EVALUATING PARSL: FOR IMPROVED SPEEDS IN FINE GRAINED CONCURRENT TASKS

Reddy Anish
Illinois Institute of Technology,
Chicago, IL
aravula@hawk.iit.edu

Naik Pranjal
Illinois Institute of Technology,
Chicago, IL
pnaik13@hawk.iit.edu

Abstract: For a high-level language like Python, Parsl[4] is an excellent parallel scripting library that augments Python with simple, scalable, and flexible constructs for encoding parallelism. Parsl helps us in executing programs concurrently on multiple processors and is very important for High Performance Computing (HPCs). Thus, it makes sense that optimizing the execution of parsl is very important for scientific computing. But we believe that for fine-grained parallel tasks there is high latency and low throughput in completing the tasks. Here, we propose various ways to analyze the components of Parsl and Python in parallel workloads and pinpoint time consuming components using profilers, so that Parsl can be as efficient for fine grained parallel tasks. By analyzing the output of the profilers, we can pinpoint the bottleneck causing function which induces low latency in parallel computing.

Keywords - High Performance Computing [HPC], Parsl

I. Introduction

Parsl is a flexible and scalable parallel programming library for Python. Parsl augments Python with simple constructs for encoding parallelism. Developers annotate Python functions to specify opportunities for concurrent execution. These annotated functions, called apps, may represent pure Python functions or calls to external applications. Parsl further allows invocations of these apps, called tasks, to be connected by shared input/output data (e.g., Python objects or files) via which Parsl constructs a dynamic dependency graph of tasks to manage concurrent task execution where possible.

Parsl programs are portable, enabling them to be easily moved between different execution resources: from laptops to supercomputers. Parsl helps us in executing programs concurrently on multiple processors and is very important for High Performance Computing (HPCs). However, Python is not favored in parallel programming, as it is quite slower than parallel programming implementations in other languages. Thus, we need to pinpoint the exact cause of this slowness.

II. Background

Parallel computing is a type of computing architecture in which several processors simultaneously execute multiple, smaller calculations broken down from an overall larger, complex problem. The primary goal is to increase the utilization of computation power for faster processing of the problem. Swift, a parallel programming system with few lines of SwiftScript to specify computations involving extremely large numbers (tens or hundreds of thousands) of files and tasks, and for those computations to be executed efficiently and reliably on many distributed computers with a throughput of almost 10^3.

Xtask, a runtime system is to enable the execution of fine-grained tasks on shared memory multi-core architectures with very low latency and high throughput of almost 10^6 tasks per second. In this paper, we evaluate parallel computing, using a python library called Parsl to run a parallel workload. The throughput achieved using Parsl is in the order of 10^2 and 10^3 tasks per second. We will try and evaluate the bottlenecks or resource constraints that are preventing it to give similar throughputs as parallel computing solutions provided by other programming languages.

Parsl is designed to run in clouds, clusters and supercomputers but is relatively slow when running fine grained tasks. However, there is latency in its execution and that is why python is not a preferred parallel programming language. Thus, parallel programming implementations in C or C++ is a preferred language.

III. Motivation

Parsl gives a very good and simplistic way to execute programs efficiently on one or many processors. But it is slow in the order of 10^2 to 10^3 when compared to Swift and XTask implementations to execute parallel programs. The problem in throughput of Parsl and is most likely due to high latency in tasks execution - which is highly fine-grained. This project aims to evaluate the reason for this low throughput and high latency.

This is very important for fine-grained problems to execute very quickly like if 10^6 fine-grained tasks with each taking 1 second to execute and 1-millisecond latency to get to the next task then there is an extra 1000 second overhead of execution time if there is a 1-microsecond latency.

Parsl helps us in executing programs concurrently on multiple processors and is very important for High Performance Computing (HPCs). Thus, it makes sense that optimizing the execution of parsl is very important for scientific computing. But we believe that for fine-grained parallel tasks there is high latency and low throughput in completing the tasks. Here, we propose various ways to analyze the components of Parsl and Python in parallel workloads and pinpoint time consuming components using profilers such as Pyinstrument, so that Parsl can be as efficient for fine grained parallel tasks. By analyzing the output of the profilers, we can pinpoint the bottleneck causing function which induces low latency in parallel computing.

