ENGENHARIA DE REDES E SISTEMAS DE COMPUTADORES
ESTG-IPVC

A Framework for Machine Learning Tasks Distribution

José Antunes & Marco Macedo

supervised by

Prof. Pedro Pinto and Prof. Silvestre Malta

14 February, 2021

Abstract

Machine Learning (ML) has become popular and it is now implemented in various domains as it allows to extract valuable information out of the massive amounts of data that are being generated these days. The process of building and deploying a ML model is divided in data related tasks and modeling related tasks. In modeling related tasks the training one is the most time-consuming. [8]

As we said before, the training part is the most time-consuming and that is the part that we focus on this project. If we had more computers to execute the same tasks/code, logically the output result will become ready earlier than if we were just using one machine. So, we don't have the computers/servers needed and there is were Fed4Fire takes place, all the computation that we need to execute this project is inside of this platform. Now that we have computation we just need to program this system in order to distribute all the code that it will be executed, shortening the execution time spent.

This project intends to build a framwework to implement distributed ML tasks in Fed4FIRE+, the largest federation worldwide of Next Generation Internet (NGI)) testbeds. Our proposal, has the object of do this experience using 5 nodes(Machines) of Fed4Fire, and later if all comes fine use more and more machines because, more machines means less time wasted.

We present final tests with all the features that we have.

Contents

Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CLI	Command Line Interface
CPU	Computer Processing Unit
CPUs	Central Process Unit
ERSC	Engenharia de Redes e Sistemas de Computadores
GENI	Global Environment for Network Innovations
GPU	Graphic Processing Unit
GUI	Graphical User Interface
ID	Identification
IoT	Internet of Things
IP	Internet Protocol
LAN	Local Area Network
ML	Machine Learning
NGI	Next Generation Internet
PPP	Public Private Partnership

SDK	Software development kit
VM	Virtual Machine
WAN	Wide Area Network
WN	Wireless Network

Chapter 1

Introduction

This first chapter presents what is a brief explanation of how this project came about, what was the motivation for doing so and the objectives on which we focused on the general and several specific ones.

1.1 Context and Motivation

”His project arises from a suggestion proposed by professors of the Engenharia de Redes e Sistemas de Computadores (ERSC) course. This project appears as a way to implement for the first time, on a platform that is growing in the market of network virtualization, distribution frameworks and computational parallelization involving Machine Learning (ML) scripts.

1.2 Objective

The main goal on this project is to develop a framework based on software ”Ray” for Machine Learning Tasks Distribution on the Fed4Fire platform. As complementary objectives we want to discover some new platforms/software, bring Machine Learning as an subject were the interest of the team is shared, get some knowledge with related works, get out of our comfort zone and embrace new projects in innovative areas.

1.3 Organization

In Chapter two presents the Background and Related Work, where we approach ML Fundamentals, how it works, ML methods of learning, and some platforms.

In Chapter 3, we talk about Machine Learning and we show how can we distribute tasks using one platform.

In Chapter Four it is presented the final results and analysis of the tests that we performed during the work with machine learning and associated work in distributed computing

In the last and fifth chapter, which is the conclusion chapter, we point out some conclusions and possible future works.

Chapter 2

Background and Related Work

In this chapter we summarize all the topics that we had to study.

2.1 Distributed System

A distributed system is a collection of independent computers that appears to its users as a single coherent system. Distributed Computing is a model in which components of a software system are shared among multiple computers to improve performance and efficiency All the computers are tied together in a network either a Local Area Network (LAN) or Wide Area Network (WAN), communicating with each other so that different portions of a Distributed application run on different computers from any geographical location. A computer program that runs within a distributed system is called a distributed program. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other via message passing.

The Advantages of deploying a distributed system are presented in.

- Reliability, high fault tolerance.
- Scalability: In distributed computing systems you can add more machines as needed.
- Flexibility: It makes it easy to install, implement and debug new services.
- Fast calculation speed: A distributed computer system can have the computing power of multiple computers, making it fast1er than other systems.

- Openness: Since it is an open system, it can be accessed both locally and remotely.
- High performance: Compared to centralized computer network clusters, it can provide higher performance and better cost performance

The Disadvantages of deploying a distributed system:

- Difficult troubleshooting:Troubleshooting and diagnostics are more difficult due to distribution across multiple nodes.
- Less software support
- High network infrastructure costs
- Security issues

2.2 Parallel Computing Platforms

In this next section we present what are some of the most used distribution and parallelization platforms.

Parallel computing is a programming method that harnesses the power of multiple processors at once. Once of concern only to programmers of large supercomputers, modern computers now almost always have multi-core processors.

- Dask: From the outside, Dask looks a lot like Ray. It, too, is a library for distributed parallel computing in Python, with its own task scheduling system, awareness of Python data frameworks like NumPy, and the ability to scale from one machine to many
- Pandaral·lel, as the name implies, is a way to parallelize Pandas jobs across multiple nodes. The downside is that Pandaral·lel works only with Pandas. But if Pandas is what you're using, and all you need is a way to accelerate Pandas jobs across multiple cores on a single computer, Pandaral·lel is laser-focused on the task.
- Ipyparallel: is another tightly focused multiprocessing and task-distribution system, specifically for parallelizing the execution of Jupyter notebook code across a cluster.

Projects and teams already working in Jupyter can start using Ipyparallel immediately.

- Joblib has two major goals: run jobs in parallel and don't recompute results if nothing has changed. These efficiencies make Joblib well-suited for scientific computing, where reproducible results are sacrosanct. Joblib's documentation provides plenty of examples for how to use all its features.

- Python-related platforms

  - Python default libraries: Python does include a native way to run a Python workload across multiple Central Process Unit (CPUs). The multiprocessing module spins up multiple copies of the Python interpreter, each on a separate core, and provides primitives for splitting tasks across cores. But sometimes even multiprocessing isn't enough. Sometimes the job calls for distributing work not only across multiple cores, but also across multiple machines. That's where these six Python libraries and frameworks come in. All six of the Python tool kits below allow you to take an existing Python application and spread the work across multiple cores, multiple machines, or both.

  - Dispy: lets you distribute whole Python programs or just individual functions across a cluster of machines for parallel execution. It uses platform-native mechanisms for network communication to keep things fast and efficient, so Linux, MacOS, and Windows machines work equally well.

  - Ray Developed by a team of researchers at the University of California, Berkeley, Ray underpins a number of distributed machine learning libraries. But Ray isn't limited to machine learning tasks alone, even if that was its original use case. Any Python tasks can be broken up and distributed across systems with Ray.

2.3 Platforms for Distributed Computing

The are a few platforms that allow us to implement and test our experiments on their own hardware for free as long as our test beds are federated.

- GENI
- Fed4Fire

Global Environment for Network Innovations (GENI) [3] is a project under an American Programme with severals users around the world, provides a virtual laboratory for networking and distributed systems research and education.

- Obtain compute resources from locations around the United States;
- Connect compute resources using Layer 2 networks in topologies best suited to their experiments;
- Install custom software or even custom operating systems on these compute resources;
- Control how network switches in their experiment handle traffic flows;
- Run their own Layer 3 and above protocols by installing protocol software in their compute resources and by providing flow controllers for their switches.

Fed4FIRE+ [1] is a project under the European Union's Programme Horizon 2020, offering the largest federation worldwide of Next Generation Internet (NGI) testbeds, which provide open, accessible and reliable facilities supporting a wide variety of different research and innovation communities and initiatives in Europe, including the 5G Public Private Partnership (PPP) projects and initiatives. It started in January 2017 and will run for 60 months, until the end of December 2021. The Fed4FIRE+ project is the successor of the Fed4FIRE project. About Fed4Fire, we can only thanks for the resources provided and the hostility. Without them, we weren't able to present you this.

In particular, inside of this whole structure that Fed4Fire is there are sections, and one of the most matter sections to our project was GPULab.

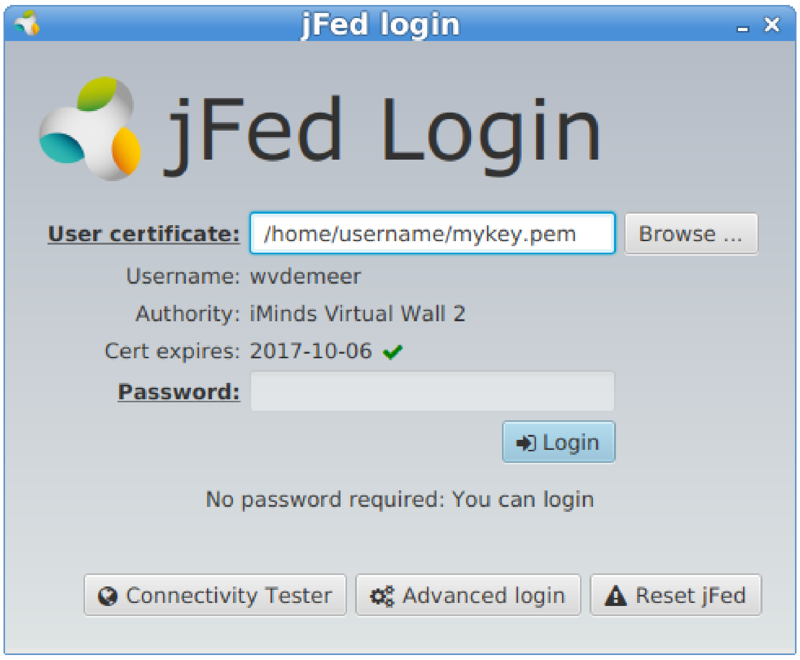
GPULab is a distributed system for running jobs in GPU-enabled Docker-containers. GPULab consists out of a set of heterogeneous clusters, each with their own characteristics (GPU model, CPU speed, memory, bus speed, ...), allowing you to select the most appropriate hardware. Each job runs isolated within a Docker containers with dedicated CPU's, GPU's and memory for maximum performance.

Fed4Fire makes it possible to learn the testbed federation architecture, workflows and Application Programming Interfaces (APIs), and makes it also easy to develop java based client tools for testbed federation. The suite is built around the low level library, which implements the client side for all the supported APIs; and a high level library, which manages and keeps track of the lifecycle of an experiment. On top of these libraries various components were developed to allow thorough examination and testing of these APIs, as well as an user-friendly graphical Experimenter GUI to allow end-users to use the testbeds.

The most important components are presented in.

- jFed Experimenter Graphical User Interface (GUI) allows end-users to provision and manage experiments.
- jFed Experimenter GUI allows end-users to provision and manage experiments.
- jFed Automated tester performs extensive full-automated tests of the testbed APIs, in which the complete workflow of an experiment is followed. This tool is used as part of the Fed4FIRE testbed monitor.

All testbeds in Fed4FIRE+ are available through an AM API. Within Fed4FIRE+ a GUI and Command Line Interface (CLI) called jFed was developed for easy provisioning of resources and topologies.



User certificate: /home/username/mykey.pem
Browse ...
Username: wvdemeer
Authority: iMinds Virtual Wall 2
Cert expires: 2017-10-06 ✓
Password:
Login
No password required: You can login
Connectivity Tester
Advanced login
Reset jFed

Figure 2.1: Fed4Fire JFed Experiments

2.4 Distributed Machine Learning

Machine learning is the science of getting computers to act without being explicitly programmed. In the past decade, machine learning has given us self-driving cars, practical speech recognition, effective web search,etc. Machine learning is so pervasive today that you probably use it a lot of times a day without knowing it. Many researchers also think that's the best way to make progress towards human-level AI.

The study of computer algorithms that improve automatically through out experience. It is seen as a part of Artificial Intelligence (AI). ML algorithms build a model based on sample data (Datasets), known as ”training data”, in order to make predictions or decisions without being explicitly programmed to do so. This algorithms are used on a several variety of applications, such as email filtering and computer vision, where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks.[18]

Data set represent the data that machine learning model will ingest to solve the problem it's developed to solve. In some cases, the training data is labeled data—'tagged' to call out features and classifications the model will need to identify. Other data is unlabeled, and the model will need to extract those features and assign classifications on its own.[5]

Training the algorithm is an iterative process–it involves running variables through the algorithm, comparing the output with the results it should have produced, and running the variables again until the algorithm returns the correct result most of the time. On the script used do execute ML it had already 8 models defined.[6] Machine learning methods:

- Supervised machine learning: Trains itself on a labeled data set. That is, the data is labeled with information that the machine learning model is being built to determine and that may even be classified in ways the model is supposed to classify data. This is the method we use on our Architecture.
- Unsupervised machine learning: Ingests unlabeled data and lots of it uses algorithms to extract meaningful features needed to label, sort, and classify the data in real-time, without human intervention. Unsupervised learning is less about automating decisions and predictions, and more about identifying patterns and relationships in data that humans would miss. Take spam detection, for example—people generate more email than a team of data scientists could ever hope to label or classify in their lifetimes.
- Semi-supervised learning: Offers a happy medium between supervised and unsupervised learning. During training, it uses a smaller labeled data set to guide classification and feature extraction from a larger, unlabeled data set. Semi-supervised learning can solve the problem of having not enough labeled data (or not being able to afford to label enough data) to train a supervised learning algorithm.

The demand for AI has grown significantly over the last decade and this growth has been fueled by advances in machine learning techniques and the ability to leverage hardware acceleration.

However, in order to increase the quality of predictions and render machine learning solutions feasible for more complex applications, a substantial amount of training data is required.

The state of the art is based on related works within the scope of the work in question, in this case we illustrate some examples of task distribution, parallelization and serialization.

The Azure Machine Learning Software development kit (SDK) in Python supports integrations with popular frameworks, PyTorch and TensorFlow. Both tables employ data parallelism for distributed training, and can use horovod to optimize base.

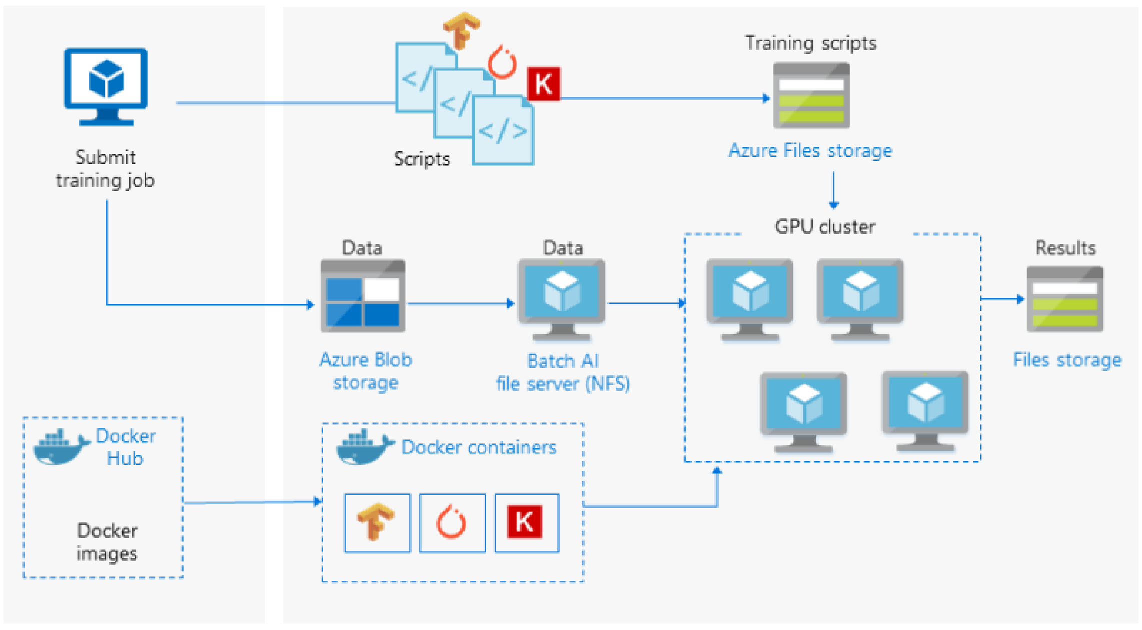


Figure 2.2: Reference Architecture: Distributed training of deep learning models on Azure - from [9]

This reference architecture shows how to conduct distributed training of deep learning models across clusters of Computer Processing Unit (CPU)-enabled Virtual Machines (VMs). The scenario is image classification, but the solution can be generalized for other deep-learning scenarios, such as segmentation and object detection.

In [17] the authors propose an distributed learning system that enables edge devices to collaboratively learn a shared model while keeping all the raw data stored distributively at the edge. The system estimates parameters related to data distribution and resource consumption, and adapts the learning process based on these estimations in real time.

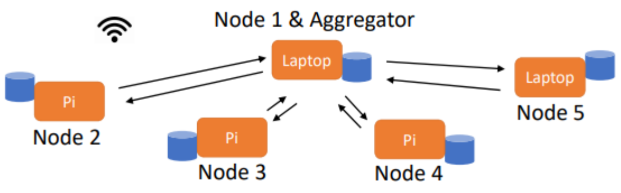


Figure 2.3: Setup example - from [17]

Chapter 3

Machine Learning Tasks Distribution Platform

On our test bed created in Fed4Fire we decided that we just have time to test with one framework to distribute tasks. So for that reason, we make some research and decide that for our project only two frameworks correspond to our necessities. We preformed a table with some aspects to compare between Ray Framework and Dispy Framework.

The work developed consisted of the following steps:

- Parallel Platforms Analysis;
- Testbed Architecture;
- Machine Learning environment setup;
- Framework Ray Installation;
- Non-Parallelized and Parallelized Scripts;
- Distributed and Parallelized Scripts:
- Machine Learning Script.

3.1 Parallel Platforms Analysis

Between these two platforms we chose ray, which in addition to being more efficient and more used has a very important feature that makes any programmer choose this platform, which is ray, because for the execution of tasks you have the possibility to use Graphic Processing Unit (GPU), the dispy on the other hand does not.

	Dispy	Ray
Asynchronous	✓	✓
Distributed/parallel computing	✓	✓
Scalability	✓	✓
Cluster manager	✗	✓
Fault recovery	✓	✓
Uses CPU	✓	✓
Uses GPU	✗	✓
Easy to manage and store	✓	✓
Use SSL encryption when transferring data	✓	✓
Can return provisional or partially completed results	✓	✗
Syntax is minimal	✗	✓
Open Source	✓	✓

The Dispy features are the following.

- Client-side and server-side fault recovery are supported.
- Asynchronous.
- Support distributed/parallel computing .
- Computation nodes can be anywhere on the Network.
- After each execution is finished, the results of execution, output, errors and exception trace are made available for further processing.
- Nodes may become available dynamically: dispy will schedule jobs whenever a node is available.
- If callback function is provided, dispy executes that function when a job is finished; this can be used for processing job results as they become available.

- dispy can be used in a single process to use all the nodes exclusively (with JobCluster - simpler to use) or in multiple processes simultaneously sharing the nodes (with SharedJobCluster and dispyscheduler program).
- Cluster can be monitored and managed with web browser.

These are the Ray features:

- Provides recovery from process and machine failures.
- Asynchronous.
- Support distributed/parallel computing .
- Ray's syntax is minimal, so you don't need to rework existing apps extensively to parallelize them.
- Ray even includes its own built-in cluster manager, which can automatically spin up nodes as needed on local hardware or popular cloud computing platforms.

As Robert Nashihara quote [12]: *Ray is a fast, simple framework for building and running distributed applications. Ray leverages Apache Arrow for efficient data handling and provides tasks and actors abstractions for distributed computing.*

On the Fig. 3.1, a machine with 48 physical cores, Ray is 9x faster than Python multiprocessing and 28x faster than single-threaded Python, the benchmark of three workloads, compares Ray, Python Multiprocessing and serial Python code.

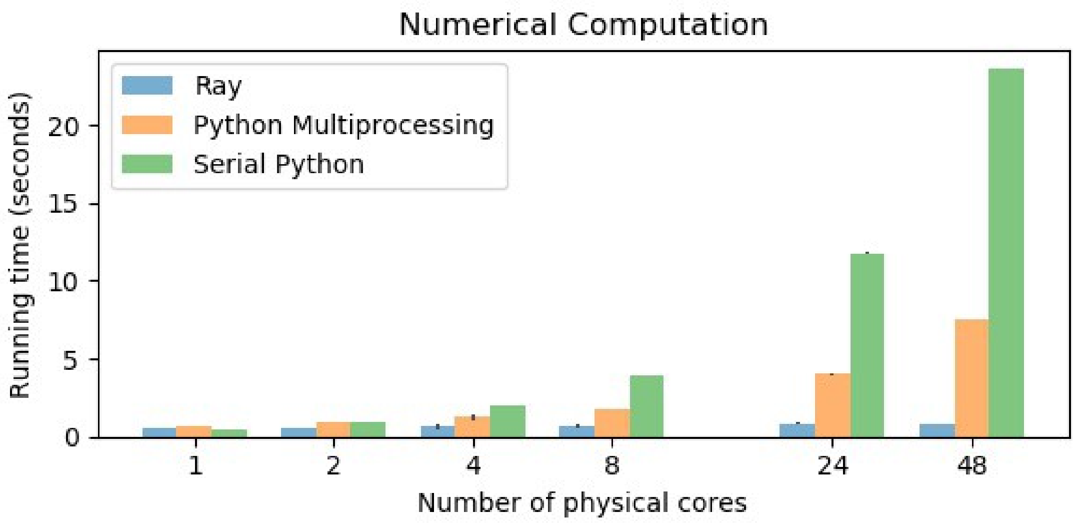


Figure 3.1: Image from- [15]

As we can noticed Ray is 10-30x faster than Serial Python, is 5-25x faster than multiprocessing, and 5-15x faster than the faster of these two on a large machine.[13]

While Python is used for a wide range of applications:

- it lacks on handling of numerical data.
- The abstractions for stateful computation(i.e tasks can't share variables between separated "tasks").

On a machine with 48 physical cores, Ray is 6x faster than Python multiprocessing and 17x faster than single-threaded Python. Python multiprocessing does not outperform single-threaded Python on fewer than 24 cores. The workload is scaled to the number of cores, so more work is done on more cores (which is why serial Python takes longer on more cores).

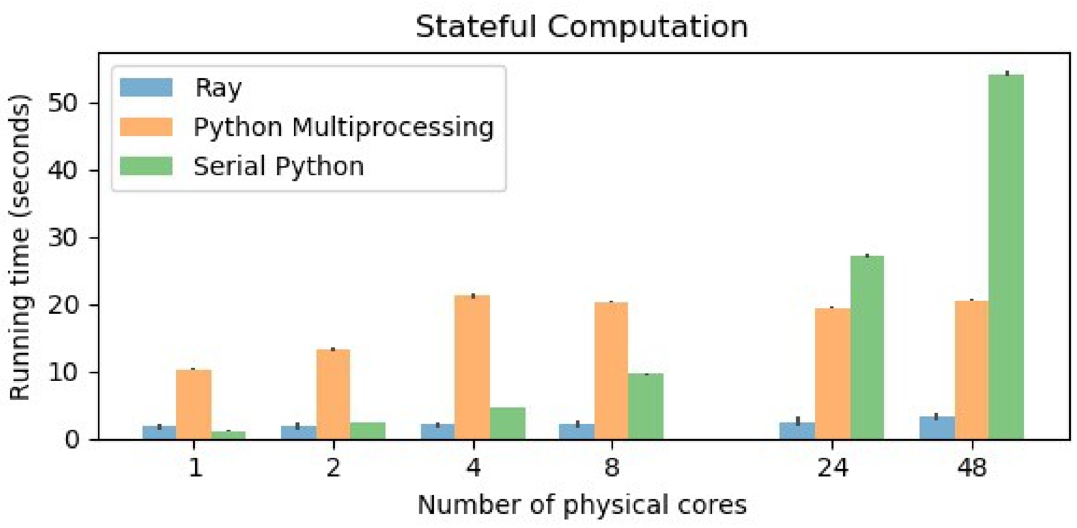


Figure 3.2: Image from- [15]

3.2 Testbed Architecture

Initially our topology, was composed by 5 nodes, since one node of them was declared as Head Node, Node0 and the others were claimed by Workers, Node1/2/3/4.

