GRAPH MASSIVIZER

# D4.1: BGO AND COMPOSITION RULES FOR HETEROGENEOUS INFRASTRUCTURE

This project has received funding from the European Union´s Horizon Research and Innovation Actions under Grant Agreement n° 101093202.

| Title | Document Version |
|---|---|
| BGO and composition rules repository for heterogeneous devices | 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* |
|---|---|---|
| M15 (March 2024) | M17 (May 2024) | R-PU |

| Responsible | Organisation | Contributing WP |
|---|---|---|
| Ana-Lucia Varbanescu | UTW | WP4 |


| Authors | Organisation |
|---|---|
| Ana-Lucia Varbanescu, Duncan Bart, Kuan-Hsun Chen | UTW |
| Radu Prodan | KLU |
| Andrea Bartolini, Martin Molan, Junaid Ahmed Khan | UNIBO |
| Jože Rožanec | JSI |

## 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 presents the current status of the BGO repository, the modeling principle for hardware and BGO software, and examples of performance and energy consumption models that Graph-Optimizer will use to predict overall workload performance. The **draft** BGO repository is present on Github, and it will be continuously updated with new BGOs.



## Keywords

BGO optimization, BGO implementation, BGO models, high-performance, energy-efficiency, models calibration, microbenchmarking

# REVISION HISTORY

| Revision | Date | Description | Author (Organisation) |
|---|---|---|---|
| v0.1 | 15.01.2024 | First version of the document. | Ana Lucia Varbanescu (UTW) |
| v0.2 | 01.02.2024 | First BGO description. | Ana Lucia Varbanescu (UTW) |
| v0.3 | 01.04.2024 | BGO code, benchmarking | Duncan Bart (UTW) |
| v0.4 | 08.04.2024 | Model communication | Duncan Bart (UTW) |
| v0.5 | 03.05.2024 | Finalized draft for reviewing | Ana Lucia Varbanescu (UTW) |
| v0.6 | 05.05.2024 | Revision 1 | Andrea Bartolini (UNIBO) |
| v0.7 | 06.05.2024 | Revision 2 | Jože Rožanec (IJS) |
| v0.8 | 14.05.2024 | Final consolidation and editing. | Ana Lucia Varbanescu (UTW) |
| v1.0 | 15.05.2024 | 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 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 code based on the selected workflow.

This report presents the architecture and current state of the basic graph components (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**, 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**, 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 D6.1**, 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, 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. 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 presents the platforms we use for Graph-Optimizer, while Section 5 presents the current list of BGOs, following a similar structure for each. The detailed platform descriptions, BGO code, models, and benchmarking data are all available in our repository.

Section 6 describes the composition rules and models for combining BGOs. Specifically, we discuss the conceptual ideas behind BGO composition in workflows, the different types of compositions, and the models we use for each.

Section 7 concludes this report with a summary of the current BGO status and an outline of 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). 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). In this deliverable, we present concrete examples of such models, with the comprehensive list constantly updated in the repository on GitHub.

Finally, the models are *calibrated* and *validated* using benchmarking. We differentiate between *microbenchmarking*, which focuses on measuring the performance of the 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 BGOs in operation, thus enabling model validation.

# 3 METHODS AND TOOLS

## 3.1 Execution

All BGOs can be executed stand-alone or in workflows. The only requirement for execution is for the data to be available and provided in the right format (in-memory, by message-passing, per-file).

All BGOs can be tested for correctness against a basic reference implementation (CPU-based, sequential).

All BGOs can be benchmarked for performance in terms of execution time and energy consumption. The performance data is reported using the following metrics:

- wall-clock time
- computation and data movement time (where needed)
- throughput, using graph- and BGO-specific metrics (like, for example, TEPS for traversal algorithms).
- energy consumption
- energy efficiency, using graph- and BGO-specific metrics (like, for example, Te/W or FLOPS/W).

We use high-resolution timers for performance measurement and propose a simple benchmarking framework (see Section 3.2) for collecting and storing the data. For energy consumption we use either integrated counters (i.e., using tools like PAPI or LIKWID) or estimates based on power measurements.

## 3.2 Benchmarking

We benchmark the performance of the proposed implementation at two levels: microbenchmarking and BGO benchmarking.