IV. Proposed Solution

The project implements a real system for the evaluation of the parallel programming execution in python.

Multiple python programs were executed using parallel programming library of parsl. On every such program, the profilers were tested to check for latencies in the execution.

For the purpose of evaluation, the following programs were evaluated -

- Double.py {Doubles the number}
- Increment.py {Increment a number by 1}
- Fibonacci.py {Calculate sum of n fibonacci number}

Parsl has 2 executors, the HighThroughputExecutor, and ThreadPoolExecutor. These programs were iterated through both these executors for evaluations.

The HighThroughputExecutor uses maximum number of processes. The number of processes used were 24, 48, 96, and 192. The ThreadPoolExecutor uses maximum number of threads. The number of threads used were 24, 48, 96, and 192.

The Pyinstrument profiler was used which gives out a tree structure, that can be viewed in HTML format, It also gives a json file that is used for analysis and plots. It outputs a cumulative execution time in seconds. This profiler is run against multiple programs using mulitple configurations for 2 types of executors.

The package, along with the version needed are mentioned in the requirements.txt file on github, and is also mentioned in the submitted source code.

The snippets from programs are as follows -

The @python_app is a decorator which wraps around the standard Python function calls as mentioned in the snippet. The start and end time of the program is monitored too.

Snippet from increment.py -

```python
import time
import argparse
import parsl
from parsl.app.app import python_app

from executors.all import execs


@python_app
def increment(x):
    return x + 1


@python_app
def slow_increment(x, dur):
    import time

    time.sleep(dur)
    return x + 1


def perform_increment(depth=5):
    futs = {0: 0}
    for i in range(1, depth):
        futs[i] = increment(futs[i - 1])

    x = sum([futs[i].result() for i in futs if not isinstance(futs[i], int)])
    return x


def perform_slow_increment(depth=5):
    futs = {0: 0}
    for i in range(1, depth):
        futs[i] = slow_increment(futs[i - 1], 0.1)

    x = sum([futs[i].result() for i in futs if not isinstance(futs[i], int)])

    return x


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-d", "--num", default="5", action="store", dest="d", type=int)
    parser.add_argument(
        "-e",
        "--exec",
        action="store",
        dest="exec",
        type=str,
    )
    args = parser.parse_args()
    parsl.clear()
    executor = execs.get(args.exec)
    print("Loading executor with config:", executor)
    parsl.load(executor)
    start = time.time()
    print(perform_increment(args.d))
    end = time.time()
    print(end - start)
    start = time.time()
    print(perform_slow_increment(args.d))
    end = time.time()
    print(end - start)
```

Snippet from fibonacci.py -

```python
import argparse
from parsl.app.app import join_app, python_app
from executors.all import execs
from logs.logger import rootLogger

import parsl


@python_app
def add(*args):
    """Add all of the arguments together. If no arguments, then
    zero is returned (the neutral element of +)
    """
    accumulator = 0
    for v in args:
        accumulator += v
    return accumulator


@join_app
def fibonacci(n):
    if n == 0:
        return add()
    elif n == 1:
        return add(1)
    else:
        return add(fibonacci(n - 1), fibonacci(n - 2))


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-d", "--num", default="5", action="store", dest="d", type=int)
    parser.add_argument(
        "-e",
        "--exec",
        action="store",
        dest="exec",
        type=str,
    )
    args = parser.parse_args()
    parsl.clear()
    executor = execs.get(args.exec)
    print("Loading executor with config:", executor)
    parsl.load(executor)
    print(fibonacci(args.d).result())
```

Snippet from double.py -

```python
import time
import argparse
import parsl
from parsl.app.app import python_app

from executors.all import execs
from logs.logger import rootLogger


@python_app
def double(x):
    return x * 2


def parallel_execution(n):
    res = []
    for i in range(n):
        res.append(double(i))
    x = sum([fut.result() for fut in res if not isinstance(fut, int)])
    return x


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-n", "--num", default="10", action="store", dest="n", type=int)
    parser.add_argument(
        "-e",
        "--exec",
        action="store",
        dest="exec",
        type=str,
    )
    args = parser.parse_args()
    parsl.clear()
    executor = execs.get(args.exec)
    print("Loading executor with config:", executor)
    parsl.load(executor)
    start = time.time()
    print(parallel_execution(args.n))
    end = time.time()
    print(end - start)
```

All the above evaluations are automated using python and bash scripts and run on mystic.

