D4.2: PREDICTING GRAPH PROCESSING WORKLOADS

This project has received funding from the European Union´s Horizon Research and Innovation Actions under Grant Agreement nº 101093202.

Title	Version
Predicting Graph Processing Workloads	1.0

Project Number	Project Acronym	Project Title
101093202	Graph-Massivizer	Extreme and Sustainable Graph Processing for Urgent Societal Challenges in Europe

Contract Delivery Date	Actual Delivery Date	Deliverable Type* Security*
M24 (December 2024)	M25 (January 2025)	R-PU

Responsible	Organisation	Contributing WP
Andrea Bartolini	UNIBO	WP4

Authors	Organisation
Andrea Bartolini, Junaid Ahmed Khan	UNIBO
Ana-Lucia Varbanescu, Duncan Bart, Kuan-Hsun Chen	UTW
Radu Prodan, Reza Farahani	KLU

Abstract

Graph-Massivizer researches and develops a high-performance, scalable, and sustainable platform for information processing and reasoning based on the massive graph representation of extreme data. It delivers a toolkit of five open-source software tools and FAIR graph datasets covering the sustainable life cycle of processing extreme data as massive graphs. There are four major use cases for the Graph-Massivizer project from domains such as finance, global news and event tracking, auto manufacturing, and high-performance computing data centers.

One of the five tools in Graph-Massivizer is Graph-Optimizer, which informs the selection of the most suitable implementation and mapping of graph processing workloads on a given infrastructure. Graph workloads are built using basic graph operations (BGOs) optimized for different hardware platforms. Graph-Optimizer bases its prediction on models of these BGOs calibrated for given hardware platforms and input graphs. This deliverable builds upon D4.1 (which presents the Graph-Massivizer BGO repository, available on Github), and details the performance prediction methodology (as defined and used in T4.3 and T4.4), the current models for graph processing workloads, and their accuracy. An up-to-date version of the models and prediction data is also available on GitHub, and updated monthly[1].

Keywords

Graph-workload, performance prediction, analytical modeling, statistical modeling, microbenchmarking, automated benchmarking, Graph Neural Networks, SPARQL query

[1] Tasks T4.3 and T4.4 run till M30 and M33, respectively.

REVISION HISTORY

Revision	Date	Description	Author (Organization)
v0.1	27.12.2024	First version of the document.	Andrea Bartolini (UNIBO)
v0.2	15.01.2025	Restructure, adding content.	Ana Lucia Varbanescu (UTW)
v0.3	20.01.2025	Text added	Junaid Ahmed Khan (UNIBO)
v0.8	27.01.2025	Deliverable revision	Ana Lucia Varbanescu (UTW)
v 0.9	30.01.2025	Deliverable revision	Reza Farahani (AAU)
v 1.0	31.01.2025	Reviewer's comment fix	Andrea Bartolini and Junaid Ahmed Khan (UNIBO)
v 1.1	31.01.2025	Final review and formatting.	Radu Prodan (AAU)

This project has received funding from the European Union´s Horizon Research and Innovation under Grant Agreement nº 101093202.

More information is available at https://graph-massivizer.eu/.

COPYRIGHT STATEMENT

INDEX

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION.

1.1 PURPOSE OF THE DOCUMENT

Graph-Optimizer is responsible for deriving the best-performing implementation of graph processing workflows by (a) selecting the most suitable Basic Graph Operations (BGO) implementations from existing implementations available in a BGO repository based on their performance models, (b) composing the best version of the workload using composition rules and their models to interconnect the BGOs, and (c) generating executable code for the selected workflow.

This report presents the architecture and current state of the BGOs already developed in the project and the composition rules that enable them to be combined in complex workflows. Specifically, in the following sections, we provide examples of how BGOs are constructed, modeled, and benchmarked, and further link to the active development repository for the code, performance models, and calibration data. As such, this document also includes reading guidelines for the repository itself.

1.2 RELATIONSHIP TO OTHER DELIVERABLES

Deliverable D2.1 [1], entitled "Graph-Massivizer Requirements Elicitation and First Architecture Design" defined an initial high-level architecture for the Graph-Massivizer toolkit. This deliverable uses the proposed architecture but focuses only on aspects related to the implementation of Graph-Optimizer, including its input/output links to the rest of the architecture.
Deliverable D3.1 [2], entitled "Massive Graph Inception Architecture and Data Management" defines the concept of basic graph operations (BGOs) in the scope of Graph-Massivizer and introduces a Use Case 0 (UC-0) example to better explain the role of BGOs in the project. This deliverable builds upon the BGO concept and focuses on the design, implementation, benchmarking, and modeling aspects of the BGOs. The UC-0 example is also present in this deliverable, as the BGOs we use as examples are those used to build UC-0.
Deliverable D4.1 [3], entitled "BGO and composition rule for heterogeneous infrastructure" defines the concept of basic graph operations (BGOs) in the scope of Graph-Massivizer and introduces a set of BGOs used in the projects. It further introduces the analytical models for performance prediction of BGOs based on a microbenchmark approach. In this deliverable, we generalize the analytical model and extend it with statistical performance prediction models.
Deliverable D6.1 [4], entitled "Graph-Massivizer Infrastructure, Tools, and Use Cases" focuses on the Graph-Massivizer infrastructure comprising all tools in the Graph-Massivizer toolkit, thus expanding on the high-level architecture from D2.1. This deliverable contributes details on Graph-Optimizer implementation and use-case UC-0.