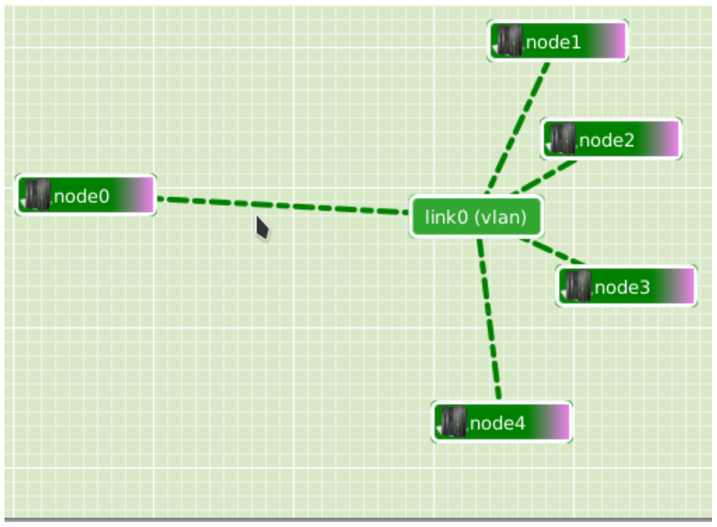
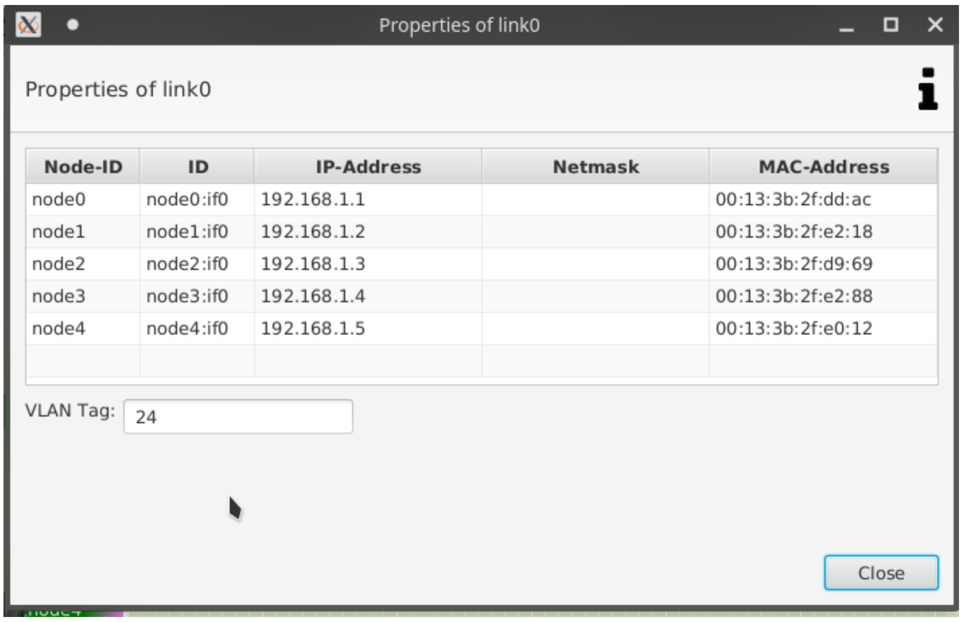


Figure 3.3: Network Topology

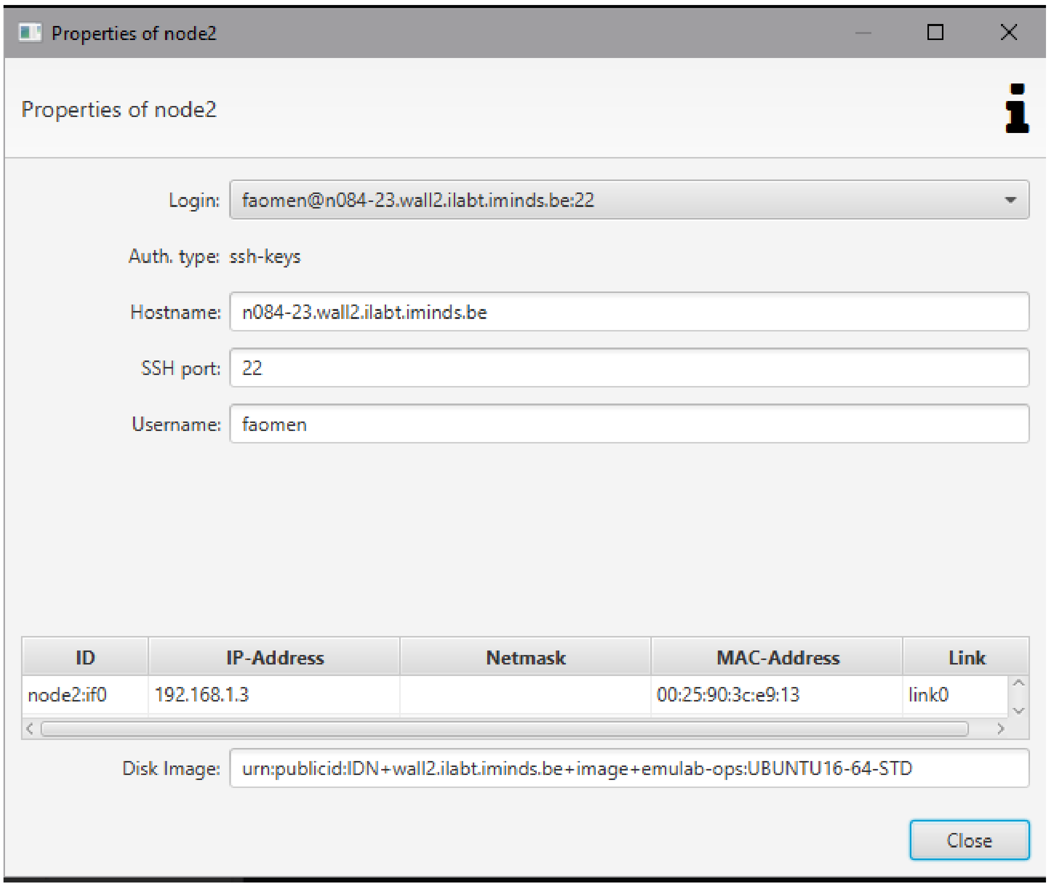
These nodes are Machines provided by Fed4Fire, they are connect to an Vlan(link0), their Internet Protocol (IP) we choose to be automatic but you can configure them manually.


Properties of link0

Node-ID	ID	IP-Address	Netmask	MAC-Address
node0	node0:if0	192.168.1.1		00:13:3b:2f:dd:ac
node1	node1:if0	192.168.1.2		00:13:3b:2f:e2:18
node2	node2:if0	192.168.1.3		00:13:3b:2f:d9:69
node3	node3:if0	192.168.1.4		00:13:3b:2f:e2:88
node4	node4:if0	192.168.1.5		00:13:3b:2f:e0:12

VLAN Tag: 24

Figure 3.4: Link 0 Configuration


Properties of node2

Login: faomen@n084-23.wall2.ilabt.iminds.be:22

Auth. type: ssh-keys

Hostname: n084-23.wall2.ilabt.iminds.be

SSH port: 22

Username: faomen

ID	IP-Address	Netmask	MAC-Address	Link
node2:if0	192.168.1.3		00:25:90:3c:e9:13	link0

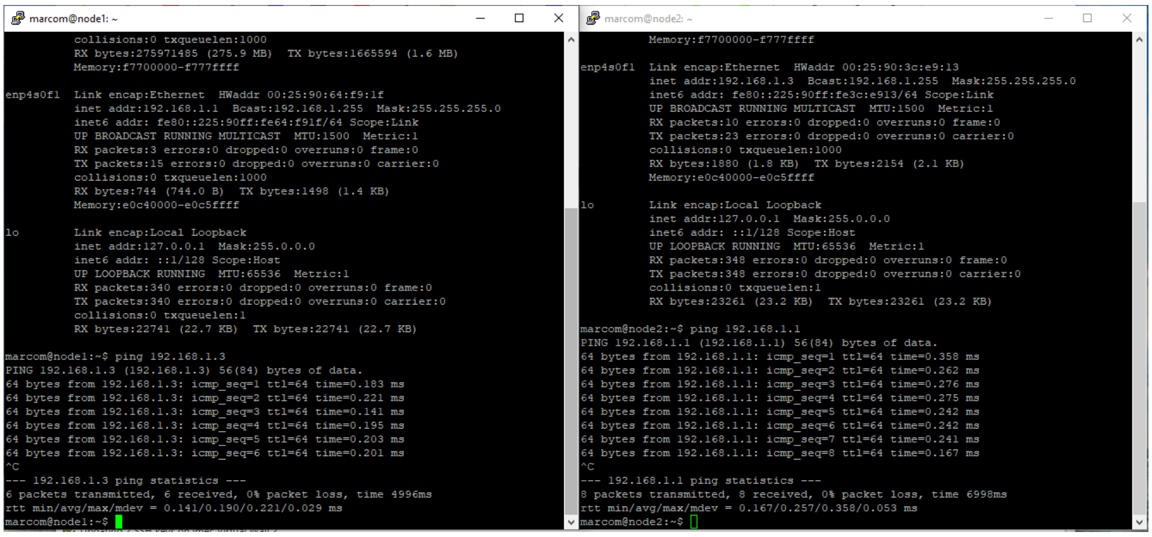
Disk Image: urn:publicid:IDN+wall2.ilabt.iminds.be+image+emulab-ops:UBUNTU16-64-STD

Figure 3.5: Topology

Fig. 3.2 shows an example of a node configuration on the Fed4Fire platform, in this case node 2, which will be executed on Virtual Wall 2, which is available on Jfed and we use the "image": UBUNTU20.04 at the latest updates.

We chose to use Virtual Wall 2 because only on this testbed is it possible to use GPU nodes and in order to be able to do Machine learning using the RAY framework it will be necessary to use GPU in its execution.

In Fig. 3.6, we present some communication tests.



```
          collisions:0 txqueuelen:1000
          RX bytes:275971485 (275.9 MB)  TX bytes:1665594 (1.6 MB)
          Memory:f7700000-f777ffff

enp4s0f1  Link encap:Ethernet  HWaddr 00:25:90:64:f9:1f
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::225:90ff:fe64:f91f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:744 (744.0 B)  TX bytes:1498 (1.4 KB)
          Memory:e0c40000-e0c5ffff

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:340 errors:0 dropped:0 overruns:0 frame:0
          TX packets:340 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:22741 (22.7 KB)  TX bytes:22741 (22.7 KB)

marcom@node1:~$ ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.183 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=0.221 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=0.141 ms
64 bytes from 192.168.1.3: icmp_seq=4 ttl=64 time=0.195 ms
64 bytes from 192.168.1.3: icmp_seq=5 ttl=64 time=0.203 ms
64 bytes from 192.168.1.3: icmp_seq=6 ttl=64 time=0.201 ms
^C
--- 192.168.1.3 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 4996ms
rtt min/avg/max/mdev = 0.141/0.190/0.221/0.029 ms
marcom@node1:~$
```

```
          Memory:f7700000-f777ffff

enp4s0f1  Link encap:Ethernet  HWaddr 00:25:90:3c:e9:13
          inet addr:192.168.1.3  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::225:90ff:fe3c:e913/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:23 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1880 (1.8 KB)  TX bytes:2154 (2.1 KB)
          Memory:e0c40000-e0c5ffff

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:348 errors:0 dropped:0 overruns:0 frame:0
          TX packets:348 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:23261 (23.2 KB)  TX bytes:23261 (23.2 KB)

marcom@node2:~$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.358 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.262 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.276 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=0.275 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=0.242 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=0.242 ms
64 bytes from 192.168.1.1: icmp_seq=7 ttl=64 time=0.241 ms
64 bytes from 192.168.1.1: icmp_seq=8 ttl=64 time=0.167 ms
^C
--- 192.168.1.1 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 6998ms
rtt min/avg/max/mdev = 0.167/0.257/0.358/0.053 ms
marcom@node2:~$
```

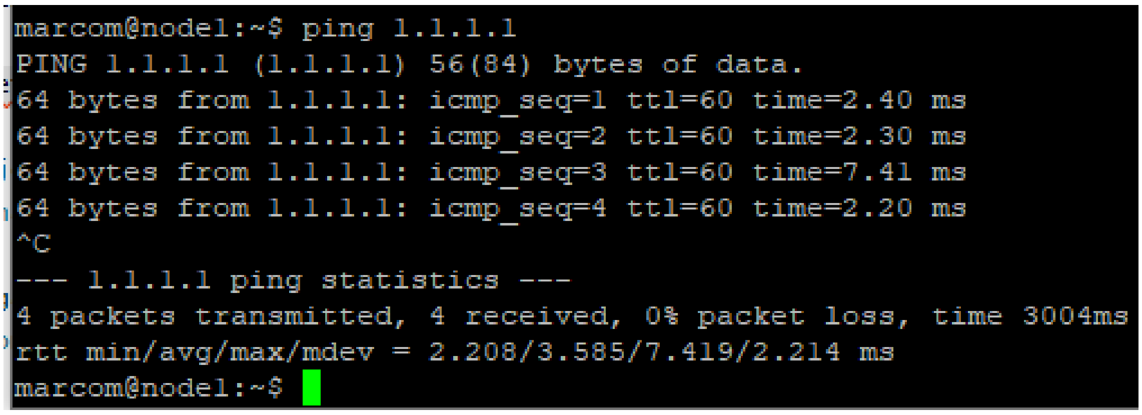
Figure 3.6: Machines communicating

We set the nodes 1 and 2 exchanging "pings" between the two to see if there was communication, and there it was. The topology is ready to go.

After having the topology set up and the nodes in communication, we noticed that the machines did not have access to the internet, DNS/Gateway Problems (this problem surger when we tried to update the nodes). To solve this question, just a little research on imce.lab documentation and we found the solution:

```
wget -O - -nv https://www.wall2.ilabt.iminds.be/enable-nat.sh | sudo bash
```

After the previously command we should test your internet connection as we have done it on Fig.3.2.



```
marcom@node1:~$ ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=60 time=2.40 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=60 time=2.30 ms
64 bytes from 1.1.1.1: icmp_seq=3 ttl=60 time=7.41 ms
64 bytes from 1.1.1.1: icmp_seq=4 ttl=60 time=2.20 ms
^C
--- 1.1.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 2.208/3.585/7.419/2.214 ms
marcom@node1:~$
```

Figure 3.7: Machines now have access to Internet

In order to update all the machines you must execute the following commands.

```
sudo apt-get update \&\& sudo apt-get -full-upgrade -y
```

3.3 Environment setup

In order to setup the ML environment some tools some tools were installed according to the requisites defined in the be Anaconda was installed is very useful in this setup, being possible to aggregate all the tools we need on a virtual environment making much easier the installation of the tools using Anaconda bash. Download the Anaconda installation script with your web browser or wget :

```
wget -P /tmp https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-
    x86_64.sh
```

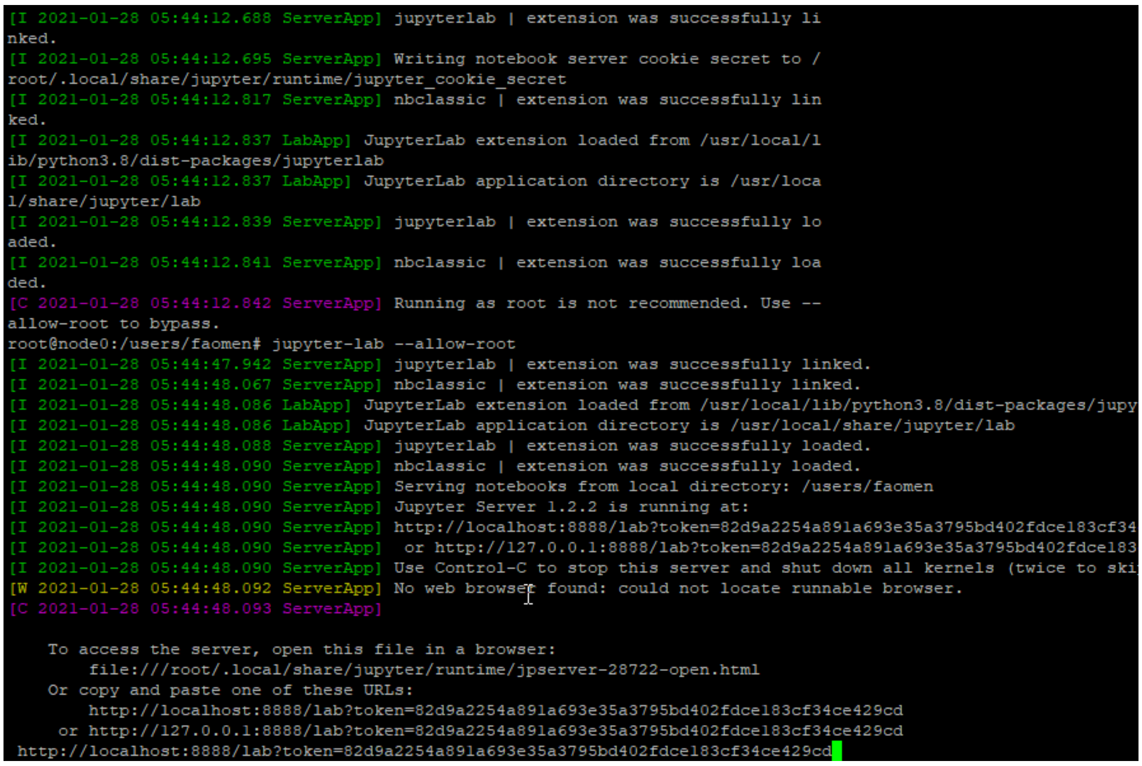
```
bash /tmp/Anaconda3-2020.02-Linux-x86_64.sh
```

```
source ~/.bashrc
```

In order to use an text editor to create and edit our scripts, instead of using 'nano' we use Jupyter Editor. JupyterHub [4] allows you to launch Jupyter notebooks on the imec iLab.t computing infrastructure with a simple mouseclick. It gives you access to an interactive environment where you can use Python, R, Julia, etc. for your research and data processing. We offer both plain and GPU-enabled environments Install Jupyter-Lab on the head Node, in our case, Node0 by typing the command.

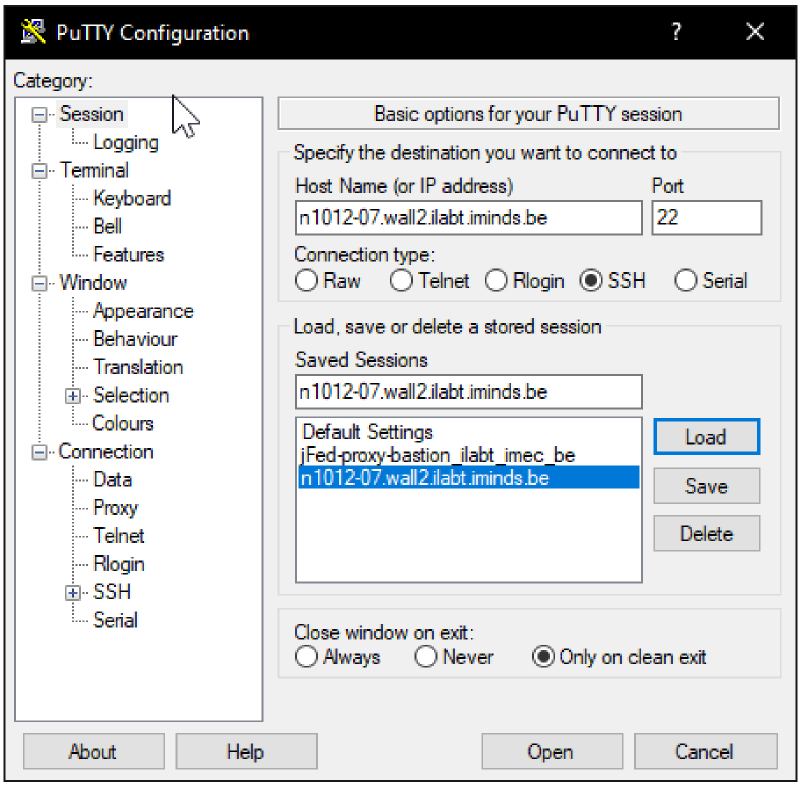
```
pip3 install jupyter-lab
```

Now to start the server you just have to call the program, jupyter-lab, and it's done (usually the port is 8888 but if that port is being occupied it will choose another free.See Fig. 3.8 and consult documentation [14],[7]). So, by now you have installed the service but our ”problem” was that the Jupyter Server was installed on a ”server”, as we know jupyter launches on LocalHost, and we can not just type the command *http://localhost:8888/* cause we do not have any service on our localhost listening on that port. So as a solution after a long research, we decided to use Putty to preform a tunnel to the ”localHost” of our Node0 (see Figs. 3.9 and 3.10). Now is possible to type *http://localhost:8888/* on adddress bar on browser and have access to the editor(see Fig. 3.11).



```
[I 2021-01-28 05:44:12.688 ServerApp] jupyterlab | extension was successfully li
nked.
[I 2021-01-28 05:44:12.695 ServerApp] Writing notebook server cookie secret to /
root/.local/share/jupyter/runtime/jupyter_cookie_secret
[I 2021-01-28 05:44:12.817 ServerApp] nbclassic | extension was successfully lin
ked.
[I 2021-01-28 05:44:12.837 LabApp] JupyterLab extension loaded from /usr/local/l
ib/python3.8/dist-packages/jupyterlab
[I 2021-01-28 05:44:12.837 LabApp] JupyterLab application directory is /usr/loca
l/share/jupyter/lab
[I 2021-01-28 05:44:12.839 ServerApp] jupyterlab | extension was successfully lo
aded.
[I 2021-01-28 05:44:12.841 ServerApp] nbclassic | extension was successfully loa
ded.
[C 2021-01-28 05:44:12.842 ServerApp] Running as root is not recommended. Use --
allow-root to bypass.
root@node0:/users/faomen# jupyter-lab --allow-root
[I 2021-01-28 05:44:47.942 ServerApp] jupyterlab | extension was successfully linked.
[I 2021-01-28 05:44:48.067 ServerApp] nbclassic | extension was successfully linked.
[I 2021-01-28 05:44:48.086 LabApp] JupyterLab extension loaded from /usr/local/lib/python3.8/dist-packages/jupy
[I 2021-01-28 05:44:48.086 LabApp] JupyterLab application directory is /usr/local/share/jupyter/lab
[I 2021-01-28 05:44:48.088 ServerApp] jupyterlab | extension was successfully loaded.
[I 2021-01-28 05:44:48.090 ServerApp] nbclassic | extension was successfully loaded.
[I 2021-01-28 05:44:48.090 ServerApp] Serving notebooks from local directory: /users/faomen
[I 2021-01-28 05:44:48.090 ServerApp] Jupyter Server 1.2.2 is running at:
[I 2021-01-28 05:44:48.090 ServerApp] http://localhost:8888/lab?token=82d9a2254a891a693e35a3795bd402fdce183cf34
[I 2021-01-28 05:44:48.090 ServerApp]  or http://127.0.0.1:8888/lab?token=82d9a2254a891a693e35a3795bd402fdce183
[I 2021-01-28 05:44:48.090 ServerApp] Use Control-C to stop this server and shut down all kernels (twice to ski
[W 2021-01-28 05:44:48.092 ServerApp] No web browser found: could not locate runnable browser.
[C 2021-01-28 05:44:48.093 ServerApp]

    To access the server, open this file in a browser:
        file:///root/.local/share/jupyter/runtime/jpserver-28722-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/lab?token=82d9a2254a891a693e35a3795bd402fdce183cf34ce429cd
     or http://127.0.0.1:8888/lab?token=82d9a2254a891a693e35a3795bd402fdce183cf34ce429cd
 http://localhost:8888/lab?token=82d9a2254a891a693e35a3795bd402fdce183cf34ce429cd
```

Figure 3.8: Jupyter Server on Node0


PuTTY Configuration

Category:

- Session
  - Logging
- Terminal
  - Keyboard
  - Bell
  - Features
- Window
  - Appearance
  - Behaviour
  - Translation
  - Selection
  - Colours
- Connection
  - Data
  - Proxy
  - Telnet
  - Rlogin
  - SSH
  - Serial

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): n1012-07.wall2.ilabt.iminds.be

