Distributed Systems (521290S) Course Project Report: Pystributor: Distributed Python Computing and Prime Number Checking

https://github.com/hirsimaki-markus/pystributor

Patrik Pyykkönen, 2434137, pyykkonen.patrik@gmail.com
Markus Hirsimäki, 2433921, hirsimaki.markus@gmail.com

Course project overview. The artefact we created is a Python library that allows for distributing computationally intensive workloads to multiple workers that are managed by one central hub. We implement a completely new tool. We also include an example workload which checks number primeness; the purpose of the sample task is demonstrating the distribution in and of itself and its related implications.

The library uses symmetrically encrypted TCP/IP connections to send JSON for communication purposes. In our project we assume a secure channel is used to distribute the necessary encryption keys. The overlay network created follows star topology. The library can be deployed by using Docker containers or by running Python processes natively on a given machine. For the evaluation we needed the flexibility of running the Python files natively, and opted out using Docker. Relevant working Docker containers are, however, provided within the repository for normal operation with the Pystributor.

The hub manages a pool of workers and sends parts of the computationally intensive task to individual workers, who then asynchronously provide their answers. The hub keeps track of results obtained and compiles them. Each worker is individually identifiable based on its socket connection, thus allowing a single physical machine or Docker container to run multiple worker instances. The library substantially improves Python's computational power when running certain types of tasks even on a single physical machine as it will bypass the global interpreter lock, GIL. The Evaluation section elaborates on this.

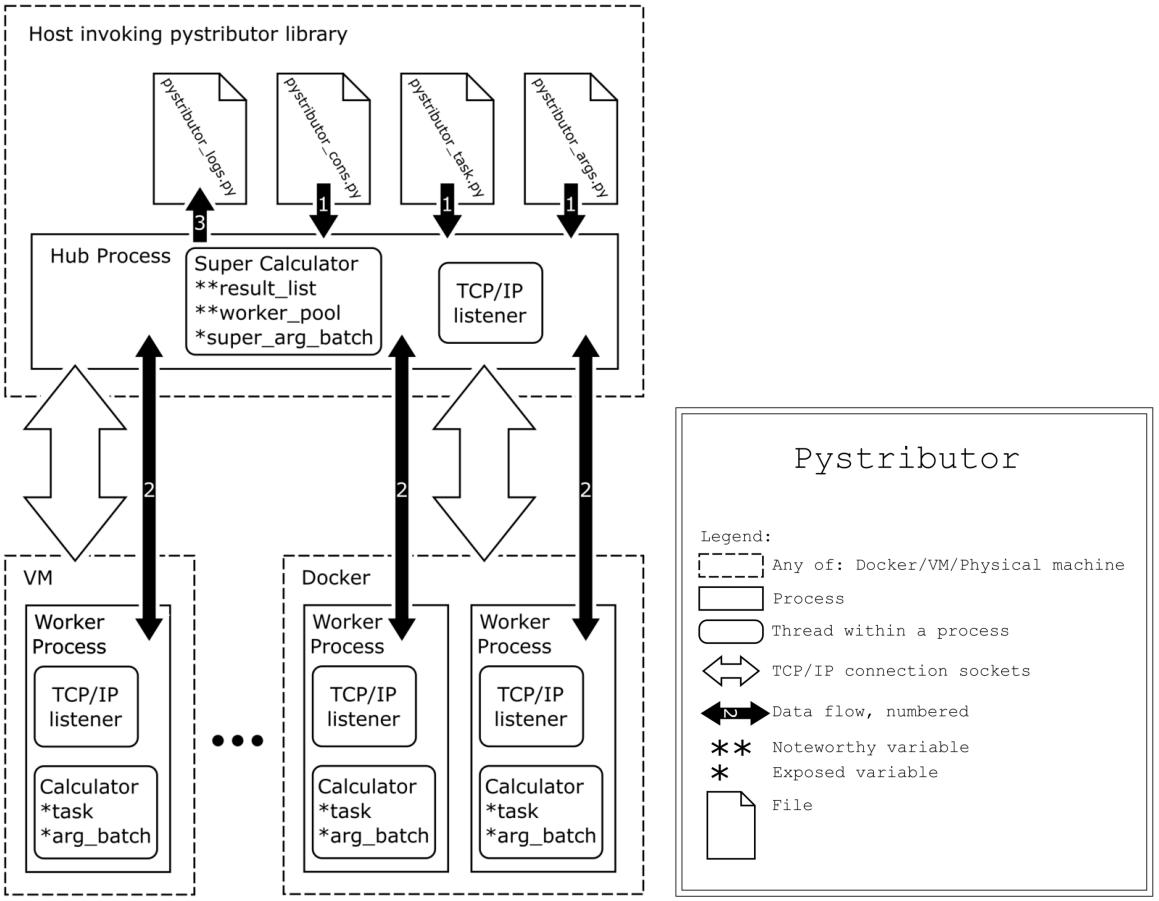
1. Architecture. The Pystributor library follows a simple star topology; a central hub first distributes a task to workers that then report back to the hub after with their calculation results as they receive arguments to be processed with the task.

