Nitocris - A distributed data processing system for streamlining crowdsourced map functions

Distributed Data Processing Systems 2021

András Schmelczer[1] **and Leonardo Pohl**[1]

[1]Leiden Institute for Advanced Computer Science, Universiteit Leiden, the Netherlands

It is widely believed that Pharaoh Nitocris ruled at the end of the 6th Dynasty from 2182 to 2179 BC in ancient Egypt. It is also believed that she built the smallest pyramid, the third pyramid at Giza. To achieve that, she needed people to help her accomplish this feat. Similar to her, we try to achieve a larger goal by distributing work. Another similarity is that, as of yet, we are not aiming to build the greatest pyramid.

In this report, we introduce NITOCRIS, a proof-of-concept distributed data processing system.

1 Introduction

Our team uses Python along with several scientific packages — for instance, *numpy* [1], *gensim* [2], etc. — to process larger amounts of data on a daily basis. This usually entails applying a function to a large list of items (providing an opportunity for input partitioning-based parallelisation) for, for instance, lemmatising text or hyperparameter-optimising simple artificial intelligence models. Stemming from the inherently limited parallelisation of most Python interpreters [3], extra steps need to be taken to make full use of the machine's computing resources. However, if we still require more speedup, we have to look further and involve a number of remote hosts. Compared to classical solutions, NITOCRIS[1] makes this step as easy as clicking on a link.

2 Background

Many have faced our problems before, there have been numerous approaches utilised to solve it as well. In the following, let us look at some of them in order to position our work in the subdomain.

One of the most well-known distributed processing solutions is MapReduce. In their paper about MapReduce, Dean et al. [4] described a paradigm only using a sequence of *Map* and *Reduce* operations to implement most large-scale algorithms. This turned out to be an adequately versatile framework, of course, later works improved on it but the core ideas remained. Opposed to MapReduce, we would have liked to arrive at a simple Python package that can be integrated into pre-existing projects without the need to restructure or rewrite them. We also noticed that in our experience, the performance bottlenecks in our applications tend to be the *Map* operations, hence, we only focused on them for now.

A Python specific library already exists which supports remote execution called *Ray* [5]. It could be easy to imagine wrapping and repurposing this functionality. However, *Ray* requires setting up a cluster beforehand and is also tailored towards data centre-like deployments. As for NITOCRIS, we wished to be able to involve machines from all around the globe without a need to explicitly set up a cluster for them. Supporting dynamic, runtime changes, and having fault tolerance are something we cannot imagine our application without.

From this, we concluded that no well-known software matches our expectations and requirements, hence, we implemented our own approach.

3 Implementation

To help better understand our approach, we describe the requirements, design, and implementation details in the following subsections. For an overview of the use of the library refer to Listing 3, and for the design, Figure 1.

3.1 Requirements

Let us detail a use-case for the library. A programmer/data scientist is conducting some experiments. They have one or multiple large lists that they need to map over. This is a surprisingly common task, as most input is array-like. They install the NITOCRIS package to speed-up the calculations. After adding a couple of lines of code they can already take full advantage of the library. When they run their program again (or execute the cells in a Jupyter notebook), they are given a link. Through that link, they can connect all their household, work, cloud machines to join in on the data processing. They can even share the link on social media asking for help from their contacts. The handling of scaling and fault-tolerance should be transparent

[1]github.com/LeonardoPohl/DPS-2-Nitocris

and automated since clients joining and leaving may occur frequently. Stemming from the possibly wildly different computational resources of the clients, balancing the load should also be taken care of.

To achieve the scenario detailed above, we have to solve a number of issues. Most importantly: security. Running arbitrary Python code from an unknown source is not something people should be comfortable with doing. Additionally, setting up Python on each machine with the appropriate package version can be a menial task. Setting up Python can even prove to be impossible, for example, on mobile devices[2].