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

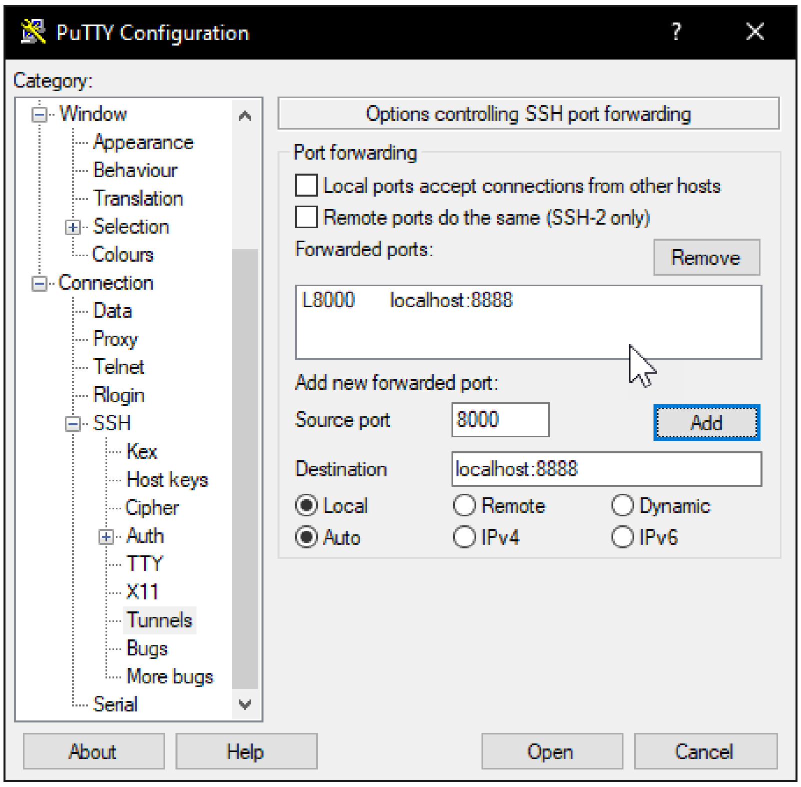
Load, save or delete a stored session

Saved Sessions: n1012-07.wall2.ilabt.iminds.be

- Default Settings
- jFed-proxy-bastion_ilabt_imec_be
- n1012-07.wall2.ilabt.iminds.be

Close window on exit: Always Never [✓ Only on clean exit]

Figure 3.9: Putty Configuration


PuTTY Configuration

Category:

- Window
  - Appearance
  - Behaviour
  - Translation
  - Selection
  - Colours
- Connection
  - Data
  - Proxy
  - Telnet
  - Rlogin
  - SSH
    - Kex
    - Host keys
    - Cipher
    - Auth
    - TTY
    - X11
    - Tunnels
    - Bugs
    - More bugs
  - Serial

Options controlling SSH port forwarding

Port forwarding

☐ Local ports accept connections from other hosts

☐ Remote ports do the same (SSH-2 only)

Forwarded ports:

```
L8000     localhost:8888
```

Add new forwarded port:

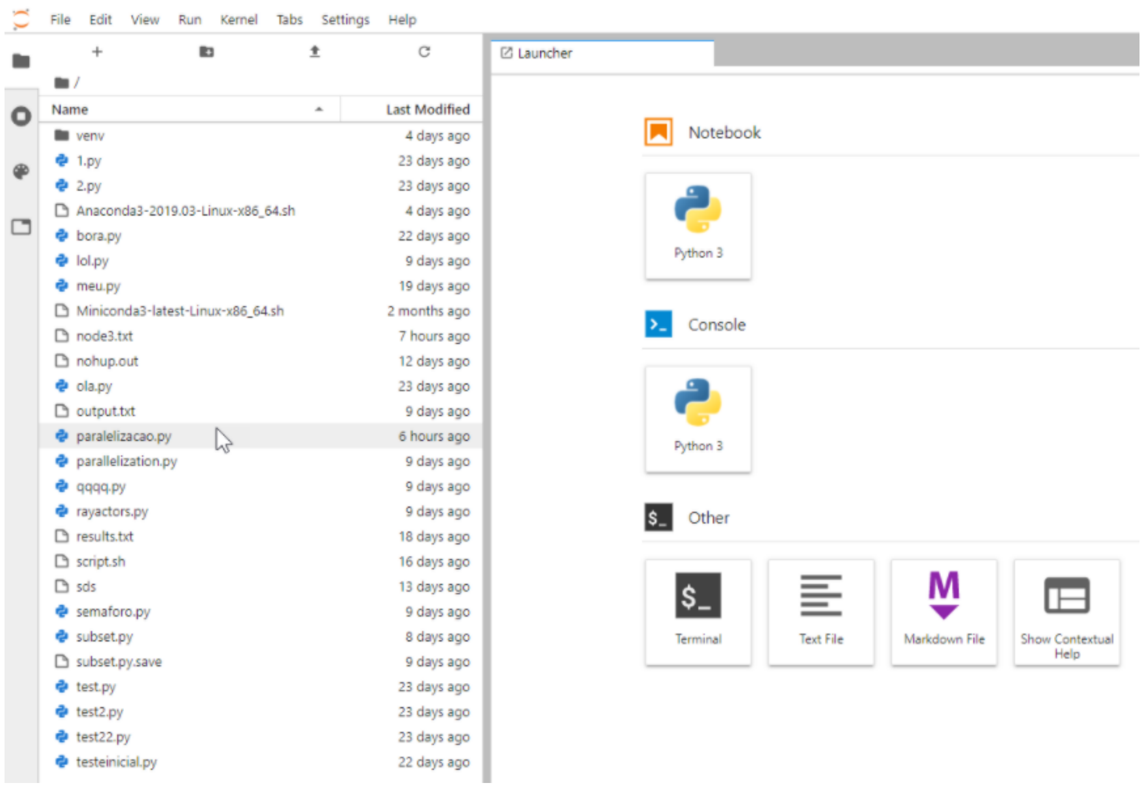
Source port: 8000

Destination: localhost:8888

[✓ Local] Remote Dynamic

[✓ Auto] IPv4 IPv6

Figure 3.10: Putty Tunnel to the localHost on the Fed4Fire infastructure



Name	Last Modified
venv	4 days ago
1.py	23 days ago
2.py	23 days ago
Anaconda3-2019.03-Linux-x86_64.sh	4 days ago
bora.py	22 days ago
lol.py	9 days ago
meu.py	19 days ago
Miniconda3-latest-Linux-x86_64.sh	2 months ago
node3.txt	7 hours ago
nohup.out	12 days ago
ola.py	23 days ago
output.txt	9 days ago
paralelizacao.py	6 hours ago
parallelization.py	9 days ago
qqqq.py	9 days ago
rayactors.py	9 days ago
results.txt	18 days ago
script.sh	16 days ago
sds	13 days ago
semaforo.py	9 days ago
subset.py	8 days ago
subset.py.save	9 days ago
test.py	23 days ago
test2.py	23 days ago
test22.py	23 days ago
testeinicial.py	22 days ago

Figure 3.11: Jupyter Lab on Node0

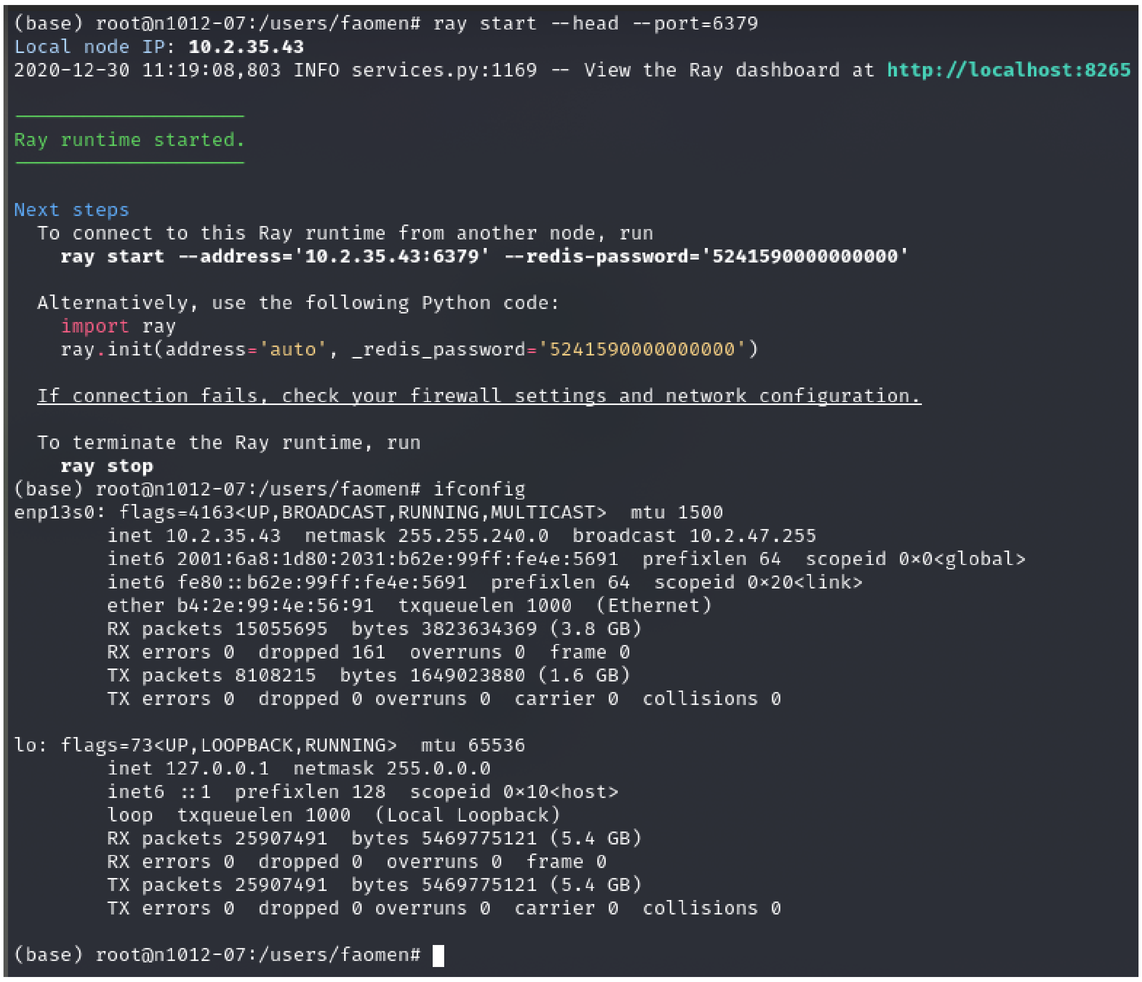
Then, for the environment to be more complete we will install ray. The best way and the only way that we found way to install Ray, was on their website `https://docs.ray.io/en/latest/installation.html`. Normally the pip installer is enough but if not follow the tutorial.[16]

```
pip3 install -U ray
```

Check if the version on all nodes is the same, if not you must run the command, and update all too the last wheel.

```
ray install-nightly
```

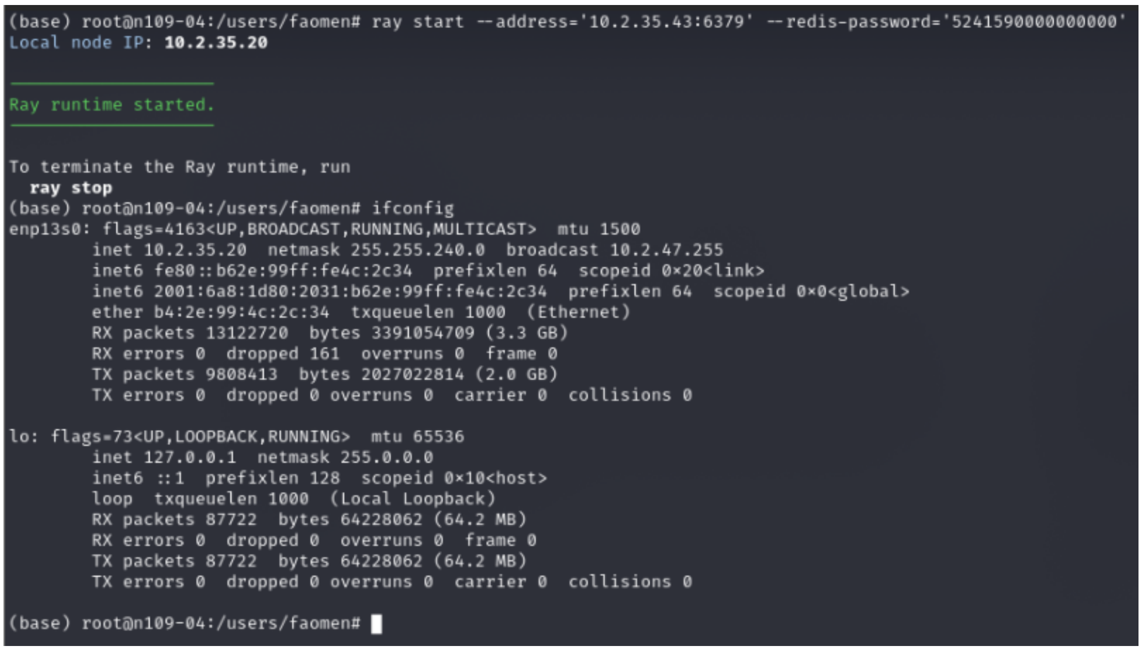
The next step will be to create a Ray cluster in our topology, that is, within what the illustration in Fig. 3.3 we will define that node 0 will be the head and all other nodes will be works. Moving on to the practical part, this is possible using Ray start commands.



```
(base) root@n1012-07:/users/faomen# ray start --head --port=6379
Local node IP: 10.2.35.43
2020-12-30 11:19:08,803 INFO services.py:1169 -- View the Ray dashboard at http://localhost:8265

--------------------
Ray runtime started.
--------------------

Next steps
  To connect to this Ray runtime from another node, run
    ray start --address='10.2.35.43:6379' --redis-password='5241590000000000'

  Alternatively, use the following Python code:
    import ray
    ray.init(address='auto', _redis_password='5241590000000000')

  If connection fails, check your firewall settings and network configuration.

  To terminate the Ray runtime, run
    ray stop
(base) root@n1012-07:/users/faomen# ifconfig
enp13s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.2.35.43  netmask 255.255.240.0  broadcast 10.2.47.255
        inet6 2001:6a8:1d80:2031:b62e:99ff:fe4e:5691  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::b62e:99ff:fe4e:5691  prefixlen 64  scopeid 0x20<link>
        ether b4:2e:99:4e:56:91  txqueuelen 1000  (Ethernet)
        RX packets 15055695  bytes 3823634369 (3.8 GB)
        RX errors 0  dropped 161  overruns 0  frame 0
        TX packets 8108215  bytes 1649023880 (1.6 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 25907491  bytes 5469775121 (5.4 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 25907491  bytes 5469775121 (5.4 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

(base) root@n1012-07:/users/faomen# 
```

Figure 3.12: Ray start the cluster on Head Node

From now on, the cluster is created, as we can see on Fig. 3.12, so we just need to add do this cluster the number of Workers as we want.



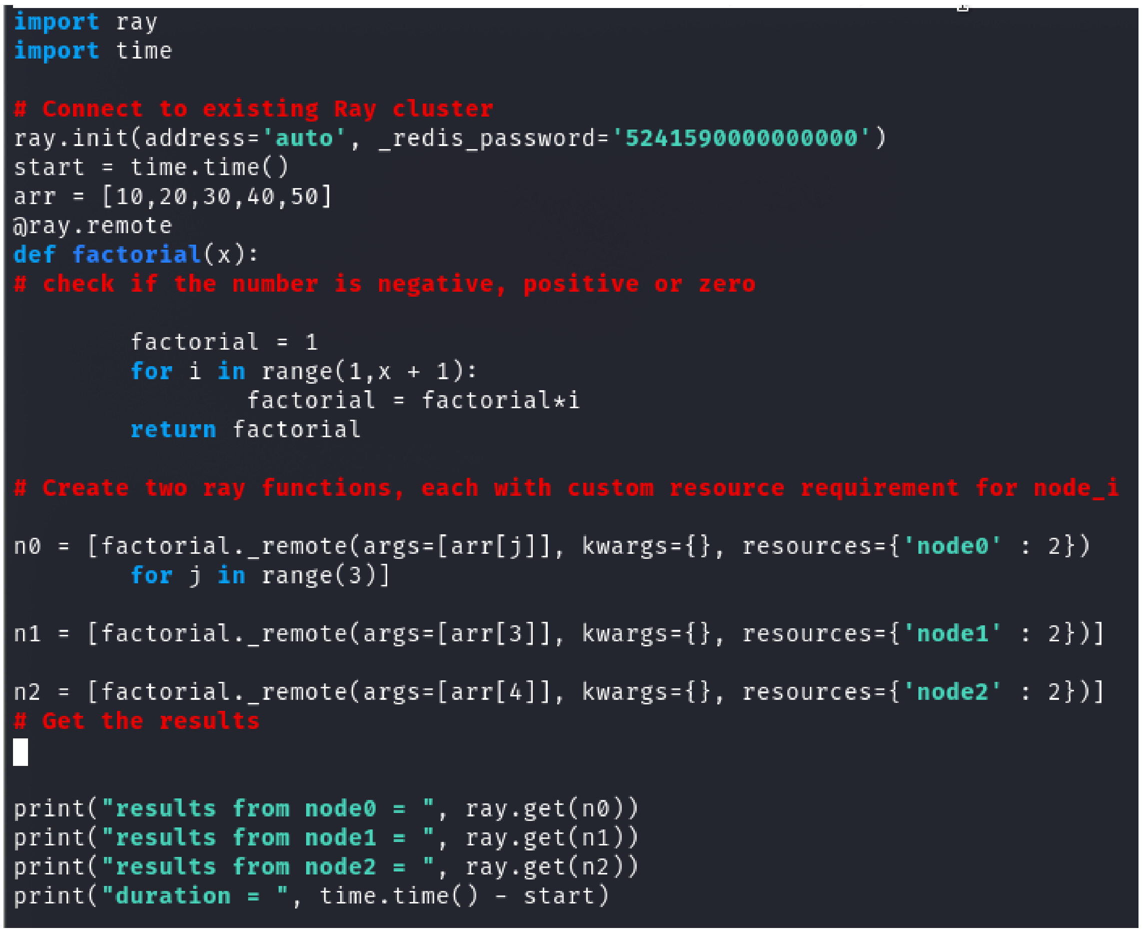
```
(base) root@n109-04:/users/faomen# ray start --address='10.2.35.43:6379' --redis-password='5241590000000000'
Local node IP: 10.2.35.20

--------------------
Ray runtime started.
--------------------

To terminate the Ray runtime, run
  ray stop
(base) root@n109-04:/users/faomen# ifconfig
enp13s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.2.35.20  netmask 255.255.240.0  broadcast 10.2.47.255
        inet6 fe80::b62e:99ff:fe4c:2c34  prefixlen 64  scopeid 0x20<link>
        inet6 2001:6a8:1d80:2031:b62e:99ff:fe4c:2c34  prefixlen 64  scopeid 0x0<global>
        ether b4:2e:99:4c:2c:34  txqueuelen 1000  (Ethernet)
        RX packets 13122720  bytes 3391054709 (3.3 GB)
        RX errors 0  dropped 161  overruns 0  frame 0
        TX packets 9808413  bytes 2027022814 (2.0 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 87722  bytes 64228062 (64.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 87722  bytes 64228062 (64.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

(base) root@n109-04:/users/faomen# █
```

Figure 3.13: Ray join on the cluster with IP:10.2.35.43

This step is repeated as times as workers that we want/use.

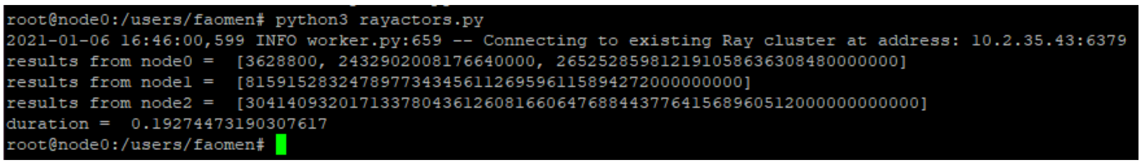
3.4 Distributed / Non-Parallelized and Parallelized Basic Tasks

At this moment, we create, a Python Non-Parallelized Script to test the functionality of the cluster Fig.3.14. We implemented an array, created 3 functions with custom resources for each node. On the variable(n0) we gave some inputs like the position of the array (in this case 3 numbers for that node execute the factorial of them 3), and the label for each node(2 in this case). In this script we were able to decide which numbers we execute on each node. Is important that if you are a new user of this tool, read all the documentation on how to improve the efficiency of the code principally you must delay that operation as much as possible, it's function that take some time and we don't want to waste time, we want to gain it. You can check the output in Fig. 3.15.



```
import ray
import time

# Connect to existing Ray cluster
ray.init(address='auto', _redis_password='5241590000000000')
start = time.time()
arr = [10,20,30,40,50]
@ray.remote
def factorial(x):
# check if the number is negative, positive or zero

        factorial = 1
        for i in range(1,x + 1):
                factorial = factorial*i
        return factorial

# Create two ray functions, each with custom resource requirement for node_i

n0 = [factorial._remote(args=[arr[j]], kwargs={}, resources={'node0' : 2})
        for j in range(3)]

n1 = [factorial._remote(args=[arr[3]], kwargs={}, resources={'node1' : 2})]

n2 = [factorial._remote(args=[arr[4]], kwargs={}, resources={'node2' : 2})]
# Get the results


print("results from node0 = ", ray.get(n0))
print("results from node1 = ", ray.get(n1))
print("results from node2 = ", ray.get(n2))
print("duration = ", time.time() - start)
```

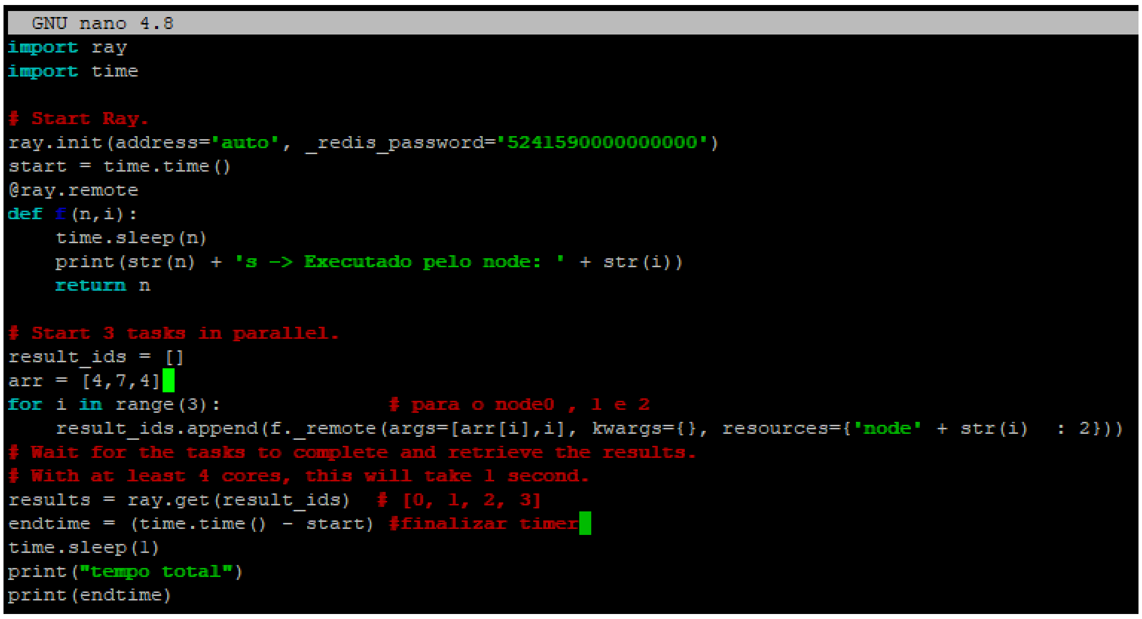
Figure 3.14: Distributed but no Parallelized fatorial task Script



```
root@node0:/users/faomen# python3 rayactors.py
2021-01-06 16:46:00,599 INFO worker.py:659 -- Connecting to existing Ray cluster at address: 10.2.35.43:6379
results from node0 =  [3628800, 2432902008176640000, 265252859812191058636308480000000]
results from node1 =  [815915283247897734345611269596115894272000000000]
results from node2 =  [30414093201713378043612608166064768844377641568960512000000000000]
duration =  0.19274473190307617
root@node0:/users/faomen# 
```