1.3 STRUCTURE OF THE DOCUMENT

Section 2 presents the terminology needed to understand this document. This is a brief recap of the definitions and terms introduced in D2.1, D3.1, D4.1 and D6.1, combined with a brief introduction to graph processing terminology, a brief overview of CPU and GPU architectural features, and basic modeling techniques.

Section 3 presents the methodological aspects of our BGO modeling and benchmarking using the analytic model. Should new methods be added and/or old methods become deprecated, we will update the list accordingly. A live version of this list is also hosted in our repository.

Section 4 discusses the statistical modeling of BGOs using machine learning and artificial intelligence techniques. It focuses on the implementation of these methods for two specific BGOs within UC-4 of the Graph-Massivizer project. The detailed platform descriptions, BGO code, models, and benchmarking data are all available in our repository.

Section 5 concludes this deliverable with a summary and outlines the next steps.

2 BACKGROUND

This section presents the basic terminology needed to understand this document.

Graph-Massivizer executes graph processing workloads expressed as workflows of BGOs. BGOs are well-defined graph operations whose performance can be predicted and combined into workflows to generate complex processing algorithms. Both these concepts are formally defined in D3.1 (section 2.1.1) [2]. This deliverable provides examples of BGOs and examples of composition rules and acts as a reading guideline for the constantly updated repository of BGOs and composition rules, available on GitHub here.

To optimize the implementation of these workloads and select the right hardware platforms for executing them, Graph-Optimizer uses performance and energy models to enable a performance- and/or energy-based ranking of alternative implementations. These models are formally defined in D3.1 (section 2.1.1) [2]. In this deliverable, we present concrete examples of such models, with the comprehensive list constantly updated in the repository on GitHub.

The models are *calibrated* and *validated* using benchmarking. We differentiate between *microbenchmarking*, which focuses on measuring the performance of the individual components of the performance models (e.g., single operations like additions or multiplications or memory read/writes), and *benchmarking*, which focuses on collecting performance data for the actual BGO executions, enabling model validation. The formal definition of *microbenchmarking* and *benchmarking* can be found in D4.1.

Finally*, Analytical models* capture the functional behaviour of the BGO in a closed-form expression. Such models are created by the BGO developer from source code and lead to a symbolic model expressed as a closed-form equation with parameters related to the input graph. In contrast, *statistical models* target the creation of prediction models based on performance data collected from representative inputs and basic statistical/machine learning techniques. In this deliverable, we extend the *analytical model* presented in D4.1 and develop the *statistical model* for the UC4 BGOs.

3 ANALYTICAL MODELING

The Analytical Modeling was previously presented in D4.1 as a methodology to predict the execution time (in milliseconds) for a given BGO or a workflow of BGOs represented as a Directed Acyclic Graph (DAG) on a specified hardware configuration. In this DAG, nodes represent individual operations, while edges define dependencies between them.

This methodology targets the creation of a closed-form performance model. Such models are created by the BGO developer from source code and lead to a symbolic model expressed as a closed-form equation with parameters related to the input graph. The symbolic model is universally applicable on various platforms where the BGO algorithm can be executed. In this deliverable, we extend the formulation, by generalizing the approach describing the tool developed. The analytical modeling tool operates via an API that allows users to obtain symbolic models of graph operations and evaluate their execution times based on input parameters.
Its key functionalities are:

- **Execution time prediction** for graph operations across different hardware configurations.
- **Symbolic model generation**, including graph properties as variables for analytical performance estimation.
- **API integration**, providing endpoints for accessing calibrated models and evaluating execution times.

3.1 Performance Modeling

Each BGO implementation is associated with a symbolic performance model that expresses execution time as a function of graph properties and hardware characteristics. For example, a calibrated execution time model could be:

- $T_{BGO} = 561*n*924*m + 91*n^2$

where:

- (n) represents the number of nodes in the graph;
- (m) represents the number of edges in the graph;
- (T_{BGO}) is the execution time in milliseconds.

This model is calibrated using micro-benchmarking techniques (presented in D4.1) that measure the performance of atomic operations such as memory reads/writes and arithmetic computations.

3.2 Workflow and Usage

The analytical modeling tool follows a four-step process:

1. **Define Input DAG of BGOs:** Users define a workload as a DAG of BGOs, specifying dependencies between operations. Each BGO is assigned a unique identifier and a list of preceding operations on which it relies. For instance, a Betweenness Centrality operation may serve as an input to a Find Max operation, which, in turn, could feed into a Find Path operation.

2. **Specify Hardware Configuration:** Users provide details about their hardware setup, including processor specifications such as clock speed, number of cores,

and power consumption. Additionally, benchmark values for basic operations (e.g., memory reads and arithmetic operations) are collected through an automated micro-benchmarking process. These benchmarks help calibrate the execution models to reflect actual hardware performance.