Lastly, developer and user experience should also be prioritised. The users helping the developer by volunteering their resources must be able to specify the amount of resources, and they should also be provided some feedback. So as the data scientist who initiated a processing round; they should also get feedback on the state of the processing.

3.2 API

As for the application programming interface (API), we would have liked to provide the developers with something familiar. Listing 1 shows a simple mapping operation in Python run on a single thread. It is common to rewrite it as Listing 2, to take full advantage of every core by creating multiple Python processes.

Listing 1. A simplified version of a typical performance bottleneck of the authors' scripts. The first five natural integers are raised to their second power and returned as a list.

```
import numpy as np

list(map(np.square, range(5)))
# [0, 1, 4, 9, 16]
```

Listing 2. Speeding up the algorithm of Listing 1 by distributing the input among multiple Python processes running on all CPU cores.

```
import numpy as np
from concurrent.futures import *

with ProcessPoolExecutor() as p:
  list(p.map(np.square, range(5)))
  # [0, 1, 4, 9, 16]
```

We provide a way to replace the multiprocessing-based version with a distributed processing solution as shown in Listing 3. Notice how similar Listing 2 and 3 are. We need to keep in mind that this is only a simple example. It is also expected of libraries to enable their users to customise all major steps of the operation. In our case, this is possible by passing in optional keyword arguments such as: `websocket_port`, `server_port`, `packages`, `timeout_in_seconds`, and `chunk_size`.

Listing 3. Speeding up the algorithm of Listing 1 by distributing the input among multiple remote machines using NITOCRIS

```
import numpy as np
from distributed_execution import *

with DistributedExecution() as d:
  d.map(np.square, range(5))
  # [0, 1, 4, 9, 16]
```

3.3 Design

The common denominator of most problems mentioned earlier is that they have a common solution. By using the browser and running a Python (3.9) interpreter along with the required packages in WebAssembly [6] (WASM) solves the security (WASM runs in a sandbox), compatibility (browsers are available on all major platforms), and manual installation issues all at once. It also makes joining and leaving the cluster as easy as opening and closing a website. We rely on the *Pyodide* library [3] for the WASM-based interpreter.

Figure 1 should provide the reader with a bird's eye view picture of our library. It can be clearly separated into client and server-side. They mostly communicate through a WebSocket [7] connection. There is an additional REST API for querying the required packages to execute the map function. The server keeps track of the clients and their tasks, handles failures by reexecution and provides the users with a progress bar. The client maintains a pool of WebWorkers running in WASM mode with *Pyodide*. The number of workers can be set on the website and feedback is given on their performance.

3.4 Server

The most crucial part of the library is the server implementation, this is the part responsible for orchestrating the task execution and providing an interface for the programmer.

Initialisation. A single class is exposed for the developer, namely, `DistributedExecution`. It supports the ContextManager protocol — allowing it to be used with the `with` operator — in order to make its deinitialisation clean. This class exposes a public method called `map`. Thus far, the API

[2]The question of whether mobile devices are even worth using for resource-intensive applications might arise. We think — in some cases — yes: for instance, the latest iPad Pro has the exact same CPU as the latest 13" MacBook Pro, with both of them being reasonably powerful machines.