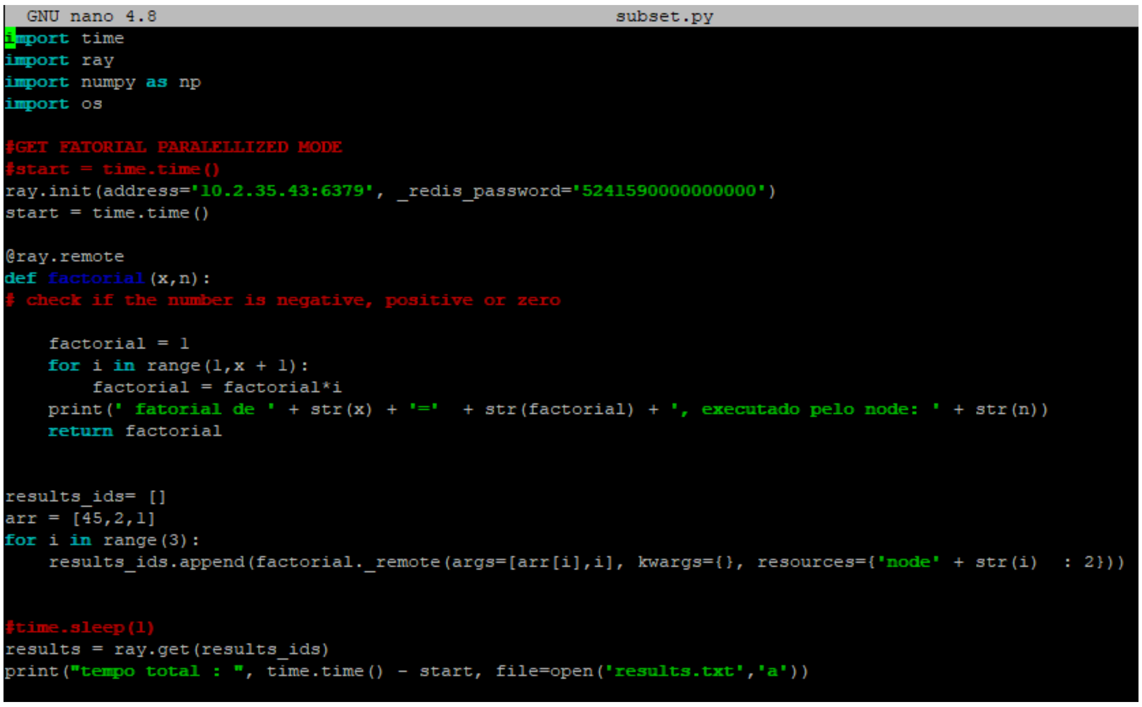
Figure 3.15: Output from each Node on Head Node(0)

The Figs. 3.16 and 3.17 illustrate two scripts that were made in the implementation of this project as a way of understanding how the distribution and parallelization worked with the use of a ray cluster. The results and analysis of the implementation are in the following chapter (4)



```
 GNU nano 4.8
import ray
import time

# Start Ray.
ray.init(address='auto', _redis_password='5241590000000000')
start = time.time()
@ray.remote
def f(n,i):
    time.sleep(n)
    print(str(n) + 's -> Executado pelo node: ' + str(i))
    return n

# Start 3 tasks in parallel.
result_ids = []
arr = [4,7,4]
for i in range(3):                # para o node0 , 1 e 2
    result_ids.append(f._remote(args=[arr[i],i], kwargs={}, resources={'node' + str(i)   : 2}))
# Wait for the tasks to complete and retrieve the results.
# With at least 4 cores, this will take 1 second.
results = ray.get(result_ids)  # [0, 1, 2, 3]
endtime = (time.time() - start) #finalizar timer
time.sleep(1)
print("tempo total")
print(endtime)
```

Figure 3.16: Distributed and Paralellized time sleep tasks Script



```
 GNU nano 4.8                                          subset.py
import time
import ray
import numpy as np
import os

#GET FATORIAL PARALELLIZED MODE
#start = time.time()
ray.init(address='10.2.35.43:6379', _redis_password='5241590000000000')
start = time.time()

@ray.remote
def factorial(x,n):
# check if the number is negative, positive or zero

    factorial = 1
    for i in range(1,x + 1):
        factorial = factorial*i
    print(' fatorial de ' + str(x) + '='  + str(factorial) + ', executado pelo node: ' + str(n))
    return factorial


results_ids= []
arr = [45,2,1]
for i in range(3):
    results_ids.append(factorial._remote(args=[arr[i],i], kwargs={}, resources={'node' + str(i)   : 2}))


#time.sleep(1)
results = ray.get(results_ids)
print("tempo total : ", time.time() - start, file=open('results.txt','a'))
```

Figure 3.17: Parallel fatorial task Script

But to understand it better we will use a basic example which is the case of the task of (time sleep).

Emphasize that in addition to being parallelized/non-parallelized, tasks are distributed among the various nodes as we presented in fig. 3.18 and fig. 3.18.

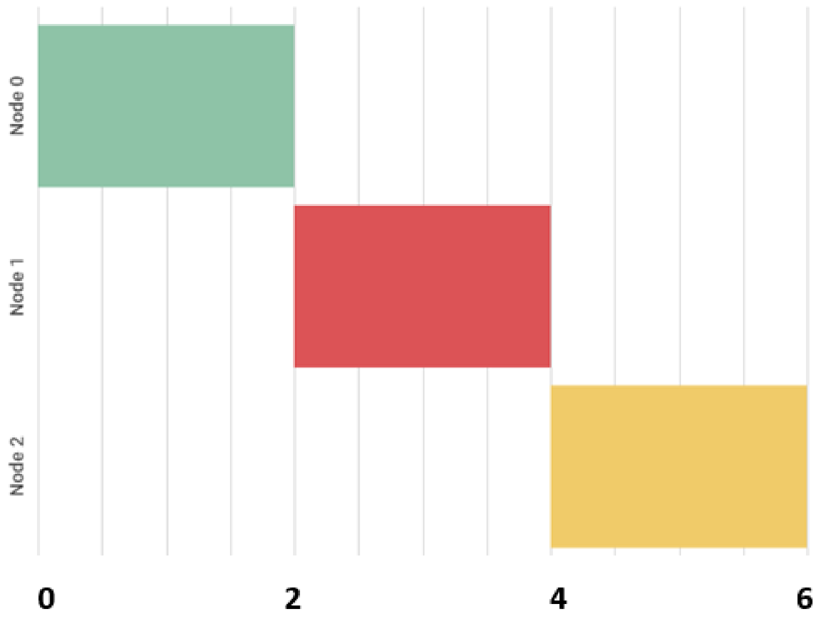


Figure 3.18: Distributed and Non-Paralellized Tasks

There are several gaps in what is the distribution of unparalleled tasks and it is not something we want for our work, because has we can see in the Fig. 3.18 we will need 6 minutes to end this process. It would be important to have tasks distributed by the various nodes in a parallelized way, because it is a matter of time and process execution as showed in fig.3.19.

Only after control these concepts we can adapted the project to machine learning scripts, so that there would be great efficiency in the work above.

Therefore, the graph in fig. 3.19 shows in a simplified way what is the parallelization of tasks (in this case a simple time sleep task).

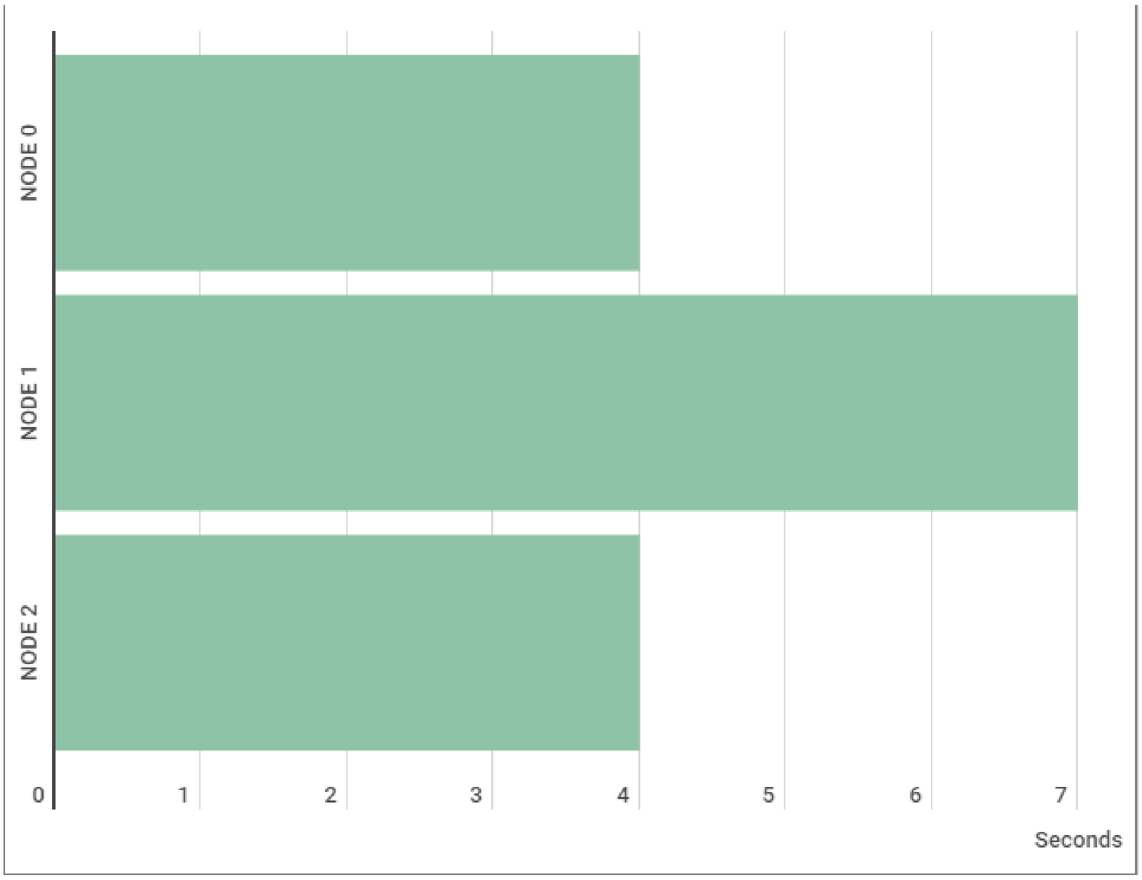


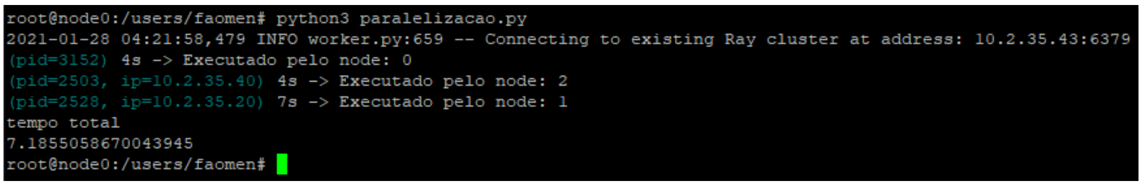
Figure 3.19: Distributed and Paralellized Tasks

In this bar chart, we have 3 nodes each performing a task, time.sleep.

- node 0 : 4 second time sleep
- node 1 : 7 second time sleep
- node 2 : 4 second time sleep

As the tasks are parallelized we will have to execute the 3 tasks in total it will take only 7 seconds, because they will be executed at the same time in the different nodes. Therefore, node 0 and node 2 will finish their task before node 1.

We can see that the total task took 7 seconds through the command line output.



```
root@node0:/users/faomen# python3 paralelizacao.py
2021-01-28 04:21:58,479 INFO worker.py:659 -- Connecting to existing Ray cluster at address: 10.2.35.43:6379
(pid=3152) 4s -> Executado pelo node: 0
(pid=2503, ip=10.2.35.40) 4s -> Executado pelo node: 2
(pid=2528, ip=10.2.35.20) 7s -> Executado pelo node: 1
tempo total
7.1855058670043945
root@node0:/users/faomen#
```

Figure 3.20: SCRIPT Distributed and Paralellized Tasks

The 7.19 (leased) seconds was the total time, which is nothing less than the total 7 seconds of the task of node 2.

Distributed and Paralellized Tasks Now that we know how task parallelization works, we will implement a more complex task that is, for example, calculating factorials, see Fig. 3.21.

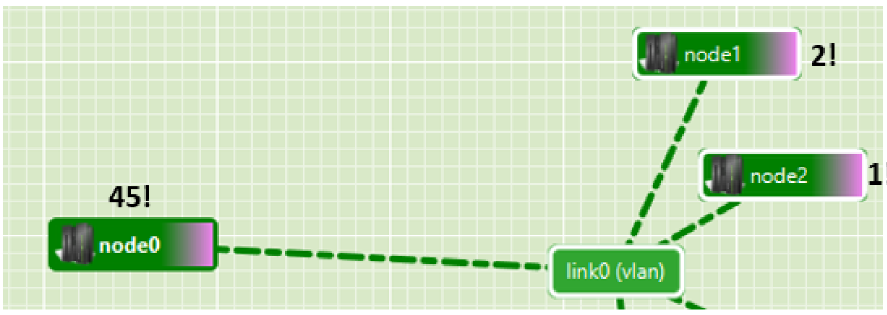
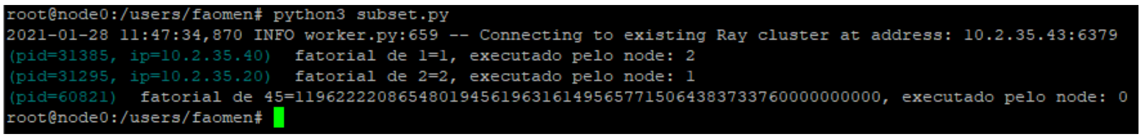


Figure 3.21: Topology Scheme

As shown in the Fig. 3.21, each node will perform a function, being that:

- node 0 : 45! (factorial)
- node 1 : 2!
- node 2 : 1!

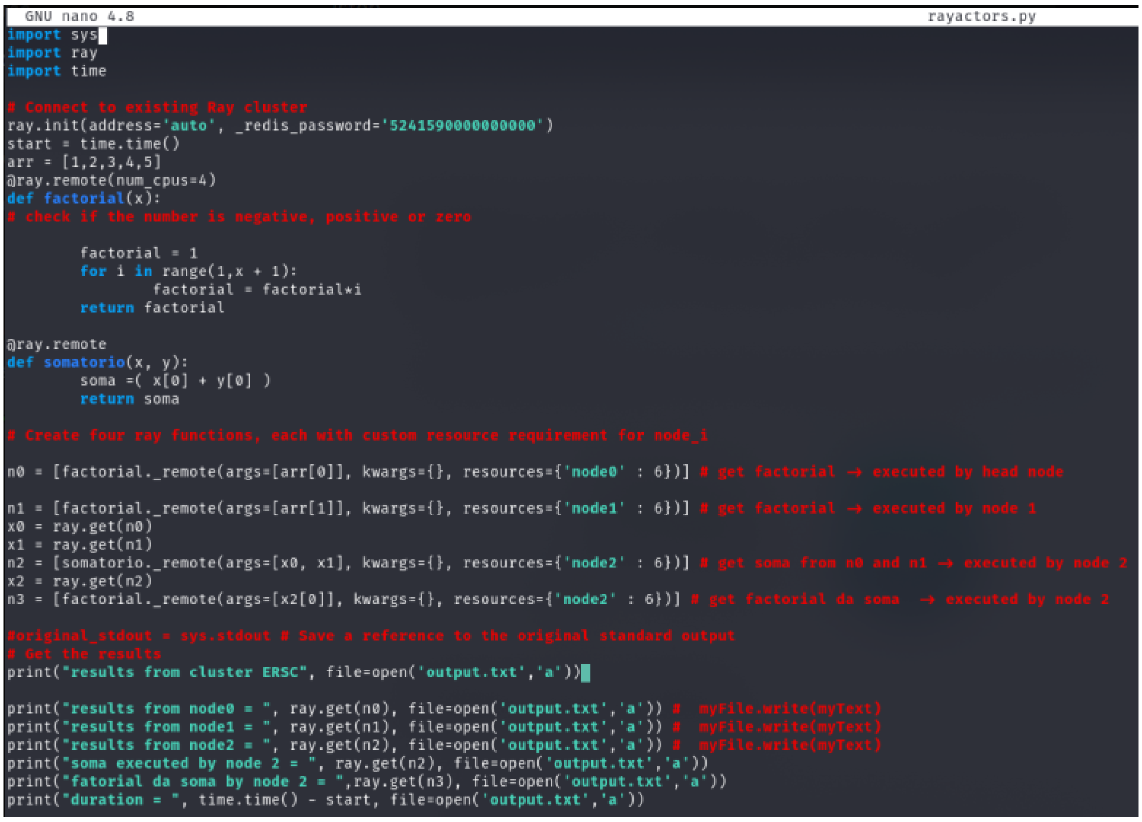
The expectation is that the nodes 1 and 2 finish their task before the node finishes theirs, since the factorial calculation is heavier. We show now some results (Fig. 3.22.



```
root@node0:/users/faomen# python3 subset.py
2021-01-28 11:47:34,870 INFO worker.py:659 -- Connecting to existing Ray cluster at address: 10.2.35.43:6379
(pid=31385, ip=10.2.35.40)  fatorial de 1=1, executado pelo node: 2
(pid=31295, ip=10.2.35.20)  fatorial de 2=2, executado pelo node: 1
(pid=60821)  fatorial de 45=119622220865480194561963161495657715064383733760000000000, executado pelo node: 0
root@node0:/users/faomen#
```

Figure 3.22: Distributed and Paralellized tasks - Results

The results were showing up on the monitor as they finished ... Saving results in .txt file, so later we can check them and troubleshoot if necessary (see Fig. 3.23). After get a script where we are using our workers in a dependent way now we can advance and add more functionalities to our script. So we are now storage our results from this script in a file, as you can see in the Figure below.

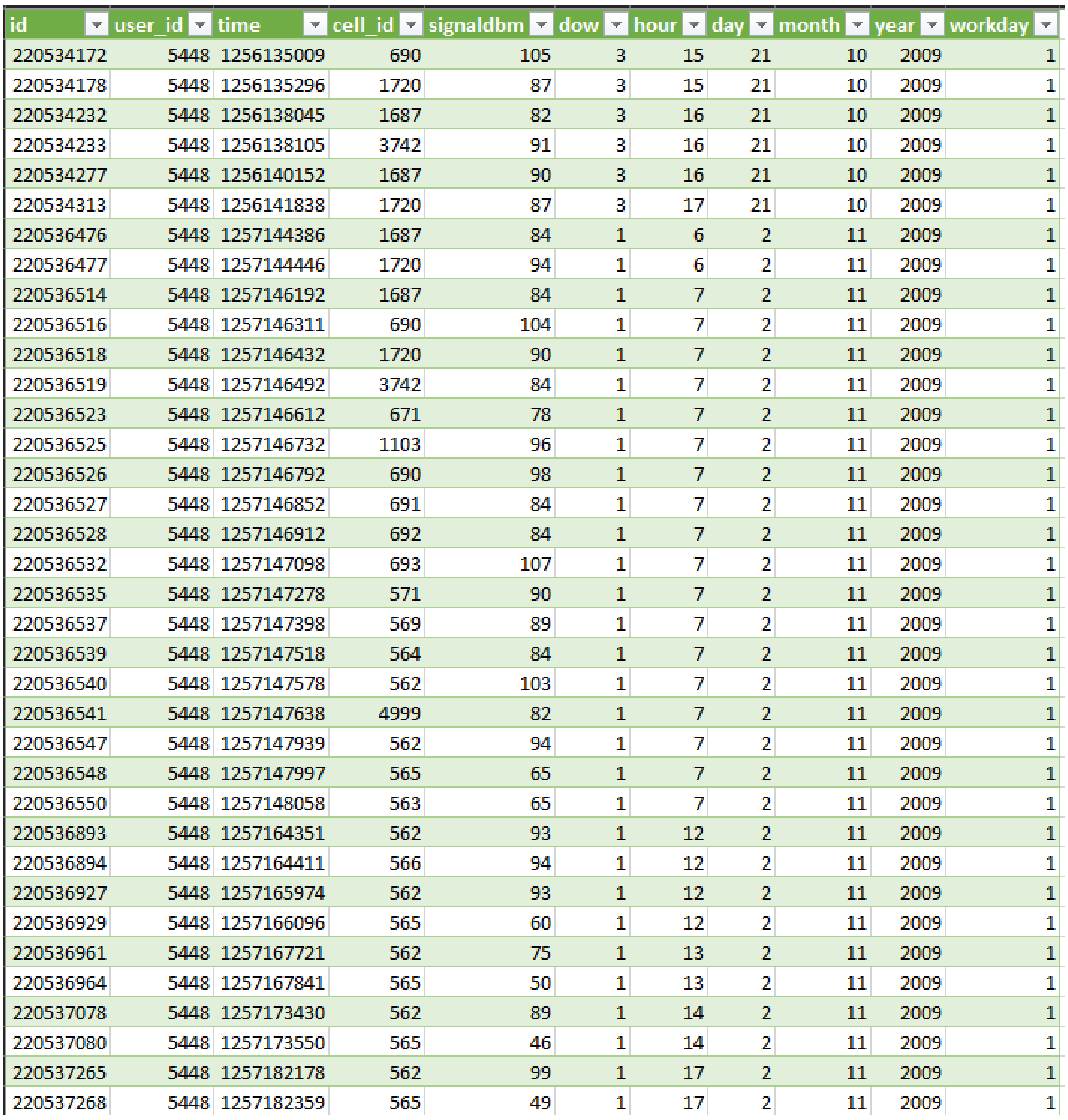


```
  GNU nano 4.8                                                                              rayactors.py
import sys
import ray
import time

# Connect to existing Ray cluster
ray.init(address='auto', _redis_password='5241590000000000')
start = time.time()
arr = [1,2,3,4,5]
@ray.remote(num_cpus=4)
def factorial(x):
# check if the number is negative, positive or zero

        factorial = 1
        for i in range(1,x + 1):
                factorial = factorial*i
        return factorial

@ray.remote
def somatorio(x, y):
        soma =( x[0] + y[0] )
        return soma

# Create four ray functions, each with custom resource requirement for node_i

n0 = [factorial._remote(args=[arr[0]], kwargs={}, resources={'node0' : 6})] # get factorial → executed by head node

n1 = [factorial._remote(args=[arr[1]], kwargs={}, resources={'node1' : 6})] # get factorial → executed by node 1
x0 = ray.get(n0)
x1 = ray.get(n1)
n2 = [somatorio._remote(args=[x0, x1], kwargs={}, resources={'node2' : 6})] # get soma from n0 and n1 → executed by node 2
x2 = ray.get(n2)
n3 = [factorial._remote(args=[x2[0]], kwargs={}, resources={'node2' : 6})] # get factorial da soma  → executed by node 2

#original_stdout = sys.stdout # Save a reference to the original standard output
# Get the results
print("results from cluster ERSC", file=open('output.txt','a'))

print("results from node0 = ", ray.get(n0), file=open('output.txt','a')) #  myFile.write(myText)
print("results from node1 = ", ray.get(n1), file=open('output.txt','a')) #  myFile.write(myText)
print("results from node2 = ", ray.get(n2), file=open('output.txt','a')) #  myFile.write(myText)
print("soma executed by node 2 = ", ray.get(n2), file=open('output.txt','a'))
print("fatorial da soma by node 2 = ",ray.get(n3), file=open('output.txt','a'))
print("duration = ", time.time() - start, file=open('output.txt','a'))
```