### 3.2.1 Microbenchmarking

We use microbenchmarking for model calibration. Therefore, we focus on basic computing operations - e.g., additions or multiplications - and memory operations - e.g., memory read and write operations. Intuitively, the performance and energy consumption per operation are determined by measuring the execution time taken by a sufficiently large number of repetitions of identical operations and calculating an average execution time as the ratio between the measured time and the number of operations. Care needs to be taken that the additional operations - beyond the targeted ones - are kept at the minimum: enough to prevent compilers from optimizing the code out, but not sufficiently many to impact the microbenchmarking data.

Additionally, for parallel CPU and GPU code, synchronization primitives and communication primitives - the so-called H2D (host-to-device) and D2H (device-to-host) data transfer times must be microbenchmarked.

For microbenchmarking, we use our own code, data, and code from additional sources, like uops.info or the microbenchmarks from LIKWID for CPUs, and the GPU performance tools here.

### 3.2.2 BGO benchmarking

BGO benchmarking is primarily used for model validation. This entails the actual performance measurement for the BGOs using a diverse set of graphs - real (from repositories and/or use-cases) or synthetic - and recording the same two basic metrics: execution time and energy consumption. Derived metrics like TEPS or TEPS/W can be further calculated and reported from these metrics.

We are automating the BGO measurement for data management purposes, i.e., for collecting and managing the performance data. For each BGO, we include in the repository the graphs (as actual data for the small graphs and as links for the larger graphs) we used for validation and the results of the benchmarking operations.

## 3.3 Modeling

We employ two types of performance models: analytical models and statistical models.

### 3.3.1 Analytical models

Analytical models capture the functional behavior 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 (see section 5.1 for an example). The symbolic model is universally applicable on various platforms where the BGO algorithm can be executed.

However, the symbolic model is only partially useful for performance prediction and/or selection of best-performing BGO. For such analysis, model calibration is needed. We calibrate all models using microbenchmarks, which effectively "personalize" the symbolic models with the actual observed values for the system where the BGO is running.

### 3.3.2 Statistical models

There are cases when symbolic models are not feasible. For example, where AI models are used for certain analyses, creating analytical models is difficult due to the multiple components of such AI models. Instead, in such cases, we use statistical models to collect performance data from representative inputs, and, using basic statistical/machine learning techniques, we create 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 specifics are part of the data being collected. However, we do require BGO benchmarking and representative input data, to create data for creating such models. The BGO benchmarking leverages the same benchmarking tools as the BGOs, while the representative input data creation/selection is left to the BGO developer.

# 4 PLATFORMS

This section presents the platforms under consideration for measuring BGO performance. Like many other aspects of this deliverable, this live platform will include more platforms as the project develops further BGOs and use cases. For each one of these platforms, we present the basic architecture features.

| Platform ID | Type | Machine/ Location | Architectural features |
|---|---|---|---|
| DAS5-CPU<br>DAS5-NGPU-<cc,m[1]> | CPU, GPU | DAS5 (VU) | The DAS5 cluster is an older generation heterogeneous machine, featuring 68 nodes, some of which are also GPU-accelerated.<br>The CPUs are Intel Xeon machines, using a dual 8-core CPU running at 2.4 GHz, and 64 GB of RAM.<br>The accelerators are NVIDIA GPUs (NGPU) from various generations, including TitanX, Titan, GTX980, K20, and K40. |
| DAS6-CPU<br>DAS6-NGPU-<cc,m> | CPU, GPU | DAS6 (VU) | The DAS6 cluster features 32 single-socket 24-core nodes alongside 2 dual 24-core fat nodes. All 34 nodes have 128GB of RAM. 10 have an NVIDIA RTX A4000 GPU, and 4 have a multi-GPU configuration with 2-4 RTX 4000 GPUs. 3 nodes have an NVIDIA RTX A6000 GPU, one of which is a fat node. The second fat node has an NVIDIA A100 GPU. 4 nodes have an NVIDIA RTX A2 GPU, and one node has 4 NVIDIA RTX A2 GPUs. |
| ANT-CPU<br>ANT-NGPU-<cc,m> | CPU, GPU | ANTON (UTW) | Anton is the heterogeneous computing server of the CAES group at the University of Twente. It contains 2x Intel Xeon Silver 4216 2.1 GHz 16-core processors and 64GB of RAM. The accelerators are an NVIDIA RTX 3080 Ti, a Xilinx Alveo U55C, and a Xilinx VCK5000 Versal Development Card. |

