

ENS 491-492 – Graduation Project
Final Report

Project Title: Real-Time Super Computer Monitoring

Group Members:

Alperen Yıldız

Bilgehan Çağiltay

Elif Cemre Durgut

Supervisor: Kamer Kaya

Date: 04/06/2023



1. EXECUTIVE SUMMARY

The real-time supercomputer monitoring project (SuperTwin) is a part of SparCity, which is an international project where 6 partners from 4 different countries collaborate with 2.6 M € funding by the European High-Performance Computing Joint Undertaking (EuroHPC JU).

SuperTwin aims to create an automated monitoring tool by utilizing machine learning. The goal of this tool is to monitor for, recognize, and respond to reductions in performance. Within this project, the objective is not only to create a tool that can be utilized to view the performance of supercomputers with respect to their configurations but also to extract data and shed further light on performance anomalies associated with some configurations and cache accesses.

SuperTwin gathers data from supercomputers and creates a structured dashboard representation by managing data input from a collection of tools. Some capabilities of SuperTwin are knowledge retrieval, modeling, monitoring, and automated profiling. In the end, it is expected to optimize data retrieval and storage, improve the data monitoring visualizations, use machine learning to predict future performance and detect anomalies.

SuperTwin aims to extract data from standalone Linux servers and HPC clusters and process this large volume of data so that it can be used to train machine learning models. Additionally, the data is also processed and visualized to system owners on dashboards so as to clearly show what is currently happening within the system. The trained machine learning model will then be used to detect anomalies, provide insights on how to achieve better performance and energy efficiency and report performance characteristics.

Furthermore, the tool will feature detailed documentation on its usability and features. This will be complemented by visual animations for providing a user-friendly insight into the inner workings of the system.

2. PROBLEM STATEMENT

The main drawbacks of existing supercomputer monitoring systems are

- Cost: Supercomputers are expensive to purchase and maintain. The cost of implementing and maintaining a monitoring system can add significantly to the overall cost.
- Complexity: Supercomputers are complex systems with many different components that need to be monitored, which can make the monitoring process itself complex.
- Scalability: As supercomputers grow in size and complexity, it can be difficult to scale the monitoring system to keep up with the increased demands.
- Integration with other systems: Supercomputers often need to integrate with other systems and technologies. It can be challenging to ensure that the monitoring system works seamlessly with these other systems.
- Accuracy: It is important for monitoring systems to be accurate in order to effectively identify and diagnose problems. However, achieving high levels of accuracy can be difficult, especially as supercomputers become more complex.

SuperTwin aims to address these issues.

2.1. Objectives/Tasks

As we stated in our Proposal Report in October 2022, our objectives were as follows:

- Extraction of data from a standalone Linux server and an HPC cluster
- Visualization of the extracted data
- Usage of the data to train machine learning models to detect anomalies, and performance characteristics, and provide insights for better performance and energy efficiency
- Documentation of the tool and visual animation to provide insight into its inner workings of it.

2.2. Realistic Constraints

Economic: The main resource needed for this project is data coming from supercomputers. Since supercomputers are expensive tools to build and operate because of their high energy demand and complexity they constitute the main cost for the continuation of this project. Fortunately, EuroPHC JU covers the cost related to the supercomputers. Furthermore, other resources such as software version control tools and clusters are required and compose an albeit lesser portion of the budget. For this purpose, computing resources allocated by Sabancı University, open source tools such as Git and free-to-use remote repository services such as GitHub are used.

Environmental: Apart from some projects related to computer science such as blockchain applications, most research done in this area has a negligible environmental impact. Albeit still negligible, the energy consumption of supercomputers constitutes a major part of the aforementioned impact.

Social: Some computer science-related projects have a direct or indirect impact on society such as some machine learning projects. However, this particular project poses no hazard on the social lives of any people.

Health and Safety: Due to this project having no overlap with medicine or any health critical field; health and safety are not a major concern.

Manufacturability: As a software project, it requires computing resources, necessary software, tools, and IDE. GNU Linux operating system is needed to use the necessary libraries. However, the necessary computers already exist, the computers in the HPC clusters are open to using when needed, and the software and the operating system are free.

Sustainability (social, economic, and environmental): Our project aims to be sustainable in terms of energy efficiency. The main SparCity project that our project belongs to is briefly about “creating a supercomputing framework that will provide efficient algorithms and coherent tools specifically designed for maximizing the performance and energy efficiency of sparse computations on emerging HPC systems” as it is written on the project website. One of the objectives of this project is to “create digital SuperTwins of supercomputers to evaluate and simulate what-if hardware scenarios and to gather real-time performance and energy intel from node- and system-level components for application optimization on the current and future hardware” which encapsulates our project. By considering what-if scenarios and analyzing the

data collected from the super-computers, energy consumption will also be detected and will be minimized by optimizations.

Another aspect of this constraint is to make the code and the reports sustainable for future readers and users so that others will be able to understand, make use of, and develop the project.

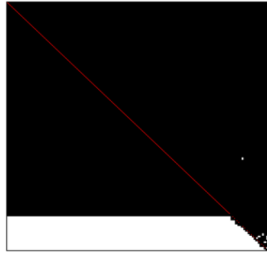
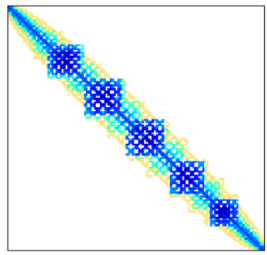

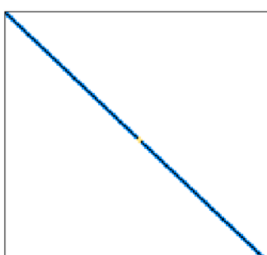

3. METHODOLOGY

One of the main objectives of SparseBase is to accelerate the sparse matrix-vector multiplication by reordering the matrices. To this end, we set one of our objectives to test orderings offered by Sparsebase with different matrices on SuperTwin. Sparsebase has 7 different ordering implementations namely

- 1) Amd
- 2) Degree
- 3) Generic
- 4) Gray
- 5) Metis
- 6) Rabbit
- 7) Rcm

We chose 10 different matrices with our project supervisor's advice by considering dimension size, number of non-zero values, symmetry, and structure to test a wide range of matrices as shown on Table 1. Their dimensions vary from 60K to 41M. While some of the matrices are denser, some of them are more sparse. As Bian et. al (2021) did in their SpMV research, we grouped the matrices into two, regular and irregular. Regarding whether the matrix is regular or not if the difference between the maximum nnz elements in a row and the minimum nnz elements in a row is in (0, 999), it is considered as regular. Else if the difference is 1000+, it is regarded as an irregular matrix.

We selected 8 metrics as it is demonstrated in Table 2 to test the performance during the orderings and the SpMV algorithms.

