GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Analyzing IO performance when using Dask-ML

Zoya Masih

MatrNr: -

Supervisor: Prof. Julian Kunkel

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2023

Abstract

This report aims to assess the performance of a computing cluster when executing Dask-ML, a robust machine learning framework, with a specific focus on several key performance metrics. The investigation concentrates on execution time, CPU utilization, bandwidth between workers, bytes stored in memory, and memory usage. Dask-ML, known for its distributed computing capabilities and integration with popular libraries like Scikit-learn, is well-suited for handling extensive datasets and complex machine learning models.

Additionally, this study evaluates the I/O performance associated with reading LAZ files, a compressed variant of LAS files commonly employed in LiDAR data processing, on the computing cluster. The results underscore the system's scalability in response to varying input LAZ file sizes. Furthermore, this report delves into the challenges associated with efficient LAZ file processing, emphasizing the use of memory-efficient techniques employing the Laspy library.

Contents

List of Tables

List of Figures

List of Listings

1 Introduction

This report reviews both theoretical explorations and practical efforts related to the course titled: 'Seminar with Practical: Scalable Computing Systems and Applications in AI, Big Data and HPC'.The main goal of this project is analyzing I/O performance during execution of a ML pipeline on GWDG's scientific supercomputer cluster. The source codes are available in the GWDG's repository.

1.1 Motivation

In the context of contemporary computing, the effective handling and enhancement of Input/Output (I/O) processes play a vital role. This function cannot be accomplished unless there is a thorough and efficient analysis of I/O patterns and behavior. Such analysis can help us in upholding the system's efficiency, scalability, dependability, and overall well-being. Also, in the case of performance obstacles, it facilitates the detection and resolution of the issues, and aids in fine-tuning resource utilization, and guarantees a seamless and prompt user interaction. This importance grows when confronting the difficulties presented by Big Data, supercomputing, and parallel/distributed computing models.

This is well known that large-scale data is being produced constantly, and nonstop. This has forced analysts to face analyzing huge data that can not fit into the memory. Obviously, this large-scale data can not be processed by regular ML techniques and algorithms. In such a scenario, other options are subsampling, Cloud Computing, and out-of-core machine learning. In the last option, the data is loaded from a secondary storage device in mini-batches such that the batches can fit into the main memory. The next step is extracting features from that data to be used as the ML model's input. In the last step, the model is trained incrementally. In incremental learning, models learn from new information in real time or on-the-fly. The model starts learning with the mini-batch and when new data becomes available, the model gets updated, while preserving its existing knowledge[STA23].

Not all ML algorithms support out-of-core ML. One, among the ones with this capability, is Dask-ML that is a library built on top of Dask framework. Dask, itself is a library for parallel computing in Python. Dask-ML provides machine learning algorithms and tools that are designed to work efficiently with large datasets using Dask's parallel computing capabilities[Wan+20].

1.2 Background

The main concepts and tools which are used in the current project, are briefly explained in this section. To ensure that there is no distortion, most of the Dask information and the information on LAZ files, given in this section are from the official websites [Das] and [LAZ].

1.2.1 Dask Library

Dask is a flexible library for parallel computing in Python. This library is used by the world's largest banks, national labs, retailers, technology companies, and government agencies. It is used in highly secure environments. The most common problem that Dask solves is connecting Python analysts to distributed hardware, particularly for data science and machine learning workloads. Dask, provided in 2015, is a relatively thin wrapper on top of NumPy, Pandas, Jupyter, Scikit-Learn libraries. All these, have made Dask trusted and simple to use.
Dask usually communicates over TCP, using *msgpack* for small administrative messages, and its own protocol for efficiently passing around large data. The scheduler and each worker host their own TCP server, making Dask a distributed peer-to-peer network that uses point-to-point communication.
Some Dask deployments have been on around 1000 multicore machines, perhaps 20,000 cores in total, although most institutional-level problems (1-100 TB) are well solved by deployments of 10–50 nodes.
Technically, each task (an individual Python function call) in Dask has an overhead of around 200 microseconds. So if these tasks take 1 second each, then Dask can saturate around 5000 cores before scheduling overhead dominates costs. As workloads reach this limit, they are encouraged to use larger chunk sizes to compensate.
Dask is resilient to the failure of worker nodes. It knows how it came to any result and can replay the necessary work on other machines if one goes down. If Dask's centralized scheduler goes down, then resubmission of the computation will be needed.
Dask supports most formats that NumPy and Pandas support, however, not all formats are well-suited for parallel access. The following formats are usually fine to be used by Dask.