[3]pyodide.org

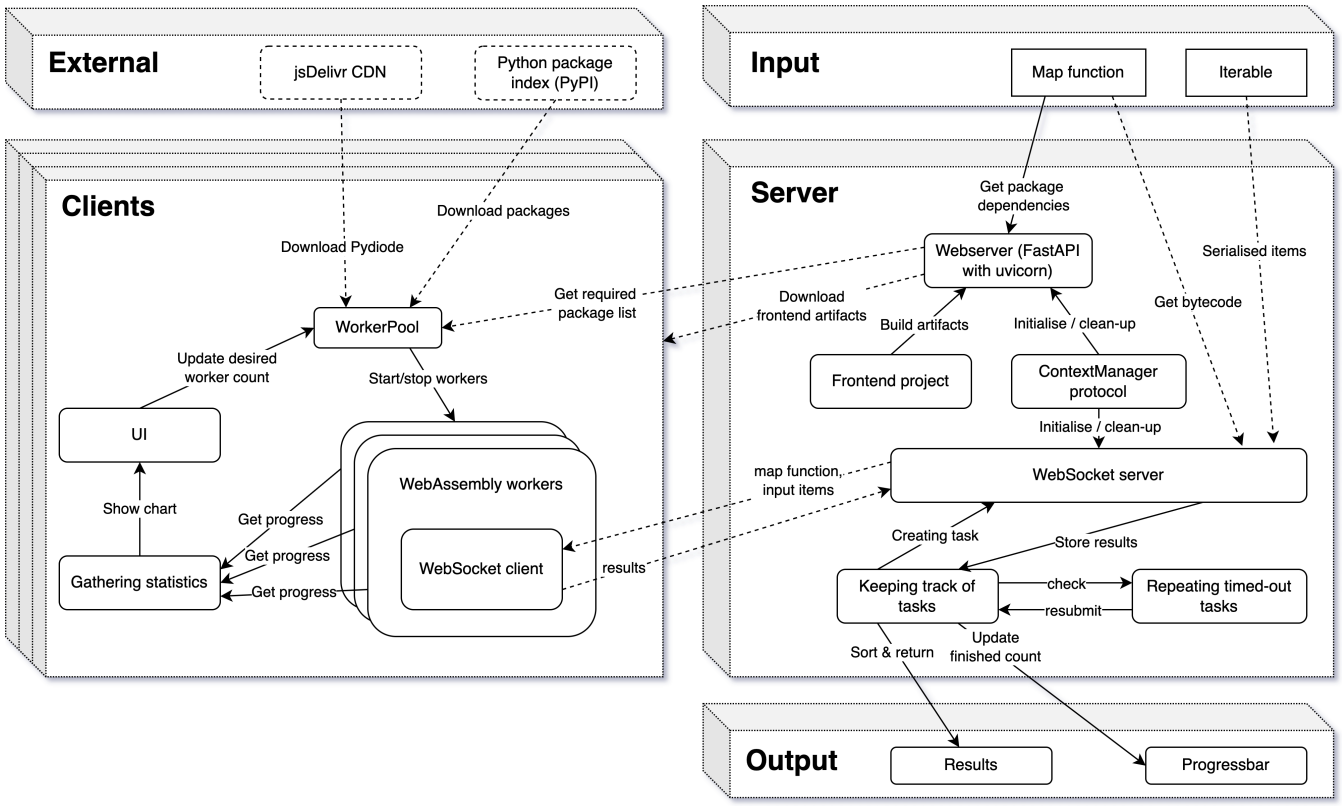

Fig. 1. High-level architectural view of NITOCRIS. The dashed arrows represent data-oriented flows, while the solid ones control-oriented flows.

is mimicking the the one of `ProcessPoolExecutor` presented in Listing 2. However, its inner workings could not be any different.

Processing. As a result of a call to `map`, a web server is started serving the client and the list of dependencies. Meanwhile a WebSocket server is also started waiting for said clients to come online. The function to be used for the *map* operation is serialised along with the data it is to be applied on. When a client joins, it receives the serialised function and data from the server. The latter happens in `chunk_size` chunks. Multiple records are kept for bookkeeping purposes, such as the list of tasks, the mapping between active tasks and clients, their time left to live, the returned results and their original indices, etc.

The data pieces are serialised one by one, sent to the client, meanwhile, a watchdog keeps track of the active tasks; if a worker times out for a task, by losing the connection or plainly just taking too long for other reasons (like threshing), the task is reset and added back to the task pool.

After everything has been done, the result is sorted based on the items' original position in the input, the servers are shutdown and the resulting list is returned.