ID	Matrix Name	Dimension	Nof nonzeros	Symmetry	Structure
R1	amazon-2008	735,323 x 735,323	5,158,388	no	
R2	Ga41As41H72	268,096 x 268,096	18,488,476	yes	
R3	road_central	14,081,816 x 14,081,816	33,866,826	yes	
R4	cant	62,451 x 62,451	4,007,383	yes	
R5	road_usa	23,947,347 x 23,947,347	57,708,624	yes	

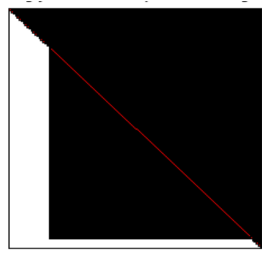

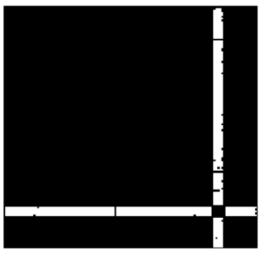
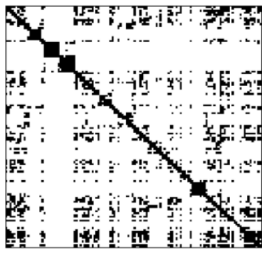
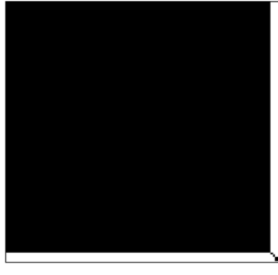
ID	Matrix Name	Dimension	Nof nonzeros	Symmetry	Structure
IR1	twitter7	41,652,230 x 41,652,230	1,468,365,182	no	
IR2	rail4284	1,096,894 x 1,096,894	11,284,032	no	
IR3	uk-2005	39,459,925 x 39,459,925	936,364,282	no	
IR4	cnr-2000	325,557 x 325,557	3,216,152	no	
IR5	webbase-2001	118,142,155 x 118,142,155	1,019,903,190	no	

Table 1. Matrix names & features to be used in testing

PMU Metric Name	Description
L1D:REPLACEMENT	Represents the number of L1 cache lines that were replaced due to conflicts or capacity limitations. L1 cache is the first level of cache in a computer processor, and it is used to store frequently accessed data for quick access. If the cache is full or there are multiple requests for the same cache line, replacement occurs. The L1D:REPLACEMENT metric can help identify if there are issues with cache utilization or if there are memory access patterns that are causing cache conflicts.
MEM_UOPS_RETIRED: ALL_LOADS	Shows the number of micro-ops (uops) that were retired for all loads from memory. Uops are small instructions that the processor executes, and this metric counts the number of uops that were executed for memory loads. This metric can help identify memory-bound performance issues, such as when there are bottlenecks in accessing memory.
MEM_UOPS_RETIRED: ALL_STORES	Represents the number of micro-ops that were retired for all stores to memory. Similar to the previous metric, this metric counts the number of uops executed for memory stores. It can help identify memory-bound performance issues and identify patterns in memory access.
MEM_LOAD_RETIRED: L1_HIT	Represents the number of retired load uops that hit in the L1 cache. It can help identify the effectiveness of the L1 cache and whether data is being cached effectively.
MEM_LOAD_RETIRED: L2_HIT	Shows the number of retired load uops that hit in the L2 cache. The L2 cache is the second level of cache in a computer processor and is larger than the L1 cache. This metric can help identify the effectiveness of the L2 cache and whether data is being cached effectively.
MEM_LOAD_RETIRED: L3_HIT	Represents the number of retired load uops that hit in the L3 cache. The L3 cache is the third level of cache in a computer processor and is even larger than the L2 cache. This metric can help identify the effectiveness of the L3 cache and whether data is being cached effectively.
FP_ARITH: SCALAR_SINGLE	Represents the number of scalar single-precision floating-point arithmetic instructions executed. This can include operations such as addition, subtraction, multiplication, and division on single-precision floating-point values. This metric can help identify whether there are any bottlenecks in floating-point arithmetic operations.
FP_ARITH: SCALAR_DOUBLE	Shows the number of scalar double-precision floating-point arithmetic instructions executed. This can include operations such as addition, subtraction, multiplication, and division on double-precision floating-point values. This metric can help identify whether there are any bottlenecks in floating-point arithmetic operations for double-precision values.

Table 2. Performance metrics names and descriptions

We implemented and adopted two SpMV algorithms. The first one is a classic SpMV product as demonstrated in Algorithm 1.

Algorithm 1: *SpMV Computation using CSR matrix format*

Input: *CSR matrix A , vector x*

Output: *Vector y such that $y \leftarrow Ax$*

```
for i = 1 : A.n
    A[i,] = 0
    for j = A.row_ptr[i] : A.row_ptr[i]-1
        A[i] += A.vals[j] + v[A.cols[j]]
```

Secondly, we adopted Merge-SpMV to SparseBase matrix format so that it works with SparseBase functions and classes. As pointed out by Merrill and Garland (2016), the main idea is to present parallel CsrMV decomposition as a merger of the CSR row-offsets and CSR non-zero data, which is then split equally among parallel processors/threads. This partitioning method ensures that no single processing element can be overwhelmed by assignment to arbitrarily-long rows or an arbitrarily-large number of zero-length rows. The pseudocode of the merge-based SpMV algorithm can be found below. If you would like to get more information, click on this [link](#) to access their GitHub repository.

Algorithm 2: *Merge-SpMV Computation using CSR format by Merrill, Garland*

Input: *Number of parallel threads, CSR matrix A , vector x*

Output: *Vector y such that $y \leftarrow Ax$*

```
Merge list A: row end-offsets
Merge list B: Natural numbers(NZ indices)
Merge path total length
Merge items per thread
Spawn parallel threads
    Find starting and ending MergePath coordinates
    (row_ptr, nonzero_idx) for each thread
    Consume merge items, whole rows first
    Consume merge items, whole rows first
    Save carry_outs
Carry-out fix-up (rows spanning multiple threads)
```

To conduct these tests, first we need to create a SuperTwin instance and then later on run the demo to create performance dashboards. Below, you will find how these operations are done in detail. You can find our project repository in this [link](#).

1) Creating SuperTwin instance

To create a SuperTwin instance, you need to run `supertwin.py` file after you start `mongodb`, `grafana-server` and `influxdb`. In this file, `SuperTwin` class constructor is called. Inside this constructor, `__build_twin_from_scratch` method is called. Since no arguments are given in the command previously run, this function asks 4 inputs for the user to enter, the address of the remote system, and the hostname.

The function named `main` is called which is defined in `remote_probe.py` that performs remote probing in Linux. Firstly, this function prompts the user to enter the SSH username and password. The `paramiko` library is used to create an SSH client object. It creates an instance of the `SSHClient` class from the `paramiko` library and assigns it to the `ssh` variable. This object represents an SSH client session that can be used to connect to a remote system and execute commands over an encrypted channel.

Later, it sets the policy for automatically adding the remote system's host key to the local system's `known_hosts` file. This is done to prevent potential security risks, such as a man-in-the-middle attack.

The function then connects to the remote host using SSH and executes the `"hostname"` command to obtain the name of the remote host. The function then creates two directories on the remote system and copies files from the local system to the remote system using Secure Copy (SCP).

The `scp.put()` method is used to copy a file or directory from the local system to a remote system over an encrypted SCP (Secure Copy) connection.

The `pmu_query_path` variable holds the path of the file or directory to be copied, and the recursive parameter is set to `True` to recursively copy the entire directory and its contents.

The `remote_path` parameter specifies the destination path on the remote system where the file or directory will be copied to.

After copying the files, the function runs several commands on the remote system, including removing files, creating files, compiling code, and running Python scripts. Finally, the function retrieves the probing results from the remote system using SCP and saves them to a file.

The `scp.get()` method is used to download a file or directory from the remote system to the local system over an encrypted SCP (Secure Copy) connection.

The function returns the remote host name, the probing results file name, SSH username, and SSH password.

The function `read_env()` reads environment variables from a file named `"env.txt"` and returns a tuple containing the values of four specific variables: `"MONGODB_SERVER"`, `"INFLUX_1.8_SERVER"`, `"GRAFANA_SERVER"`, and `"GRAFANA_TOKEN"`. This txt file contains the ports that run the Influx, Grafana and MongoDB servers and the tokens.

The function `create_grafana_datasource()` is used to create a datasource in Grafana for connecting to an InfluxDB server. It takes several parameters:

- `hostname`: The hostname used to identify the datasource.
- `uid`: A unique identifier used to identify the datasource.
- `api_key`: The API key used for authentication with Grafana.
- `grafana_server`: The URL of the Grafana server.