3. **Run the Prediction Server:** Users start the prediction server once the workload and hardware configuration are set up. The server processes requests by matching the workload with the calibrated performance models and returns symbolic formulas describing execution time. The models can be obtained by submitting a request to the server, which returns a detailed report of estimated execution times based on the specified hardware characteristics.

4. **(Optional) Specify Graph Characteristics:** To obtain precise execution time predictions for a particular dataset, users can provide specific graph properties such as the number of nodes and edges. The tool then evaluates these properties against the symbolic models to return refined performance estimates tailored to the given input size.

3.3 Execution example

Let's consider the DAG shown in Figure 1, which consists of the following BGOs: betweenness centrality (BC), Breadth-First Search (BFS), Find Max and Find Path.

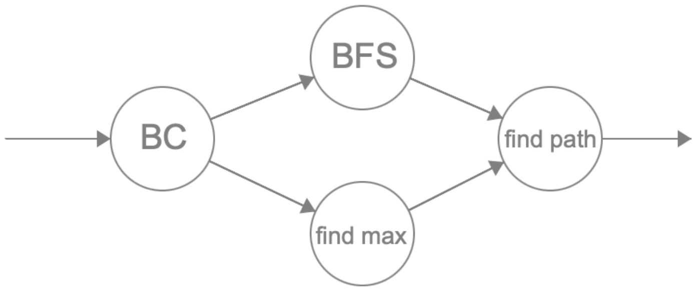

Figure 1: Example DAG of BGO

We define the hardware configurations as follows:

- The system comprises a single host (*host1*) equipped with two Intel Xeon processors. Each processor features 16 cores and 32 threads, operating at a clock speed of 2.10 GHz with a power consumption of 35 watts per unit.
- The CPU's performance is characterised by various benchmark metrics: integer addition takes 2.4 ns, floating-point greater-than comparison executes in 0.8 ns, and queue operations such as push, pop, and front retrieval take 16.1 ns, 11.2 ns, and 14.5 ns, respectively.
- The memory hierarchy performance includes L1 cache reads at 1.26 ns, L2 cache reads at 4.24 ns, L3 cache reads at 20.9 ns, and DRAM reads at 62.5 ns. Additionally, heap operations exhibit times of 52.7 ns for max insertion, 123.3 ns for min extraction, and 12.7 ns for key decrease.
- The system's cache line sizes for L1, L2, and L3 are uniformly 64 bytes, ensuring efficient data handling. Other vector and heap operations, such as push_back, execute within 12 ns, highlighting the processor's efficiency in managing computational workloads.

The input graph is an undirected, unweighted network consisting of 15,763 nodes and 171,206 edges, resulting in an average node degree of 21. The graph has a diameter of 7, indicating that the longest shortest path between any two nodes spans at most seven edges. Despite its sparsity, with a clustering coefficient of 0.0132526, the graph exhibits some local connectivity, forming 591,156 triangles. Additionally, a sample subgraph of 1,000 nodes is considered for analysis.

3.4 Analytical Model Output

The computational workflow is structured as a DAG with four tasks. The process begins with the BC task, which has no dependencies. The two tasks, "Find Max" and BFS, depend on the completion of BC before execution. The final task, "Find Path", relies on both "Find Max" and BFS, indicating a dependency on results from both computations before proceeding. This structured dependency ensures an orderly execution sequence where intermediate results feed into subsequent stages.

The annotated DAG, generated as output by the analytical modeling tool, includes estimated runtime predictions for each task based on the analytical model and the performance of the Intel Xeon processor (host1). The runtime is measured in milliseconds.

Here's the annotated DAG:

```
[
    {
        "id": 0,
        "name": "bc",
        "dependencies": [],
        "performances": [
            {
                "host": "host1",
                "runtime": 23199732.859271962,
            }
        ]
    },
    {
        "id": 1,
        "name": "find_max",
        "dependencies": [
            0
        ],
        "performances": [
            {
                "host": "host1",
                "runtime": 94280.15671874999,
            }
        ]
    },
    {
        "id": 2,
        "name": "bfs",
        "dependencies": [
            0
        ],
        "performances": [
            {
                "host": "host1",
                "runtime": 1354.9894153153125,
            }
        ]
    },
    {
        "id": 3,
        "name": "find_path",
        "dependencies": [
            1,
            2
        ],
        "performances": [
            {
                "host": "host1",
                "runtime": 106.650390625,
            }
        ]
    }
]
```

Since *host1* is the only available system, it is assigned to all tasks in the DAG. If there were multiple systems, an optimizer could suggest the best system for each task, considering user-defined constraints such as performance.

3.5 Validation and outlook

The analytical models are validated by automated benchmarking, a tool devised as part of the Graph-Optimizer toolkit. The validation compares the predicted performance with the measured one and calculates the deviation as a percentage. For example, for the BGOs presented in 3.3, the prediction error is within 3%. All results are included in the project repository here.

Similar predictions are done for communication channels, where we evaluate the speed of data transfers between BGOs. This is done in collaboration with Graph-Greenifier, and results will be presented in the same repository

The following steps are to validate the full workflow over multiple devices and BGOs.