- Tabular: Parquet, ORC, CSV, Line Delimited JSON, Avro, text
- Arrays: HDF5, NetCDF, Zarr, GRIB

1.2.2 Dask-ML

Dask-ML provides scalable machine learning in Python using Dask alongside popular machine learning libraries like Scikit-Learn, XGBoost, and others. Although there are technics like sampling for scaling an ML model, Dask-ML is a powerful tool in scaling data size and model size. It is particularly well-suited for scenarios in which we are working with massive datasets or training complex machine learning models, and we need to efficiently leverage distributed computing capabilities.

1.2.3 LAZ File Format

In this experience, LAZ files are used as input. In the following, a brief introduction about this file format is provided. LAS and LAZ are popular vector formats for storing LiDAR data as point clouds. They're widely embraced in the surveying industry due to their compatibility with numerous software tools, making them essential for LiDAR data storage and sharing. LAS, short for LIDAR Aerial Survey, is a prevalent binary format for 3D LiDAR point cloud data. Developed by ASPRS in 2003, it's the industry standard. LAS files comprise LiDAR points with attributes like X, Y, and Z coordinates,

intensity values, return numbers, and classification codes. This format is adaptable, allowing users to include specific attributes as needed. LAS is known for its extensibility, enabling the addition of new attributes. It also includes vital metadata like projection information and survey timestamps. LAZ, or LASzip, is a compressed variant of LAS, designed to reduce file size. Created in 2007 as open-source software, it compresses LAS files, often reducing their size by up to 90%, without data loss. Both LAS and LAZ formats excel in interoperability with various LiDAR software tools, simplifying data analysis and visualization. Their standardized nature ensures efficient data sharing and processing across diverse applications without the need for custom software development or data conversion. This interoperability streamlines workflows and facilitates analysis with different tools, regardless of the data's source or processing software.

1.3 Report Outline and Goals

In spite of great capabilities of Dask, using this tool has been neglected in many ML projects. In the current project, I embarked on the task of running an out-of-core machine learning pipeline on the GWDG's SCC and accordingly assessed its I/O performance within this environment.
My primary objective was to evaluate the I/O performance of the cluster, when executing an IO-intensive job. In this path, I tried to gain an understanding of Dask and leverage it to develop an ML pipeline as the job.
The outline of the report is structured as follows. Section two delves into the details of how the ML pipeline is implemented using Dask. The software packages and hardware modules employed are also discussed in this section. Section 3 presents information about the pre-processing and trainig of the model, and in section 4 empirical results obtained from running the job are presented. Finally, in section 5 the report is concluded.

2 Methodology

In this chapter some details about the setup of this experiment, and data resource are provided.

2.1 Data Collection

In this project, a machine learning pipeline is developed, that aims at predicting tree species within a forest environment. Input data collection process involves sourcing forest datasets and information from a reputable source, primarily the GWDG GitLab repository [Hos].
The forest dataset used in this project is generated by SynForest, a tool designed for the creation of large-scale, realistic point clouds representing forested areas. SynForest achieves this by simulating a Light Detection and Ranging (LiDAR) scanning process from either a stationary or moving platform. This simulation method employs pulsed laser technology to accurately measure variable distances to the Earth's surface.
For this project, each execution of the 'run.py' script with the specified configuration results in the creation of a LAZ file and a CSV file, from a RES file. These files serve as the primary input data for our machine learning model.