[1] The notation "cc,m" stands for "computing capability" and "model", and will uniquely identify the GPU model and family where the data is collected.

| | | | |
|---|---|---|---|
| MC-RISCV | CPU | Monte Cimone (UNIBO) | The Monte Cimone compute system consists of 8 compute nodes based on the unmatched U740 SoC from SiFive (4 cores and 8 GB DRAM ) and three compute nodes consisting of the MILk-V Pioneer box system which features a SOPHON SG2042 processor w.64 cores, 128 GB of DDR4 memory and two PCIe4 slots. The MILk-v Pioneer system is accelerated with a set of RISC-V-based PCIe accelerators which consists of Esperanto Et-soc-1, Axellera.ai Metis, Tenstorrent Grayskull, and an Alveo u55c used to emulate RISC-V open-HW accelerations. Being prototype accelerators that are not yet supported on RISC-V hosts, their current usage has limitations. |
| Leo-CPU<br>Leo-NGPU-<cc,m> | CPU, GPU | Leonardo (CINECA) | Two main computing modules:<br><br>The booster module comprises 3456 computing nodes, each with four NVidia A100 SXM6 64GB GPUs and 32-core Intel Ice Lake CPU. Computational performance over 240 Pflops.<br><br>A data-centric module comprises 1536 nodes equipped with two Intel Sapphire Rapids CPUs, each with a core count over 50. Capable of 9 PFlops of sustained performance. |

Table 1. Target platforms for BGOs

The performance data included in this deliverable is extracted using Anton and DAS6. However, we expect all these machines to be used for (some of) the BGOs developed in the project. Therefore, all performance data included in the deliverables and the repository will be annotated with the platform ID where the data was collected.

# 5 BASIC GRAPH OPERATIONS

This section presents the high-level description of the BGOs from UC-0, as available in our repository on May 1st, 2024 (v1.0). For each BGO, we follow a similar structure, where we describe its goal, the basic algorithm, the API, and the current implementations. supported platforms, benchmarking data, and performance models (where available/applicable). We start this presentation with the complete example of BFS in Section 5.1, we continue with the design and implementation of the remaining UC-0 BGOs in Sections 5.2-5.5 and include two examples of more complex BGOs from UC-4 in Sections 5.6-5.7.

## 5.1 BREADTH-FIRST SEARCH (BFS)

### 5.1.1 Design

Breadth-first search is an algorithm that calculates the minimal number of hops it takes to get from a source node to any other node by traversing the graph layer by layer. The basic BFS algorithm is listed in Listing 1.

```
1 procedure BFS(graph, source):
2     Q <- new queue
3     visited[source] = true
4     Q.push(source)
5     while !Q.isEmpty() do
6           v <- Q.pop()
7           for node in graph.neighbours(v) do
8                 if !visited[node] then
9                       visited[node] = true
10                      parent[node] = v
11                      level[node] = level[v] + 1
12                      Q.push(node)
13    return parent, level
```

Listing 1: Pseudocode for BFS.

The input for BFS is a directed or undirected graph, along with a source node from which the search is initiated. The output is a list of the distance from the source to each node, and a second list for each node's parent, which can be used for finding the shortest path between two nodes.

### 5.1.2 Available implementations

We have two different implementations of this algorithm, as listed in Table 2.

| Version | Platform | Features | Code | Runtime | Model |
|---|---|---|---|---|---|
| V0 | CPU | Seq | GitHub | $171.429n+3.36897n^2$ | GitHub |
| V1 | CPU | LAGraph | LAGraph GitHub | - | - |

Table 2 BFS implementations

### 5.1.3 Benchmarking

We benchmark the proposed version on Anton (ANT-CPU in Table 1, Section 4), a local server at the University of Twente, using the input graphs presented in Table 3, stored in the MatrixMarket (mtx) format and available in our repository.

| Graph Name | #Nodes | #Edges | Source |
|---|---|---|---|
| as20000102 | 6474 | 26467 | GitHub |
| chess | 7115 | 111558 | GitHub |
| librec-ciaodvd-trust | 4562 | 66116 | GitHub |
| maayan-vidal | 2783 | 12445 | GitHub |
| moreno_blogs | 1222 | 33431 | GitHub |
| moreno_innovation | 117 | 930 | GitHub |
| wikilens-ratings | 5111 | 53709 | GitHub |