3.5 Client

The client is wrapped in the form of a website. The website is built and bundled using webpack[4] and the build artifacts are saved into the server's file hierarchy. From there, they can be easily served.

Technologies. We chose to implement the website using the *React* framework[5]. This made it easy to declaratively handle dynamic behaviour which the website certainly does not lack. The rich library of NPM[6] provided is us with important building blocks for the application, for example, with a responsive charting library, *Recharts*.

As for the programming languages: instead of the more classical JavaScript programming language, we opted for TypeScript which facilitates development by doing a compile-time type checking of the entire project. This way, each of the models have their typed TypeScript interfaces which helps immensely in avoiding mistakes during the API calls. Additionally, we used the SASS[7] language which ex-

[4] webpack.js.org
[5] reactjs.org
[6] www.npmjs.com
[7] sass-lang.com

tends the capabilities of Cascading Style Sheets (CSS) and let us handle the complexities of styling every component and page more easily and robustly by allowing the use of variables and functions.

Many challenges had to be overcome to end up with a functional web application. For instance, React's relatively new approach of *function components* and *hooks* required some trial and error to get used to. Nevertheless, they eased later development by making the code more expressive and less verbose.

Worker pool. The client first requests the list of required packages from the server. It then downloads both the Pyodide library from a content delivery network (CDN) and the required packages from the Python Package Index (PyPI). To load and build these packages, we use an automatically generated Python file. After the initialisation steps, the client has a pool of workers (their desired count can be set by the user). Each worker creates a WebSocket connection with the server, requests the map function and streams the data items while signaling when it has finished with them. We chose to run separate WebSocket client for each worker so as not to overwhelm the main thread with all the communication. The client tries to connect to the server in a polling manner, hence, after finishing with one map workload, it can idle after a possible next one is requested by the server. The main (JavaScript) process is also kept in touch with using a callback function.

Design. We prioritised creating a responsive and cross-browser compatible frontend. The results — which ended up being subjectively aesthetic and functional — can be seen in Figures 2 and 3.

Headless Mode. For benchmarking purposes, we implemented a headless version of the client. Instead of running inside the browser in a WebWorker using WASM to execute a Python interpreter which runs our script, we can just run the script using a regular Python binary. This removes most benefits of our solution, however, it should be useful to highlight the strenghts and weaknesses of the server implementation.

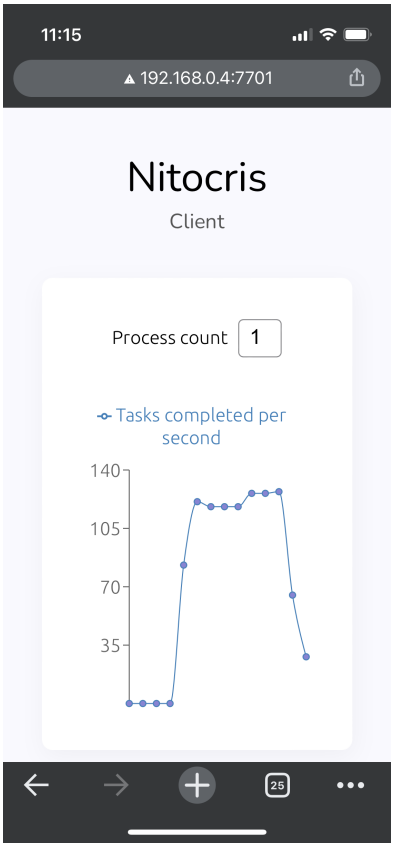

Fig. 2. The client shown using Google Chrome on an iPhone 13 mini. The website fits correctly even on very small screens. The charts show the time it took to multiply 1000 matrices and vectors of size 1024. Each time step corresponds to 5 seconds, the trapezoid shape comes from starting and finishing all the tasks.