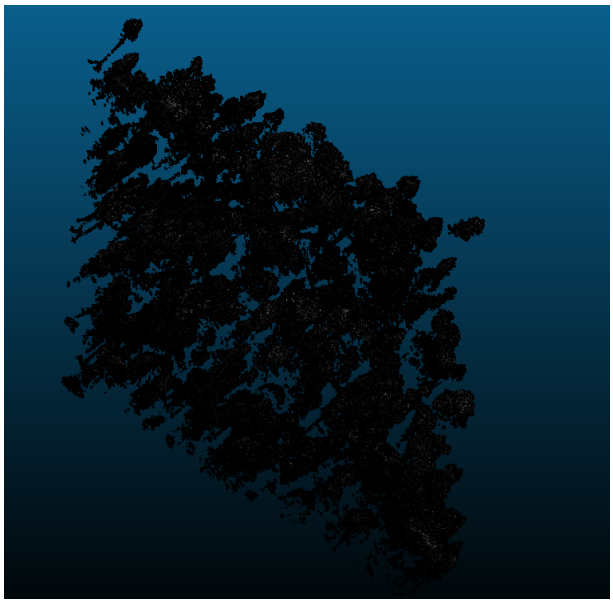

Figure 1: A LAZ file, showing a forest

2.2 Experimental Setup

In this section, a brief overview of the experimental setup used in the project is provided.

2.2.1 Hardware Environment

This experiment was conducted on the scientific computing cluster of GWDG. The key components of the hardware environment are as follows. The cluster is powered by Intel(R) Xeon(R) Silver 4214 CPUs with 48 CPU cores, supporting virtualization (VT-x) and organized into two NUMA nodes for efficient memory management. Memory (RAM) capacity is 187 GB (23 GB in use, 131 GB available), with 10 GB allocated for shared memory and a 32GB buffer/cache[GWD].

2.2.2 Software Environment

The software environment for this project is based on Scientific Linux 7.9, Dask version 2023.2.0, and the execution environment is JupyterHub.
In the cluster setup, a Dask cluster consisting of 4 worker nodes is used, each equipped with 3 cores, to distribute the computational workload.

2.3 Metrics and Measurements

The metrics which are considered here are Bandwidth workers, Bytes stored, CPU utilization, memory utilized, and execution time of the modelling.

3 Implementation

At the very beginning, a Dask client for distributed computing is initialized to enhance scalability[Bas].

```
The laz file:
     X    Y    Z  id
0  x11  y12  z13   1
1  x12  y22  z23   1
2  x13  y23  z33   0
3  x14  y24  z34   2
4  x15  y25  z35   0

The CSV file:
    d   h   s  id
0  d1  h1  s1   0
1  d2  h2  s2   1
2  d3  h3  s3   2
```