The project briefly address many of the topics seen in the course; we define a custom communication protocol for connecting the hub and the workers, naming is addressed as individual workers are identifiable, coordination is achieved by the central hub tracking current progress, fault tolerance is implicitly considered and, a level of security is implemented by using encrypted messaging to avoid sending plain executable code over the network.

The overall description of the design can be seen in the below figure. The library is not dependent on any specific physical configuration when it comes to workers as long as they are reachable via network. Multiple workers could be deployed on one physical machine or workers could be distributed to Docker containers with each container hosting any number of workers. The workers could also reside on the same physical hardware as the hub; most configurations will work as long as network connection is available. Should VMs or Dockerization be used, taking care of opening necessary ports and tunnels is required. The choice of network connection should also not matter; the machines can be connected to the same LAN or multiple networks connected via the internet.

The intended use case is to effectively bypass Python's GIL's limitations via multiprocessing for tasks that can be distributed sensibly among multiple processes. Additionally further concurrency is achieved by allowing the use of multiple physical machines for this purpose. For maximal performance benefit, the worker counts should be chosen so that there is one-to-one distribution of the worker processes among physical CPU

cores. It should be noted that the hub process will not automatically initialise the workers; they must be instantiated separately on each (worker) machine with their respective network configuration steps taken care of.



Pystributor is intended to be a Python library invoked by some other (Python) program. The library aims to provide a means for solving computational problems which can benefit from bypassing GIL or being distributed to several machines. Checking prime numbers is used as an example task to demonstrate the library; any other task could also be distributed among the workers. The method used for checking primes is intentionally crude to simulate high CPU loads by a task that is efficiently split to multiple workers.

The graph below was created before beginning actual programming. The end result which we achieved resembles the graph extremely closely with the only notable exception being that the files seen are implemented via Python objects and calls instead of requiring disk operations as originally thought.

Some of the less important differences were that the connection and log files also proved to be redundant with these changes. Additionally, support for argument batching was left for future work and multithreading was not necessary for workers as the calculator thread was able to sufficiently handle listening due to lockstepping.

2. Implementation. The entire library is implemented in pure Python except for encryption which is provided by Fernet. There are no specific hardware or system requirements. The overall view regarding the implementation can be seen in the above figure. We now elaborate more on the details in this section.

The final implementation, from the perspective of the invoker, creates a blocking call when a task is being processed by workers. Checking the status of answers collected is left to the end user. This could be done by whatever means they deem sufficient (e.g. by simply waiting or by polling from another thread).

The task being distributed is, in essence, an executable Python function that should be run thousands of times with varying arguments. Pystributor splits these different runs of the same function to multiple physical CPU cores among multiple physical machines to achieve local and distributed concurrency.