4 STATISTICAL MODELING

In cases where symbolic models are not feasible, such as when AI models are used for specific analyses, creating analytical models becomes challenging due to their multiple components. Instead, in such cases, we use statistical models to collect performance data from representative inputs and employ basic statistical or machine-learning techniques to construct a prediction model.
These models are bound to the system they are created on, but the process of creating them is reproducible on any other machines with limited user intervention. Furthermore, these models do not require separate calibration, as system-specific characteristics are inherently captured within the collected data.

Among the various use case BGOs in the Graph-Massivizer project, we will focus on two BGOs from the UC-4, which are based on the following basic graph operations: (1) Graph Neural Networks (GNN) inference for anomaly prediction models and (2) SPARQL queries on the UC-4 KG.

The statistical modeling tool's key functionalities are:

- **Systematic design-space exploration** comprises the following sub-steps:
    a. defining the domain of the BGO configuration;
    b. executing the BGO under various configurations;
    c. analyzing execution times across different configurations to identify correlations between parameters and execution time and the creation of a dataset consisting of multiple executions of the same BGO in the target HW while varying its internal configurations.
- **Statistical model fitting/training** includes (i) the selection of the basic machine learning tool to model the relation between the performance data and the BGO's configuration, as well as (ii) the training of the machine learning model weights.
- **Performance Prediction and validation** consist of the machine learning model inference to estimate the performance of a given BGO and assessment of the final accuracy.

4.1 UC-4 Graph Neural Network Inference

This BGO consists of a Graph Neural Network Inference (GNN). The topology of the neural network targets the UC-4 need and has been trained to predict compute nodes unavailability.

The unavailability of compute nodes is a critical issue for data centers, affecting the reliability and performance of high-performance computing (HPC) systems. Such unavailability, caused by hardware failures, software malfunctions, or maintenance, leads to disruptions, delays, and reduced system efficiency. It also increases operational costs due to job rescheduling, workload redistribution, and additional resource usage. These events are classified as anomalies deviating from expected operational norms.

The UC-4 anomaly prediction GNN model is illustrated in Figure 1. The architecture of the GNN model has 3 Graph Convolutional Network (GCN) layers, followed by 2 fully connected dense layers. The final architecture is as follows:

- GCNConv (input channels = 417, output channels = 300)
- GCNConv (input channels = 300, output channels = 100)
- GCNConv (input channels = 100, output channels = 16)
- Fully connected dense layer (input channels = 16, output channels = 16)
- Fully connected dense layer (input channels = 16, output channels = 1)

The final layer gives the probability value of an anomaly for a compute node.

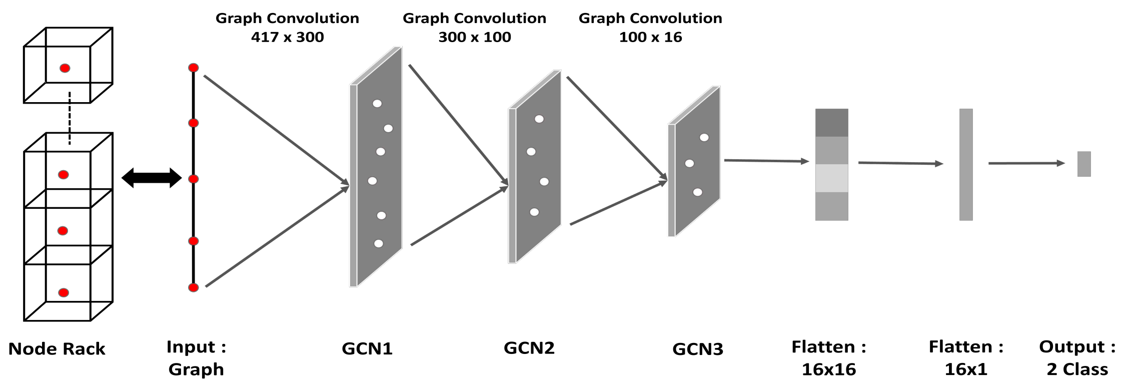
Graph Convolution 417 x 300
Graph Convolution 300 x 100
Graph Convolution 100 x 16
Node Rack
Input : Graph
GCN1
GCN2
GCN3
Flatten : 16x16
Flatten : 16x1
Output : 2 Class

Figure 2: Architecture of the GNN anomaly prediction model.

In UC-4, different input-size GNN models have been proposed, showing different performance/accuracy tradeoffs. The first model that was tested was a per-rack model, where the inputs are the features for all the nodes in one rack. The output of this model is the anomaly predicted class for all the nodes in the rack. The second model which has been tested is a room-level model where inputs are all the node's features. The output of this model is the anomaly predicted class for all the nodes in the room.

In this section, we explore the impact of various hyperparameters on the execution time of the GNN inference models. The primary goal is to develop a predictive model that estimates execution time based on model configurations. The GNN models are characterized by key hyperparameters such as the number of input nodes (*input_size*), the number of layers (*number_of_layers*), and the convolution kernel size (*kernel_size*). These parameters play a crucial role in determining the computational complexity and performance of the model.