The bash script snippet -

```bash
#!/bin/sh

declare -a ParslExecutors=("local_few_htex" "local_high_htex" "local_few_threads" "local_high_threads")
declare -a arguments=(3 10 31)
program=mandlebrot
argm="-w"

for exec in ${ParslExecutors[@]}
do
        for d in ${arguments[@]}
        do
                echo $exec $d
                someString=`python ${program}.py ${argm} $d -e ${exec}`
                someString=`python -m pyinstrument -r json -o outputs/${program}-${d}-${exec}.json ${program}.py ${argm} ${d} -e ${exec}`
                echo $someString
                echo "$someString"
        done
done
```

The bash script automates the execution of the python program, using executors as arguments and the number of threads.

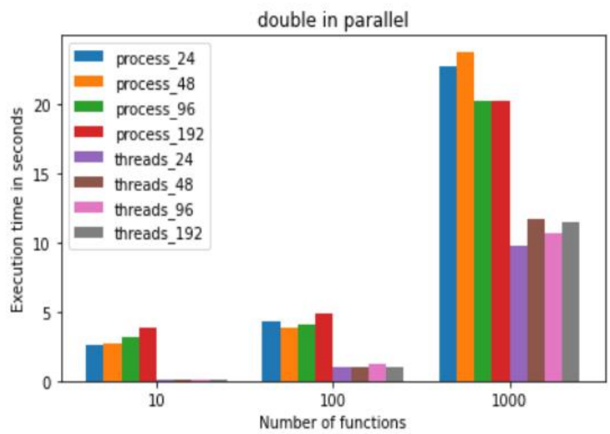
All the above source code is completely implemented in Python, and executed on Linux environment. There are approximately 1100 lines of code and we have used github as a method of source control.

V. Evaluation

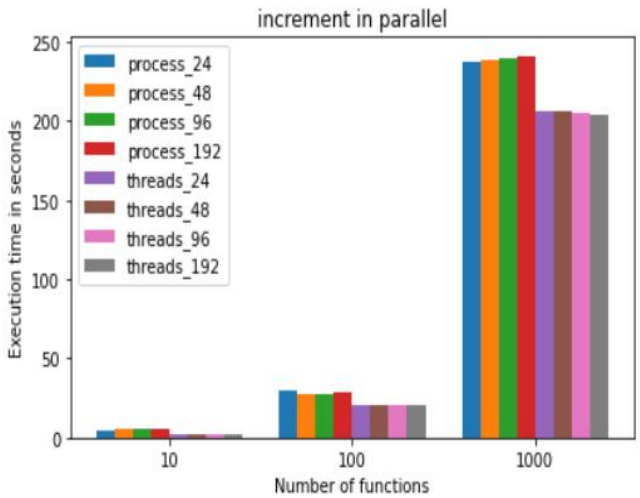
The graphs have the number of functions on the X-axis and execution time in seconds on the Y-axis. For plotting, we consider 10, 100 and 1000 functions. Execution of the mentioned python programs are plotted using these measures.