Table 3. Datasets used for BGO benchmarking

These graphs are selected from the KONECT Project [10] but have been modified to only contain the single largest connected component. This was done to increase the applicability to UC0, which also contains a single connected component, and thus to increase the accuracy of the models. The results of the benchmarking are recorded in a .csv file on GitHub. A snapshot of the data is included in Table 4.

| | Execution time | |
|---|---|---|
| Dataset | Measured [ms] | Predicted [ms] |
| as20000102_largest_cc | 134.607 | 142.312 |
| librec-ciaodvd-trust_largest_cc | 71.800 | 70.897 |
| chess_largest_cc | 167.599 | 171.7687 |
| moreno_blogs_blogs_largest_cc | 5.561 | 5.240 |
| maayan-vidal_largest_cc | 25.550 | 26.570 |
| moreno_innovation_innovation_largest_cc | 0.279 | 0.066 |
| wikilens-ratings_largest_cc | 85.671 | 88.881 |

Table 4. Predicted and measured performance for the BFS V0 BGO on ANT-CPU.

### 5.1.4 Models and validation

This section lists a short overview of the analytical models for BFS and their accuracies. The full models and calibration are presented in GitHub.

For the naive sequential version, we derived the symbolical model presented in Figure 1.

$$
\begin{aligned}
T_{BFS} &= nT_{init} + nT_{while\_loop} + nT_{visit\_node} \\
T_{init} &= 2T_{mem\_write} \\
T_{while\_loop} &= T_{q\_front} + T_{q\_pop} + n\left(T_{add} + T_{G\_mem\_read} + T_{mem\_read}\right) \\
T_{visit\_node} &= 2T_{DRAM} + T_{L1} + T_{add} + T_{q\_push} \\
T_{mem\_write} &= T_{mem\_read} = (1 - MR_{L1})T_{L1} + MR_{L1}(1 - MR_{L2})T_{L2} + \\
&\quad MR_{L1}MR_{L2}(1 - MR_{L3})T_{L3} + MR_{L1}MR_{L2}MR_{L3}T_{DRAM} \\
MR_{L\{1,2,3\}} &= \frac{1}{cacheLineSize_{L\{x\}}/sizeof(\texttt{int})} \\
T_{G\_mem\_read} &= (1 - G\_MR_{L1})T_{L1} + G\_MR_{L1}(1 - G\_MR_{L2})T_{L2} + \\
&\quad G\_MR_{L1}G\_MR_{L2}(1 - G\_MR_{L3})T_{L3} + \\
&\quad G\_MR_{L1}G\_MR_{L2}G\_MR_{L3}T_{DRAM} \\
G\_MR_{L\{1,2,3\}} &= \frac{1}{cacheLineSize_{L\{x\}}/sizeof(\texttt{float})}
\end{aligned}
$$

Figure 1. Analytical model for the BFS BGO V0.

After calibration on ANT-CPU (see Table 1), this symbolical model is simplified to:

$$171.429n + 3.36897n^2$$

In Figure 2, we plot the predicted execution times for the graphs listed in section 5.3.2 alongside the benchmarked values, with a regular y-axis and a logarithmic y-axis to better visualize the execution times of the smallest graphs.