Figure 3.23: Saving Results in a file

3.5 Machine Learning Script

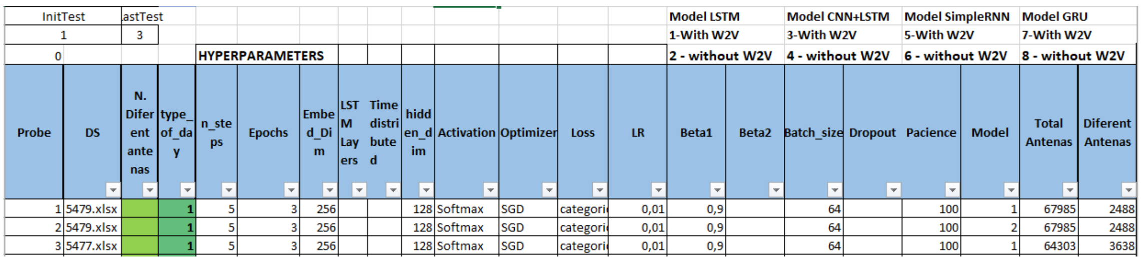
After we get into the Machine Learning setup we must first, contextualize our Script as ours Datasets. We used three different datasets where each one represent a different mobile user. Each dataset represent the mobility of a user over a cellular network, each record is an user association to an antenna (base station) and 11 features are available as seen in Fig.3.24.



id	user_id	time	cell_id	signaldbm	dow	hour	day	month	year	workday
220534172	5448	1256135009	690	105	3	15	21	10	2009	1
220534178	5448	1256135296	1720	87	3	15	21	10	2009	1
220534232	5448	1256138045	1687	82	3	16	21	10	2009	1
220534233	5448	1256138105	3742	91	3	16	21	10	2009	1
220534277	5448	1256140152	1687	90	3	16	21	10	2009	1
220534313	5448	1256141838	1720	87	3	17	21	10	2009	1
220536476	5448	1257144386	1687	84	1	6	2	11	2009	1
220536477	5448	1257144446	1720	94	1	6	2	11	2009	1
220536514	5448	1257146192	1687	84	1	7	2	11	2009	1
220536516	5448	1257146311	690	104	1	7	2	11	2009	1
220536518	5448	1257146432	1720	90	1	7	2	11	2009	1
220536519	5448	1257146492	3742	84	1	7	2	11	2009	1
220536523	5448	1257146612	671	78	1	7	2	11	2009	1
220536525	5448	1257146732	1103	96	1	7	2	11	2009	1
220536526	5448	1257146792	690	98	1	7	2	11	2009	1
220536527	5448	1257146852	691	84	1	7	2	11	2009	1
220536528	5448	1257146912	692	84	1	7	2	11	2009	1
220536532	5448	1257147098	693	107	1	7	2	11	2009	1
220536535	5448	1257147278	571	90	1	7	2	11	2009	1
220536537	5448	1257147398	569	89	1	7	2	11	2009	1
220536539	5448	1257147518	564	84	1	7	2	11	2009	1
220536540	5448	1257147578	562	103	1	7	2	11	2009	1
220536541	5448	1257147638	4999	82	1	7	2	11	2009	1
220536547	5448	1257147939	562	94	1	7	2	11	2009	1
220536548	5448	1257147997	565	65	1	7	2	11	2009	1
220536550	5448	1257148058	563	65	1	7	2	11	2009	1
220536893	5448	1257164351	562	93	1	12	2	11	2009	1
220536894	5448	1257164411	566	94	1	12	2	11	2009	1
220536927	5448	1257165974	562	93	1	12	2	11	2009	1
220536929	5448	1257166096	565	60	1	12	2	11	2009	1
220536961	5448	1257167721	562	75	1	13	2	11	2009	1
220536964	5448	1257167841	565	50	1	13	2	11	2009	1
220537078	5448	1257173430	562	89	1	14	2	11	2009	1
220537080	5448	1257173550	565	46	1	14	2	11	2009	1
220537265	5448	1257182178	562	99	1	17	2	11	2009	1
220537268	5448	1257182359	565	49	1	17	2	11	2009	1

Figure 3.24: Example Dataset used

This script, based on past record of the mobile user, predicts the next antenna where the user will connect. The process of building and deploying a ML model is divided in a set of tasks, where the training one is the most time-consuming. The paralelization and distribution of a ML script can speed up the finding of the best set of hyperparameter using a grid search as seen on Fig.3.25. As this process is iterative and repetitive, several tests can be launched at same time in order to find the best set of hyperparameters that prove to be good solving this or other specific problem.



InitTest		astTest												Model LSTM		Model CNN+LSTM		Model SimpleRNN		Model GRU	
1		3												1-With W2V		3-With W2V		5-With W2V		7-With W2V	
0				HYPERPARAMETERS										2 - without W2V		4 - without W2V		6 - without W2V		8 - without W2V	
Probe	DS	N. Diferent antenas	type_of_day	n_steps	Epochs	Embed_Dim	LSTM Layers	Time distributed	hidden_dim	Activation	Optimizer	Loss	LR	Beta1	Beta2	Batch_size	Dropout	Pacience	Model	Total Antenas	Diferent Antenas
1	5479.xlsx		1	5	3	256			128	Softmax	SGD	categori	0,01	0,9		64		100	1	67985	2488
2	5479.xlsx		1	5	3	256			128	Softmax	SGD	categori	0,01	0,9		64		100	2	67985	2488
3	5477.xlsx		1	5	3	256			128	Softmax	SGD	categori	0,01	0,9		64		100	1	64303	3638

Figure 3.25: Hyper-parameters

Chapter 4

Results and Analysis

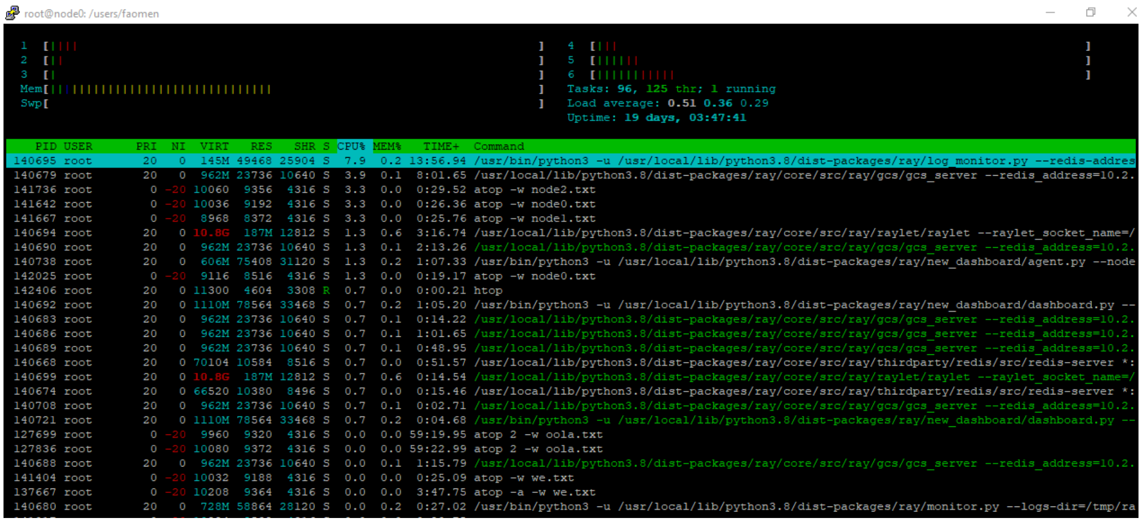
This chapter presents the results obtained from all the scripts and implementations that we carry out, as well as the analyzes carried out on all the following tests:

- Paralellized / Non-Paralellized Tasks
- Distributed and Paralellized Tasks
- Analyzing/Monitoring Performance
- Machine Learning Tasks

4.1 Analyzing/Monitoring Performance

For our Project, we will need to see the results and compare them, so we need to have a tool that can help us with this, Linux have many monitoring software which does this out of the box.

Command htop, this command was already used by use the results are presented in Fig. 4.1.



```
root@node0: /users/faomen
  1  [||||                                                   ]   4  [|||                                                   ]
  2  [|||                                                    ]   5  [|||||||                                               ]
  3  [||                                                     ]   6  [|||||||||||||                                         ]
  Mem[||||||||||||||||||||||||||||||                         ]   Tasks: 96, 125 thr; 1 running
  Swp[                                                       ]   Load average: 0.51 0.36 0.29
                                                                 Uptime: 19 days, 03:47:41

    PID USER       PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 140695 root        20   0  145M 49468 25904 S  7.9  0.2 13:56.94 /usr/bin/python3 -u /usr/local/lib/python3.8/dist-packages/ray/log_monitor.py --redis-addres
 140679 root        20   0  962M 23736 10640 S  3.9  0.1  8:01.65 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/gcs/gcs_server --redis_address=10.2.
 141736 root         0 -20 10060  9356  4316 S  3.3  0.0  0:29.52 atop -w node2.txt
 141642 root         0 -20 10036  9192  4316 S  3.3  0.0  0:26.36 atop -w node0.txt
 141667 root         0 -20  8968  8372  4316 S  3.3  0.0  0:25.76 atop -w node1.txt
 140694 root        20   0 10.8G  187M 12812 S  1.3  0.6  3:16.74 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/raylet/raylet --raylet_socket_name=/
 140690 root        20   0  962M 23736 10640 S  1.3  0.1  2:13.26 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/gcs/gcs_server --redis_address=10.2.
 140738 root        20   0  606M 75408 31120 S  1.3  0.2  1:07.33 /usr/bin/python3 -u /usr/local/lib/python3.8/dist-packages/ray/new_dashboard/agent.py --node
 142025 root         0 -20  9116  8516  4316 S  1.3  0.0  0:19.17 atop -w node0.txt
 142406 root        20   0 11300  4604  3308 R  0.7  0.0  0:00.21 htop
 140692 root        20   0 1110M 78564 33468 S  0.7  0.2  1:05.20 /usr/bin/python3 -u /usr/local/lib/python3.8/dist-packages/ray/new_dashboard/dashboard.py --
 140683 root        20   0  962M 23736 10640 S  0.7  0.1  0:14.22 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/gcs/gcs_server --redis_address=10.2.
 140686 root        20   0  962M 23736 10640 S  0.7  0.1  1:01.65 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/gcs/gcs_server --redis_address=10.2.
 140689 root        20   0  962M 23736 10640 S  0.7  0.1  0:48.95 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/gcs/gcs_server --redis_address=10.2.
 140668 root        20   0 70104 10584  8516 S  0.7  0.0  0:51.57 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/thirdparty/redis/src/redis-server *:
 140699 root        20   0 10.8G  187M 12812 S  0.7  0.6  0:14.54 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/raylet/raylet --raylet_socket_name=/
 140674 root        20   0 66520 10380  8496 S  0.7  0.0  0:15.46 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/thirdparty/redis/src/redis-server *:
 140708 root        20   0  962M 23736 10640 S  0.7  0.1  0:02.71 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/gcs/gcs_server --redis_address=10.2.
 140721 root        20   0 1110M 78564 33468 S  0.7  0.2  0:04.68 /usr/bin/python3 -u /usr/local/lib/python3.8/dist-packages/ray/new_dashboard/dashboard.py --
 127699 root         0 -20  9960  9320  4316 S  0.0  0.0 59:19.95 atop 2 -w oola.txt
 127836 root         0 -20 10080  9372  4316 S  0.0  0.0 59:22.99 atop 2 -w oola.txt
 140688 root        20   0  962M 23736 10640 S  0.0  0.1  1:15.79 /usr/local/lib/python3.8/dist-packages/ray/core/src/ray/gcs/gcs_server --redis_address=10.2.
 141404 root         0 -20 10032  9188  4316 S  0.0  0.0  0:25.09 atop -w we.txt
 137667 root         0 -20 10208  9364  4316 S  0.0  0.0  3:47.75 atop -a -w we.txt
 140680 root        20   0  728M 58864 28120 S  0.0  0.2  0:27.02 /usr/bin/python3 -u /usr/local/lib/python3.8/dist-packages/ray/monitor.py --logs-dir=/tmp/ra
```

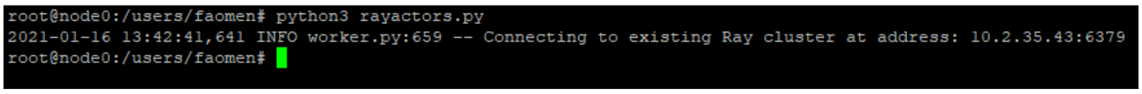
Figure 4.1: Saving Results in a file

As presented, it got the command column, which is handy to identify the process path. And also it is colorful.

A similar framework as "htop" is atop, but with a brilliant feature to record the output in a file so you can view them later. Imagine there is a pattern of having an issue at a specific time window. You can schedule to write the output in a file through crontab or other, and later you can playback.

Once atop is launched, by default it shows system activity for CPU, memory, swap, disks, and network in 10-second intervals. In addition, for each process and thread, you can analyze CPU utilization, memory consumption, disk I/O, priority, username, state, and even exit codes:

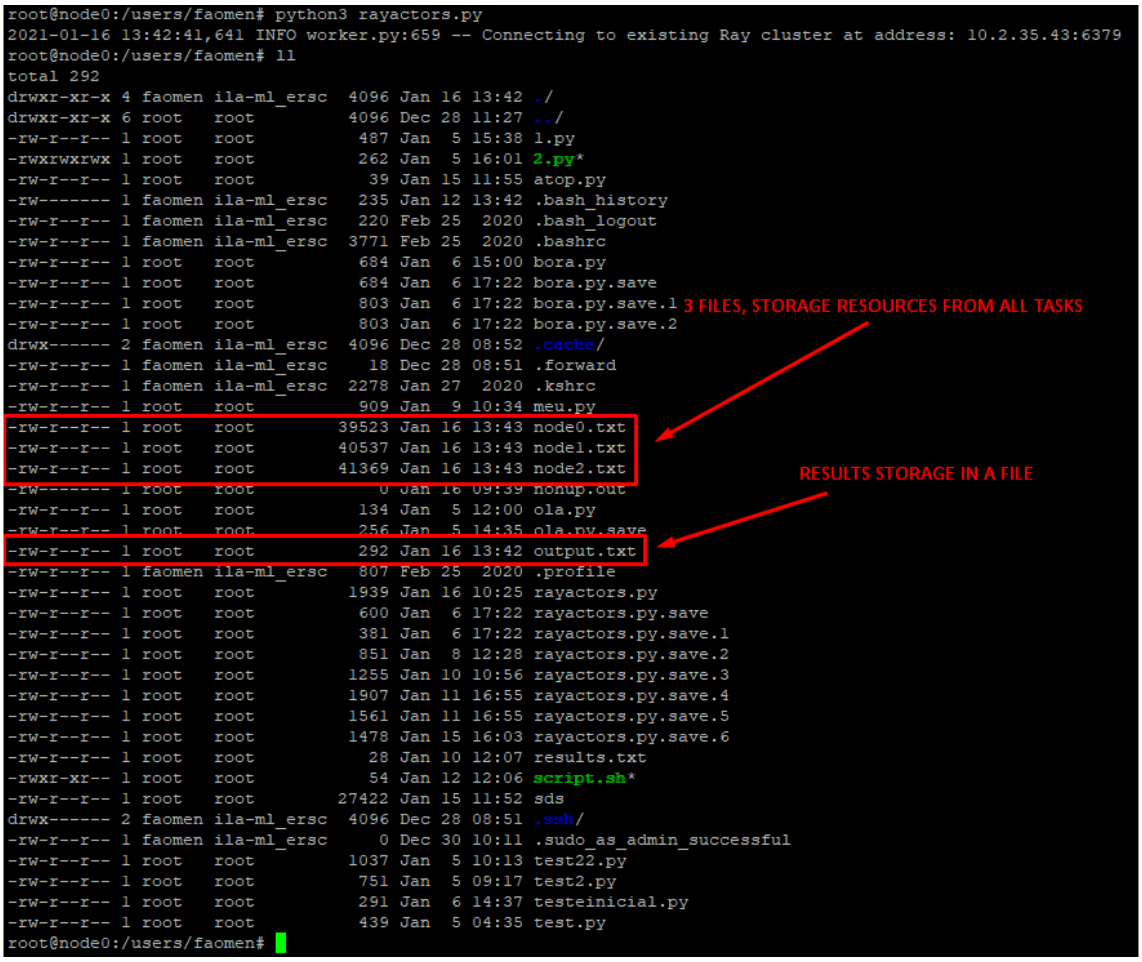
So, we can use the command to monitor our tasks, and the command is *atop -w filename* and then if we want to playback we use *atop -r filename*. So, we add this on our script "rayactors.py". First of all we can launch the cluster script, as presented in Fig. 4.2.



```
root@node0:/users/faomen# python3 rayactors.py
2021-01-16 13:42:41,641 INFO worker.py:659 -- Connecting to existing Ray cluster at address: 10.2.35.43:6379
root@node0:/users/faomen#
```

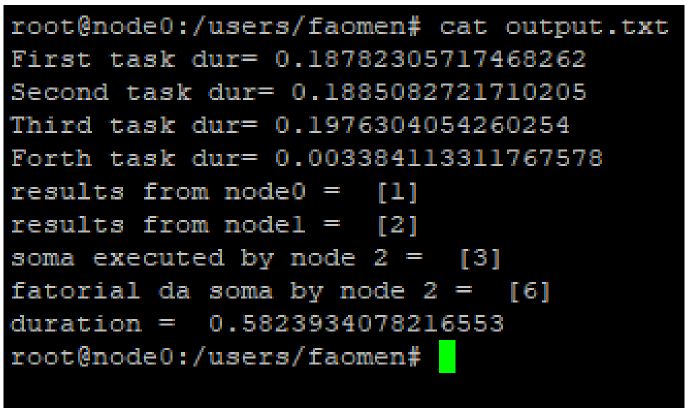
Figure 4.2: Execute Script

Now we can check if everything did fine. Checking if we have our 3 files with resources information, and if we have our ”output” file with the results in fig.4.3 and 4.4.


3 FILES, STORAGE RESOURCES FROM ALL TASKS
RESULTS STORAGE IN A FILE

```
root@node0:/users/faomen# python3 rayactors.py
2021-01-16 13:42:41,641 INFO worker.py:659 -- Connecting to existing Ray cluster at address: 10.2.35.43:6379
root@node0:/users/faomen# ll
total 292
drwxr-xr-x 4 faomen ila-ml_ersc  4096 Jan 16 13:42 ./
drwxr-xr-x 6 root   root         4096 Dec 28 11:27 ../
-rw-r--r-- 1 root   root          487 Jan  5 15:38 1.py
-rwxrwxrwx 1 root   root          262 Jan  5 16:01 2.py*
-rw-r--r-- 1 root   root           39 Jan 15 11:55 atop.py
-rw------- 1 faomen ila-ml_ersc   235 Jan 12 13:42 .bash_history
-rw-r--r-- 1 faomen ila-ml_ersc   220 Feb 25  2020 .bash_logout
-rw-r--r-- 1 faomen ila-ml_ersc  3771 Feb 25  2020 .bashrc
-rw-r--r-- 1 root   root          684 Jan  6 15:00 bora.py
-rw-r--r-- 1 root   root          684 Jan  6 17:22 bora.py.save
-rw-r--r-- 1 root   root          803 Jan  6 17:22 bora.py.save.1
-rw-r--r-- 1 root   root          803 Jan  6 17:22 bora.py.save.2
drwx------ 2 faomen ila-ml_ersc  4096 Dec 28 08:52 .cache/
-rw-r--r-- 1 faomen ila-ml_ersc    18 Dec 28 08:51 .forward
-rw-r--r-- 1 faomen ila-ml_ersc  2278 Jan 27  2020 .kshrc
-rw-r--r-- 1 root   root          909 Jan  9 10:34 meu.py
-rw-r--r-- 1 root   root        39523 Jan 16 13:43 node0.txt
-rw-r--r-- 1 root   root        40537 Jan 16 13:43 node1.txt
-rw-r--r-- 1 root   root        41369 Jan 16 13:43 node2.txt
-rw------- 1 root   root            0 Jan 16 09:39 nohup.out
-rw-r--r-- 1 root   root          134 Jan  5 12:00 ola.py
-rw-r--r-- 1 root   root          256 Jan  5 14:35 ola.py.save
-rw-r--r-- 1 root   root          292 Jan 16 13:42 output.txt
-rw-r--r-- 1 faomen ila-ml_ersc   807 Feb 25  2020 .profile
-rw-r--r-- 1 root   root         1939 Jan 16 10:25 rayactors.py
-rw-r--r-- 1 root   root          600 Jan  6 17:22 rayactors.py.save
-rw-r--r-- 1 root   root          381 Jan  6 17:22 rayactors.py.save.1
-rw-r--r-- 1 root   root          851 Jan  8 12:28 rayactors.py.save.2
-rw-r--r-- 1 root   root         1255 Jan 10 10:56 rayactors.py.save.3
-rw-r--r-- 1 root   root         1907 Jan 11 16:55 rayactors.py.save.4
-rw-r--r-- 1 root   root         1561 Jan 11 16:55 rayactors.py.save.5
-rw-r--r-- 1 root   root         1478 Jan 15 16:03 rayactors.py.save.6
-rw-r--r-- 1 root   root           28 Jan 10 12:07 results.txt
-rwxr-xr-- 1 root   root           54 Jan 12 12:06 script.sh*
-rw-r--r-- 1 root   root        27422 Jan 15 11:52 sds
drwx------ 2 faomen ila-ml_ersc  4096 Dec 28 08:51 .ssh/
-rw-r--r-- 1 faomen ila-ml_ersc     0 Dec 30 10:11 .sudo_as_admin_successful
-rw-r--r-- 1 root   root         1037 Jan  5 10:13 test22.py
-rw-r--r-- 1 root   root          751 Jan  5 09:17 test2.py
-rw-r--r-- 1 root   root          291 Jan  6 14:37 testeinicial.py
-rw-r--r-- 1 root   root          439 Jan  5 04:35 test.py
root@node0:/users/faomen#
```

Figure 4.3: Execute Script

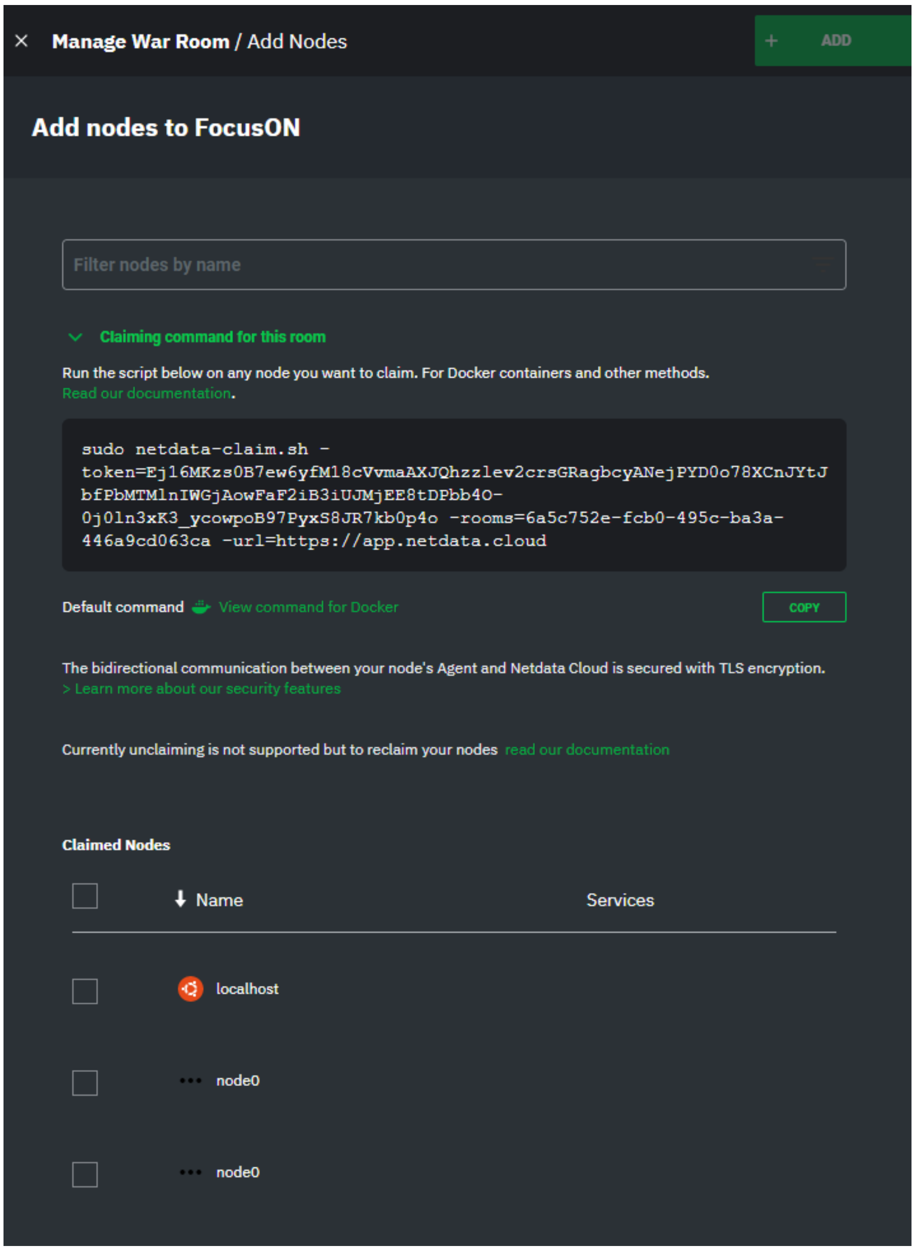


```
root@node0:/users/faomen# cat output.txt
First task dur= 0.18782305717468262
Second task dur= 0.1885082721710205
Third task dur= 0.1976304054260254
Forth task dur= 0.0033841133117675778
results from node0 =  [1]
results from node1 =  [2]
soma executed by node 2 =  [3]
fatorial da soma by node 2 =  [6]
duration =  0.5823934078216553
root@node0:/users/faomen#
```

Figure 4.4: File with the results

Netdata [10] as our Web-Monitor, [11] the authors propose a distributed, real-time performance troubleshooting and health monitoring platform for infrastructures of any size. The entirely free, open-source Agent works in collaboration with Netdata Cloud to provide visibility into the performance of both single nodes and entire infrastructures.