To better understand the relationship between the model parameters and execution time, we systematically explore various configurations. Using a grid of parameters, we create a set of GNN models with different combinations of *input_size*, *number_of_layers*, and *kernel_size*. We then perform inference with these models, recording the time taken for its execution. This dataset, consisting of configuration-inference time pairs, serves as the foundation for training a predictive model that can estimate execution time for future configurations.

4.1.1 Experimental Setting

The parameter grid consists of three sets of values:

- Number of input features (*input_size*): [4, 6, 12, 24, 32, 64, 128, 256, 417, 512, 750, 1000].
- Number of GCN layers (*number_of_layers*): [1, 2, 3, 4, 5, 6].
- Kernel size (*kernel_sizes*): [4, 8, 16, 24, 32, 64, 128].

Using this grid of parameters, we reach 504 different combinations of GNN models to be used to train predictive models. Furthermore, we will evaluate the GNN inference time on both CPU and GPU settings.

The experiments are conducted on a server featuring an Intel Xeon E5-2630 v3 CPU with an x86_64 architecture. The processor comprises 16 physical cores across two sockets, with a 256 KiB L1 data cache, a 256 KiB L1 instruction cache, a 2 MiB L2 cache, and a 20 MiB L3 cache. It operates at a base frequency of 2.40 GHz, with a maximum frequency of 3.20 GHz. The server also includes an NVIDIA Quadro RTX 6000 GPU, running CUDA 12.1 and PyTorch 2.4.0+cu121.

We tested two types of predictive models:

- Linear Regressor (LR): A basic regression model used for prediction.
- Neural Network (NN): A model with three fully connected layers. The architecture is as follows:
    - The first layer transforms the 3 input channels into 64 output channels.
    - The second layer reduces the output to 32 channels.
    - The final layer outputs a single value, representing the predicted execution time.

The NN model is trained using the Adam optimizer with a learning rate of 0.01 and mean-squared-error (MSE) loss. The data is split 80:20 for training and testing.

4.1.2 Performance on CPU

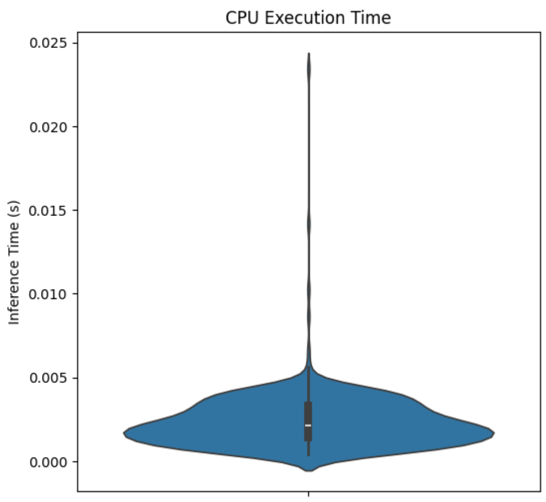

Figure 3: Execution time of GNN models on CPU.

The average CPU execution time is 0.00235s, with a standard deviation of 0.00646s. The minimum recorded time is 0.0004s, while the maximum is significantly higher at 0.023421s, highlighting the presence of occasional outliers. The 25th, 50th (median), and 75th percentiles are 0.001398s, 0.002113s, and 0.003309s, respectively, suggesting that most inference times fall within a narrow range, but the occasional extreme values increase the spread. These are presented in Figure 2 as a violin plot.

Figure 3 presents three scatter plots, each illustrating the relationship between CPU execution time and key parameters: *input_size*, *number_of_layers*, and *kernel_size*. The plots reveal that variations in these parameters do not significantly impact the execution time for GNN inference on the CPU, with most execution times clustering below 0.005s. A slight increase in execution time is observed as the number of layers grows; however, the values remain consistently below 0.005s, reaffirming the limited influence of these parameters.

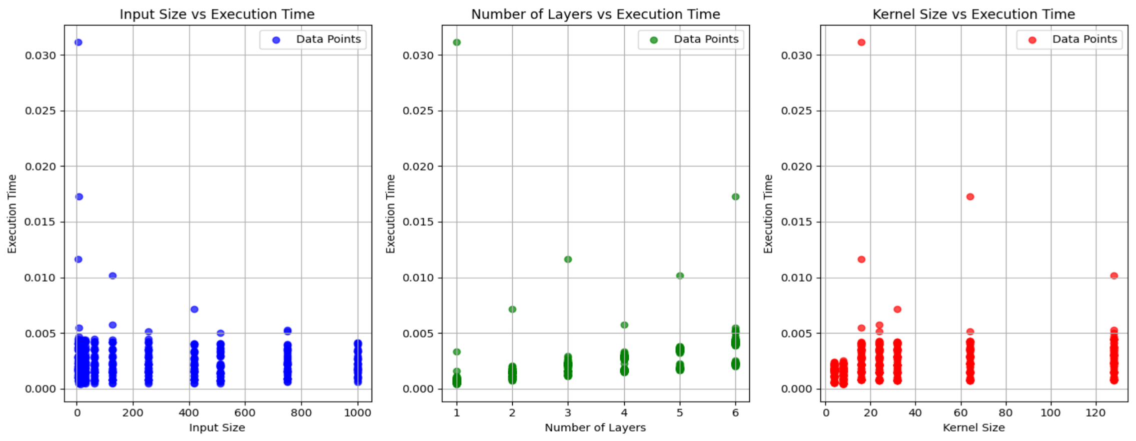

Figure 4: Scatter plots showing the relationship between input size, number of layers, kernel size, and execution time on CPU.