- `influxdb_server`: The URL of the InfluxDB server.
- `verify`: Optional parameter to verify SSL/TLS certificates (default is True).

The function begins by setting the necessary headers for the API request. It includes the Authorization header with the provided `api_key` and sets the Content-Type header to "application/json".

Next, a dictionary data is created to define the properties of the datasource. It includes the name of the datasource, which combines the hostname and uid, and sets the type to "influxdb". The url property is set to the `influxdb_server` URL, and the database property is set to the hostname. The access property is set to "proxy" to allow Grafana to proxy requests to the InfluxDB server, and `basicAuth` is set to False to indicate that basic authentication is not used.

The function then makes a POST request to the Grafana API endpoint for creating datasources (`/api/datasources`). It sends the data dictionary as JSON in the request body and includes the headers for authentication. The `verify` parameter is used to control SSL/TLS certificate verification. It returns the response JSON content as a dictionary.

The function `get_pcp_pids()` begins by creating an SSH client object using `paramiko.SSHClient()` and sets the missing host key policy to automatically add the remote system's host key. It then establishes an SSH connection to the remote system using the provided address, SSH username, and password.

The function executes the command `ps aux | grep pcp` on the remote system using `ssh.exec_command()`. This command lists all running processes (`ps aux`) and filters the output to only show lines containing the string "pcp" (`grep pcp`). The function reads the output from the command and stores it in the output variable.

The function initializes an empty dictionary `pids` to store the process IDs of the PCP processes.

It then iterates over each line in the output and checks for specific PCP process names using `item.find()`.

Finally, the function prints the modified `pids` dictionary and returns it as the result of the function.

`get_influx_database` parses an address string to extract the host and port information, and then returns an `InfluxDBClient` object initialized with the extracted host and port values for connecting to an InfluxDB server.

`create_influx_database` function creates a new database with the specified name using the provided InfluxDB data source

`create_database` creates a new database in InfluxDB

`insert_twin_description` function inserts a twin description document into a MongoDB collection. It retrieves relevant information from the provided supertwin object, such as hostname, address, date, and various configuration details. The document is inserted into the "twin" collection, and the ID of the inserted document is returned

`get_mongo_database` establishes a connection to a MongoDB server using the provided connection string. It then returns the specified MongoDB database with the given name.

`get_twin_description_from_file` reads a JSON file specified by `hostProbFile`, loads its content into a dictionary `_sys_dict`, and passes that dictionary along with other parameters (`alias`, `SSHuser`, `SSHpass`, `addr`) to a function `generate_dt.main()` to generate a twin description `_twin`. The generated twin description is then returned.

`pmu_to_pcp` converts PMU (Performance Monitoring Unit) metrics into PCP (Performance Co-Pilot) metric names and appends them to the metrics list. It iterates over each key in the PMUs dictionary. If the key does not contain the substring "perf", it checks if the key should be added based on whether it has already been added to the added list. If it should be added, it iterates over the events associated with that key and appends the corresponding PCP metric name, original metric name, and event value to the metrics list. The added list keeps track of the keys that have been processed to avoid duplicates.

`add_my_metrics_mapped` adds metrics to a dictionary of models. It retrieves specific metrics based on given categories, maps them to the appropriate format, and appends them to the dictionary under the corresponding model.

`add_cpus` It iterates over each key in the PMUs dictionary. If the key does not contain the substring "perf", it checks if the key should be added based on whether it has already been added to the added list. If it should be added, it iterates over the events associated with that key and appends the corresponding PCP metric name, original metric name, and event value to the metrics list. The added list keeps track of the keys that have been processed to avoid duplicates. Finally, the function returns the updated metrics list.

`add_memory` function adds custom metrics to a dictionary based on provided parameters, appending either a supertwin telemetry or regular telemetry based on the type of the custom metric. The updated dictionary is returned.

`add_disk` function adds a disk component to a digital twin represented by a dictionary (`models_dict`) based on the provided parameters. It connects the disk component to the system, adds properties and custom metrics as telemetry, and calls another function `add_phy_disks()` to add physical disks. The updated `models_dict` is returned.

`add_network` function adds a network component to a digital twin represented by a dictionary (`models_dict`) based on the provided parameters. It connects the network component to the system, adds custom metrics as telemetry, and calls another function `add_subnets()` to add subnets. The updated `models_dict` is returned.

`get_pcp_pids_by_credentials` adds a network component to a digital twin represented by a dictionary (`models_dict`) based on the provided parameters. It creates a top-level network interface, connects it to the system, adds custom metrics as telemetry, and calls another function `add_subnets()` to add subnets. The updated `models_dict` is returned.

`get_monitoring_metrics` retrieves monitoring metrics from a supertwin object based on the specified metric type. It accesses a database, extracts the twin data, and filters out

metrics that match the given metric type. The function then returns a list of dictionaries, where each dictionary contains the metric name and its corresponding type.

`get_metric_type` function determines the type of a metric based on the given metric name. It checks for specific patterns in the metric name and assigns the corresponding type. The function returns a string representing the type of the metric.

`reconfigure_observation_events_beginning` used to reconfigure the observation events at the beginning. It first checks for metrics that should always be present in the "observation_metrics" list and adds them if they are missing. Then, it calls the "reconfigure_perfevent()" method and registers the twin state using the "register_twin_state()" function from the "utils" module. There is commented out code that writes the metrics to a file named "last_observation_metrics.txt".

`reconfigure_perfevent` is used to reconfigure the "perfevent" component on a remote server. It establishes an SSH connection to the server using the provided credentials. Then, it uses SCP to transfer a file named "perfevent.conf" to a temporary location on the server. It creates a shell script named "reconfigure_perf.sh" that contains a series of commands to perform the reconfiguration. The shell script is also transferred to the server. Finally, it executes the shell script with sudo privileges on the remote server to reconfigure the "perfevent" component. A message is printed to indicate that the remote "perfevent" pmda has been reconfigured.

`generate_perfevent_conf` function is used to generate a new configuration file for the "perfevent" pmda component. It takes a "SuperTwin" object as input. It retrieves the observation metrics from the object and ensures that any metrics that should always be present are included. It also retrieves the MSR (Model-Specific Register) configuration using the "get_msr()" function from the "utils" module.

`reconfigure_perfevent` The function then creates a new file named "perfevent.conf" and writes the MSR configuration and the list of metrics to it. Finally, it prints a message to indicate that a new configuration for the "perfevent" pmda has been generated. The function then creates a new file named "perfevent.conf" and writes the MSR configuration and the list of metrics to it. Finally, it prints a message to indicate that a new configuration for the "perfevent" pmda has been generated.

`generate_pcp2influx_config` generates a configuration file for PCP2InfluxDB integration based on the attributes of a "SuperTwin" object. It retrieves the necessary information such as database name, tags, source IP, and metrics. It then constructs the configuration file by adding options, including InfluxDB server details and source information, as well as the specified metrics. The resulting configuration file is written to disk, and the file name is returned.

`update_state` updates the state by appending a new line of information to a file named "supertwin.state". It takes in the parameters name, addr, twin_id, and collection_id, and writes them in a specific format separated by the "|" character. The function then closes the file after writing the information.

`kill_zombie_monitors` This function is used to kill zombie monitoring samplers running on the system. It retrieves the process information for processes matching the name `"/usr/bin/pcp2influxdb"` by executing the command `"ps aux | grep pcp2influxdb"`. It extracts the process ID, state, and configuration file from the output and compares the process ID with the `monitor_pid` attribute. If they do not match, it forcefully kills the process.

`generate_monitoring_dashboard` generates a monitoring dashboard using the `generate_monitoring_dashboard()` function from the `monitoring_dashboard` module. It assigns the generated URL of the dashboard to the variable `url`. Then, it calls the `update_twin_document__add_monitoring_dashboard()` method of the current object, passing the URL as a parameter to update the twin document with the monitoring dashboard URL.