Get up and running within minutes with one-line deployment that requires zero configuration. Immediately access prebuilt charts and alarms with opinionated, intelligent defaults to help you realize value immediately. We can join our nodes to Netdata by simpling type the command, auto-generated by Netdata as you can see on figure:4.5.



Add nodes to FocusON

Claiming command for this room

Run the script below on any node you want to claim. For Docker containers and other methods.
Read our documentation.

```
sudo netdata-claim.sh -
token=Ej16MKzs0B7ew6yfM18cVvmaAXJQhzzlev2crsGRagbcyANejPYD0o78XCnJYtJ
bfPbMTMlnIWGjAowFaF2iB3iUJMjEE8tDPbb4O-
0j0ln3xK3_ycowpoB97PyxS8JR7kb0p4o -rooms=6a5c752e-fcb0-495c-ba3a-
446a9cd063ca -url=https://app.netdata.cloud
```

The bidirectional communication between your node's Agent and Netdata Cloud is secured with TLS encryption.
> Learn more about our security features

Currently unclaiming is not supported but to reclaim your nodes read our documentation

Claimed Nodes

	Name	Services
☐	localhost	
☐	node0	
☐	node0	

Figure 4.5: Script to claim Nodes

At this point we have joined all the nodes to Netdata as we can see on the figure: 4.6.

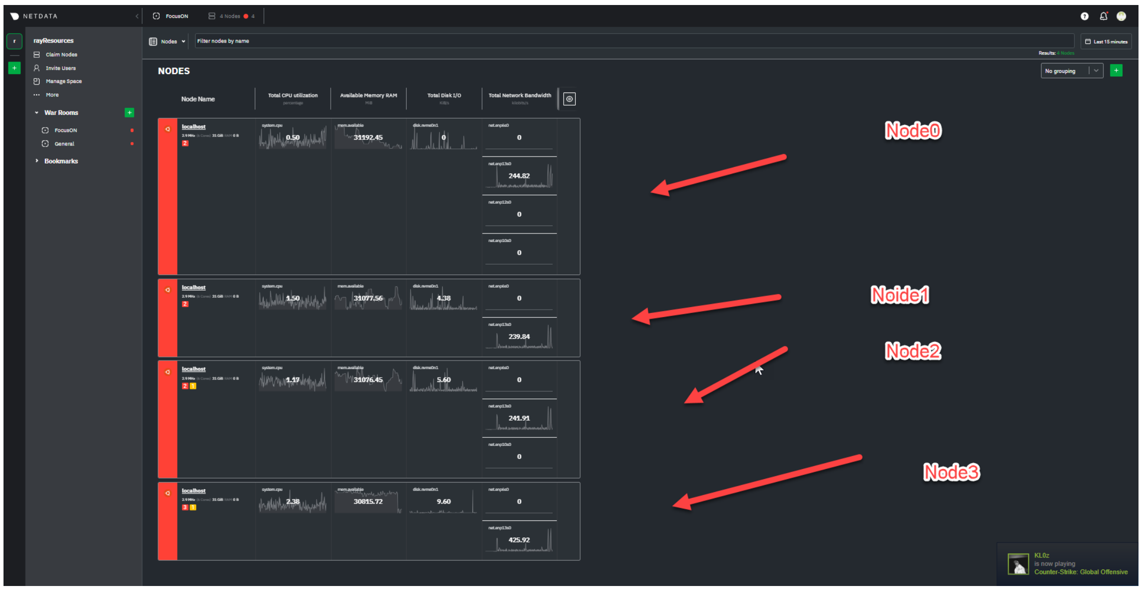
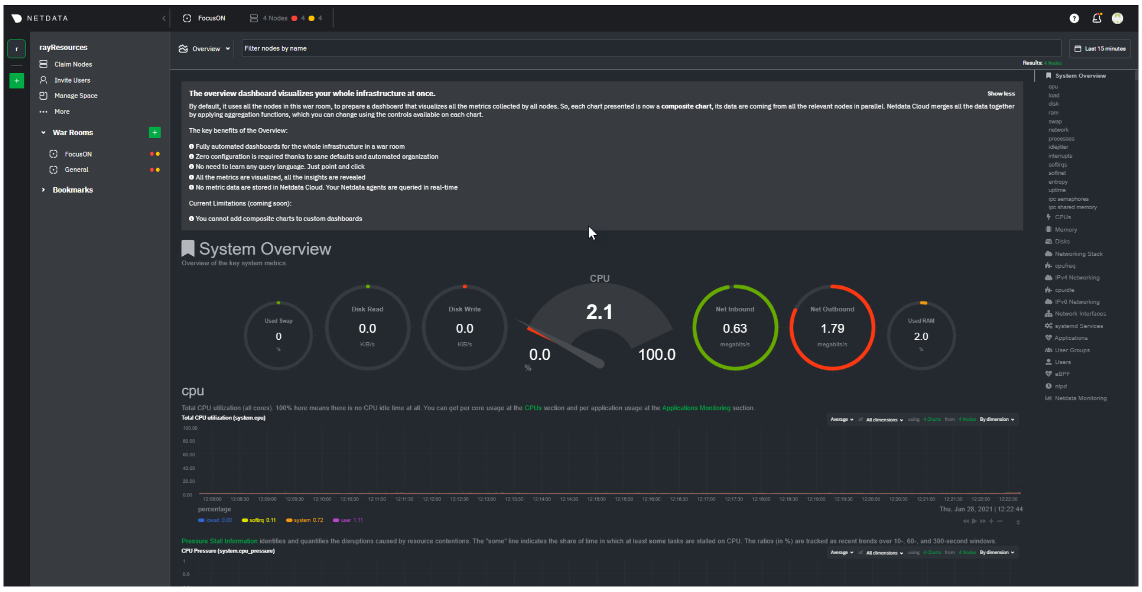


Figure 4.6: All the nodes



The overview dashboard visualizes your whole infrastructure at once.

By default, it uses all the nodes in this war room, to prepare a dashboard that visualizes all the metrics collected by all nodes. So, each chart presented is now a **composite chart**, its data are coming from all the relevant nodes in parallel. Netdata Cloud merges all the data together by applying aggregation functions, which you can change using the controls available on each chart.

The key benefits of the Overview:

- Fully automated dashboards for the whole infrastructure in a war room
- Zero configuration is required thanks to sane defaults and automated organization
- No need to learn any query language. Just point and click
- All the metrics are visualized, all the insights are revealed
- No metric data are stored in Netdata Cloud. Your Netdata agents are queried in real-time

Current Limitations (coming soon):

- You cannot add composite charts to custom dashboards

System Overview

Overview of the key system metrics.

cpu

Total CPU utilization (all cores). 100% here means there is no CPU idle time at all. You can get per core usage at the CPUs section and per application usage at the Applications Monitoring section.

Total CPU utilization (system.cpu)

Pressure Stall Information identifies and quantifies the disruptions caused by resource contentions. The "some" line indicates the share of time in which at least some tasks are stalled on CPU. The ratios (in %) are tracked as recent trends over 10-, 60-, and 300-second windows.

CPU Pressure (system.cpu_pressure)

Figure 4.7: Detailed Node

Plugins Netdata orchestrators may also be described as modular plugins. They are modular since they accept custom made modules to be included. Writing modules for these plugins is easier than accessing the native Netdata API directly. You will find modules already available for each orchestrator under the directory of the particular modular plugin (e.g. under python.d.plugin for the python orchestrator). .

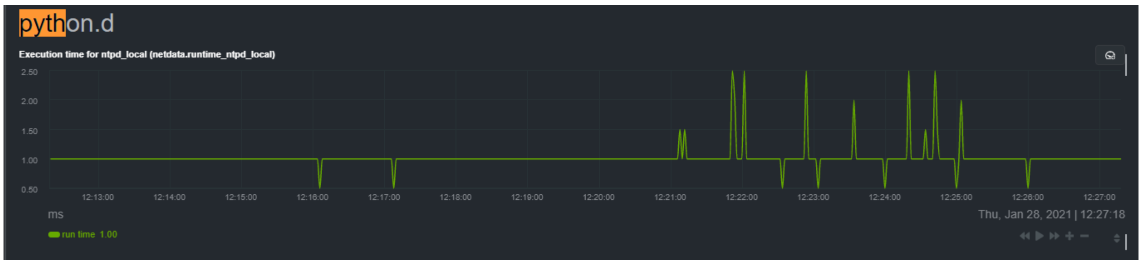
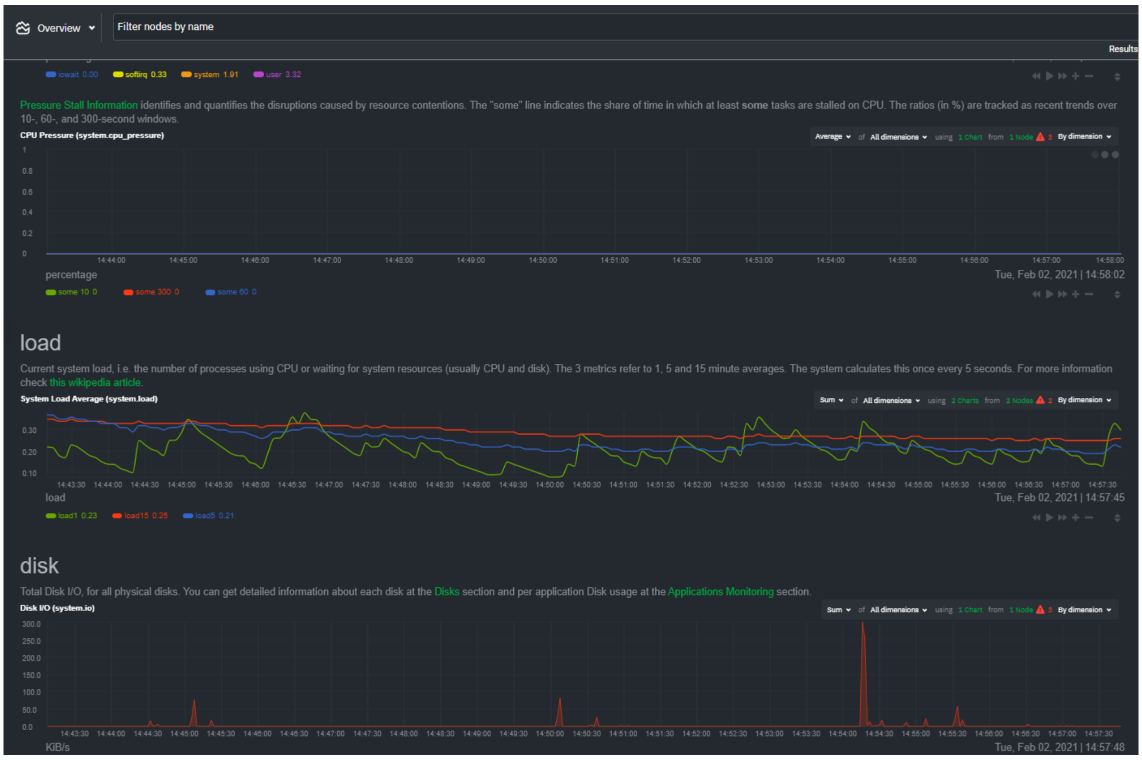


Figure 4.8: CPU usage of Python based Frameworks



Overview
Filter nodes by name
Results
iowait 0.00
softirq 0.33
system 1.91
user 3.32

Pressure Stall Information identifies and quantifies the disruptions caused by resource contentions. The "some" line indicates the share of time in which at least **some** tasks are stalled on CPU. The ratios (in %) are tracked as recent trends over 10-, 60-, and 300-second windows.

CPU Pressure (system.cpu_pressure)
Average of All dimensions using 1 Chart from 1 Node 3 By dimension
percentage
Tue, Feb 02, 2021 | 14:58:02
some 10 0
some 300 0
some 60 0
load

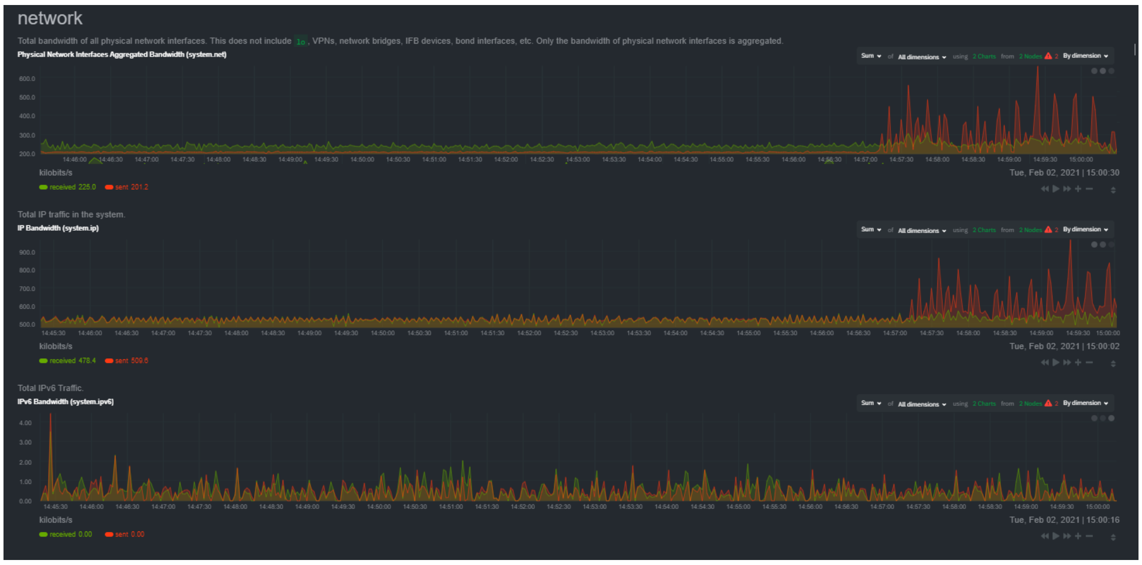
Current system load, i.e. the number of processes using CPU or waiting for system resources (usually CPU and disk). The 3 metrics refer to 1, 5 and 15 minute averages. The system calculates this once every 5 seconds. For more information check this wikipedia article.

System Load Average (system.load)
Sum of All dimensions using 2 Charts from 2 Nodes 2 By dimension
load
Tue, Feb 02, 2021 | 14:57:45
load1 0.23
load15 0.25
load5 0.21
disk

Total Disk I/O, for all physical disks. You can get detailed information about each disk at the Disks section and per application Disk usage at the Applications Monitoring section.

Disk I/O (system.io)
Sum of All dimensions using 1 Chart from 1 Node 3 By dimension
KiB/s
Tue, Feb 02, 2021 | 14:57:48

Figure 4.9: Disk usage Plugin


network

Total bandwidth of all physical network interfaces. This does not include `lo`, VPNs, network bridges, IFB devices, bond interfaces, etc. Only the bandwidth of physical network interfaces is aggregated.

Physical Network Interfaces Aggregated Bandwidth (system.net)

kilobits/s

received 225.0 sent 201.2

Tue, Feb 02, 2021 | 15:00:30

Total IP traffic in the system.

IP Bandwidth (system.ip)

kilobits/s

received 478.4 sent 509.6

Tue, Feb 02, 2021 | 15:00:02

Total IPv6 Traffic.

IPv6 Bandwidth (system.ipv6)

kilobits/s

received 0.00 sent 0.00

Tue, Feb 02, 2021 | 15:00:16

Figure 4.10: Network usage Plugin

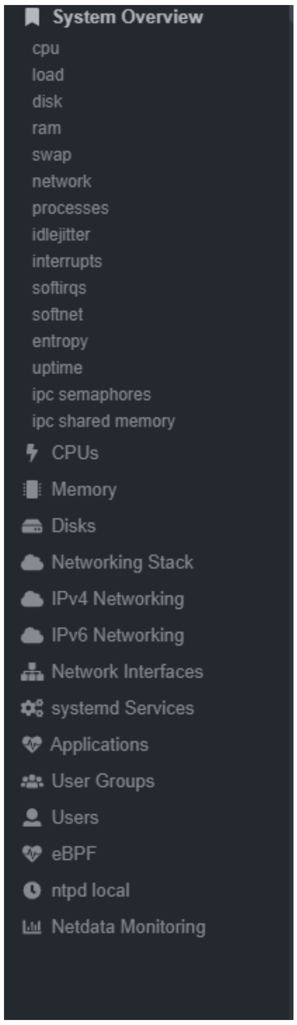


Figure 4.11: Others plugins

Each of these modular plugins has each own methods for defining modules. Please check the examples and their documentation.On all of the plugins the one that we care the most was this *python.d Plugin* see fig.4.9, we mount a plugin to monitorize the disk usage see fig.4.9, a plugin to monitorize the Network usage see fig.4.10, and some other plugins to explore fig,4.11

4.2 Machine Learning Tasks Results and Analysis

Inside this section we divide the following section ML tasks executed by one node from the distributed and parallelized ML tasks, and serialized tasks section, because on the first topic we approach ML tasks only executed by one node. The section 4.2.2, introduces the parallelization paradigm and distributed ML tasks on the architecture and for last on section4.2.3 we Serialize all the tasks with the paradigm previously explained.

4.2.1 Machine Learning Tasks executed by one Node

Starting now for this part of the work, we know that now the objective will be to test and obtain results of ML tasks with and without distribution so we can compare times and have the notion that ML tasks with distribution is more competent and efficient, however, first it is necessary to graphically perceive this whole structure.

There are some Datasets prepared, it´s used three Excel files, with labeled data. The format of the excel should be properly prepared, because if some space is blank or even wrong filled it you can have unexpected outputs.

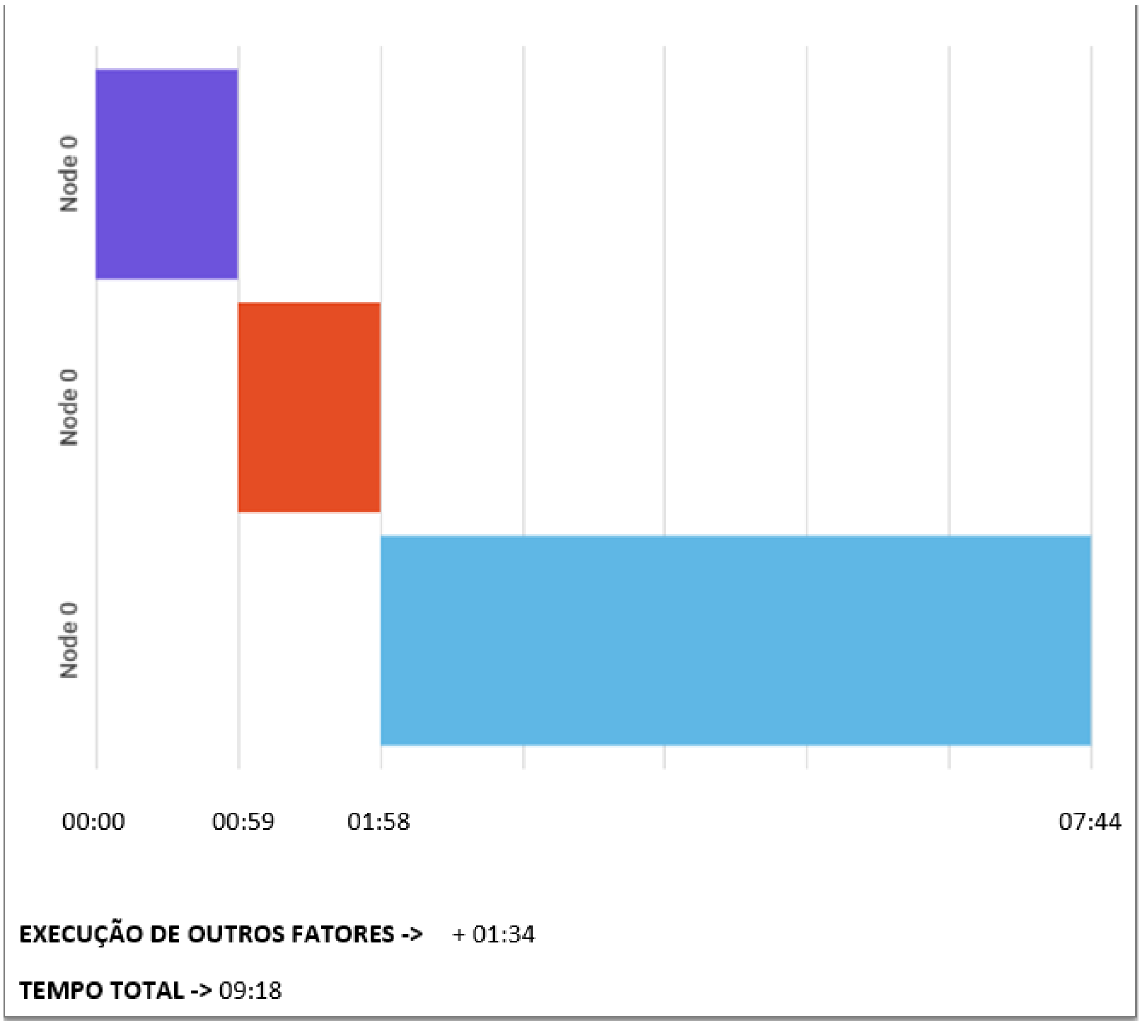


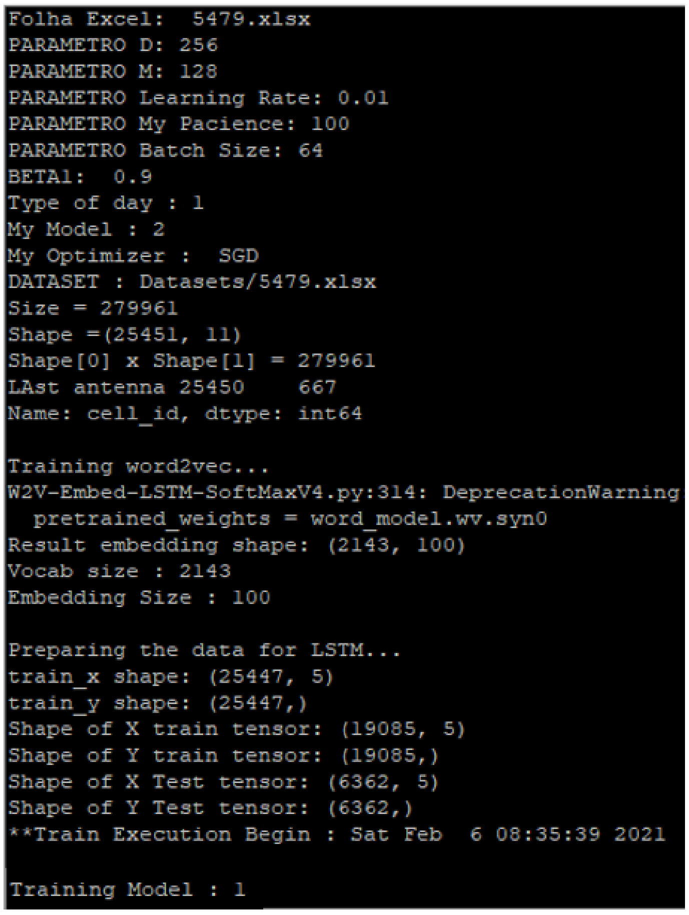
Figure 4.12: ML tasks executed by Node 0

In the figure above we have: Node 0 will execute 3 datasets tasks, one execute after the other ends.