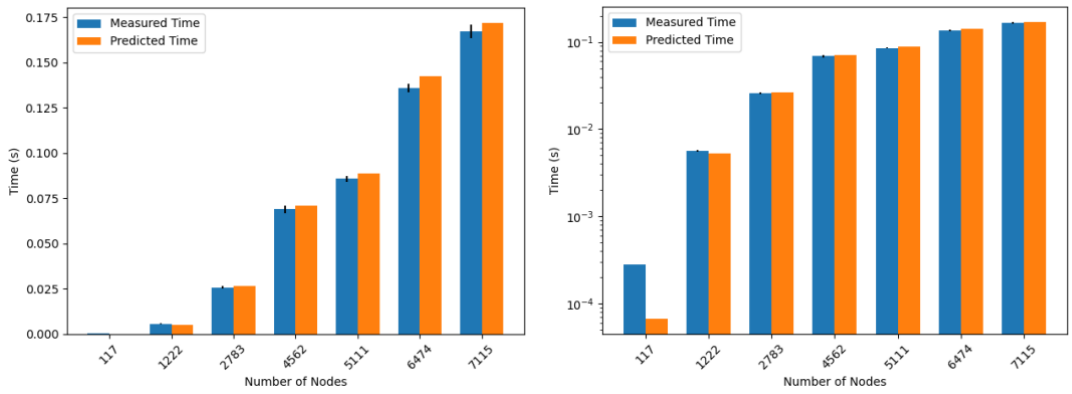


Figure 2. Prediction vs. measurement execution time for BFS.

From these figures, we note that the accuracy seems to increase as the graphs get larger, and stabilises around a 96% accuracy.

## 5.2 BETWEENNESS CENTRALITY (BC)

### 5.2.1 Basic design

Betweenness centrality is an algorithm that calculates the importance of a vertex in a graph by calculating how many critical paths (out of all critical paths) include this vertex. The basic algorithm is presented in the following pseudocode. One of the versions we include is this basic, sequential algorithm:

```
1 procedure BC(graph, sources):
2   ASSP <- [sources.count()]
3   for i in sources.count() do
4     ASSP[i] = sssp(graph, sources[i])
5
6   for s in sources.count() do
7     for t in sources.count() do
8       if sources[i] != sources[t] do
9         ds = ASSP[s]
10        dt = ASSP[t]
11        for n in graph.nodes() do
12          if n != sources[s] &&
13             n != sources[t] &&
14             ds[n].dist + dt[n].dist == ds[sources[t]].dist &&
15             ds[sources[t]].num_paths > 0 do
16            BC[n] += ds[v].num_paths * dt[v].num_paths /
17                     ds[sources[t]].num_paths
18   return BC
```

Listing 2: Pseudocode for BC.

BC receives as input an undirected graph and a list of source nodes, and provides a list of centrality scores for each vertex. The list of source nodes is used to calculate the critical paths from, and serves as a means of limiting the execution time.

### 5.2.2 Available implementations

We have implemented the following versions of BC:

| Version | Platform | Features | Code |
|---|---|---|---|
| V0 | CPU | Naive | GitHub |
| V1 | CPU | Brandes | GitHub |
| V2 | CPU | GraphBLAS | GitHub |
| V3 | CPU | LAGraph | LAGraph GitHub |

Table 5. BC implementations

*At the time of deliverable submission, we are modeling and benchmarking these implementations. The data will be added to the repository.*

## 5.3 FIND MAX

### 5.3.1 Design

Find max is a simple algorithm for finding the index of the maximum value in a list or array. We use the following pseudocode:

```
1 procedure find_max(list):
2     max = -inf
3     for i, n in enumerate(list) do
4           if n > max do
5                 max = n
6                 max_index = i
7     return max_index
```

Listing 3: Pseudocode for FindMax.

### 5.3.2 Available implementations

We have two implementations, one which takes a normal array as input and one that takes a graphBLAS vector:

| Version | Platform | Features | Code |
|---|---|---|---|
| V0 | CPU | Array | GitHub |
| V1 | CPU | GrB_Vector | GitHub |

Table 6. FindMax implementations

*At the time of deliverable submission, we are modeling and benchmarking these implementations. The data will be added to the repository.*

## 5.4 SINGLE-SOURCE SHORTEST PATH (SSSP)

### 5.4.1 The design

Whenever the edges of a graph have weights attached to them, BFS does not necessarily return the shortest path. In this case, an algorithm is needed that takes these weights into account. These algorithms are Single-Source Shortest Path (SSSP) algorithms.
There are many variants of this algorithm. Perhaps the most famous one is Dijkstra's algorithm, which is one of the implementations we made. The pseudocode for Dijkstra's algorithm is presented in Listing 4.

```
1 procedure dijkstra(graph, source):
2     Q <- new priority_queue
3     for node in graph.nodes() do
4           dist[node] = inf
5           parents[node] = -1
6     Q.push(source, 0)
7     dist[source] = 0
8     while !Q.isEmpty() do
9           v <- Q.pop()
10          for node in graph.neighbours(v) do
11                if dist[v] + graph.edges(v, node) < dist[node] do
12                      parent[node] = v
13                      dist[node] = dist[v] + graph.edges(v, node)
14                      Q.push(node, dist[node])
15    return dist, parent
```

Listing 4: Pseudocode for SSSP using Dijkstra's algorithm.