4 Benchmarking

In our experimentation, we ran different *Map* functions both locally and in a distributed manner using NITOCRIS to see if we achieved a speedup. We derived a multitude of experiments to test both the performance of our system in different scenarios, as well as the reliability in case of failures.

To test the performance of NITOCRIS, we took inspiration for the design of our experiments from [8]. The authors, Zaharia et al. introduced both Macro- and Microbenchmarks

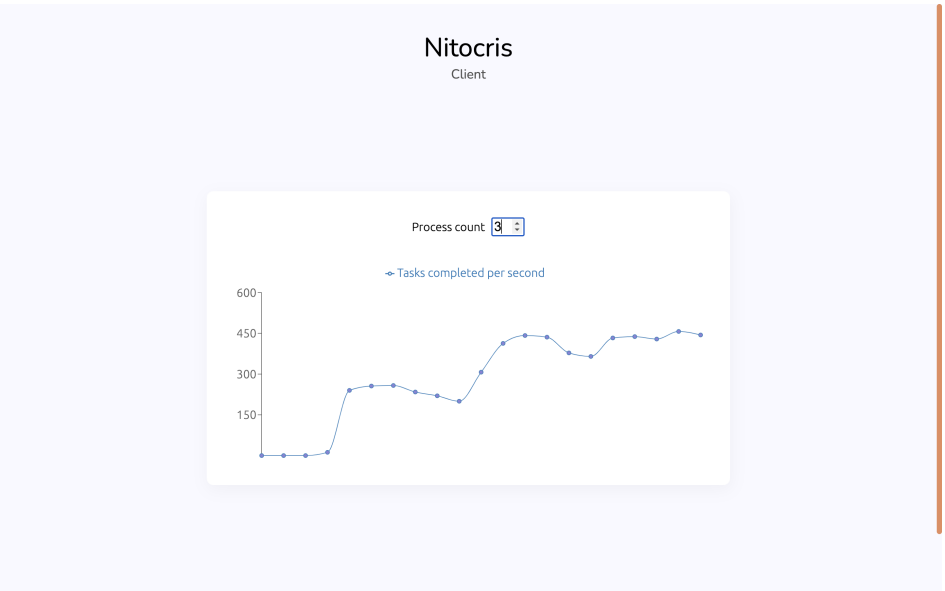

Fig. 3. The client shown using Chrome on a display size of a MacBook Air. The chart reflects how the process count was increased in 2 steps from 1 to 3.

in their paper. Microbenchmarks focused on specific features of the Delay Scheduling system, which was introduced in their paper. Due to the limited scope of this project, we opted to implement their proposed Macrobenchmarks, since their classification seemed to be a fitting distinction for our project. The Macrobenchmarks contain IO-heavy and CPU-heavy workloads.

We implemented both a map function that is highly reliant on IO, as well as one which barely has any IO but is resource intensive. The IO-heavy map function is a Matrix-Vector Multiplication. Each worker receives the matrix and one of several vectors with which it is multiplied. The CPU-heavy workload calculates the nth prime number, based on an input of n. Each worker receives a number and then calculates the nth prime using the Sieve of Eratosthenes. This algorithm takes constantly long for each prime, since it calculates all prime numbers until a set number and then finds the nth one, this is not very efficient, but staining on the CPU in a controllable manner.

For a baseline, we are calculating the built-in Python3 `map()` function and measuring the execution time. We repeat each experiment 10 times to calculate the standard deviation and a mean value. The experiments are conducted on the Distributed ASCI Supercomputer 5 (DAS-5). DAS-5 is a six-cluster wide-area distributed system designed by the Advanced School for Computing and Imaging.

We conducted further experiments to test the reliability of our system, by calculation of the error rate. We concluded that the results are hard to present since there were no errors in the execution by definition and we will therefore not go into further detail.

In this section, we will go further into detail about the results of our experiments.