Figure 2: A Miniature Forest's Input File

3.1 Preprocessing

The primary input dataset, generated by SynForest is in the LAZ file format, in which all the features are providing technical information related to LIDAR data, besides coordination of the forest's points. To align the machine learning pipeline more closely with tree-related attributes, a CSV file is also generated by SynForest, with minor modifications. This CSV file consists of 'height' and 'diameter' of the trees, in addition to the specie of the trees.
The main objective was to create a model capable of predicting the species of trees based on the geographic coordinates, diameter, and height of the trees that the point is located on.
The first challenge in this step, was to load the datasets properly. Each input compressed LAZ file is corresponding to a forest and is around 2 GiB, and the uncompressed file is around 10 GiB. Although this is not a big dataset, regular data loading using *laspy.read("/file/path")* did not work properly for more than 2 files. As there is no native support for LAZ files in Dask, an efficient reading and processing method was applied. In this method, smaller and more manageable pieces were loaded, rather than loading the entire file into memory at once, to prevent memory issues. For the sake of lack of references that address this issue, the corresponding code snippet was recorded in Listing 2.
Subsequently, Dask dataframes were created from both the CSV and LAZ files. The next step involved merging these two Dask dataframes to create the final dataset, which would serve as the input for the machine learning model. The only remaining pre-processing step was encoding the tree species into integer values for compatibility with our model. To achieve this, the 'LabelEncoder' from the 'dask_ml' library was employed. Before starting the main implementation, to clarify what is happening in this experiment, a miniature forest is provided in this section, that shows how the files are processed.
For the sake of clarity, a tiny forest is provided here, that consists of 5 points which are located on 3 trees. Figure 2 shows that what corresponding LAZ file and CSV file look like, in a real scale forest. In this figure, diameter, height, and species of the trees are demonstrated. The attribute *id* shows the number of the tree. The merged file also is

```
import dask
import dask.delayed
import dask.dataframe as dd
import laspy
from functools import partial
filename = "\Path\to\laz\file"
@dask.delayed
def offset_ reader(filename, offset, npoints):
    with open(filename, "rb") as fid:
        reader = laspy.LasReader(fid)
        reader.seek(offset)
        data = reader.read_ points(npoints).array
    return pd.DataFrame(data)
def partitions(filename, npartitions=100):
    with open(filename, "rb") as fid:
        reader = laspy.LasReader(fid)
        npoints = reader.header.point_ count
        batchsize = round(npoints/npartitions/10)*10
        batches = list(range(0, npoints, batchsize))
        # include the last hanging strip
        batches.append(npoints - batches[-1])
        pairs =[(batch_index, batchsize) for batch_index in batches]
    lazy_ load = []
    for pos, points in pairs:
        lazy_func = offset_reader(filename, pos, points)
        lazy_ load.append(lazy_func)
    return lazy_ load
def dask_reader(filename, npartitions=100):
    with open(filename, "rb") as fid:
        reader = laspy.LasReader(fid)
        dtype = reader.header.point_ format.dtype()
        meta = pd.DataFrame(np.empty(0, dtype=dtype))
    return dd.from_delayed(partitions(filename, npartitions), meta=meta)}

```

Listing 1: Loading LAZ Files

shown in figure 3 that gives a better understanding of the input file used in the predicting model.

	X	Y	Z	d	h	s
0	x11	y12	z13	d2	h2	s2
1	x12	y22	z23	d2	h2	s2
2	x13	y23	z33	d1	h1	s1
3	x15	y25	z35	d1	h1	s1
4	x14	y24	z34	d3	h3	s3

Figure 3: The Model's Input File

3.2 Training and Testing

In this section, an overview of the model training and evaluation process is provided. The beginning point is splitting the dataset into training and testing sets with proportion 80-20.
Next, the features were standardized by using *StandardScaler*, that prepares the data for model training. The XGBRegressor model was employed and fit to the scaled training data for this task [XGB].
To evaluate the model, K-Fold Cross-Validation was performed. The results are visualized by using a confusion matrix, where class labels are defined and displayed for clarity.

4 Results and Analysis

To conduct an evaluation and gain insights into the system's behavior, the code was executed using varying input scenarios consisting of 1, 2, 3, 4, and 5 dense forest LAZ files. These LAZ input files, each approximately 2GB in size, undergo significant expansion to approximately 10GB when uncompressed. This deliberate selection of input variations was intended to provide a rigorous assessment of system performance and scalability.

4.1 I/O Performance Comparison

In this section, a detailed analysis of the results is provided. At the end of the section the results are also included in a table.

4.1.1 Time

Increasing the size of the initial LAZ input file from 2 to 10 GB (LAS file 10 to 50), increases the execution time significantly from 1034 to 5147 seconds, and the growth is almost linear. In the figure 4 the changes are shown.

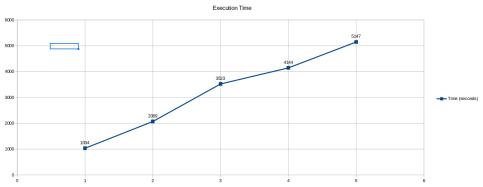

Figure 4: Execution Time Changes

4.1.2 CPU Utilizing Percentage

This factor did not change significantly during different experiments, and remained almost constant around 25 percent, during the modelling.