The graphical output is as shown below for the programs -

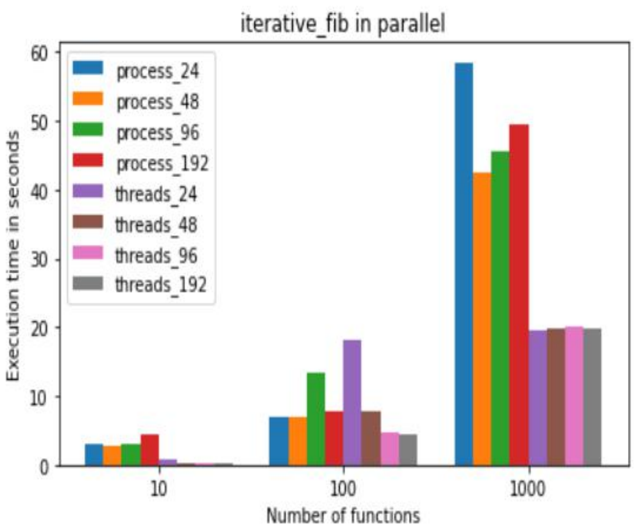
Execution of double.py in parallel -



Execution of increment.py in parallel -



Execution of fibonacci in parallel -

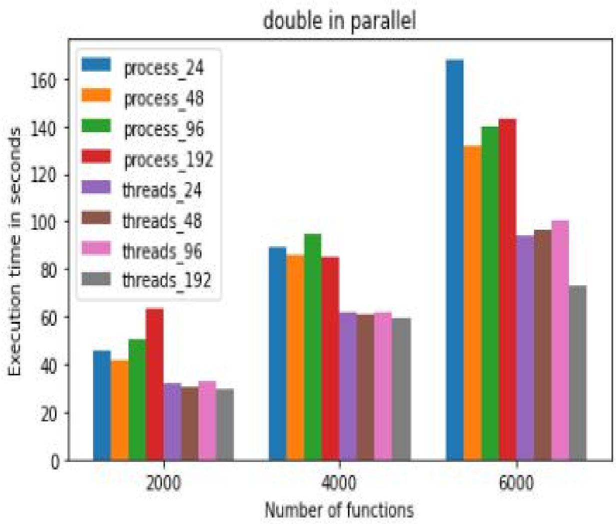


We observed that the processes are slower than threads i.e the execution time is more for processes as compared to the threads for all three programs. This is an unexpected result, as the execution of processes should take lesser time than the execution of threads.

To further check for the same response and to confirm if the findings are true, the double.py program is now run against 2000, 4000 and 6000 functions to check for execution time trends. We can see that the findings still stand and that the execution time of processes is more than the execution time of the threads.

Execution of double.py in parallel after increasing the number of functions -

We can see that there is a bottleneck in parsl handling processes. To check further on the handling issue, the double.py program is evaluated against 196 processes.

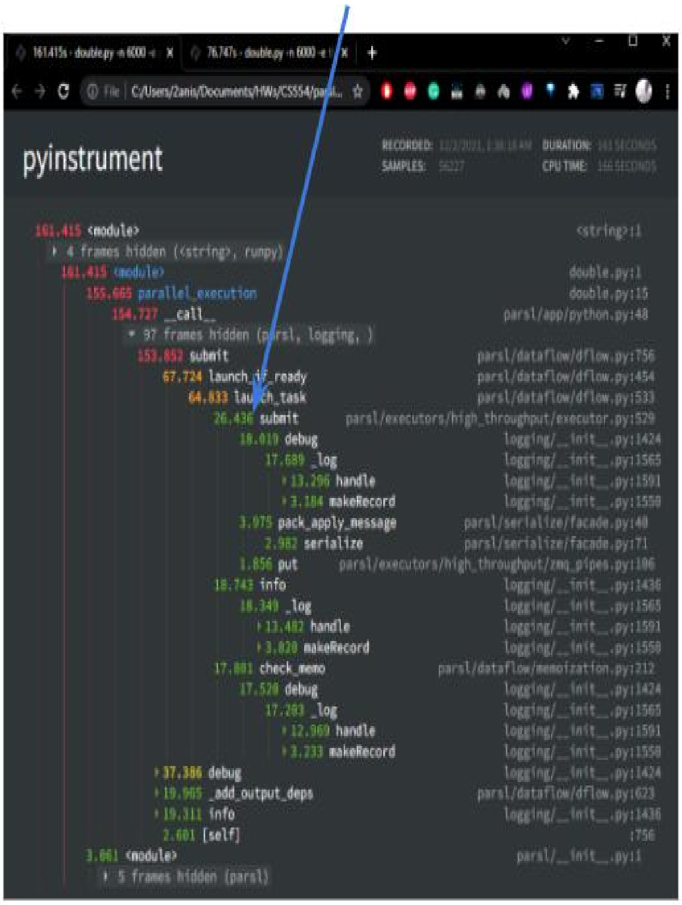


We now will have a granular look at the flow of program, we make use of pyinstrument profiler to evaluate the findings. Pyinstrument focuses on the slowest part of the program and returns detailed output. We run double.py for evaluations of the slow components in the program.