4.1 IO-heavy workload

As expected, the overwhelmingly large communication overhead makes our approach inefficient for IO-heavy applications. We conducted a few experiments, however, it was clear that IO heavy workloads are a matter for future work. The results of the IO-heavy workload can be seen in Figure 4. It is important to note that the x-axis is logarithmic. With this in mind, we can see the effect of the high communication overhead. The sequential execution took around 0.04s, while the execution using NITOCRIS starts at approximately 15s. These results were to be expected.

A significant part of the communication overhead stems from the server needing to serialise the function and the input for each chunk that is sent to the client. A way to mitigate this would be to increase the chunk size and therefore reduce the number of communications needed for a specific execution.

As a reference, the experiment from Figure 4 was executed with a chunk size of one, i.e. for each communication, one element of the input list is computed. In Figure 5, we can see that with an increase in chunk size we can indeed see a performance improvement. With 15 clients and a chunk size of 50, we can see quite an odd behaviour, this behaviour of instability can be explained if we look at the size of the input vector. The input vector has 1000 elements. Once we start the experiment, 15 clients begin to calculate 15 chunks with

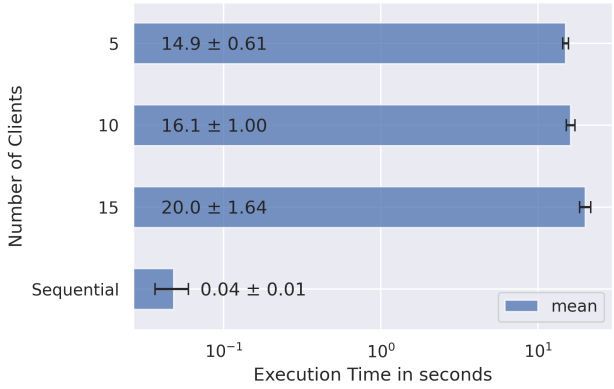

Fig. 4. The execution of the IO-heavy workload. A matrix-vector multiplication with a 1000x1000 matrix.

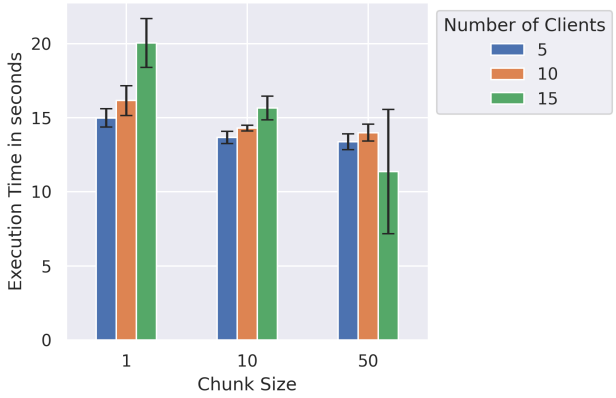

Fig. 5. The execution of the IO-heavy workload using different chunk sizes, ranging from 1 to 50 chunks.

50 elements each, i.e., 750 elements are calculated in the first "iteration". After the first clients are done fewer chunks than clients are left. Therefore, some clients will be idle and do not contribute to the execution. Consequently, the execution time is depending more on only a few clients, leading to a large variation in the values, which can be observed in Figure 5.

4.2 CPU-heavy workload

In Figure 6 we can see the results of our experimentation with a CPU-heavy workload. We observe that there is almost a linear speedup to the sequential execution, at least with five clients. We expected the CPU heavy workload to perform better than the IO-heavy workload, due to the lack of communication as a bottleneck. The results exceeded our humble expectations.