Node 0 performs the first task in 59 seconds, the second also ends in 59 seconds and the third takes 5 minutes and 46 seconds. The total take 7 minutes and 44 seconds to perform.

However, there are also other factors in the execution of the python command, which in this case ends up having an additional value of 1 minute and 34 seconds which gives a total of 9 minutes and 18 seconds to make the total execution.

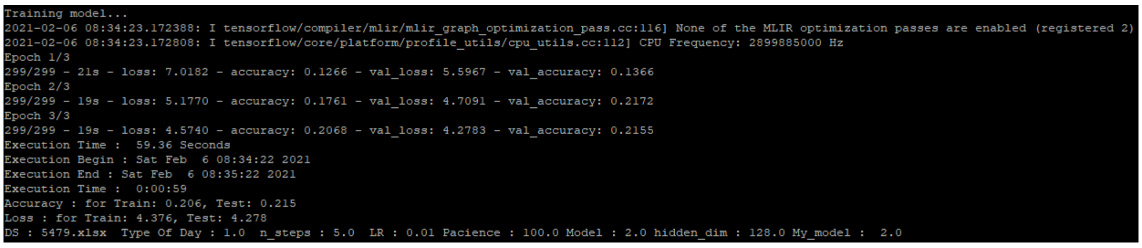
After running the command *python3 manager.py*, the first thing that appears in the output is all the information and hyperparameters about the first dataset to be trained, in this case, the 5479 dataset model 1, see Fig. 4.13.



```
Folha Excel:  5479.xlsx
PARAMETRO D: 256
PARAMETRO M: 128
PARAMETRO Learning Rate: 0.01
PARAMETRO My Pacience: 100
PARAMETRO Batch Size: 64
BETA1:  0.9
Type of day : 1
My Model : 2
My Optimizer :  SGD
DATASET : Datasets/5479.xlsx
Size = 279961
Shape =(25451, 11)
Shape[0] x Shape[1] = 279961
LAst antenna 25450    667
Name: cell_id, dtype: int64

Training word2vec...
W2V-Embed-LSTM-SoftMaxV4.py:314: DeprecationWarning
  pretrained_weights = word_model.wv.syn0
Result embedding shape: (2143, 100)
Vocab size : 2143
Embedding Size : 100

Preparing the data for LSTM...
train_x shape: (25447, 5)
train_y shape: (25447,)
Shape of X train tensor: (19085, 5)
Shape of Y train tensor: (19085,)
Shape of X Test tensor: (6362, 5)
Shape of Y Test tensor: (6362,)
**Train Execution Begin : Sat Feb  6 08:35:39 2021

Training Model : 1
```

Figure 4.13: 5479 dataset model 1

Now that the dataset was prepared, it is time to pick a percentage of samples, a bigger percentage for training the model and the other to test it, as in Fig. 4.14.

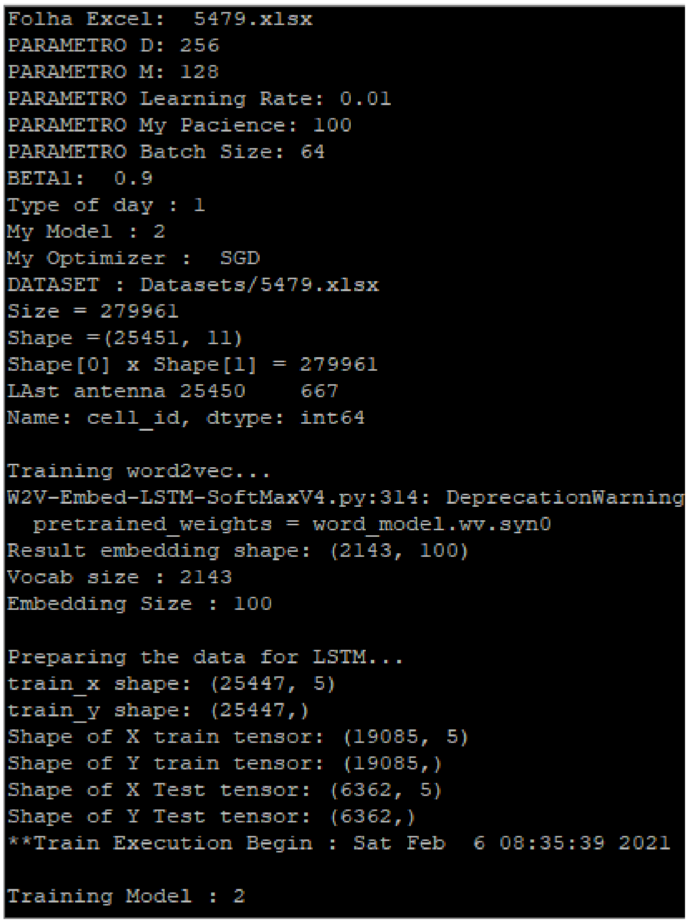


```
Training model...
2021-02-06 08:34:23.172388: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the MLIR optimization passes are enabled (registered 2)
2021-02-06 08:34:23.172808: I tensorflow/core/platform/profile_utils/cpu_utils.cc:112] CPU Frequency: 2899885000 Hz
Epoch 1/3
299/299 - 21s - loss: 7.0182 - accuracy: 0.1266 - val_loss: 5.5967 - val_accuracy: 0.1366
Epoch 2/3
299/299 - 19s - loss: 5.1770 - accuracy: 0.1761 - val_loss: 4.7091 - val_accuracy: 0.2172
Epoch 3/3
299/299 - 19s - loss: 4.5740 - accuracy: 0.2068 - val_loss: 4.2783 - val_accuracy: 0.2155
Execution Time :  59.36 Seconds
Execution Begin : Sat Feb  6 08:34:22 2021
Execution End : Sat Feb  6 08:35:22 2021
Execution Time :  0:00:59
Accuracy : for Train: 0.206, Test: 0.215
Loss : for Train: 4.376, Test: 4.278
DS : 5479.xlsx  Type Of Day : 1.0  n_steps : 5.0  LR : 0.01 Pacience : 100.0 Model : 2.0 hidden_dim : 128.0 My_model :  2.0
```

Figure 4.14: 5479 dataset model 1 Training

In the figure, it is impossible to see that this dataset took 59 seconds to execute 3 epoch training (execution time), with a val accuracy of 0.2155 (hit percentage).

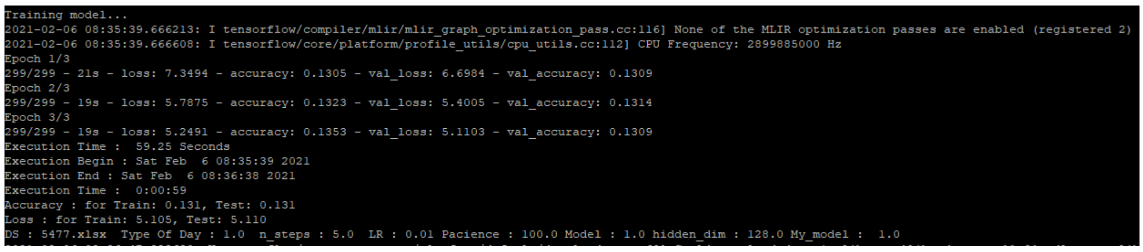
This time, the same process on Fig. 4.13, but now with another dataset and another model, see Fig. 4.15.



```
Folha Excel:  5479.xlsx
PARAMETRO D: 256
PARAMETRO M: 128
PARAMETRO Learning Rate: 0.01
PARAMETRO My Pacience: 100
PARAMETRO Batch Size: 64
BETA1:  0.9
Type of day : 1
My Model : 2
My Optimizer :  SGD
DATASET : Datasets/5479.xlsx
Size = 279961
Shape =(25451, 11)
Shape[0] x Shape[1] = 279961
LAst antenna 25450    667
Name: cell_id, dtype: int64

Training word2vec...
W2V-Embed-LSTM-SoftMaxV4.py:314: DeprecationWarning
  pretrained_weights = word_model.wv.syn0
Result embedding shape: (2143, 100)
Vocab size : 2143
Embedding Size : 100

Preparing the data for LSTM...
train_x shape: (25447, 5)
train_y shape: (25447,)
Shape of X train tensor: (19085, 5)
Shape of Y train tensor: (19085,)
Shape of X Test tensor: (6362, 5)
Shape of Y Test tensor: (6362,)
**Train Execution Begin : Sat Feb  6 08:35:39 2021

Training Model : 2
```

Figure 4.15: 5479 dataset model 2

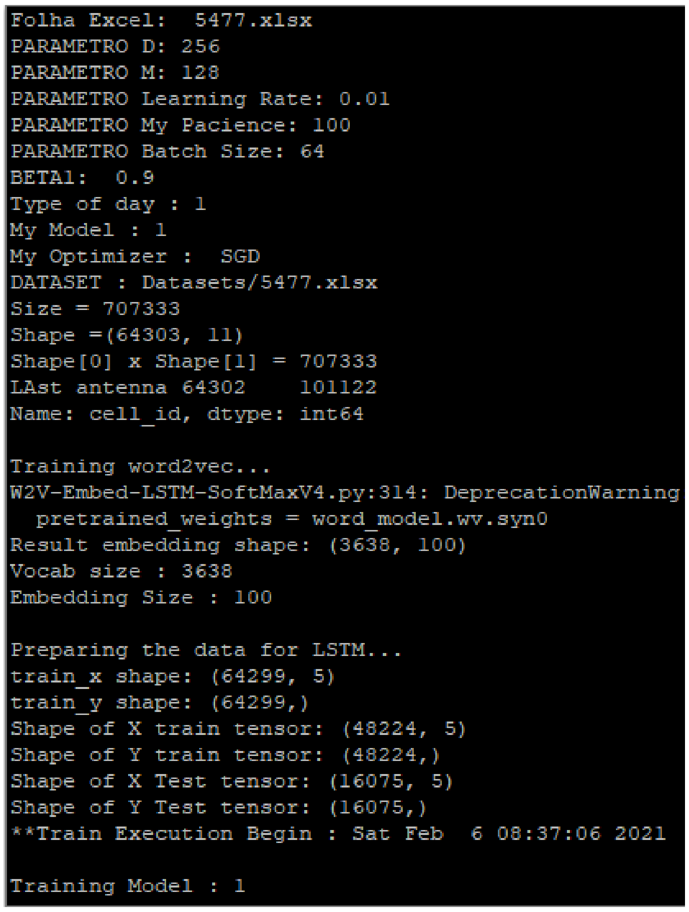
Now that the dataset was prepared, again it is time to pick a percentage of samples, a bigger percentage for training the model and the other to test it, see Fig. 4.16. *Dataset 5479 Model 2: Execution and training.*



```
Training model...
2021-02-06 08:35:39.666213: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the MLIR optimization passes are enabled (registered 2)
2021-02-06 08:35:39.666608: I tensorflow/core/platform/profile_utils/cpu_utils.cc:112] CPU Frequency: 2899885000 Hz
Epoch 1/3
299/299 - 21s - loss: 7.3494 - accuracy: 0.1305 - val_loss: 6.6984 - val_accuracy: 0.1309
Epoch 2/3
299/299 - 19s - loss: 5.7875 - accuracy: 0.1323 - val_loss: 5.4005 - val_accuracy: 0.1314
Epoch 3/3
299/299 - 19s - loss: 5.2491 - accuracy: 0.1353 - val_loss: 5.1103 - val_accuracy: 0.1309
Execution Time :  59.25 Seconds
Execution Begin : Sat Feb  6 08:35:39 2021
Execution End : Sat Feb  6 08:36:38 2021
Execution Time :  0:00:59
Accuracy : for Train: 0.131, Test: 0.131
Loss : for Train: 5.105, Test: 5.110
DS : 5477.xlsx  Type Of Day : 1.0  n_steps : 5.0  LR : 0.01 Pacience : 100.0 Model : 1.0 hidden_dim : 128.0 My_model :  1.0
```

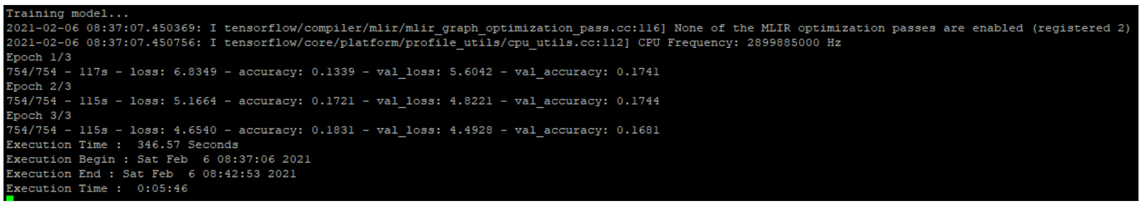
Figure 4.16: 5479 dataset model 2 Training

In the figure 4.16, it is impossible to see that this dataset took 59 seconds to execute 3 epoch training (execution time), with a val accuracy of 0.13 (hit percentage), see Fig. 4.17.



```
Folha Excel:  5477.xlsx
PARAMETRO D: 256
PARAMETRO M: 128
PARAMETRO Learning Rate: 0.01
PARAMETRO My Pacience: 100
PARAMETRO Batch Size: 64
BETA1:  0.9
Type of day : 1
My Model : 1
My Optimizer :  SGD
DATASET : Datasets/5477.xlsx
Size = 707333
Shape =(64303, 11)
Shape[0] x Shape[1] = 707333
LAst antenna 64302    101122
Name: cell_id, dtype: int64

Training word2vec...
W2V-Embed-LSTM-SoftMaxV4.py:314: DeprecationWarning
  pretrained_weights = word_model.wv.syn0
Result embedding shape: (3638, 100)
Vocab size : 3638
Embedding Size : 100

Preparing the data for LSTM...
train_x shape: (64299, 5)
train_y shape: (64299,)
Shape of X train tensor: (48224, 5)
Shape of Y train tensor: (48224,)
Shape of X Test tensor: (16075, 5)
Shape of Y Test tensor: (16075,)
**Train Execution Begin : Sat Feb  6 08:37:06 2021

Training Model : 1
```

Figure 4.17: 5477 dataset model 1

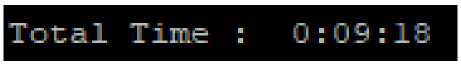


```
Training model...
2021-02-06 08:37:07.450369: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the MLIR optimization passes are enabled (registered 2)
2021-02-06 08:37:07.450756: I tensorflow/core/platform/profile_utils/cpu_utils.cc:112] CPU Frequency: 2899885000 Hz
Epoch 1/3
754/754 - 117s - loss: 6.8349 - accuracy: 0.1339 - val_loss: 5.6042 - val_accuracy: 0.1741
Epoch 2/3
754/754 - 115s - loss: 5.1664 - accuracy: 0.1721 - val_loss: 4.8221 - val_accuracy: 0.1744
Epoch 3/3
754/754 - 115s - loss: 4.6540 - accuracy: 0.1831 - val_loss: 4.4928 - val_accuracy: 0.1681
Execution Time :  346.57 Seconds
Execution Begin : Sat Feb  6 08:37:06 2021
Execution End : Sat Feb  6 08:42:53 2021
Execution Time :  0:05:46
```

Figure 4.18: 5477 dataset model 1 Training

In the figure 4.18, it is impossible to see that this dataset took 5 minutes and 46 seconds to execute 3 epoch training (execution time), with a val accuracy of 0.1681 (hit percentage).

After all tasks have been performed, in that order. We get the final time and the script is finished, see fig.4.19.



```
Total Time :  0:09:18
```

Figure 4.19: Total Time

4.2.2 Distributed and Paralellized Machine Learning tasks

In this phase, we will perform ML tasks in distributed and parallelized mode, e.g the first dataset (5479 dataset model 1) will be executed by node 0, the second (5479 dataset model 2) by node 1 and the third (5477 dataset model 1) by node 3. And they will all be executed at the same time using the ray tool.

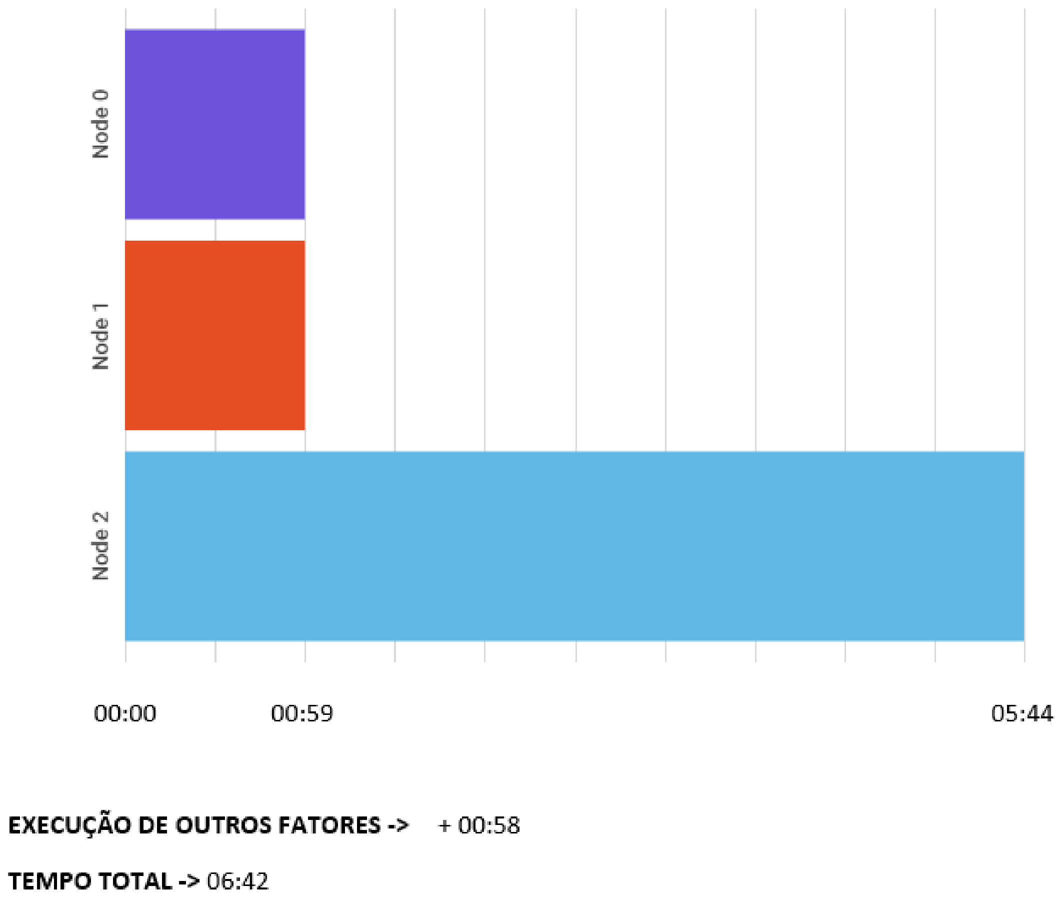
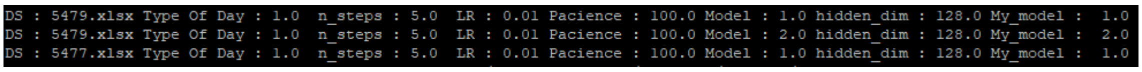


Figure 4.20: Distributed and Paralellized Machine Learning Tasks

The execution time was 5 minutes and 44 seconds, but with some additional factors we add 58 seconds to this time, so in the final we have 6 minutes and 42 seconds.

Less 3 minutes than without parallelization and ray.

In terms of output, as they are all executed at node 0 and at the same time, we will see that the execution of all datasets are sampled at the same time on the screen.



```
DS : 5479.xlsx Type Of Day : 1.0  n_steps : 5.0  LR : 0.01 Pacience : 100.0 Model : 1.0 hidden_dim : 128.0 My_model :  1.0
DS : 5479.xlsx Type Of Day : 1.0  n_steps : 5.0  LR : 0.01 Pacience : 100.0 Model : 2.0 hidden_dim : 128.0 My_model :  2.0
DS : 5477.xlsx Type Of Day : 1.0  n_steps : 5.0  LR : 0.01 Pacience : 100.0 Model : 1.0 hidden_dim : 128.0 My_model :  1.0
```

Figure 4.21: Datasets

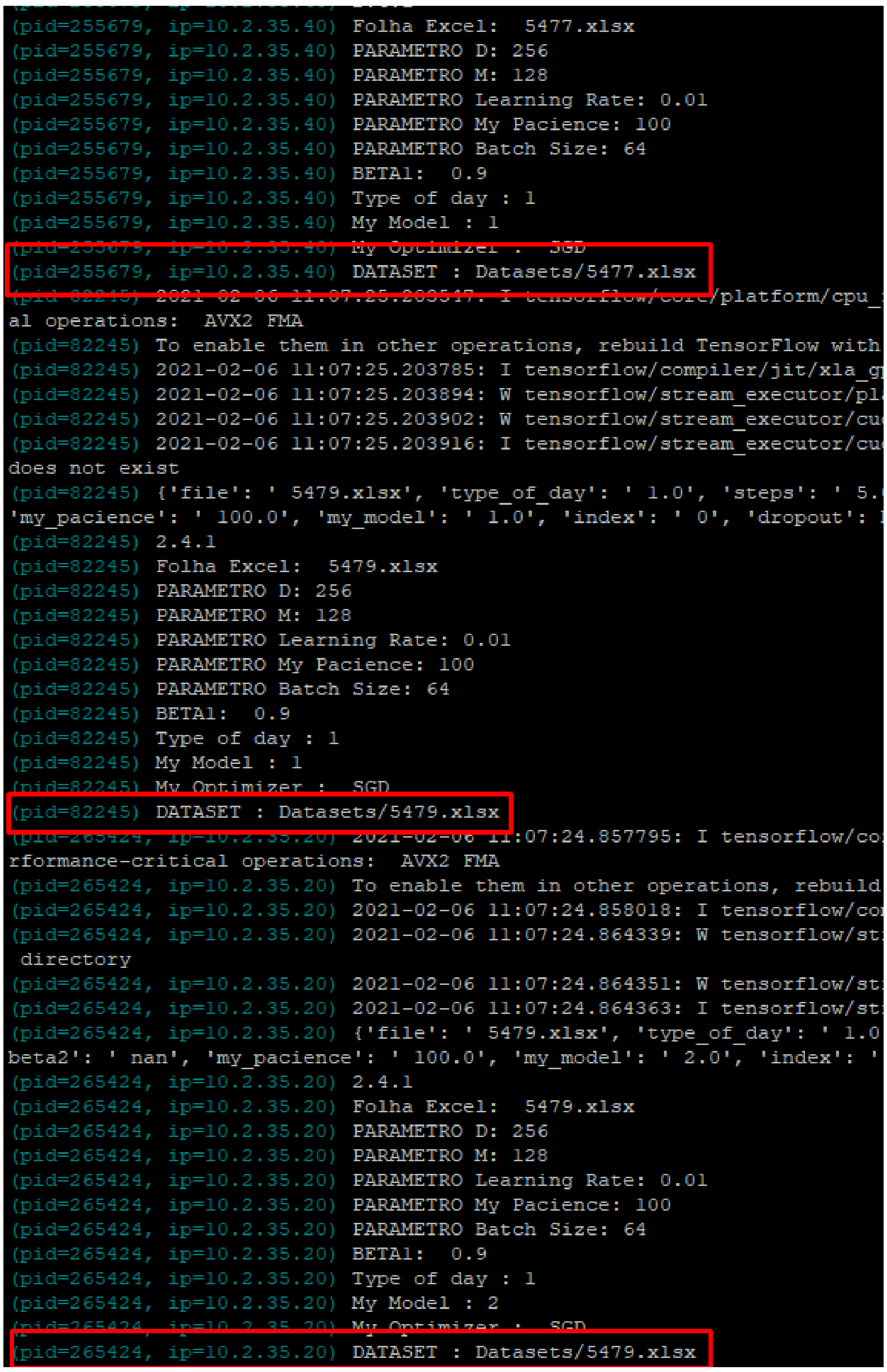
```
NODE 0 : 10.2.35.43
```

For node 0 does not appears in Fig:4.22 the IP because it´s the node that we are working at, and it is the master in the cluster.

```
NODE 1 : 10.2.35.20
```

```
NODE 2 : 10.2.35.40
```