The Pyinstrument profiler was used which gives out a tree structure, that can be viewed in HTML format as shown in the diagrams below.

We make use of double.py with 196 processes, having 6000 functions and then evaluating double.py with 196 threads, having 6000 functions.

The pyinstrument output as seen from the browser is as follows for processes and threads-



We can see that most of the program execution time is taken by the submit function, executing in the process.

submit() is used for logging in parallel programming, and is used to make the process fault-tolerant.

While for the other figure, where the threads are used, there is no logging overhead, resulting in fast execution time, and thus, provides lesser latency than the former.

That means that a significant amount of time is spent in logging the execution and submitting these logs into the queue.

VI. Related Work

Lightweight Function Monitors[6] Maintaining a pool of nodes that can execute lower latency tasks. Instead of assigning functions to whole nodes, Work Queue provides the ability to dynamically pack tasks onto available worker nodes. But to assign a function to nodes requires labeling the resource requirement so methods to monitor the function are implemented. Making the python function invocation the fundamental unit of resource management in a distributed system raises issues relating to granular parallelism, management of software environments, and adaptation to computing resources.

This paper extends Parsl and work queue by developing tools to handle these execution challenges automatically. But these implementations are useful if the tasks are of varying complexity and require varied requirements of resources, so this implementation helps in automatically managing and scheduling tasks such that utilization can be squeezed in a multi-core or distributed system.

The main difference between lightweight function monitors and parsl is that even though they both monitor concurrent fine grained tasks, each function and data is used by a management node to dynamically pack the tasks onto available worker nodes in lightweight function monitor. This work does not talk about limitations of Parsl but implements a workaround to increase utilization. While, in our paper, we try to pinpoint the exact cause of this latency, so that in the future work, it can be improved to increase utilization, instead of employing a workaround like the former paper.

VII. Conclusion

The most important takeaway from this project was that we were able to learn in depth and also implement parallel programming in python, using parsl. We made use of the parsl library to implement parallel programming and were able to figure out where exactly the latency takes place. Thus, considering that we have achieved our end result, we can term this project as a 'success'.

Several profilers and parallel programs were evaluated against multiple configurations of processes and threads and two types of executors - Parsl Process Executor and Parsl Thread Executor. All these programs and configurations were automated by using python and bash scripts.

It was found that logging is one of the issues that is causing the slow execution of parallel programming in python. This logging is done to ensure fault tolerance of the system. Parsl does aggressive logging for this, which causes significant latencies in the parallel program execution.

Future work could be where this logging could be turned off, sacrificing the fault tolerance of the system, and then check the execution parameters and trends to further zero in on the source of the slowness.

VIII. References

[1] The python profilers.https://docs.python.org/3/library/profile.html.

[2] Extrae. https://github.com/bsc-performance-tools/extrae.

[3] yappi 1.3.2. https://pypi.org/project/yappi/.

[4] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M.Wozniak, I. 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.

[5] D. POORNIMA NOOKALA, D. K. HALE, and D. I. RAICU. Xtask-extreme fine-grained concurrent task invocation runtime. DataSys, 2021.

[6] T. Shaffer, Z. Li, B. Tovar, Y. Babuji, T. Dasso, Z. Surma, K. Chard, I. Foster, and D. Thain. Lightweight function monitors for fine-grained management in large scale python applications. In 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 786–796. IEEE, 2021.

[7] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. 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.

[8] Pyinstrument profiler. https://pyinstrument.readthedocs.io/

[9] Y. Babuji, K. Chard, I. Foster, D. S. Katz, M. Wilde, A. Woodard, and J. Wozniak. 2018. Parsl: Scalable Parallel Scripting in Python. In 10th International Workshop on Science Gateways.