4.1.3 Bandwidth Workers

Bandwidth, in the Dask dashboard, refers to the rate at which data is transferred between workers. The histogram 5 taken from the dashboard, is related to the experiment with 3 initial LAZ files and displays the distribution of bandwidth across workers. The changes for this factor are also not so distinctive.

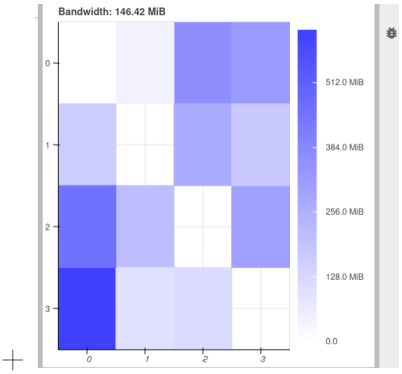

Figure 5: Bandwidth Workers

4.1.4 Bytes Stored

This critical metric provides insights into how much data is actively cached in memory, helping to manage memory resources effectively and optimizes the performance of the Dask workflows. Figure 6 demonstrates that the amount of data actively cached in memory is changing almost linearly as compared to the changing in the size of the initial input LAZ file, when running the modelling task. Figure 7 also is a screenshot of this metric, in the experiment with 4 initial files as input.

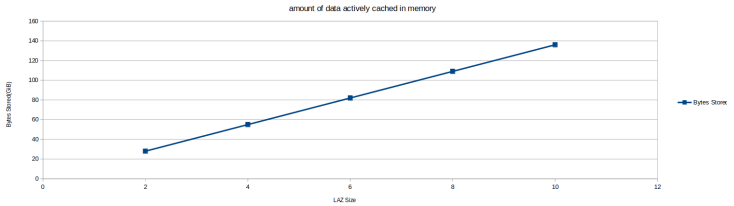

Figure 6: Cache Change

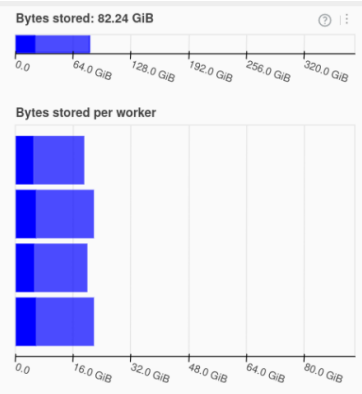

Figure 7: Bytes Stored

4.1.5 Memory

In the Dask dashboard, the *Memory* graph provides information about the memory usage and management of the Dask cluster. In this graph, the x-axis represents time, and the y-axis represents memory usage, It shows how much memory is currently being used by workers and tasks at different points in time. Figure 8 shows the *Memory* graph of running the code with 4 files. In other words, during the modelling task, memory usage increased to 59.8 GiB, in this experiment.
The memory used, increased from 7 GiB to 56.7 GiB during the 5 experiments.

Figure 8: Memory Usage

4.2 The Results

In table 4.2, all the results of the 5 experiments are shown. As expected, the execution time and active data stored on cache memory has increased in proportion to input data growth, and both of them has increased linearly. The rate at which data is transferred between workers has fluctuated among different runs.