4.1.3 Performance on GPU

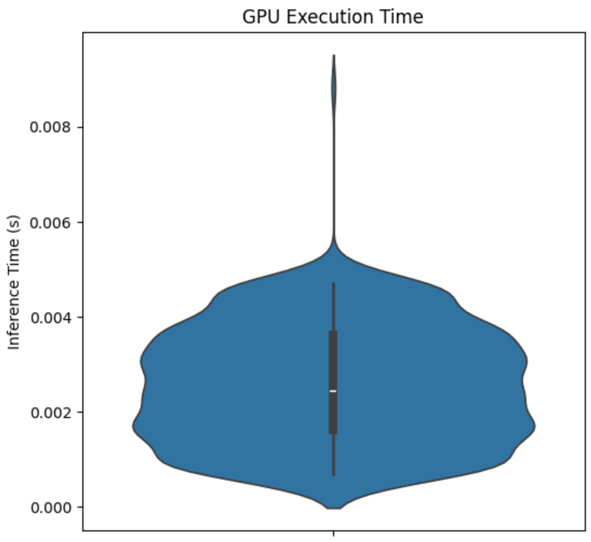

Figure 5: Execution time of GNN models on GPU.

Figure 4 shows the spread of inference time on GPU as a violin plot. The data reveals a slightly higher average inference time of 0.002577s compared to the CPU, but with a lower standard deviation of 0.001217s, demonstrating more consistent performance. The minimum and maximum times are 0.000679s and 0.00881s, respectively, showing a narrower range than the CPU. The 25th, 50th, and 75th percentiles are 0.001607s, 0.002435s, and 0.003618s, indicating a tight clustering of execution times.

From the scatter plots in Figure 5, we observe the correlation between execution time and the three key parameters. Similar to the CPU results, variations in these parameters do not significantly influence execution time, except for the number of layers. As the number of layers increases, we notice a slight increase in execution time, consistent with the CPU observations.

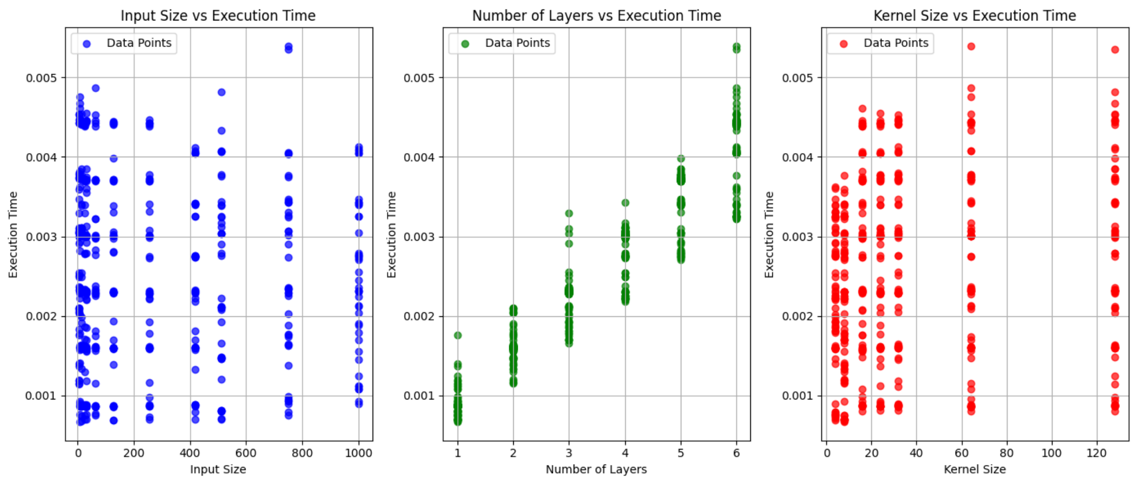

Figure 6: Scatter plots showing the relationship between input size, number of layers, kernel size, and execution time on GPU.

4.1.4 Execution Time Predictor

Since, we have execution time for both CPU and GPU, we have trained separate predictive models for each. The metric used to evaluate the prediction performance of these models on the test set is mean-squared error (MSE).

The final equation for the LR model for CPU after training is stated below:

$$y = -0.00019 \cdot x_1 + 0.00109 \cdot x_2 + 0.00022 \cdot x_3 + 0.00251$$

Similarly, the final equation for the LR model for GPU after training is stated below:

$$y = -0.00004 \cdot x_1 + 0.00115 \cdot x_2 + 0.00014 \cdot x_3 + 0.00259$$

$$where\ \ x_1\ is\ input\ size,\ \ x_2\ is\ number\ of\ layers, and\ x_3\ is\ kernel\ sizes.$$

Table 1: MSE of LR and NN models for CPU and GPU.

