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

Presented by:

- Ismail Elomari Alaoui
- Reda Chaguer

Project supervisors:

○ Dr. Ioan Raicu

○ TA: Alexandru Orhean

Plan

- Introduction
- Related work
- Proposed solution
- Evaluation
- Conclusions

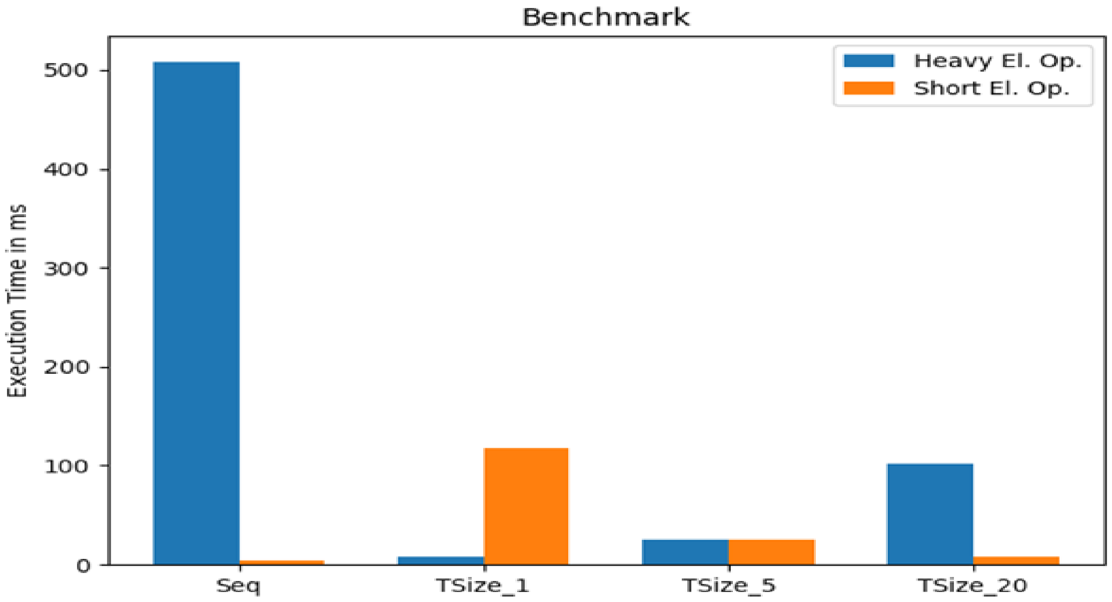
Introduction

Background:

Parsl is a parallel scripting library for **Python**.

- Excellent in heavy-computational parallelism.
- Weak in fine-grained parallelism

Problem Statement:



Motivation:

- Previous experience with parallel computing.
- Multiple multi-core programming projects conducted in C
  - Abelian Sand Model
  - Blurring Algorithm
  - …
- Take up a new challenge in discovering, proposing solutions and contributing in **Parsl**.

Related Work

Parsl’s Github: https://github.com/Parsl/parsl

Parsl’s website:

http://parsl-project.org/

Numerous benchmarks focused on heavy-computational parallelism

Served us as tutorials to understand Parsl’s functionality and performance

Proposed Solution

- Conception and implementation of numerous algorithms and workloads with an extensive variety of parameters
- Profiling numerous programs using **cprofile** and **Yappi**
- Profile visualization using **Snakeviz**
- Benchmarking Parsl’s performance
- Trying out different configurations
- Comparing Parsl with other sequential as well as parallel computing tools
- Creation of clear and concise graphs targeting precise areas in Parsl’s code