`generate_monitoring_dashboard` is responsible for generating a monitoring dashboard. It calls the `generate_monitoring_dashboard()` function from the `monitoring_dashboard` module, passing `self` as an argument. The generated dashboard URL is assigned to the variable `url`. Then, it calls the `update_twin_document__add_monitoring_dashboard()` method of the current object, passing the URL as a parameter to update the twin document by adding the monitoring dashboard URL.

`get_params_interface_known` retrieves the parameters for a known interface and measurement from a twin description (`td`). It iterates over the contents of the specified interface in the twin description and checks if the content's type contains `"Telemetry"`. If a content's database name matches the provided measurement, it assigns the content's display name as the `"Param"` parameter and the interface's display name as the `"Alias"` parameter in the `params` dictionary. There is a special case for the measurement `"hinv_cpu_clock"` where the first content's display name is used instead. If the `"Alias"` contains the word `"thread"`, it is stripped for a cleaner appearance. The function then returns the `params` dictionary.

`update_twin_document__add_monitoring_dashboard` updates the twin document by adding a monitoring dashboard URL. It retrieves the MongoDB database for the twin using the twin's name and MongoDB address. Then, it retrieves the twin document associated with the current object's MongoDB ID. The retrieved document is modified by adding a new key-value pair with the key `"monitoring_dashboard"` and the provided URL as the value. Finally, the modified document is replaced in the database using the twin's MongoDB ID. A message is printed to indicate that the monitoring dashboard has been added to the digital twin.

`register_twin_state` registers the state of a twin by updating its twin document in the MongoDB database. It retrieves the twin document based on the twin's name and MongoDB address. Then, it updates the relevant fields in the document with the corresponding values from the `SuperTwin` object. The modified document is replaced in the database, and a message is printed to indicate that the twin state has been registered.

Super Twin				
	__build_twin_from_scratch			
		main		
			set_missing_host_key_policy	
			connect	
			exec_command	
			run_sudo_command	
			run_command	
			put	
			get close	
			cmd	
		read_env		
		create_grafana_datasource		
		get_pcp_pids		
			get_pcp_pids_by_credentials	
		get_influx_database		
		create_influx_database		
			create_database	
		insert_twin_description		
			get_mongo_database	
		get_twin_description_from_file		
			main	
				pmu_to_pcp
				add_my_metrics_mapped
				add_cpus
				add_memory
				add_disk
				add_network
				get_pcp_pids_by_credentials
		get_monitoring_metrics		
			get_metric_type	

		reconfigure_observation_events_beginning		
			reconfigure_perfevent	
				generate_perfevent_conf
				reconfigure_perfevent
		main		
			generate_pcp2influx_config	
		update_state		
		kill_zombie_monitors		
		generate_monitoring_dashboard		
			generate_monitoring_dashboard	
				get_params_interface_known
			update_twin_document_add_monitoring_dashboard	
		register_twin_state		

Table 3. Creating SuperTwin instance function map

2) Generating performance dashboards

In the SuperTwin repository, you will find `pmu_demo.py` file which shows an example of how to generate a dashboard.

In this file, a superTwin instance is created by giving the IP address of the computer that you want the test to be conducted on as a parameter.

Later, `reconfigure_observation_events_parameterized` function is called from the superTwin object by giving the txt file which includes the performance observation metric names to test. This function reads the txt file, saves the observation metrics. It generates a new perfevent pmda configuration, connects to the remote host via ssh, creates a connection with MongoDB and finally registers the SuperTwin state to the database.

Then, `execute_observation_batch_parameters` is called from the same superTwin instance. This function takes three parameters as

- 1) **path:** path in the supercomputer where the executable commands work.
- 2) **affinity:** `likwid-pin` which is a command line application to pin sequential or multithreaded applications to dedicated processors is used here along with necessary options.

- 3) **commands:** a list of commands where each command is in the form of <given name> | <executable command>. The <given name> will be the indicator of this command, it will be shown on Grafana-generated graphs.

This function executes a batch of observations, where each element in the batch is observed individually. It configures the InfluxDB server and PCP, connects to the remote host via ssh and runs the commands on the remote host, saves the outputs on InfluxDB, creates a dashboard on Grafana, queries the observation results from DB, and uploads it to Grafana.

At the end of this run, a URL to access the Grafana dashboard will be printed on the screen. Graphs and mean charts for each observation are generated on this dashboard.

A SuperTwin instance is constructed from address.

reconfigure_observation_events_parameterized takes the observation metrics.

reconfigure_perfevent calls perfevent functions.

generate_perfevent_conf generates new perfevent pmda configuration.

reconfigure_perfevent connects to SuperTwin address via ssh.

register_twin_state registers to db.

get_mongo_database creates connection with MongoDB

loads deserializes a BSON-encoded string or bytes object into a Python object

execute_observation_batch_parameters creates observation dict with uid, affinity and metrics. For each command, it calls batch_element_parameters

execute_observation_batch_element_parameters executes a batch of observations, where each element in the batch is observed individually and the duration of each observation is returned.

generate_pcp2influxdb_config_observation configures InfluxDB server and PCP to a .conf file

add_pcp adds pcp configurations to influxDB configurations

observe_single_parameters connects to remote host via ssh, executes and observes the command given in pmu_demo.py on Dolap

put SCP (secure copy) is a command-line utility that allows to securely copy files and directories between two locations.

exec_command runs the script on remote host

normalize_tag provides the connection with InfluxDB, runs query on DB, and normalized data points

get_influx_database makes the InfluxDBClient connection with host and port

switch_database switches to the given database name

query executes a query against an InfluxDB database and returns the results in a list of dictionaries representing the measurements or points in the query result

write_points writes data to InfluxDB database

main performs element observations, including finding and visualizing metrics, creating a dashboard, querying a database, and uploading the dashboard to Grafana

find_from_likwid_pin splits affinity parameter to parts

ret_ts_panel gets time series panel for the metric

ret_gauge_panel gets gauge panel for the metric

get_field_and_metric function, A PMU is a unit of measure that reflects the execution of a section of code. The system starts and stops a PMU at specific code locations, and the system may update a PMU anytime between the start and stop times.

ret_query gets query for observation

add_trace adds one or more traces, represented by a Trace object, to a plot and allows for customization of their appearance.

grafana_layout_2 updates the layout of a Plotly figure by setting various properties for the x and y axes, margin, mode bar, and paper and plot background colors, and returns the modified figure

update_layout updates the layout of a Plotly figure by modifying the properties of the layout object

two_templates_two updates the layout of a Plotly figure by modifying the properties of the layout object

get_influx_database returns InfluxDBClient object with host and port

get_field_and_metric function queries a MongoDB database storing information about a digital twin and returns the display name and database name for a given performance monitoring unit (PMU) metric by matching the given socket and thread information with the corresponding component in the twin.

query method in InfluxDB is used to execute queries against the database and retrieve time-series data

get_dashboard_json get_dashboard_json generates JSON from grafanalib Dashboard object

upload_to_grafana upload_to_grafana tries to upload dashboard to grafana and prints response

update_twin_document__add_observation updates a MongoDB database by inserting a single observation document, and prints a confirmation message

get_mongo_database creates a connection to a MongoDB database using a connection string and returns an instance of that database for performing CRUD operations

SuperTwin			
reconfigure_observation_events_parameterized			
	reconfigure_perfevent		
		generate_perfevent_conf	
		reconfigure_perfevent	
	register_twin_state		
		get_mongo_database	
		loads	

execute_observation_batch_parameters			
	execute_observation_batch_element_parameters		
		generate_pcp2influxdb_config_observation	
			add_pcp
		observe_single_parameters	
			put
			exec_command
	normalize_tag		
		get_influx_database	
		switch_database	
		query	
		write_points	
	main		
		find_from_likwid_pin	
		ret_ts_panel	
		ret_gauge_panel	
		get_field_and_metric	
		ret_query	
		add_trace	
		grafana_layout_2	
		update_layout	
		two_templates_two	
		get_influx_database	
		get_field_and_metric	
		query	
		get_dashboard_json	
		upload_to_grafana	
	update_twin_document_add_observation		
		get_mongo_database	

Table 4. Generating Grafana dashboards function map

4. RESULTS & DISCUSSION

4.1 Documentation on Read The Docs

As previously mentioned in our first project report, one of our objectives was to create comprehensive documentation for the SuperTwin project, including instructions for installation on various operating systems and guidance on running and testing the project on supercomputers. To this end, we have created a ReadTheDocs document in which we provide a detailed explanation of the project. You can reach the documentation by clicking on this [link](#).