The hub invokes two daemon threads, one of which is used for masterminding the given task while the other one is used for performing I/O with the workers. Connections are initiated by workers which allows for relatively simple configuration; only the workers need connection information of the hub as the hub receives necessary information upon opening socket connections.

Workers use a single thread for their processing and listening. As they are in lockstep with the server, it is guaranteed that there are no messages to be received whilst they are performing calculations. It should be noted that a future update could change this if we decide to implement heartbeat pings for more stable connections and for more graceful handling of network errors which can currently cause the task to block infinitely despite most of the calculation results being ready for reading.

On the hub, a super calculator daemon tracks the task distribution (to worker calculators) and aggregates the results. The listener, as it is blocking, runs in a separate daemon thread and delegates the messages it receives to the super calculator.

Overall order of execution for a given task, such as prime number checking, is generally as follows:

0. Necessary network configuration steps are taken along with distributing encryption keys. Hub and workers will be configured during object instantiation with their corresponding encryption keys and IP/port pairs that were provided during this step.
1. Hub is invoked by some Python program and spawns two daemons, listener and calculator. The hub begins listening for worker connections to form a worker pool. Workers, invoked elsewhere by the same party, try to form a connection on repeat until they succeed.
2. After the pool is formed, the hub distributes a task given to it during the instantiation to the worker pool. Workers reply with ACK messages.
3. Hub begins distributing arguments for the task to different workers and updates their statuses within the pool correspondingly. Hub waits for results which now act as ACK messages for lockstepping.
4. Hub begins receiving calculation results from workers, newly ready workers receive new arguments. The cycle repeats until the worker pool has handled all arguments for the single corresponding task.
5. Having aggregated the arguments and their corresponding results, the hub kills the worker pool.
6. Workers are dead, hub ends its daemon threads, and returns control to the invoker.
7. Invoker examines the task results argument-wise from an attribute (hub.answersheet) of the hub object it instantiated.

3. Communication. The communication methods used are low level; we directly interact with sockets and send messages via the TCP/IP stream. The Hub has one socket over which every worker connection is abstracted by using a selector. The workers also open one socket each, but they have only a singular connection with the hub. No specific high level protocol is used besides using JSON to serialise data.

The communication happens in three distinct phases; worker discovery, task distribution, and argument distribution. There is no specific software or communication stack to speak of besides bare streams over TCP/IP and what we have constructed over them. It is noteworthy that the current implementation does not include message-begin or end symbols; lockstepping provides sufficient message delimiting.

4. Naming. The naming scheme used is flat and rather simplistic; each worker is addressable based on its connection information even if several workers reside on the same physical machine or within the same

Docker container. On startup, the hub performs worker discovery to form the pool. The hub uses the notion of pool when interacting with the workers and does not take great interest in addressing individual workers. This provides simplicity for the end user and for the implementation. Workers do not communicate with each other.

As the logical overlay network follows star topology, the hub will need information for all the workers and each worker will need the information required to connect with the hub. Conveniently, the hub receives the information it needs during the worker discovery phase as it will wait for workers to connect. The workers, however, need the connection information for the hub.

5. Coordination. Most of the coordination happens within the hub, more specifically in the super calculator daemon. The communication between the workers and the hub are asynchronous which would allow us to implement tools such as keep-alive pings and re-sending tasks to other nodes. These options are intentionally left outside the scope of this course work. The asynchronous nature of the processing allows for querying results even if they are incomplete or still in progress. No specific coordination scheme is used besides the hub which acts as a mastermind for the operations.

6. Consistency and replication. The project does not involve replication, each worker is assumed to respond in a timely manner and correctly. As the library we developed is a tool to perform distribution, this section is somewhat irrelevant besides the points presented in sections 5 and 7. Should fault tolerance be improved later, some level of replication could be achieved by actively leveraging abstractions like Docker. The consistency model which is most closely matched is decidedly data centric.

