: it runs generally faster

All in all, future developers could focus on one of the presented conclusions (avoiding GIL bottlenecks, creating PyPy app for Parsl, conception and implementation of a real-time monitoring tool for Parsl or for parallel computing in Python in general). All of these innovative ideas would be a great boost is Parsl's utility overall.

Acknowledgements

This project was conducted by AI major students of Illinois Institute of Technology. We would like to thank Dr. Ioan Raicu, our supervisor for this research project and for the "Data Intensive Computing" course. We are extremely glad for this opportunity to learn about numerous fields outside of our expertise such as parallel computing in Python.

References

[1] Yadu N Babuji, Kyle Chard, Ian T Foster, Daniel S Katz, Mike Wilde, Anna Woodard, and Justin M Wozniak. Parsl: Scalable parallel scripting in python. In *IWSG*, 2018.

[2] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M Wozniak, Ian Foster, et al. Parsl: Pervasive parallel programming in python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 25–36, 2019.

[3] Lisandro D Dalcin, Rodrigo R Paz, Pablo A Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

[4] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, et al. Fireworks: A dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015.

[5] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, page 136. Citeseer, 2015.

[6] Justin M Wozniak, Timothy G Armstrong, Michael Wilde, Daniel S Katz, Ewing Lusk, and Ian T Foster. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 95–102. IEEE, 2013.

[7] DR POORNIMA NOOKALA, DR KYLE HALE, and DR IOAN RAICU. Xtask-extreme fine-grained concurrent task invocation runtime.

[8] TIMOTHY G Armstrong. Integrating task parallelism into the python programming language. *University of Chicago*, 2011.

[9] Finn Årup Nielsen. Python programming—profiling. 2014.

[10] Michael Wagner, Germán Llort, Estanislao Mercadal, Judit Giménez, and Jesús Labarta. Performance analysis of parallel python applications. *Procedia Computer Science*, 108:2171–2179, 2017.

[11] Ami Marowka. On parallel software engineering education using python. *Education and Information Technologies*, 23(1):357–372, 2018.

[12] Ervin Varga. Data visualization. In *Practical Data Science with Python 3*, pages 209–253. Springer, 2019.

[13] Andreas Gocht, Robert Schöne, and Jan Frenzel. Advanced python performance monitoring with score-p. In *Tools for High Performance Computing 2018/2019*, pages 261–270. Springer, 2021.

[14] David Beazley. Understanding the python gil. In *PyCON Python Conference. Atlanta, Georgia*, 2010.

[15] Roger Eggen and Maurice Eggen. Thread and process efficiency in python. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 32–36. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2019.

[16] Guido Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, page 36, 2007.

[17] Yangguang Li. Understanding and optimizing python-based applications-a case study on pypy. 2019.

[18] Alice Lasserre, Raymond Namyst, and Pierre-André Wacrenier. Easypap: A framework for learning parallel programming. *Journal of Parallel and Distributed Computing*, 158:94–114, 2021.