In order to accommodate users who utilize various types of operating systems, we have provided installation tutorials for systems such as Ubuntu, Manjaro, and MAC. We have also included potential warnings and notes in the documentation to address some errors that users may encounter.

In addition to the installation tutorial above, the documentation includes sections such as *How to contribute to the project?* in which we explained how to add to the repository, how to report found bugs or fixes, and also the *Frequently Asked Questions* section to answer the most commonly asked questions.

Moreover, we explained some fundamental use cases of SuperTwin in our *How To Guides* section. In this section, we shed light on the SuperTwin instance creation, performance testing, and generating dashboards on the Grafana features of the project.

4.2 The automated setup code

Additionally, the setup was automated using a series of bash and Python scripts so that the setup process can proceed smoothly in any *nix operating system. The setup process works by calling a bash script which automatically detects on which operating system it is being run and adapts the setup process depending on the operating system while also maintaining a detailed log file.

For the test process, we conducted tests on new virtual machines from scratch to ensure that the installation commands and setup scripts work correctly. As a future work, next ENS 491/492 students might try this setup code, verify and/or update the lines.

4.3 Testing simple and merge-based SpMVs with different SparseBase orderings

MergeBase: From the data we have gathered, displayed within the appendix section, we can see that no ordering is faster than the other orderings, at 7.1s, by half a second for small matrices. This is followed by “degree,” “gray,” and “rcm.”

- “degree” uses the least amount of average L1 cache replacements by a significant margin. This is followed by “none” which has a similar average use of L1 replacements to “rcm” and “gray,” but as “none” finishes much faster its total resource impact is significantly lower than “rcm” and “gray.”
- The average number of total memory accesses done by “gray” and “rcm” are very similar, and this is followed by “degree” and finally “none.” Similarly, as “none” finished processing much faster, its total memory access number is still the lowest even though it accesses memory the most.

- “rcm,” “gray” and “degree” all have similar memory storage uses, but “none” utilizes much more memory usage than the first three methods.
- “rcm,” “degree,” and “gray” all have similar L1 cache accesses, while “none” uses significantly higher use of L1 accesses.
- While “rcm” and “gray” have similar total and mean L2 cache accesses, “degree” and “none” have a significantly higher access rate for this cache.
- The average L3 cache access of “none” is significantly lower than every other method, followed by “degree,” “gray,” and “rcm” where each of these have similar average accesses for L3. Similar to the other results of none, this discrepancy between it and the other methods is due to how early “none” is able to finish its processing.
- The scalar double use of “degree,” “rcm,” and “gray” have similar statistics, although the performance of “degree” is the best of the three. The “none” method, even though it finishes processing faster, uses scalar double significantly more than every other method.

Simple SpMV: For Simple SPMV, similar to Merge-Based SPMV, the “none” method finishes processing significantly faster than all the other methods. This is followed by “degree,” which is faster than “rcm” and “gray” where both these methods have the same processing speed.

- The average L1 cache replacements of “gray” is significantly lower than “degree,” “rcm,” “none.” “degree” and “rcm” have very similar results for their usage of L1 cache, and finally the usage of L1 of “none” comes last. Again, similarly to Merge-Based SPMV, while “none” uses more on average, as it runs faster, it makes significantly less L1 replacements in total.
- The access numbers of each method have very similar results, but overall “degree” has the lowest average access counts.
- Similar to the access numbers statistics above, the memory storage statistics of each method follow very similar trend lines and results. Furthermore, similar to the access numbers of each method, “degree” has the lowest average use of memory storage.
- Similar to the previous two results, the L1 cache accesses of each statistic follows similar trend lines with similar results. All methods besides “none” show a very similar average access count, however this is likely due to “none” finishing its processing faster and thus not being able to balance out its high results with lower results that are achieved at the end of execution time.
- Again, similarly to the L1 cache access results, the results of each method show very similar results with the “none” method showing worse average results due to its early calculation completion.
- Unlike L1 and L2 cache access results, the L3 cache access results show distinguishable results. Judging by the average results of each method, “gray” is the best method due to its early low trendline. This is followed by “degree,” which stops increasing its L3 cache accesses halfway through calculation until the end of its completion. The next method is “rcm” which shows a constant rise in its L3 cache accesses, which causes it to show worse results than “degree” even though they display similar results at the start of their usage of L3 cache.

- The results of “none” and “degree” for scalar double use show similar trendlines, however their final average statistics are different due to the early completion of “none.” After these two methods, the next lowest use of scalar double is by “rcm” and finally by “gray” with the highest use of scalar double.

4.4 Current result of the project

The results for the matrices of the smaller sizes have been completed with technical difficulties regarding the matrices of greater sizes due to glitches in the implementation. It is planned that the underlying technical reasons behind the glitches will be investigated therefore, additional information about matrices of smaller sizes will be provided.

5. IMPACT

High performing computing is a field of computer science where high levels of energy sources are depleted. Since a substantial percentage of energy consumed by computers are generated using fossil fuels, any and all endeavors on the path of reducing energy consumption will surely lead to a smaller carbon footprint.

Having said that, smaller energy consumption will also negatively correlate with the computing costs associated with supercomputers which will in turn make scientific investigations that utilize supercomputers much more accessible.

6. ETHICAL ISSUES

Because of the nature of the project, no ethical issues are associated with this particular project.

7. PROJECT MANAGEMENT

In our Proposal Report in October 2022, we set our objectives as follows:

- ➔ Extraction of data from a standalone Linux server and an HPC cluster
- ➔ Visualization of the extracted data
- ➔ Usage of the data to train machine learning models to detect anomalies, and performance characteristics, and provide insights for better performance and energy efficiency
- ➔ Documentation of the tool and visual animation to provide insight into its inner workings of it.

During the design and implementation phases of the project, we faced some challenges in installing and setting up the SuperTwin project. At first, we started with different operating systems, Ubuntu, MacOS, and Manjaro. However, we all switched to Ubuntu for its ease of use and the availability of extensive documentation especially in case of bugs/problems.

We managed to complete $\frac{3}{4}$ of our objectives given above. We extracted data during the SparseBase reordering tests with 8 different ordering types and 2 different SpMV product algorithms and later, visualized this data using SuperTwin by generating detailed dashboards on Grafana. Also, we provided documentation for the SuperTwin project on ReadTheDocs explaining the necessary requirements, versions and commands, demonstrating two tutorials on *How to create dashboards on Grafana?* and *How to create a SuperTwin instance?*. However, we

could not accomplish our third objective which is training ML models to detect anomalies and to create performance/energy analysis.

ENS491/492 course helped us to improve our soft skills such as effective communication among the team members, goal setting and planning, collaboration and task distribution, adaptability and problem-solving, time management, and documentation. Furthermore, the course also provided us with an opportunity to enhance our hard skills in C++ programming and command-line operations in Linux-like operating systems.