7. Fault tolerance. The current implementation makes several assumptions to keep the scope of the project reasonable; both the hub and workers assume each party is well behaved, network connections are also assumed to be flawless on TCP level; this provides some built in protection. Should a worker misbehave, the end user is left with the task of polling the state of the answersheet. We acknowledge that the following features, however, could be implemented in future and would fit the purpose; ability to reload and change worker pool on the fly, ability to detect dropped workers to re-issue arguments, more robust security measures, more general purpose task & argument structure, and heartbeat pings. All of these are left for future to avoid scope creep.

8. Security. A very basic level of security is implemented by symmetrically encrypting the messages sent between the hub and workers as it would be irresponsible to send executable code completely unsecured. Currently, the encryption is performed with the Fernet library which is baked into Pystributor. In hindsight, this ought to be refactored to be more explicit for the end user. We also assume that the user has an existing secure channel such as SSH connection to distribute the necessary keys and start worker processes.

Evaluation. We performed the evaluation using a python script with predetermined arguments. We present four graphs. In these tests we compared using Pystributor with varying worker amounts. The first two graphs show performance on a single physical machine, including a base case without Pystributor. The second graph is simply zoomed in version of the first.

The below two figures highlighting one physical machine show that with an easy task (checking small numbers) there is relatively large overhead when using Pystributor. In the graphs, worker count 0 shows the case where same task is run without Pystributor. As worker count increases on the same physical machine, we start seeing increased performance at the cost of increased CPU usage.

With low worker counts, CPU usage is increased (relative to running without workers) without a proper increase in performance, because the networking takes a relatively large slice of the time, as the

calculations are fast. We can also see that as tasks get harder (checking large numbers) there is greater performance benefit gained distributing the task.

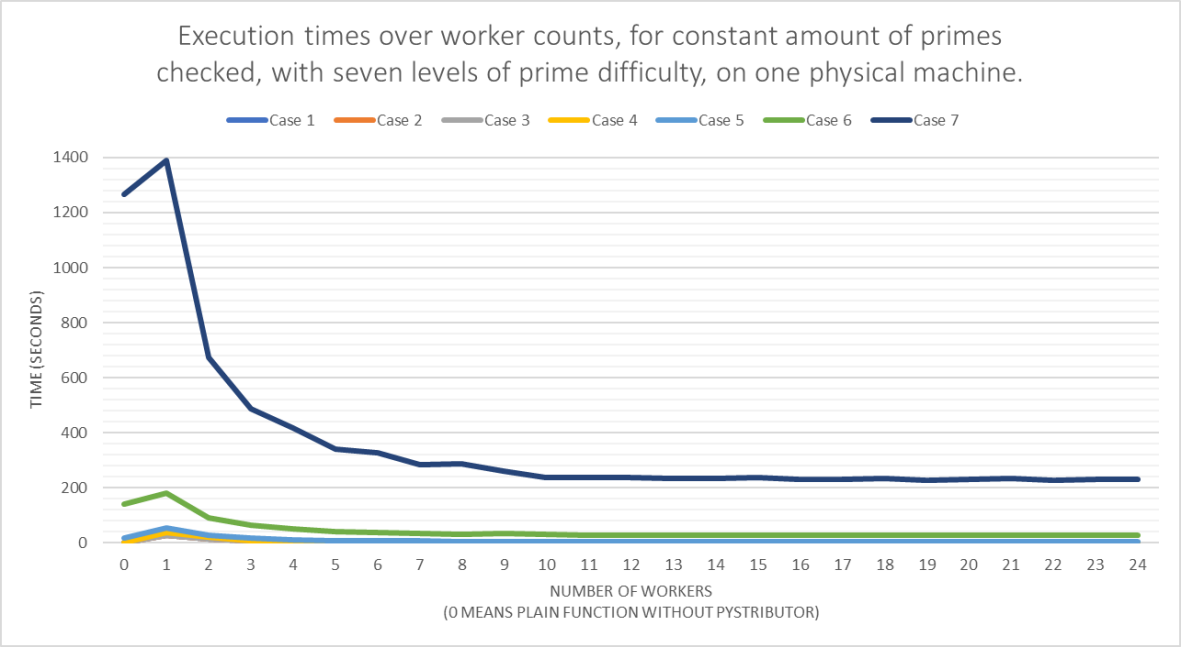
From this we can conclude that the tasks being distributed should be maximally complicated to reduce the relative amount of performance overhead. In the future, argument batching could improve this characteristic. We can also conclude that even when running on a single physical machine, bypassing GIL will produce significant performance benefits given a sufficiently difficult task and arguments.