Model	**MSE**
LR-CPU	0.000011
NN-CPU	0.000014
LR-GPU	0
NN-GPU	0.000002

Table 1 presents the evaluation of MSE for two different models—LR model and NN model for both CPU and GPU configurations. For LR-CPU model, the MSE is 0.000011, indicating a very low error, and the MSE is 0 of the NN-CPU, showing perfect accuracy. For NN-CPU, the MSE is 0.000014, slightly higher than that of LR-CPU, while for the NN-GPU, the MSE is 0.000002. In both settings, the LR model performs much better than the NN model, and even reaches a perfect accuracy for the LR-GPU model.

4.2 UC-4 Complex queries

This BGO targets the execution of complex SPARQL queries, which are essential in UC-4 data center facilities for deriving actionable insights and enabling proactive decision-making.

To build the statistical model we first benchmark the execution time of complex query implementations using the Graph-Massivizer approach, which employs SPARQL queries on the UC-4 Knowledge Graph (KG). To evaluate this, we used the ten archetypes of complex queries identified in UC-4 through a survey conducted among end users, including system administrators, user support managers, and facility managers. These archetypes progressively increase in complexity, as detailed in Table 2

Table 2: Ten query archetypes for a data center facility.

No.	Complex query archetypes
1	Give me all the nodes present in rack 1
2	Give me a list of all the racks
3	Give me the position of node 3
4	Give me the list of plugins
5	What nodes were used by the job 1001282?
6	What is the average power used by the job 1001282?
7	How many jobs are running on node 5 during the month of May 2022?
8	What is the min, max and avg temperature of node 178 when it is in use during the month of May 2022?
9	Give me a list of sensors which are of type "power"
10	Give me a list of the jobs running and the nodes they used during the month of May 2022

The UC-4 KG is constructed using M100 data from the Marconi100 facility at CINECA, collected by Borghesi et al. [5]. This KG adheres to the structure defined by the UC-4 ontology, as outlined in deliverable D6.1 [4] and is stored in a graph database.

For our evaluation, we have measured the performance of SPARQL queries executed on the UC-4 KG using the following collected metrics:

1. **Query execution time:** Measured as both CPU time and wall time.
2. **Number of data points fetched:** Total data points retrieved by the query.
3. **Code conciseness:** Quantified by the number of Lines of Code (LOC).
4. **Entities used:** The number of entities referenced in the query code.
5. **Object and data properties of KG used:** This refers to the count of objects and data properties utilized in the query. An object property represents the relationships or connections between different entities in the KG, while a data property defines the associated data types or attributes of an entity.
6. **Presence of filter conditions:** Whether the query includes filter clauses, which increase complexity by restricting the data fetched according to specific conditions.
7. **Aggregation operations (AGG OP):** Any aggregation performed for the final output, such as maximum, minimum or average.

Table 3: Query performance metrics, including CPU Time, Wall Time, Data Points fetched, Lines of code (LOC), Entities and Properties used, Filter conditions, and Aggregation Operations (OPP).

No.	CPU Time	Wall Time	Data points fetched	L O C	Entites used	Properties used	Filter	AGG OP
1	15.6ms	463.83ms	20	5	2	2	0	0
2	15.6ma	485.5ms	49	4	1	0	0	0
3	15.6ms	434.17ms	3	8	2	5	0	0
4	15.6ms	636ms	980	4	1	0	0	0
5	15.6ms	503.67ms	1	7	1	2	0	0
6	15.6ms	373.33ms	476	18	5	11	2	1
7	15.6ms	416.17ms	80	11	2	4	2	1
8	15.6ms	380.33ms	1,070	16	5	9	0	3
9	62.5ms	660.5ms	9,800	7	1	1	0	0
10	3.64s	10.35s	1,206,594	11	2	4	2	0

Table 3 reports the collected metrics for the ten query archetypes. From the table, we can see that the CPU Time and Wall Time are different, and this can be explained by the fact that in complex queries, a significant portion of the Wall Time depends on disk access time. Furthermore, there is not a strong correlation between the Wall Time and CPU Time with the input features, which seems strongly not linear.

4.2.1 Execution Time Predictor

To develop a predictive model for the execution time of complex queries on the UC-4 KG, we adopt a machine learning approach using multiple models: Linear Regression (LR), Random Forest (RF), Gradient Boosting (GB), Support Vector Machine (SVM), and K-Nearest Neighbors (KNN). Given the limited number of complex query archetypes, we train the model on these archetypes using an 80:20 split for training and testing. While the limited training data may impact the model's generalizability, stakeholders at the data center are expected to implement queries that are variants of these archetypes. Therefore, the model can serve as a reliable estimator of execution time for these scenarios. To ensure robustness despite limited data, we will evaluate the model using cross-validation and Mean-Absolute-Error (MAE) to validate its predictive performance.