The input parameters of the SSSP algorithms are the graph, and a source node. The algorithms output the distance from the source node to every other node, and an array with the parents of each node.

### 5.4.2 Available implementations

Aside from Dijkstra's algorithm, there are many more SSSP algorithms. We have implemented the following:

| Version | Platform | Features | Code |
|---|---|---|---|
| V0 | CPU | Dijkstra | GitHub |
| V1 | CPU | Seq delta step | GitHub |
| V2 | CPU | GraphBLAS delta step | LAGraph GitHub |

Table 7. SSSP implmentations

*At the time of deliverable submission, we are modeling and benchmarking these implementations. The data will be added to the repository.*

## 5.5 FIND PATH

### 5.5.1 The design

Find path is an algorithm that inputs the parent array from any of the BFS or SSSP algorithms, and extracts the path from a given source to a given destination. The pseudocode is as follows:

```
1 procedure find_path(parent, source, dest):
2     path = []
3     current = source
4     while current != dest do
5           path.append(current)
6           current = parent[current]
7     path.append(dest)
8     return path
```

Listing 5: Pseudocode for Find Path using Dijkstra's algorithm.

### 5.5.2 Available implementations

Similarly to FindMax, we have two implementations, one which takes a normal array as input and one that takes a graphBLAS vector:

| Version | Platform | Features | Code |
|---|---|---|---|
| V0 | CPU | Array | GitHub |
| V1 | CPU | GrB_Vector | GitHub |

Table 8. FindPath implementations.

## 5.6 GRAPH QUERY (UC4) DESIGN AND IMPLEMENTATION

Querying telemetry data collected by the HPC focuses on executing complex queries to be implemented, which are not directly or indirectly possible for current SOA approaches such as Examon. The queries included are tailored towards the HPC facility managers/engineers, such as: (1) average power of a job, (2) minimum, maximum, and average temperature of a compute node when in use, (3) resource spatial information, etc.

A complete list of graph queries can be found at the following GitHub Link

The historic data collected by Examon for the Marconi100 HPC center at CINECA [1] needs to be converted to RDF based on the UC-4 ontology submitted in D6.1.

The RDF data are stored in GraphDB, a graph database that allows users to load RDF files and create knowledge graphs. The RDF format chosen is TURTLE, as upon benchmarking, it has been shown to use the least amount of system memory space as compared to other RDF formats. Additionally, the turtle format provides a human-readable syntax for expressing RDF triples, which consist of subject-predicate-object statements.

The conversion of historical data, which is in the format of Parquet, is read in Python using the pandas library and then passed into a Python script as input, which converts all the data in the file to RDF representation according to the schema set by the UC4 ontology.

The converted files are then \stored in the GraphDB store, where the knowledge graph for the complete Marconi100 room will be formed. This knowledge graph allows for implementing those complex graph queries mentioned earlier.

The input parameter is the graph query to be implemented, and the output is the appropriate response from the GraphDB store query. The output may be a single value, a table of values, or a graph.

The Git repository for this BGO can be found at the following GitHub Link. We have four versions for its implementation:

| Version | Platform | Features | Code |
|---|---|---|---|
| V0 | CPU | spatial query | GitHub |
| V1 | CPU | few instances | GitHub |
| V3 | CPU | job data | GitHub |
| V3 | CPU | instances of "total_power" metric | GitHub |

Table 9. Graph query implementations.

*At the time of deliverable submission, we are modeling and benchmarking these implementations. The data will be added to the repository.*

## 5.7 GNN INFERENCE (UC4) DESIGN AND IMPLEMENTATION

The task of anomaly detection and prediction is a very prevalent issue for HPC centers. Anomalous behavior at the facility can cause unavailability of compute nodes and can also cause downtime of the whole facility. A solution is needed, state-of-the-art solutions exist using machine learning tools such as autoencoders and deep neural networks. Anomaly prediction is more difficult than anomaly detection as it predicts signs of early approaching anomalous behavior, which can alert the facility engineers to fix the problem before it occurs.