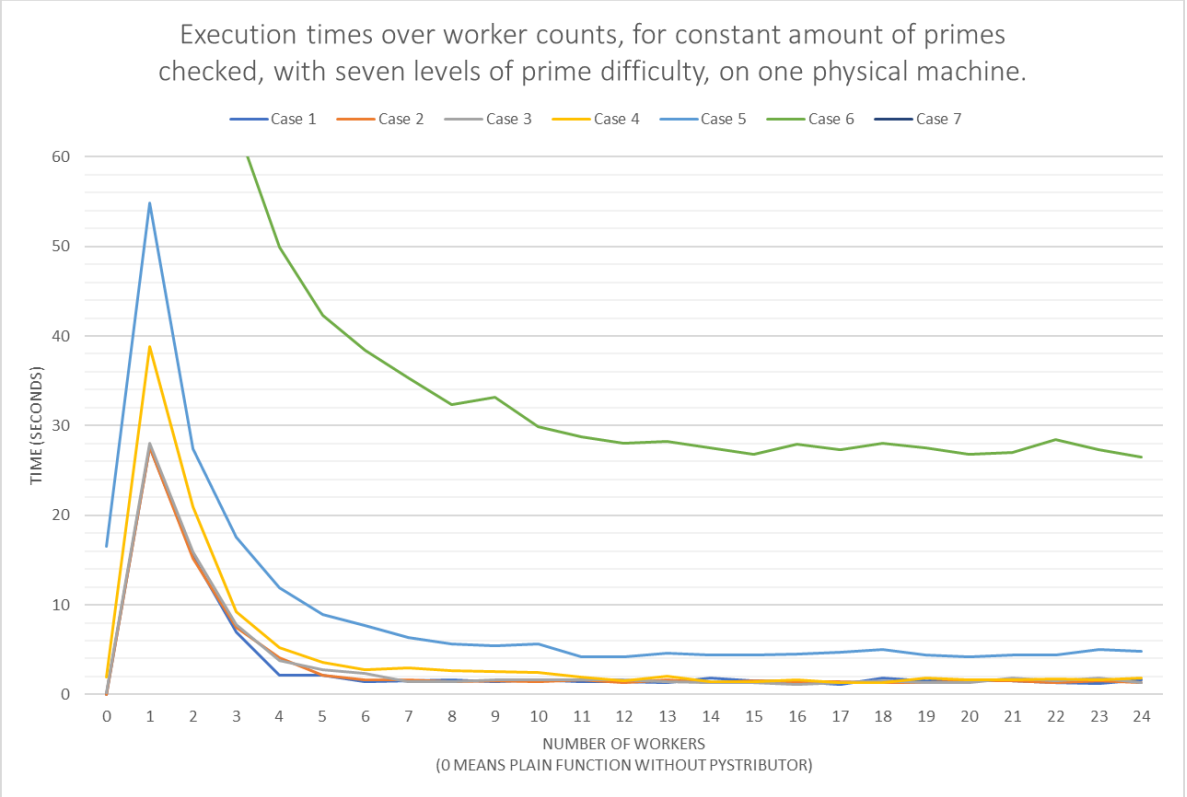
We can also see that the maximum performance with multiple workers is achieved when the number or workers matches the CPU's thread count (12 in this case), resulting in near 100% CPU utilisation. Adding some extra workers beyond CPU's core-thread count doesn't seem to hurt performance, but is pointless.

The final two figures show Pystributor's performance across multiple machines using optimal worker counts, one per core-thread, on each machine hosting workers. Hub was hosted on a machine which was also running workers. It must be noted, that each PC added has inferior performance to the previous, with a significant margin, which makes analysing the graph more difficult.