8. CONCLUSION AND FUTURE WORK

Future work includes error-investigation such as analyzing the root of the issue that is causing glitches in large matrices. The gathered data can then be used to train a machine learning model where the model can predict performance anomalies dynamically. We have also concluded that the quality of the code base can surely be improved by editing Python scripts so that they conform to the PEP8 conventions, adding docstrings, changing function signatures, and adding an automated build tool to auto-generate documentation.

Since the code has not been tested using software testing conventions, unit tests can be created for code verification. Furthermore, since a remote repository is in use, a continuous integration pipeline can be created for regression testing. Further actions in this direction such as utilizing state-of-art code testing techniques such as mutation testing or formal verification can also be used.

Additionally, we were tasked with investigating and possibly modifying the table time values normalization function. Currently, this function creates a copy of the original data tables belonging to each ordering method, modifies their time values to start from the same point, and then issues memory access to overwrite the original data tables. The issue with this approach is that all of the values are rewritten again, which causes a massive overhead. In order to circumvent this, we propose several alternative methods:

- Call this function in a separate thread. Since the subsequent computations do not depend on the value written to the database, the program execution can continue within the confinements of the main thread and join the thread when the values are to be played. Since the database is accessed exactly once during the full execution, this does not lead to concurrency problems.
- Another approach is to not change the values in the database but normalize the timestamps before uploading them to Grafana. Thus, the database is not overwritten unnecessarily.
- Changing the version of InfluxDB api (2.x) used can also remedy the problem since newer versions allow partial modification of the tables in the database.
- The last alternative approach we propose is to modify the InfluxDB table creation sequence. Currently, the measurements are fetched and written to influxDB directly but we can instead insert a piece of code to this process that normalizes the data points and then writes to the database, allowing for only a single modification to be needed for this process. We propose a pseudo-code to handle this process in Algorithm 3.

Algorithm 3: Normalizing timestamps before table insertion

```
if (firstDataPoint?)
    {time = now}
else {##table initialization sequence (pre-processing)
    minTime = earliestTimeStampFromDB
    diff = now - minTime

    ##table data writing sequence
    while(dataToProcess == true) {myTable[i].time = now - diff}
}
```