Initial LAZ file	Bandwidth	Bytes Stored	Memory	Execution Time
2GB	114	28 GiB	7.09	1034 s
4GB	156	55.57 GiB	23	2069 s
6GB	160	82.24 GiB	32.19	3523 s
8GB	163	109 GiB	45.62	4144 s
10GB	136.53	136.53 GiB	56.72	5147 s

Table 1: The effects of initial LAZ file's size

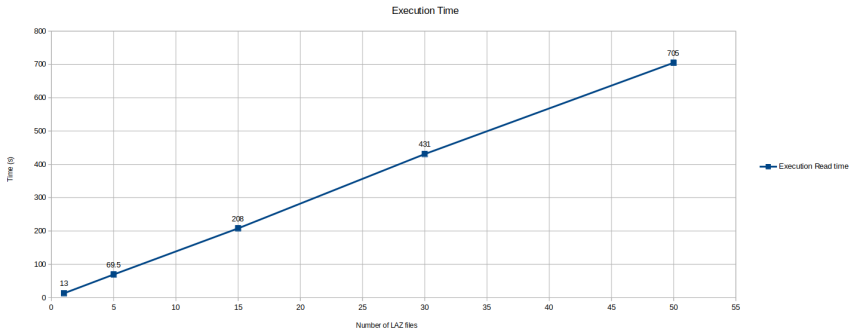

Figure 9: Reading Time

4.3 IO Evaluation

The provided results, are related to modelling a ML for predicting the species of trees in a forest. As these results, to a great extent, are effected by computation jobs, in this section the results on reading multiple LAZ files are provided. Figure 9 shows how the execution time changes while we increase the number of LAZ files. Once again here it is notified that each LAZ file is round 2 GB and the uncompressed LAS file of it is around 10 GB.
The *laspy* library employs a memory-efficient approach where it defers loading complete files into memory until necessary. LAS files are memory-mapped, which means that virtual memory is allocated to access the data as needed. This can justify the memory usage pattern that we observe, in which 9 GB for reading one file, 17 GB for reading two files, and no further increase for more than two files are seen.
Indeed, for reading one and two input files, the outputs of loading 2 are as follows.
`peak memory: 9991.87 MiB, increment: 9863.65 MiB` and
`peak memory: 17691.36 MiB, increment: 17532.84 MiB`.

```
def Multiple_ReadOnce(file_paths):
    for file in file_paths:
        las_file = laspy.read(file)
        points = las_file.points
filenames = [
    "/scratch/users/zmasih/Seminar/LAZ_files/file*  ]
Multiple_Read(file_paths)
```

Listing 2: Reading multiple LAZ files

5 Conclusion

5.1 Summary

In this study, an analysis of the I/O performance of Dask-ML was conducted, when dealing with geospatial data stored in LAZ files. Dask-ML is a powerful tool for handling large

datasets and complex machine-learning tasks, especially when leveraged in distributed computing environments. Understanding its I/O performance characteristics is essential for optimizing workflows in such scenarios.
The findings demonstrate that Dask-ML exhibits linear scalability in terms of execution time and active data storage as the size of the initial input LAZ files increases. CPU utilization remained consistent at around 25%, indicating efficient resource utilization. Bandwidth between workers displayed some variability but generally maintained adequate communication.
Efficiently reading and processing LAZ files posed a unique challenge, which is addressed using memory-mapped access and deferred loading techniques. The results illustrate that memory usage scaled moderately, reflecting the memory-efficient approach.

5.2 Outlook

Although the focus in this project is on the performance evaluation, it could be very beneficial if the ML pipeline and modelling gets more accurate and robust. The execution time in this code is so high and need to be optimized. Overall, the project has the potential to be continued and updated in some aspects

- The model needs to be more accurate
- The performance can be analyzed while changing Batch size and the number of workers.
- Partial loading of LAZ files, specifically when using Dask

References

[Bas] Luca Massaron Bastiaan Sjardin Alberto Boschetti. "Large Scale Machine Learning with Python". In: *Packt* (). URL: https://subscription.packtpub.com/book/data/9781785887215/2/ch02lvl1sec12/out-of-core-learning.

[Das] Dask. "Dask". In: *Dask Official website* ().

[GWD] GWDG. "GWDG". In: *GWDG* (). URL: https://docs.gwdg.de/doku.php?id=en:services:application_services:high_performance_computing:start.

[Hos] Ali Doost Hosseini. "SynForest". In: *GWDG-Repository* (). URL: https://gitlab-ce.gwdg.de/adoosth/synforest.