We can see very similar characteristics to the first two graphs showing a single machine; increased worker count will provide an increase in performance. There is no base case without Pystributor in these last two graphs as Python is not distributed by default.

Interestingly we can also see that performance significantly drops on the case number one. This case contains the easiest number range to check (1000-2000). It can be seen that the addition of a physical machine with a low performing processor is detrimental to the overall performance of the pool for an easy task. Even in this case, should the answersheet be polled it could be seen that most answers have been calculated in a timely manner. The same applies to more difficult tasks but to a lesser extent, because in the end the hub may need to wait for a slower PC to finish its arguments while others are already done.





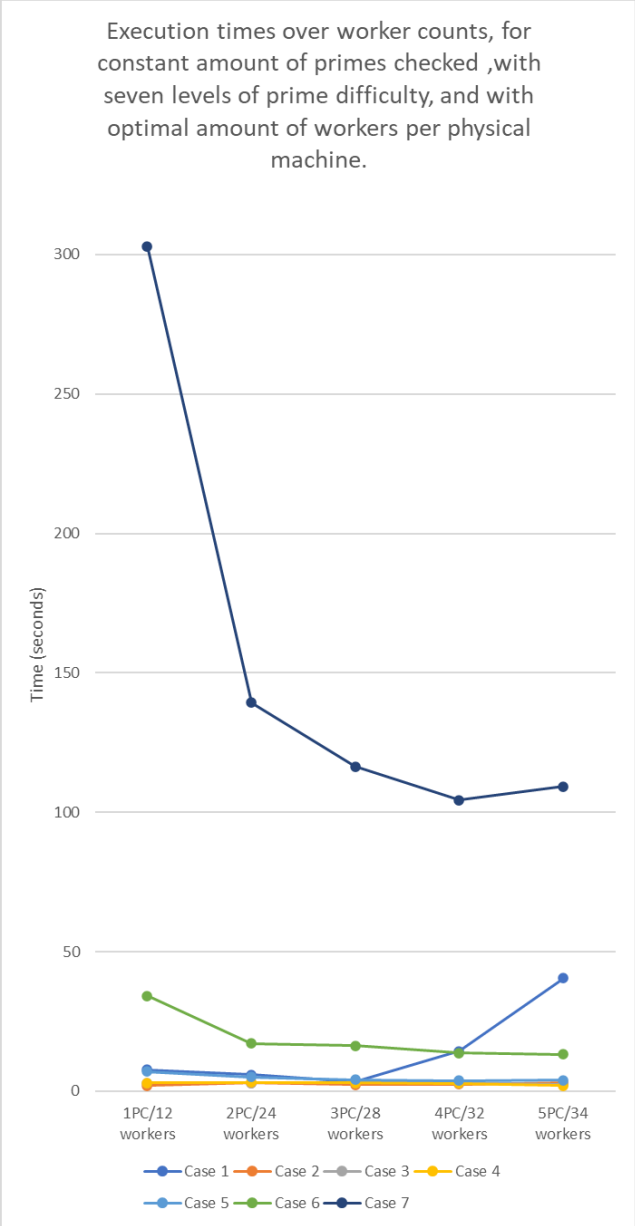
The third graph on the next page shows the performance of Pystributor using workers distributed across multiple machines to test the scalability of the system. Similarly to before, the fourth graph is zoomed in version of the third. All of the graphs show a constant sized number range being checked, one thousand. The difficulty is increased by picking the same range but with larger numbers. Smallest range was 1000 to 2000 while the largest range was 10^9 to 10^9 + 1000. In all graphs, larger case numbers correspond to harder numbers with the same range length.