9. APPENDIX

1) Merge-based SPMV on cant.mtx

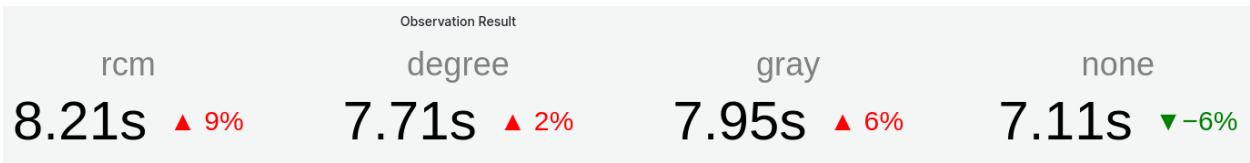


Figure 1.1: Time taken during Merge-Base SPMV with different orderings

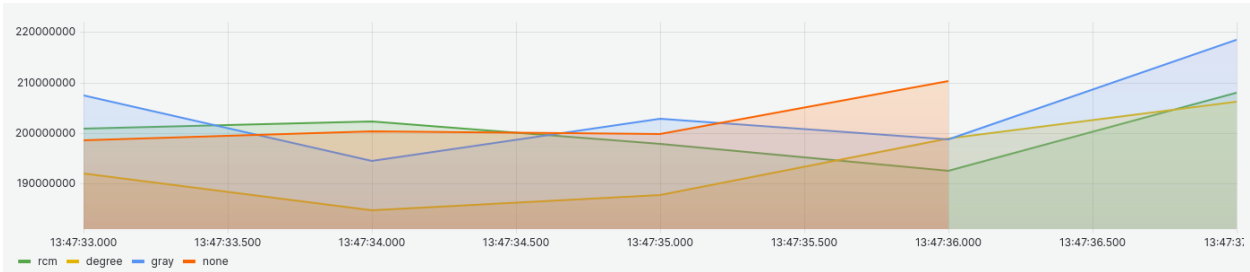


Figure 1.2: L1D:REPLACEMENT

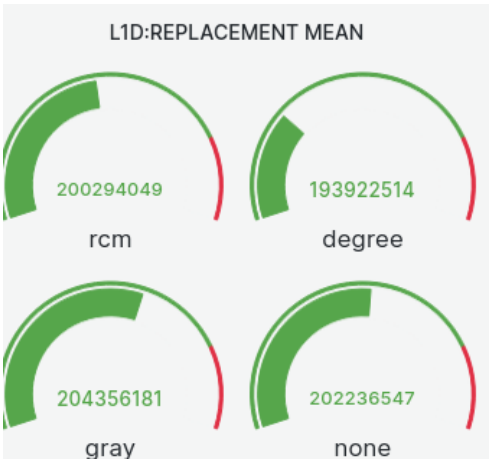


Figure 1.2.1: L1D:REPLACEMENT Mean

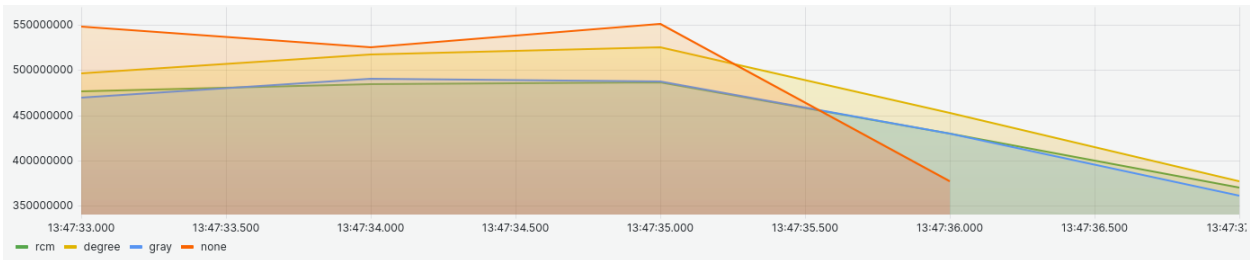


Figure 1.3: MEM_UOPS_RETIRED:ALL_LOADS

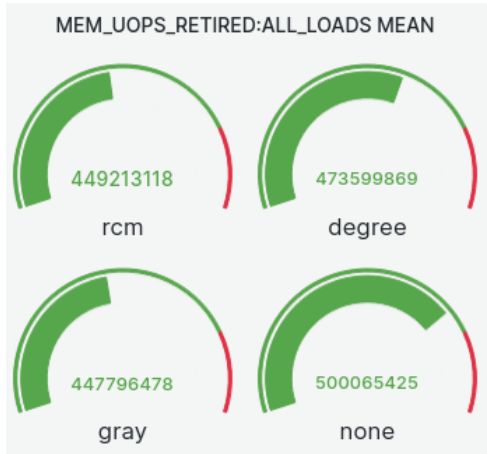


Figure 1.3.1: MEM_UOPS_RETIRED:ALL_LOADS Mean

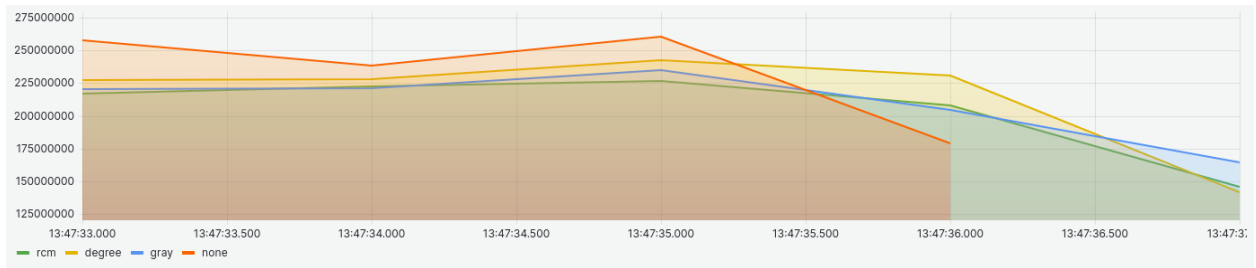


Figure 1.4: MEM_UOPS_RETIRED:ALL_STORES

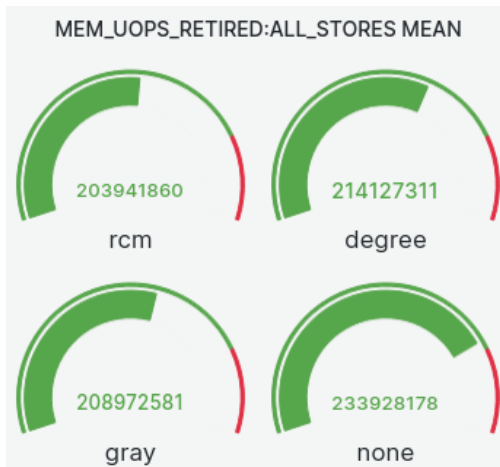


Figure 1.4.1: MEM_UOPS_RETIRED:ALL_STORES Mean

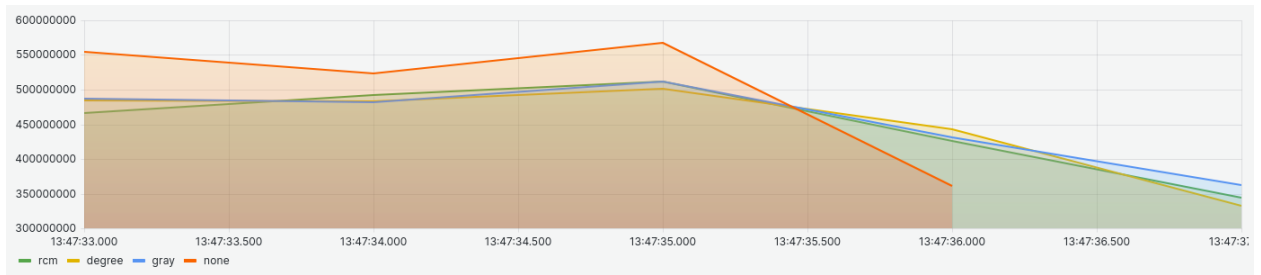


Figure 1.5: MEM_LOAD_RETIRED:L1_HIT

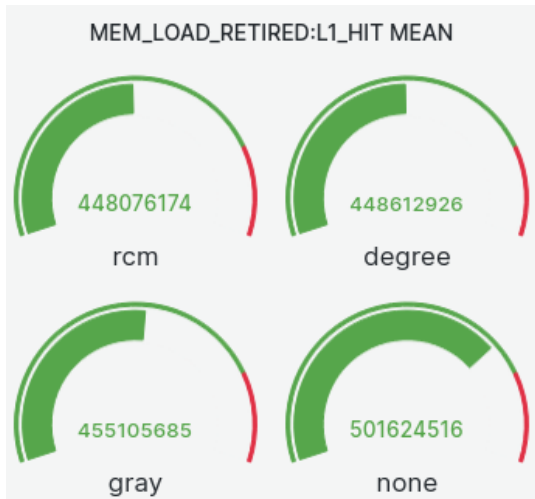


Figure 1.5.1: MEM_LOAD_RETIRED:L1_HIT Mean

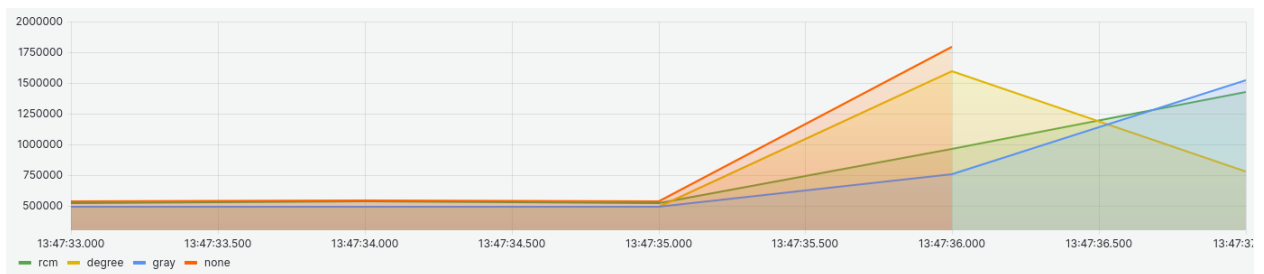


Figure 1.6: MEM_LOAD_RETIRED:L2_HIT