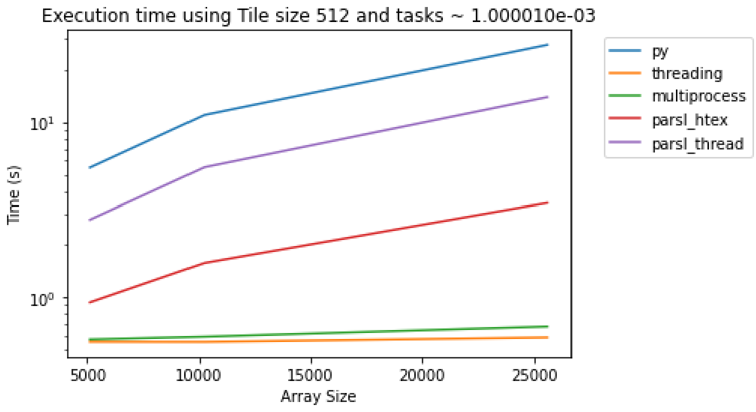
Evaluation

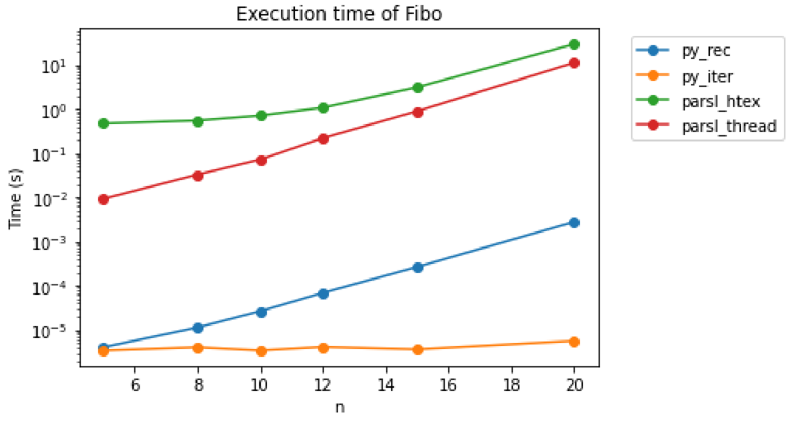
- **Fibonacci**
  - Iterative Sequential
  - Recursive Sequential
  - Recursive Parsl join app using ThreadingPool
  - Recursive Parsl join app using HTEX
- **Square array**
  - Parallel version using threading
  - Parallel version using multi-processing
  - Variety of parameters (tile size, array sizes, task duration, …)

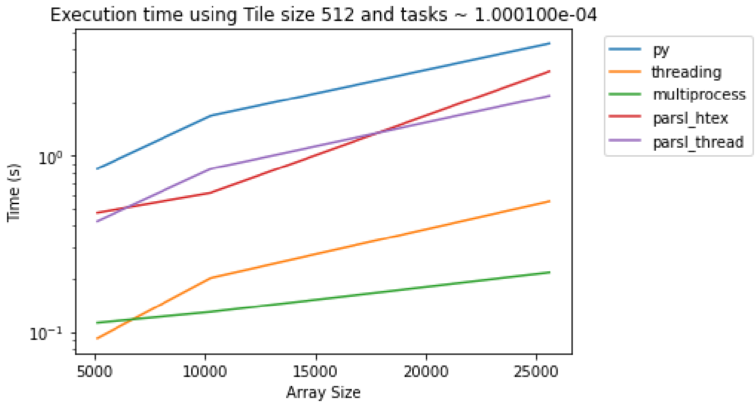
Executor	**Number of Nodes**	**Task duration for good performance**
ThreadPoolExecutor	1 (Only local)	Any
HighThroughputExecutor	<=2000	Task duration(s)/#nodes >= 0.01 longer tasks needed at higher scale