In figure: 4.22 below, all nodes are executing the dataset, which was assigned to it, at the same time.



```
(pid=255679, ip=10.2.35.40) 2.4.1
(pid=255679, ip=10.2.35.40) Folha Excel:  5477.xlsx
(pid=255679, ip=10.2.35.40) PARAMETRO D: 256
(pid=255679, ip=10.2.35.40) PARAMETRO M: 128
(pid=255679, ip=10.2.35.40) PARAMETRO Learning Rate: 0.01
(pid=255679, ip=10.2.35.40) PARAMETRO My Pacience: 100
(pid=255679, ip=10.2.35.40) PARAMETRO Batch Size: 64
(pid=255679, ip=10.2.35.40) BETA1:  0.9
(pid=255679, ip=10.2.35.40) Type of day : 1
(pid=255679, ip=10.2.35.40) My Model : 1
(pid=255679, ip=10.2.35.40) My Optimizer :  SGD
(pid=255679, ip=10.2.35.40) DATASET : Datasets/5477.xlsx
(pid=82245) 2021-02-06 11:07:25.203547: I tensorflow/core/platform/cpu_
al operations:  AVX2 FMA
(pid=82245) To enable them in other operations, rebuild TensorFlow with
(pid=82245) 2021-02-06 11:07:25.203785: I tensorflow/compiler/jit/xla_g
(pid=82245) 2021-02-06 11:07:25.203894: W tensorflow/stream_executor/pl
(pid=82245) 2021-02-06 11:07:25.203902: W tensorflow/stream_executor/cu
(pid=82245) 2021-02-06 11:07:25.203916: I tensorflow/stream_executor/cu
does not exist
(pid=82245) {'file': ' 5479.xlsx', 'type_of_day': ' 1.0', 'steps': ' 5.
'my_pacience': ' 100.0', 'my_model': ' 1.0', 'index': ' 0', 'dropout':
(pid=82245) 2.4.1
(pid=82245) Folha Excel:  5479.xlsx
(pid=82245) PARAMETRO D: 256
(pid=82245) PARAMETRO M: 128
(pid=82245) PARAMETRO Learning Rate: 0.01
(pid=82245) PARAMETRO My Pacience: 100
(pid=82245) PARAMETRO Batch Size: 64
(pid=82245) BETA1:  0.9
(pid=82245) Type of day : 1
(pid=82245) My Model : 1
(pid=82245) My Optimizer :  SGD
(pid=82245) DATASET : Datasets/5479.xlsx
(pid=265424, ip=10.2.35.20) 2021-02-06 11:07:24.857795: I tensorflow/co
rformance-critical operations:  AVX2 FMA
(pid=265424, ip=10.2.35.20) To enable them in other operations, rebuild
(pid=265424, ip=10.2.35.20) 2021-02-06 11:07:24.858018: I tensorflow/co
(pid=265424, ip=10.2.35.20) 2021-02-06 11:07:24.864339: W tensorflow/st
 directory
(pid=265424, ip=10.2.35.20) 2021-02-06 11:07:24.864351: W tensorflow/st
(pid=265424, ip=10.2.35.20) 2021-02-06 11:07:24.864363: I tensorflow/st
(pid=265424, ip=10.2.35.20) {'file': ' 5479.xlsx', 'type_of_day': ' 1.0
beta2': ' nan', 'my_pacience': ' 100.0', 'my_model': ' 2.0', 'index': '
(pid=265424, ip=10.2.35.20) 2.4.1
(pid=265424, ip=10.2.35.20) Folha Excel:  5479.xlsx
(pid=265424, ip=10.2.35.20) PARAMETRO D: 256
(pid=265424, ip=10.2.35.20) PARAMETRO M: 128
(pid=265424, ip=10.2.35.20) PARAMETRO Learning Rate: 0.01
(pid=265424, ip=10.2.35.20) PARAMETRO My Pacience: 100
(pid=265424, ip=10.2.35.20) PARAMETRO Batch Size: 64
(pid=265424, ip=10.2.35.20) BETA1:  0.9
(pid=265424, ip=10.2.35.20) Type of day : 1
(pid=265424, ip=10.2.35.20) My Model : 2
(pid=265424, ip=10.2.35.20) My Optimizer :  SGD
(pid=265424, ip=10.2.35.20) DATASET : Datasets/5479.xlsx
```

Figure 4.22: Distributed ML tasks

4.2.3 Serialized Tasks

We have a scenario in which we have 6 tasks that are performed. With our process of distribution and parallelization of tasks we will have more efficiency than if there is none of that ... As we have seen before ...

However, it is possible to improve the efficiency of these tasks ... through serialization, that is, process the data as soon it becomes available.Fortunately, Ray allows to do exactly this by calling ray.wait() on a list of object IPs. Without specifying any other parameters, this function returns as soon as an object in its argument list is ready. This call has two returns: (1) the Identification (ID) of the ready object, and (2) the list containing the IPs of the objects not ready yet. The modified program is below. Note that one change we need to do is to replace *processresults()* with *processincremental()* that processes one result at a time. However, for this ML script we can only serialize it in a pre-defined/semi-automatic way.

In the figure 4.23 it´s presented the execution of 6 ML tasks by a single node (in this case the node 0).

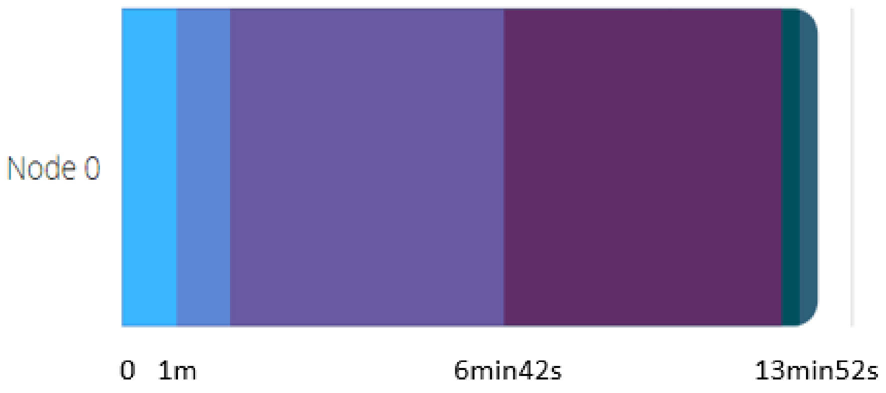
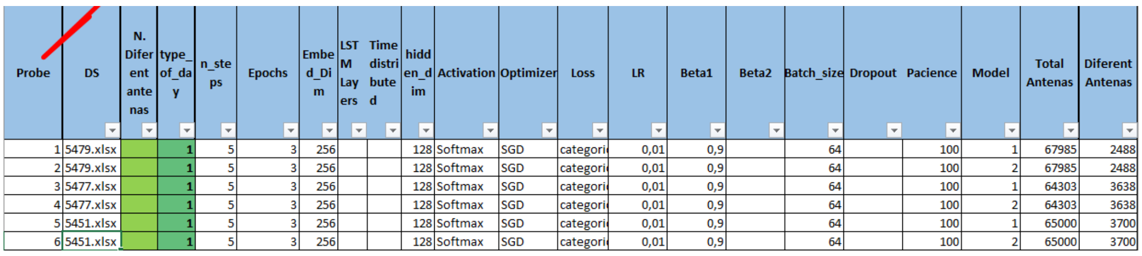


Figure 4.23: ML tasks Node 0

Once again we have the example of having only one node performing all tasks, And we can deduce that this example is not at all efficient because the time will be very extended, because in the Fig. 4.23 it takes 13 minutes and 52 seconds, but the documentation below will show how to perform a distributed, parallelized and serialized way to execute this 6 tasks. The results are in the Fig: 4.26

The Fig. 4.24 shows what are the 6 tasks (datasets), respectively.

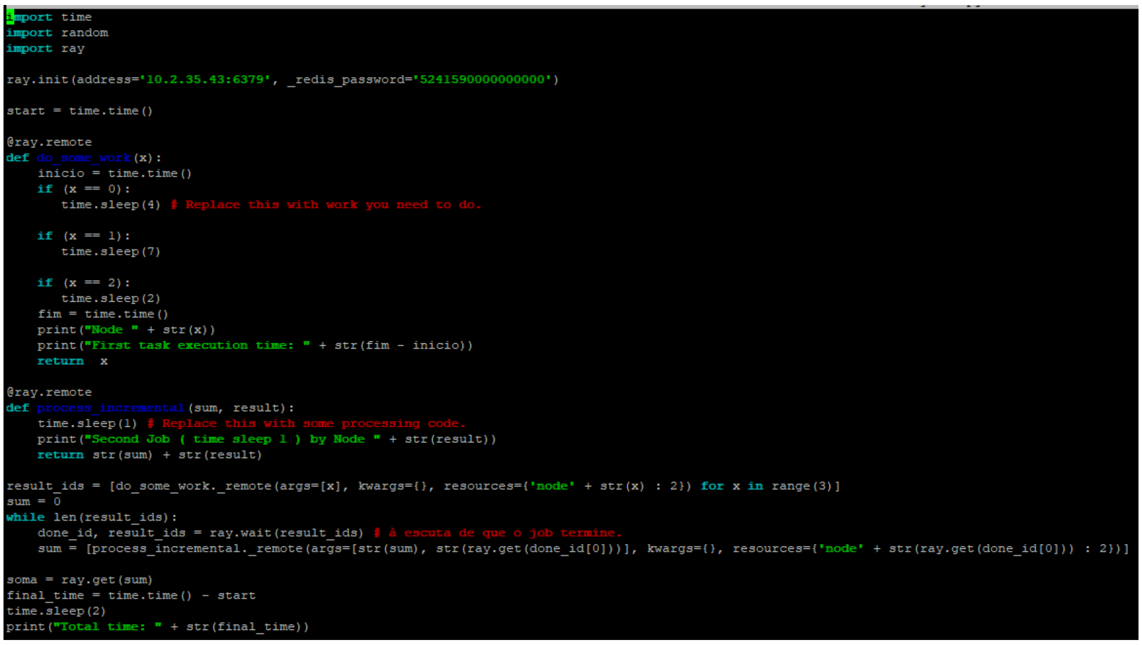


Probe	DS	N. Diferent antenas	type_of_day	n_steps	Epochs	Embed_Dim	LSTM Layers	Time distributed	hidden_dim	Activation	Optimizer	Loss	LR	Beta1	Beta2	Batch_size	Dropout	Pacience	Model	Total Antenas	Diferent Antenas
1	5479.xlsx		1	5	3	256			128	Softmax	SGD	categori	0,01	0,9		64		100	1	67985	2488
2	5479.xlsx		1	5	3	256			128	Softmax	SGD	categori	0,01	0,9		64		100	2	67985	2488
3	5477.xlsx		1	5	3	256			128	Softmax	SGD	categori	0,01	0,9		64		100	1	64303	3638
4	5477.xlsx		1	5	3	256			128	Softmax	SGD	categori	0,01	0,9		64		100	2	64303	3638
5	5451.xlsx		1	5	3	256			128	Softmax	SGD	categori	0,01	0,9		64		100	1	65000	3700
6	5451.xlsx		1	5	3	256			128	Softmax	SGD	categori	0,01	0,9		64		100	2	65000	3700

Figure 4.24: 6 Datasets

The results we obtained were all *semi-automatic serialization*, this is because we have the nodes to perform a new task whenever it is free, however, the new task to be performed is defined by us so it is not completely automatic, it would be if he automatically found a new task to perform, for that we would have to use *ray.wait* as we talked earlier, which we were unable to apply ...

However we did a *test with ray.wait* using a script that simply does *time.sleep*.



```
import time
import random
import ray

ray.init(address='10.2.35.43:6379', _redis_password='5241590000000000')

start = time.time()

@ray.remote
def do_some_work(x):
    inicio = time.time()
    if (x == 0):
       time.sleep(4) # Replace this with work you need to do.

    if (x == 1):
       time.sleep(7)

    if (x == 2):
       time.sleep(2)
    fim = time.time()
    print("Node " + str(x))
    print("First task execution time: " + str(fim - inicio))
    return  x

@ray.remote
def process_incremental(sum, result):
    time.sleep(1) # Replace this with some processing code.
    print("Second Job ( time sleep 1 ) by Node " + str(result))
    return str(sum) + str(result)

result_ids = [do_some_work._remote(args=[x], kwargs={}, resources={'node' + str(x) : 2}) for x in range(3)]
sum = 0
while len(result_ids):
    done_id, result_ids = ray.wait(result_ids) # à escuta de que o job termine.
    sum = [process_incremental._remote(args=[str(sum), str(ray.get(done_id[0]))], kwargs={}, resources={'node' + str(ray.get(done_id[0])) : 2})]

soma = ray.get(sum)
final_time = time.time() - start
time.sleep(2)
print("Total time: " + str(final_time))
```

Figure 4.25: Time.sleep tasks with ray.wait

And finally, we have the results with Distribution, parallelization and semi-serialization of tasks.

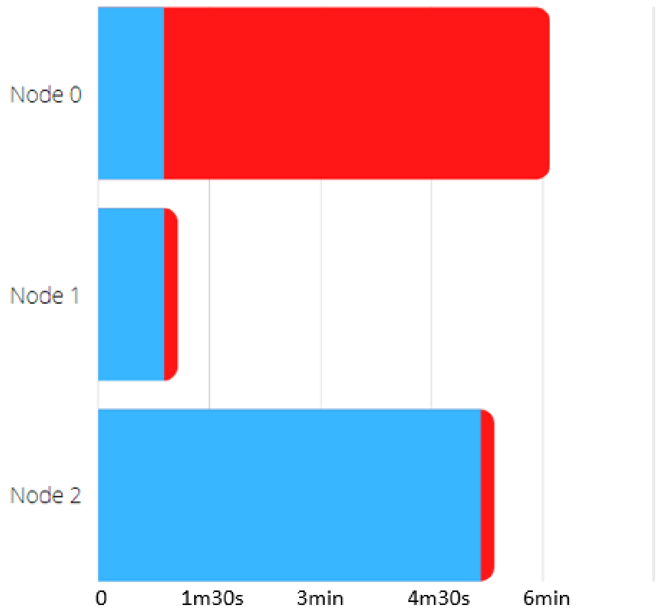
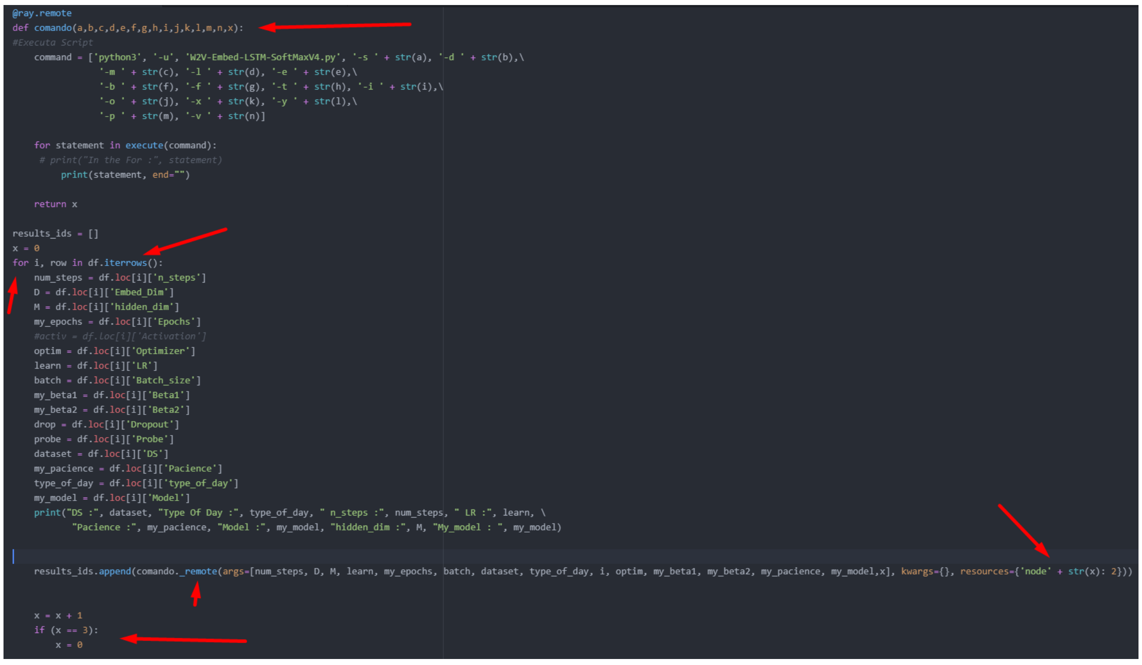


Figure 4.26: Distribution, parallelization and semi-serialization of tasks

There is a difference in the execution times between Fig. 4.23 and Fig. 4.26



```python
@ray.remote
def comando(a,b,c,d,e,f,g,h,i,j,k,l,m,n,x):
#Executa Script
    command = ['python3', '-u', 'W2V-Embed-LSTM-SoftMaxV4.py', '-s ' + str(a), '-d ' + str(b),\
               '-m ' + str(c), '-l ' + str(d), '-e ' + str(e),\
               '-b ' + str(f), '-f ' + str(g), '-t ' + str(h), '-i ' + str(i),\
               '-o ' + str(j), '-x ' + str(k), '-y ' + str(l),\
               '-p ' + str(m), '-v ' + str(n)]

    for statement in execute(command):
     # print("In the For :", statement)
        print(statement, end="")

    return x

results_ids = []
x = 0
for i, row in df.iterrows():
    num_steps = df.loc[i]['n_steps']
    D = df.loc[i]['Embed_Dim']
    M = df.loc[i]['hidden_dim']
    my_epochs = df.loc[i]['Epochs']
    #activ = df.loc[i]['Activation']
    optim = df.loc[i]['Optimizer']
    learn = df.loc[i]['LR']
    batch = df.loc[i]['Batch_size']
    my_beta1 = df.loc[i]['Beta1']
    my_beta2 = df.loc[i]['Beta2']
    drop = df.loc[i]['Dropout']
    probe = df.loc[i]['Probe']
    dataset = df.loc[i]['DS']
    my_pacience = df.loc[i]['Pacience']
    type_of_day = df.loc[i]['type_of_day']
    my_model = df.loc[i]['Model']
    print("DS :", dataset, "Type Of Day :", type_of_day, " n_steps :", num_steps, " LR :", learn, \
          "Pacience :", my_pacience, "Model :", my_model, "hidden_dim :", M, "My_model : ", my_model)


    results_ids.append(comando._remote(args=[num_steps, D, M, learn, my_epochs, batch, dataset, type_of_day, i, optim, my_beta1, my_beta2, my_pacience, my_model,x], kwargs={}, resources={'node' + str(x): 2}))


    x = x + 1
    if (x == 3):
        x = 0
```

Figure 4.27: Manager Script semi-serialized

Chapter 5

Conclusions and Future Work

So, if we applied this ”multi-core chips” on several machines, we can create a large cluster (Powerful Computer) because we have just joined all the computers onto a single machine (logically). With this much power we can just pick processes that take long time (e.g. 12 Hours) like ML jobs and execute/process them in 4 Hours, and that is revolutionary. The entire project itself was very rewarding because not only was the whole process something new, but all the platforms/software , the lack of GPU interface is not a problem just an adversity that we want together fix it and maybe take this project to another level. The entry into the AI area, as well as the application of computing parallel to the initial problem, in short, the whole process from November to February was hard, but all of it was very rewarding, allowing entry into new areas and new futures.

On a future vision, we found that some features of this project didn't were accomplished as we wish, but we see these ”problems” as a motivation to transform the project again, upgrade/patch some ”bugs” (not bugs but features unfinished), like GPU module available on the platform Fed4Fire and as another example change the whole paradigm of this project and instead of being a semi-automatic system turn it into an completely autonomous one.[2]

References

[1] fed4Fire. "What is Fed4Fire?" In: (2021). URL: `https://www.fed4fire.eu/the-project/`.

[2] Software Carpentry Foundation. "Parallel vs Sequential Computing". In: (2021). URL: `https://researchcomputingservices.github.io/parallel-computing/01-parallel-introduction/`.

[3] Geni. "What is GENI?" In: (2021). URL: `https://www.geni.net/about-geni/what-is-geni/`.

[4] GPUlab. "jupyterhub Documentation". In: (2021). URL: `https://doc.ilabt.imec.be/ilabt/jupyter/index.html`.

[5] IBM. "What is Machine Learning". In: (2020). URL: `https://www.ibm.com/cloud/learn/machine-learning`.

[6] IBM. "What is Machine Learning". In: (2020). URL: `https://www.ibm.com/cloud/learn/machine-learning`.

[7] imec iLab.t. "JupyterHub at imec iLab.t". In: (2019). URL: `https://doc.ilabt.imec.be/ilabt/jupyter/index.html#jupyterhub-at-imec-ilab-t`.

[8] H. C. Kaskavalci and S. Gören. "A Deep Learning Based Distributed Smart Surveillance Architecture using Edge and Cloud Computing". In: *2019 International Conference on Deep Learning and Machine Learning in Emerging Applications (Deep-ML)*. Aug. 2019, pp. 1–6. DOI: `10.1109/Deep-ML.2019.00009`.

[9] Microsoft. "Distributed training of deep learning models on Azure". In: (2021). URL: `https://docs.microsoft.com/pt-pt/archive/blogs/azurecat/new-`

reference-architecture-distributed-training-of-deep-learning-models-on-azure.

[10] Netdata. “Netdata Cloud Usage”. In: (2021). URL: https://learn.netdata.cloud/docs/agent/collectors/plugins.d.

[11] Netdata. “Netdata Cloud Usage”. In: (2021). URL: https://learn.netdata.cloud/.

[12] Robert Nishihara. “Modern Parallel and Distributed Python: A Quick Tutorial on Ray”. In: (Feb. 2019). URL: https://towardsdatascience.com/modern-parallel-and-distributed-python-a-quick-tutorial-on-ray-99f8d70369b8.

[13] Robert Nishihara and Philipp Moritz. “Ray: A Distributed System for AI”. In: (Jan. 2018). URL: https://bair.berkeley.edu/blog/2018/01/09/ray/.

[14] Digital Ocean. “How to Install, Run, and Connect to Jupyter Notebook on a Remote Server”. In: (2019). URL: https://www.digitalocean.com/community/tutorials/how-to-install-run-connect-to-jupyter-notebook-on-remote-server.

[15] Ray. “Numerical Computation”. In: (2021). URL: https://rise.cs.berkeley.edu/blog/modern-parallel-and-distributed-python-a-quick-tutorial-on-ray/.

[16] Ion Stoica. “Programming in Ray: Tips for first-time users”. In: (Feb. 2019). URL: https://rise.cs.berkeley.edu/blog/ray-tips-for-first-time-users/.

[17] T. Tuor et al. “Demo abstract: Distributed machine learning at resource-limited edge nodes”. In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2018, pp. 1–2. DOI: 10.1109/INFCOMW.2018.8406837.

[18] Wikipedia. “Machine Learning”. In: (2020). URL: https://en.wikipedia.org/wiki/Machine_learning.