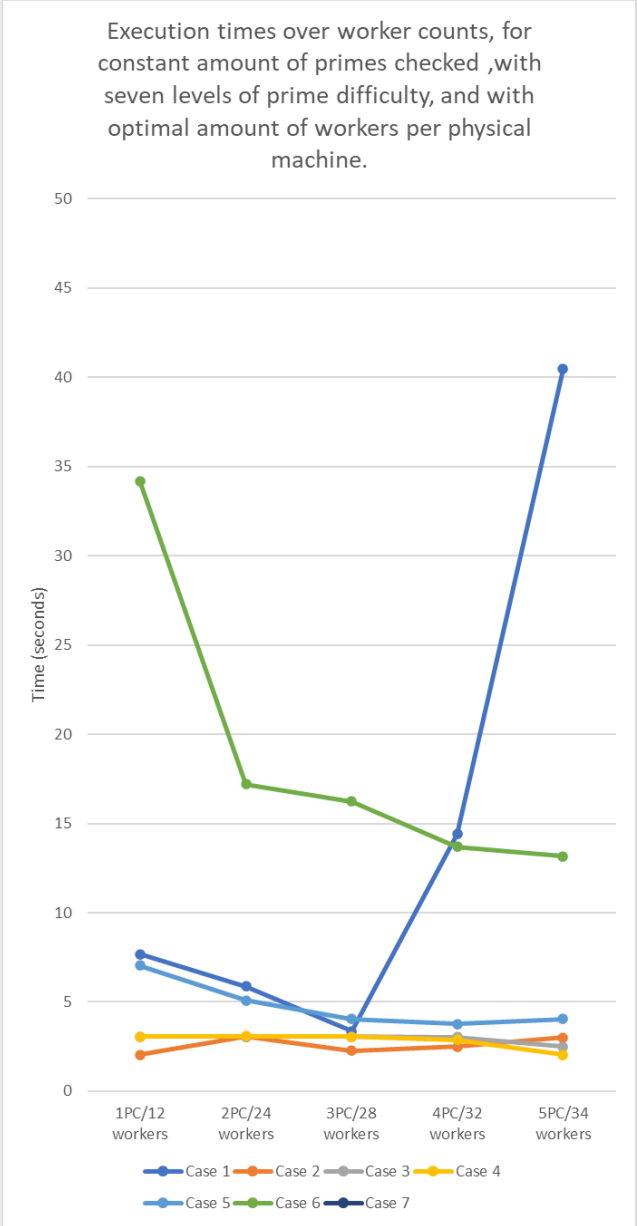
In conclusion, despite the library being very bare bones, it provides sufficient performance enhancement to warrant its usage in some use cases. This applies to both using a single physical machine as well as several physical machines. In the case of several machines a greater performance benefit is seen. It is noteworthy that all this performance benefit comes at the cost of increased CPU utilisation and an overhead.

Finally, during evaluation, we noticed that as the worker pool started to approach size of 40, we started having trouble getting workers connected. This seems to be an issue with how the worker discovery is performed. Further analysis of the issue is left for future.

The following tables provide info about the PCs used in second tests and arguments used in both.

Number or workers/PC	**Processing power**
1PC/12 workers	1 modern high performance 6-core CPU
2PC/24 workers	2x modern 6-core processor
3PC/28 workers	Previous + 4-core medium performance CPU
4PC/32 workers	Previous + old slow 4-core CPU
5PC/34 workers	Previous + old 2-core laptop CPU





Arguments ranges per case in the graphs:

Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**	**Case 6**	**Case 7**
(10^3, 10^3+1000)	(10^4, 10^4+1000)	(10^5, 10^5+1000)	(10^6, 10^6+1000)	(10^7, 10^7+1000)	(10^8, (10^8)+1000)	(10^9, (10^9)+1000)

Workload distribution including writing.

Student name	Tasks (in order of significance)	Estimated wl. (h)	Real wl. (h)
Markus	Design, coding, writing, evaluation	35	52
Patrik	Design, evaluation, writing, coding	35	57

References

Pure Python [https://www.python.org/] with the exclusion of using Fernet from [https://cryptography.io/en/latest/fernet/]. Additionally Docker was used to create containers [https://www.docker.com/].