We proposed using graph neural networks (GNNs) to predict anomalies for the facility. An HPC facility typically contains racks, and each rack contains compute nodes. This knowledge of physical proximity can be utilized to enhance analytical performance on available telemetry data from Examon. These relations can be used in GNNs to enhance the prediction capabilities of the AI models.

GNNs is a specialized neural network for graph processing. GNNs learn to extract meaningful information from the graph by iteratively updating node representations based on information from neighboring nodes and edges. This enables them to capture complex relationships and dependencies within the data.

We have trained models for the Marconi100 facility, and we have shown that they consistently outperform the current state of the art methods for prediction windows ranging from 1 hour ahead to 72 hours ahead.

This BGO for the UC4 would be to load the relevant rack model and perform model inference, which includes the generation of graphs using the real-time or historical telemetry data and the physical edge proximity matrix for the rack to make predictions for any anomalous behavior.

- **Input**: Graph depicting the relevant connectivity of the compute nodes (compute nodes in a compute rack, for instance), which includes the telemetry data as feature vectors and adjacency matrix as the edges to form the graph.
- **Output**: The prediction of anomalous behavior for each vertex in a graph (compute node) as a probability of anomaly occurrence in the prediction window.

| Version | Platform | Features | Code |
|---|---|---|---|
| V0 | CPU & GPU | GNN models inference on GPU or CPU | GitHub |

Table 10. GNN inference implementation

# 6 COMPOSITION RULES

Within Graph-Massivizer, we propose the creation of workflows as combinations of BGOs. We define **composition models** as the combination of BGOs that are found within such workflows. In this section, we present the current set of rules, and the high-level models that enable the creation of performance models of workflow combined.

## 6.1 Composition models

Based on the analysis of use-case 0, use-cases 1-4, and related work analysis (see Section 2), we propose a first set of composition models, listed in Table 11.

| Model | Template | Description |
|---|---|---|
| Seq + ShMem | Gtmp = BGO1 (Gin)<br>Gout = BGO2 (Gtmp); | Sequential execution.<br>Cost = BGO1 + BGO2<br>Zero-copy communication<br>CommCost ~ 0 |
| Seq + MsgPass | Gtmp = BGO1 (Gin);<br>send(Gtmp);<br>receive(Gtm);<br>Gout = BGO2 (Gtmp); | Sequential execution.<br>Cost = BGO1 + BGO2<br>Message passing communication<br>Cost = f(bandwidth, size(Gtmp)) |
| Seq + I/O | Gtmp1 = BGO1 (Gin);<br>tmpFile = save(Gtmp1);<br>Gtmp2 = retrieve(tmpFIle);<br>Gout = BGO2 (Gtmp); | Sequential execution.<br>Cost = BGO1 + BGO2<br>Data communication through file.<br>CommCost = f(I/O, size(Gtmp)) |
| Parallel + replicated data | Gtmp1 = BGO1 (Gin)<br>Gtmp2 = BGO2 (Gin) | Parallel BGO execution.<br>Cost = max(BGO1, BGO2)<br>Zero-copy communication<br>CommCost ~ 0 |
| Parallel + MsgPass | Gtmp1 = BGO1 (Gin);<br>receive(Gtmp);<br>Gout = BGO2 (Gtmp); | Sequential execution.<br>Cost = BGO1 + BGO2<br>Message passing communication<br>Cost = f(bandwidth, size(Gtmp)) |
| Parallel + I/O | Gtmp1 = BGO1 (Gin);<br>tmpFile = save(Gtmp1);<br>Gtmp2 = retrieve(tmpFIle);<br>Gout = BGO2 (Gtmp); | Sequential execution.<br>Cost = BGO1 + BGO2<br>Data communication through file.<br>CommCost = f(I/O, size(Gtmp)) |

Table 11. Composition models

We note that it is likely the list above is not a comprehensive list of composition rules. The list will be updated in the repository, where we keep track in "real-time" of the new composition models and rules we encounter in the project.