[10] IPython.parallel.https://github.com/ipython/ipyparallel. Accessed Apr 24, 2019

[11] N. Wilkins-Diehr, "Special issue: science gateways - common community interfaces to grid resources," Concurrency and Computation: Practice and Experience, vol. 19, no. 6, pp. 743–749, 2007.

[12] P. Kacsuk, Z. Farkas, M. Kozlovszky, G. Hermann, A. Balasko, K. Karoczkai, and I. Marton, "WS-PGRADE/gUSE generic DCI gateway framework for a large variety of user communities," Journal of Grid Computing, vol. 10, no. 4, pp. 601–630, Dec 2012

[13] T. Glatard, M. Etienne Rousseau, S. Camarasu-Pop, R. Adalat, N. Beck, ´ S. Das, R. F. da Silva, N. Khalili-Mahani, V. Korkhov, P.-O. Quirion, P. Rioux, S. D. Olabarriaga, P. Bellec, and A. C. Evans, "Software architectures to integrate workflow engines in science gateways," Future Generation Computer Systems, vol. 75, pp. 239 – 255, 2017.

[14] K. Chard, S. Tuecke, and I. Foster, "Efficient and secure transfer, synchronization, and sharing of big data," IEEE Cloud Computing, vol. 1, no. 3, pp. 46–55, Sept 2014.

[15] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," Parallel Computing, vol. 37, no. 9, pp. 633–652, Sep. 2011.

[16] Y. N. Babuji, K. Chard, and E. Duede, "Enabling interactive analytics of secure data using cloud kotta," in 8th Workshop on Scientific Cloud Computing, ser. ScienceCloud '17, 2017, pp. 9–15

[17] M. McLennan and R. Kennell, "HUBzero: A platform for dissemination and collaboration in computational science and engineering," IEEE Des. Test, vol. 12, no. 2, pp. 48–53, Mar. 2010.

[18] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "Compiler techniques for massively scalable implicit task parallelism," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '14, 2014, pp. 299–310

[19] E. Deelman, G. Singh, M.-H. Su, Y. Blythe, James Gil et al., "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," Scientific Programming, vol. 13, no. 3, pp. 219–237, 2005.

[20] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in Proc. 14th Python in Sci. Conf., 2015, pp. 130–136.

[21] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Scalable Parallel Programming in Python with Parsl. In Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning) (Chicago, IL, USA) (PEARC '19). ACM, Article 22, 8 pages. https://doi.org/10.1145/3332186.3332231

[22] Kyle Chard, Yadu Babuji "Extended Abstract: Productive Parallel Programming with Parsl"

[23] Yadu Babuji, Alison Brizius, Kyle Chard "Introducing Parsl: A Python Parallel Scripting Library"

[24] M. Wilde, M. Hategan et al., “Swift: A language for distributed parallel scripting,” Parallel Computing, vol. 37, no. 9, pp. 633–652, Sep. 2011.

[25] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, et al . 2015. FireWorks: A dynamic workflow system designed for high-throughput applications. Concurrency and Computation: Practice and Experience 27, 17 (2015), 5037–5059

[26] K. Chard, S. Tuecke, and I. Foster. 2014. Efficient and Secure Transfer, Synchronization, and Sharing of Big Data. IEEE Cloud Computing 1, 3 (Sep. 2014), 46–55. https://doi.org/10.1109/MCC.2014.52

[27] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. Wozniak, I. Foster, M. Wilde, and K. Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In 28th ACM International Symposium on HighPerformance Parallel and Distributed Computing (HPDC). ACM. https://doi.org/ 10.1145/3307681.3325400

[28] Computational Data Analysis Workflow Systems. https://s.apache.org/ existing-workflow-systems. Accessed Apr 24, 2019.

[29] I. Foster, R. Olson, and S. Tuecke. 1992. Productive parallel programming: The PCN approach. Scientific Programming 1, 1 (1992), 51–66

IX. Appendix

Work	**Contributed by**
Background Study	Anish & Pranjal
Profiler Research	Pranjal
Testing Profilers on various Python programs	Pranjal
Profiler evaluations	Anish
Analysis	Anish
Python and bash scripts	Anish
Automation of scripts	Anish