We can however also note that there is a noticeable slowdown in performance with a rising number of clients. This is

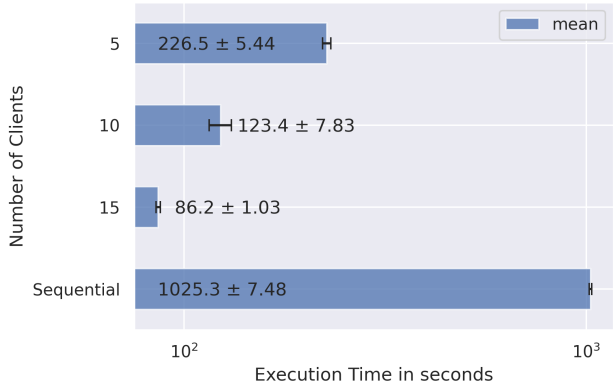

Fig. 6. The execution of the CPU-heavy workload. The workload consisted of calculating the nth prime number for 1000 numbers, using the Sieve of Eratosthenes.

to be expected and further experiments have to be conducted to find out to which extent this influences the performance when numerous clients participate in an execution. The most conspicuous reason for this decline in performance is the orchestration of tasks by the master. A future aspect of this work, therefore, is to improve the master, in terms of serialisation and deserialisation. But we are limited by the scope of the project.

Similar to the IO-heavy workload, we also experimented with different chunk sizes. But, as can be seen in Figure 5, it does not nearly have as much of an effect as with the IO-heavy workload since, as mentioned above, the communication is not the most time-consuming part of the execution.

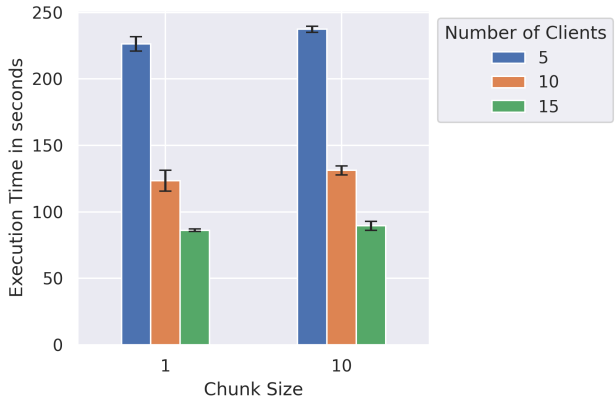

Fig. 7. The execution of the CPU-heavy workload using different chunk sizes, ranging from 1 to 50 chunks.

5 Conclusion

Overall, we designed, implemented, and benchmarked a novel, proof-of-concept distributed data processing system. We arrived at a simple-to-integrate library which provides a streamlined way of running large-scale *map* operations on a set of remote machines. Joining the calculations can be as easy as opening a link in the browser. The clients use a WebAssembly-based Python environment which automatically sets itself up, downloads the required packages, and communicates with the server while showing a user-friendly interface.

The benchmarks show a clear speedup when it comes to CPU-intensive operations. However, the serialisation/deserialisation presents immense overhead in IO-intensive computations, making our library less useful. While an increase in the chunk size shows a positive effect on the performance with the IO-heavy task, it is still not coming close to being faster than the sequential execution. A large part of the future work would therefore be a closer look at the improvement of the communication overhead, which poses a significant bottleneck.

6 Distribution of workload

We came up with the idea together after a number of brainstorming sessions. András created came up with both the server and client implementation (the Python package). Leonardo devised the experiments, ran the benchmarks and evaluated the results. Both team members documented their own work in the report.

Overall, András spent a full week working on the project amounting to about 62 hours of productive work. Leo spent 50 hours on his tasks.

References

1. T. E. Oliphant, *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.
2. R. Řehřek, P. Sojka, *et al.*, "Gensim—statistical semantics in python," *Retrieved from genism. org*, 2011.
3. D. Beazley, "Understanding the python gil," in *PyCON Python Conference. Atlanta, Georgia*, 2010.
4. J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, (USA), p. 10, USENIX Association, 2004.
5. E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning*, pp. 3053–3062, PMLR, 2018.
6. A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 185–200, 2017.
7. I. Fette and A. Melnikov, "The websocket protocol," 2011.
8. M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleey, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, (Paris, France), 04/2010 2010.