Table 4 presents the evaluation of ML models for CPU time prediction, using cross-validation. The LR model yielded cross-validation scores of 0.0235, 0.1362, 9.7727, 3.2892, and 0.0568, resulting in an average MAE of 2.6557. The RF model demonstrated significantly lower error with scores of 0.0235, 0.0015, 0.0002, 0.0203, and 0.0076, achieving an average MAE of 0.0106. GB showed comparable performance with scores of 0.0235, 1.1601e-07, 1.1601e-07, 0.0294, and 1.1601e-07, resulting in an average MAE of 0.0106. The SVM model provided cross-validation scores of 0.0294, 0.0175, 0.0175, 0.0294, and 0.0175, with an average MAE of 0.0223. Finally, the KNN model performed similarly with scores of 0.0235, 0.0059, 0.0094, 0.0154, and 0.0094, achieving an average MAE of 0.0127. The RF and GB model consistently outperformed the other approaches in terms of the lowest average MAE, suggesting their stronger predictive performance for CPU time.

Table 4: K-Fold Cross-Validation CPU Time for different models, with individual fold times and the average CPU time for each model.

Model	K-Fold Cross-Validation MAE Scores					Average MAE
	1	**2**	**3**	**4**	**5**	
LR	0.0235	0.1362	9.7727	3.2892	0.0568	2.6557
RF	0.0235	0.0015	0.0002	0.0203	0.0076	0.0106
GB	0.0235	1.1601E-07	1.1601E-07	0.0294	1.1601E-07	0.0106
SVM	0.0294	0.0175	0.0175	0.0294	0.0175	0.0223
KNN	0.0235	0.0059	0.0094	0.0154	0.0094	0.0127

Presented in Table 5 are the evaluation results of machine learning models for wall time prediction using cross-validation. The LR model yielded cross-validation scores of 0.0520, 0.1127, 7.8276, 2.6817, and 0.1488, resulting in an average MAE of 2.1646. The RF model demonstrated more variable performance with scores of 0.0890, 0.8209, 0.4276, 4.9204, and 0.7673, yielding an average MAE of 1.4050. The GB model showed better performance with scores of 0.0830, 0.0107, 0.1253, 4.9827, and 0.0817, achieving an average MAE of 1.0567. The SVM model had cross-validation scores of 0.1021, 0.2279, 0.2998, 4.9482, and 0.2053, resulting in an average MAE of 1.1567. The KNN model performed with scores of 0.0875, 1.0618, 0.1095, 4.9301, and 0.1099, yielding an average MAE of 1.2598. The GB model showed the best performance with the lowest average MAE.

Table 5: K-Fold Cross-Validation Wall Time for different models, with individual fold times and the average Wall time for each model.

Model	K-Fold Cross-Validation MAE Scores					Average MAE
	1	**2**	**3**	**4**	**5**	
LR	0.0520	0.1127	7.8276	2.6817	0.1488	2.1646
RF	0.0889	0.8209	0.4276	4.9204	0.7673	1.405
GB	0.0830	0.0107	0.1252	4.9827	0.0817	1.0567
SVM	0.1021	0.2279	0.2998	4.9482	0.2053	1.1567
KNN	0.0875	1.0618	0.1095	4.9300	0.1099	1.2598

5 SUMMARY AND FUTURE WORK

This deliverable presents two different approaches that have been implemented in the project to predict the graph workload execution time. We extended and generalized the analytical model from D4.1 with a statistical approach which is more suitable for complex BGOs. The analytical model has been implemented as a tool, and it is fully functional. Future works will extend the validation of the analytical models on the different BGOs which have been collected in the Graph-Massivizer project repository. The statistical model approach has been presented and applied to the UC-4 BGOs. Future works will extend the statistical model, validating it on other use-cases BGOs.

Our results show that predictive models can be learned with high accuracy; however, the choice of model template (analytical, neural network or linear regression) depends on the BGO itself.

6 REFERENCES

[1] Dizaji, Haleh, et al. “D2.1: Graph-Massivizer Requirements Elicitation and First Architecture Design.” Graph-Massivizer, 2023, https://graph-massivizer.eu/.

[2] Elvesæter, Brian, et at. “D3.1: Massive Graph Inception Architecture and Data Management.” Deliverable, Graph-Massivizer (2023), https://graph-massivizer.eu/.

[3] Varbanescu, Ana Lucia, et at. “D4.1: BGO and composition rule for heterogeneous infrastructure.” Deliverable, Graph-Massivizer (2024), https://graph-massivizer.eu/.

[4] Eberhart, Aaron, et at. “D4.1: Graph-Massivizer Infrastructure, Tools, and Use Cases.” Deliverable, Graph-Massivizer (2023), https://graph-massivizer.eu/.

[5] Borghesi, Andrea, et al. “M100 ExaData: a data collection campaign on the CINECA’s Marconi100 Tier-0 supercomputer.” Scientific Data 10.1 (2023): 288.

7 ACRONYMS

Acronym	Description
BGOs	Basic Graph Operations
GPU	Graphics Processing Unit
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
BFS	Breadth First Traversal
BC	Betweenness Centrality
SSSP	Single source shortest path
HPC	High performance computing
GNN	Graph Neural Networks
GCN	Graph Convolutional Networks
NN	Neural Network
MSE	Mean-Squared-Error
LR	Logistic Regression
KG	Knowledge Graph
LOC	Lines of Code
AGG OP	Aggregation Operations
RF	Random Forest
GB	Gradient Boosting
SVM	Support Vector Machine
KNN	K-Nearest Neighbors
MAE	Mean-Absolute-Error