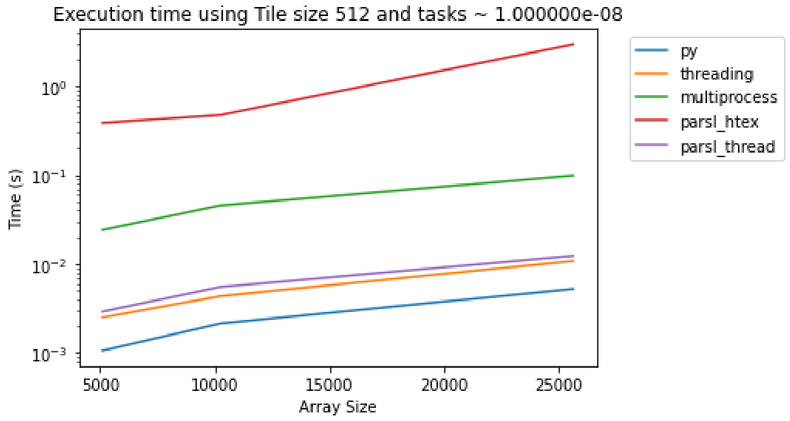
Evaluation
On a 16 cores Machine

Demo 1: Visualization and Graph Creation



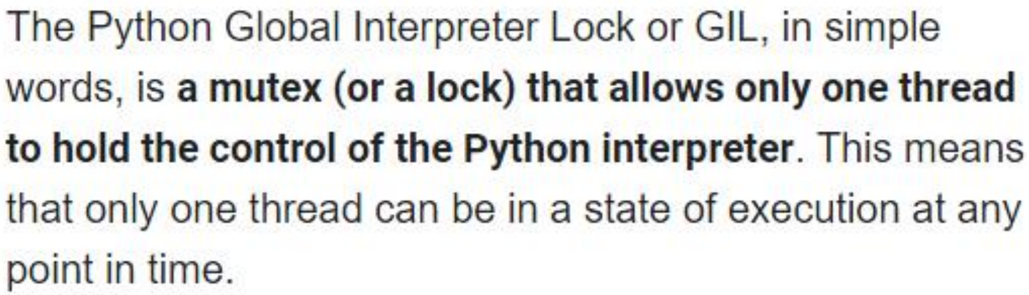




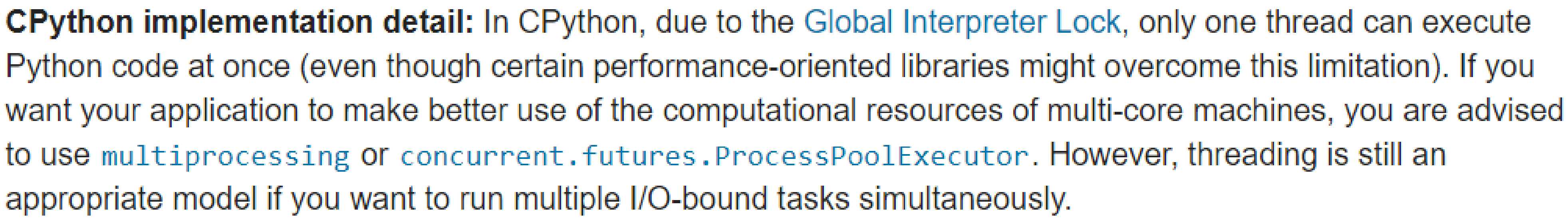


Conclusions: GIL

Source: https://realpython.com/python-gil/



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.



CPython implementation detail: 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.

Source: https://docs.python.org/3/library/threading.html

Conclusions: PyPy

Implementation of PyPy app

Great addition not only in **fine-grained** parallelism but also in **heavy-computational** parallel computing

Demo 2: PyPy, Speed in comparison to cpython

Conclusions: Lack of real-time monitoring options

Parsl creates logs of execution

Logs and profiles can **not** give all necessary information in order to discover and solve all Parsl’s bottlenecks.

A proper real-time monitoring tool would be beneficial

Demo 3: EasyPAP, Square and Abelien Sand Model

Final conclusion

- We value our project **successful**
- Multiple benchmarks conceived and implemented
- Numerous solutions and suggestions proposed
- Future work could be pursuing any one of the 3 conclusions we presented.
- In this project, we learned a lot about **parallel computing in Python**

Thank you for your attention