## 6.2 Example

Let us take the example of UC-0. From D6.1, UC-0 includes the following six BGOs, which are to be executed in sequence:

- BGO0: filter for field (SPARQL) - from Inceptor
- BGO1: run query to generate collaboration network (SPARQL) - from Inceptor
- BGO2: run BC on the collaboration network
- BGO3: select MAX = maximum BC
- BGO4: run BFS from root
- BGO5: check if MAX is visited and store path

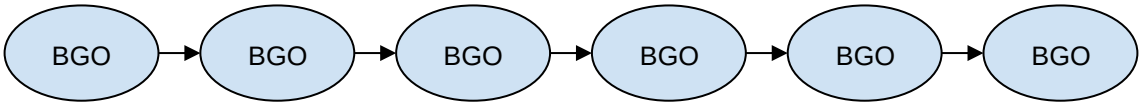


Figure 3. UC-0 workflow (from D6.1).

In this case, when all BGOs run on the same CPU, the model to be used is Seq + ShMem. When BGOs run on different CPUs (potentially on different nodes), we can use the model Seq+MsgPass.

Model validation is based on a shared-memory system. Specifically, for the Seq + ShMem model, we compare $T_m$ , the end-to-end measured execution time, against (a) $T_p$ , the predicted time as the sum of BGOs predicted times and (b) $T_{pm}$ , the sum of the BGOs measured times. The goal of this validation is twofold:

- $E_c = abs(T_m - T_{pm})/T_m$ indicates the error introduced by the composition model
- $E = abs(T_m - T_p)/T_m$ indicates the overall error, including the contributions of prediction errors for BGOs and the prediction error for the composition models.

*At the time of deliverable submission, we are benchmarking several BGOs from UC-0. We will validate the model against end-to-end measurements of UC-0. The data will be added to the repository.*

# 7 SUMMARY AND FUTURE WORK

This report presents the design and implementation principles for BGOs, for their models, and the different benchmarking aspects required for modeling and validation. Furthermore, we explain the role and design of composition models. We also include several examples of BGOs, among which the BFS includes a set of implementations, measurement and modeling data, and validation results.

However, this report is not a comprehensive list of all the BGOs in the project. Instead, we have designed it as a reading guide to accompany the repository of BGOs, currently found in the Graph-Optimizer repository. The BGO-specific information included in this report is also included in the repository, partially as guidelines and partially as ready-to-use artifacts, such as the models and/or the benchmarking data.

As Graph-Massivizer continues to evolve, the repository will expand in the following directions:

- we will add more BGOs, including those from Graph-Scrutinizer and Graph-Inceptor (e.g., BGOs from D3.1)
- we will add the analytical and/or statistical models for all these implementations
- we will add more calibration microbenchmarks (as needed for the new BGOs) and derive predictive models for the different platforms in the project
- we will add validation data for all the models
- we will add energy models and their validation for all BGOs
- we will validate the workflow composition rules and models for different implementations as they become available.

# 8 REFERENCES


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

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

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

[4] Färber, Michael, et al. "SemOpenAlex: The Scientific Landscape in 26 Billion RDF Triples." *International Semantic Web Conference*. Cham: Springer Nature Switzerland, 2023.

[5] Harris, Steve, and Andy Seaborne. "SPARQL 1.1 Query Language." World Wide Web Consortium (W3C), 2013, https://www.w3.org/TR/sparql11-query/.

[6] Hartig, Olaf, et al. "RDF-Star and SPARQL-Star." World Wide Web Consortium (W3C), 2021, https://www.w3.org/2021/12/rdf-star.html.

[7] Svetashova, Yulia, et al. "Ontology-enhanced machine learning: a Bosch use case of welding quality monitoring." *The Semantic Web–ISWC 2020: 19th International Semantic Web Conference, Athens, Greece, November 2–6, 2020, Proceedings, Part II 19*. Springer International Publishing, 2020.

[8] Turra, Roberta, et al. "D1.3: Graph-Massivizer Data Management Plan." Graph-Massivizer, 2023, https://graph-massivizer.eu/.

[9] Verstraaten, Merijn, et al. "Mix-and-Match: A Model-Driven Runtime Optimisation Strategy for BFS on GPUs". *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, IEEE, 2018, https://doi.org10.1109/ia3.2018.00014.

[10] KONECT repository, http://www.konect.cc/

# 9 ACRONYMS

| Acronym | Description |
|---|---|
| BGOs | Basic Graph Operations |
| GPU | Graphics Processing Unit |
| CPU | Central Processing Unit |
| BFS | Breadth First Traversal |
| BC | Betweenness Centrality |
| SSSP | Single source shortest path |