[LAZ] LAZ. "LAZ File Format". In: *Understanding LAZ files* (). URL: https://learn.rockrobotic.com/understanding-las-and-laz-file-formats.

[STA23] Ehsan Saeedizade, Roya Taheri, and Engin Arslan. "I/O Burst Prediction for HPC Clusters using Darshan Logs". In: *2023 IEEE 19th International Conference on e-Science (e-Science)*. IEEE. 2023, pp. 1–10.

[Wan+20] Meng Wang et al. "A survey on large-scale machine learning". In: *IEEE Transactions on Knowledge and Data Engineering* 34.6 (2020), pp. 2574–2594.

[XGB] XGBoost. "XGBoost". In: *XGBoost* (). URL: https://xgboost.readthedocs.io/en/stable/.

In this appendix some extra screenshots from Dask dashboard are provided. In the first figure, some pages of the dashboard is demonstrated. The central component of the

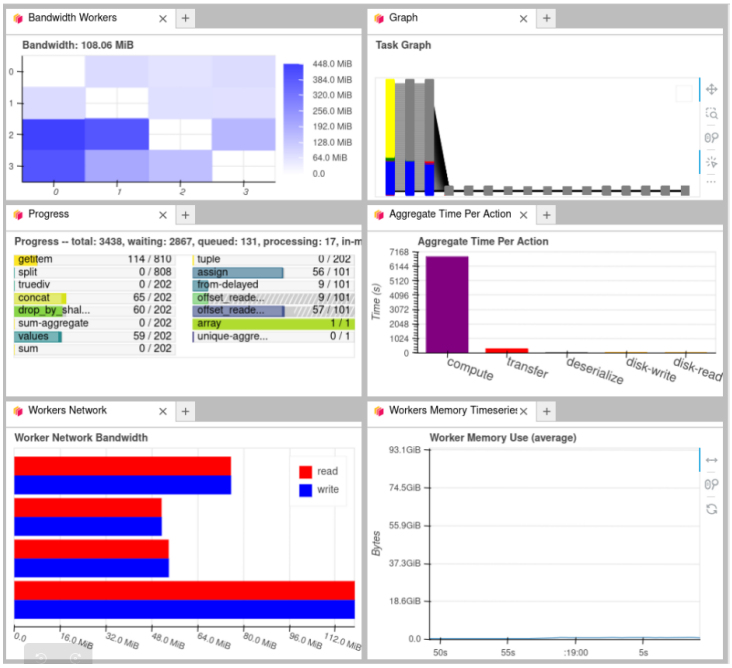

Figure 10: Dashboard while modelling

"Graph" page in Dask dashboard (the figure below), is a graphical representation of the Dask computation. This graph shows the tasks and their dependencies in the computation, allowing to visualize how data flows through the computation. It allows to see the structure of the computation, understand how tasks are scheduled, and identify any performance bottlenecks.

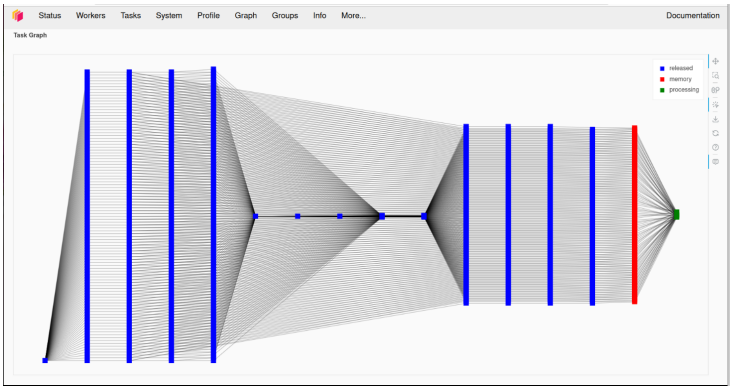

Figure 11: Page Graph in the dashboard