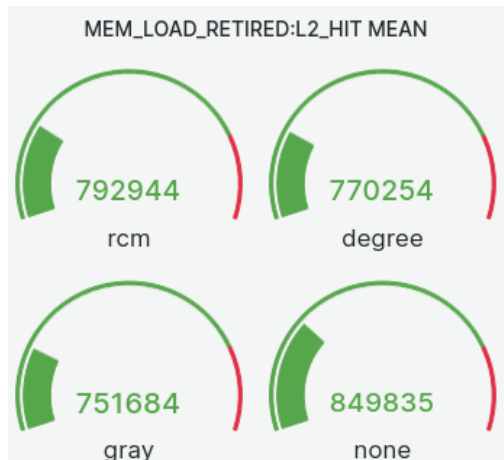


Figure 1.6.1: MEM_LOAD_RETIRED:L2_HIT Mean

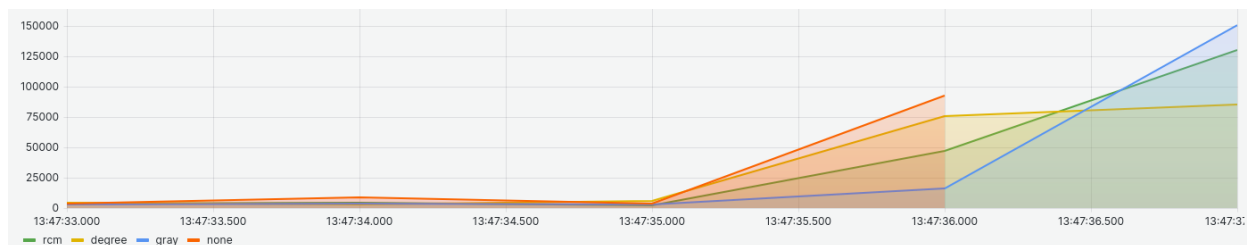


Figure 1.7: MEM_LOAD_RETIRED:L3_HIT

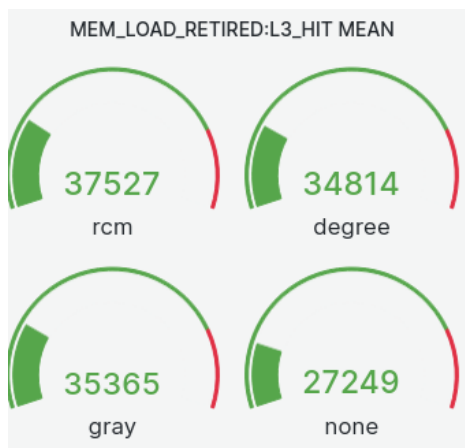


Figure 1.7.1: MEM_LOAD_RETIRED:L3_HIT Mean

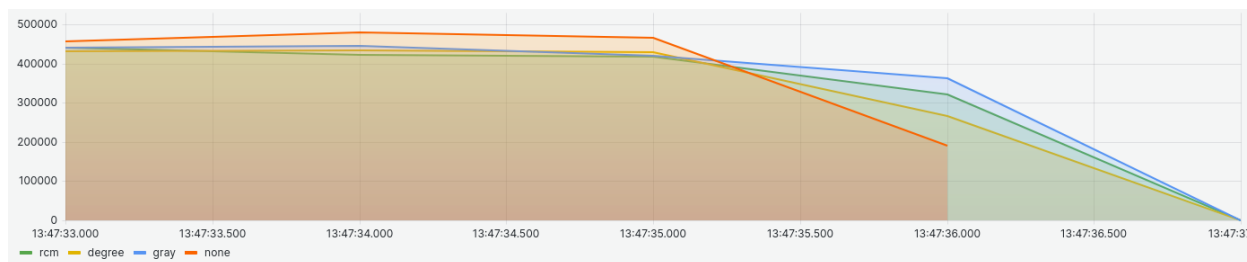


Figure 1.9: FP_ARITH:SCALAR_DOUBLE

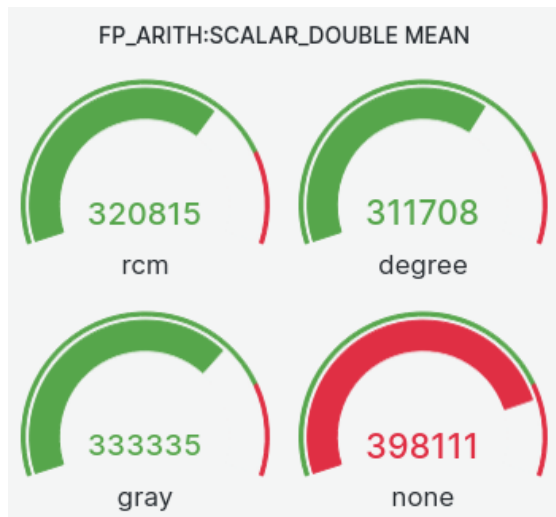


Figure 1.9.1: FP_ARITH:SCALAR_DOUBLE Mean

2) Simple SPMV on cant.mtx

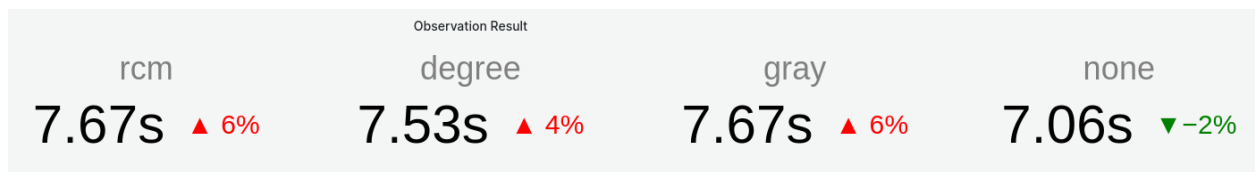


Figure 2.1: Time taken during simple SPMV with different orderings

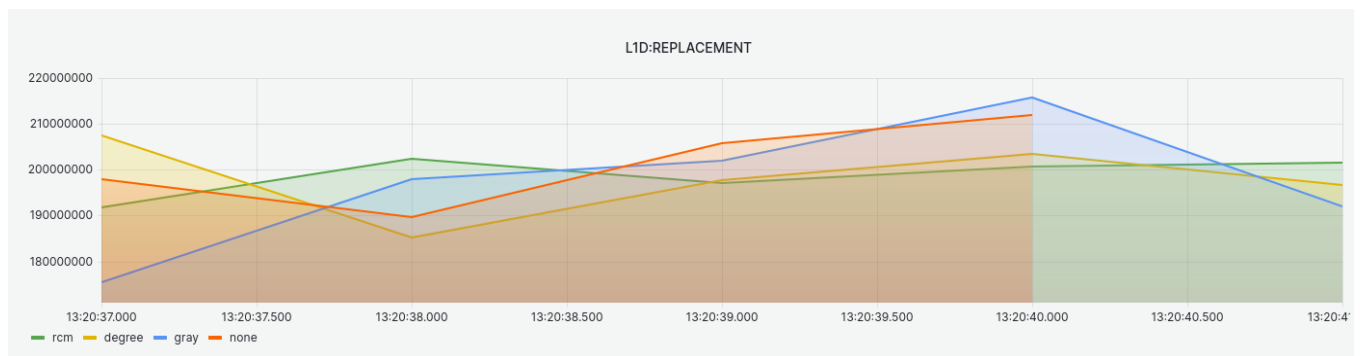


Figure 2.2: L1D:REPLACEMENT

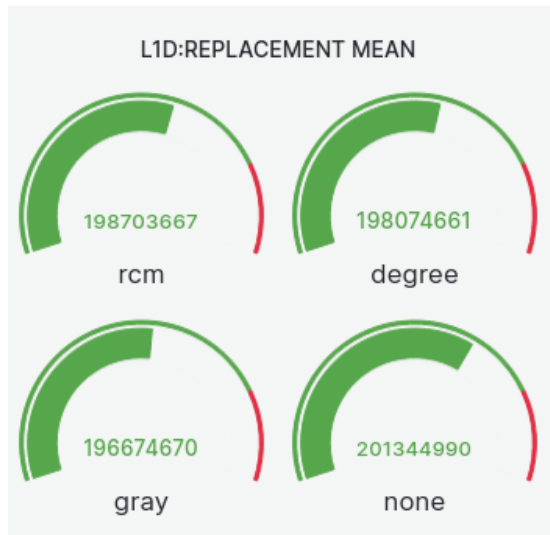


Figure 2.2.1: L1D:REPLACEMENT Mean

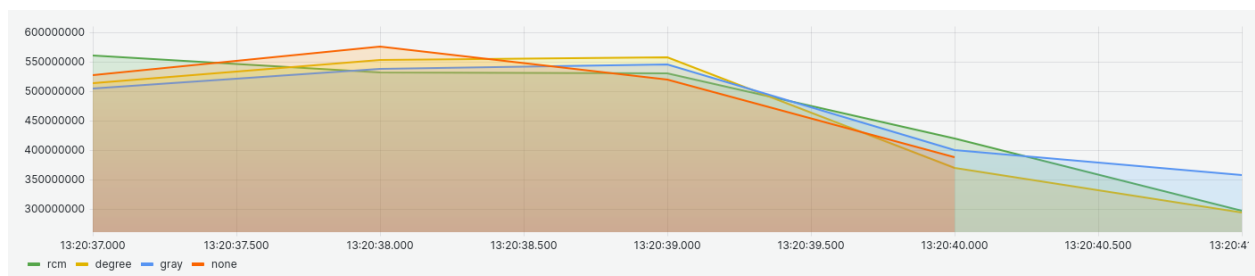


Figure 2.3: MEM_UOPS:RETIRED:ALL_LOADS

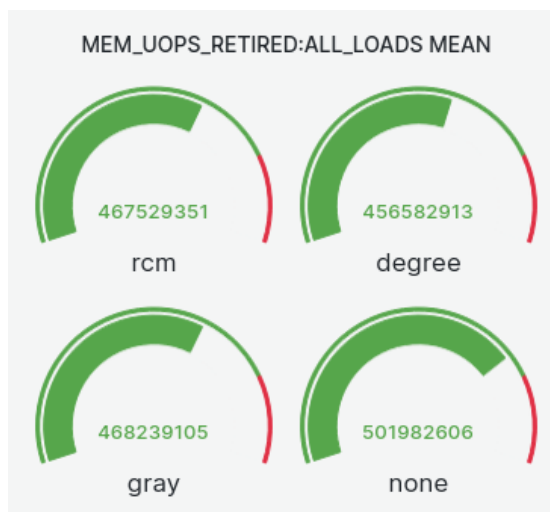


Figure 2.3.1: MEM_UOPS:RETIRED:ALL_LOADS Mean

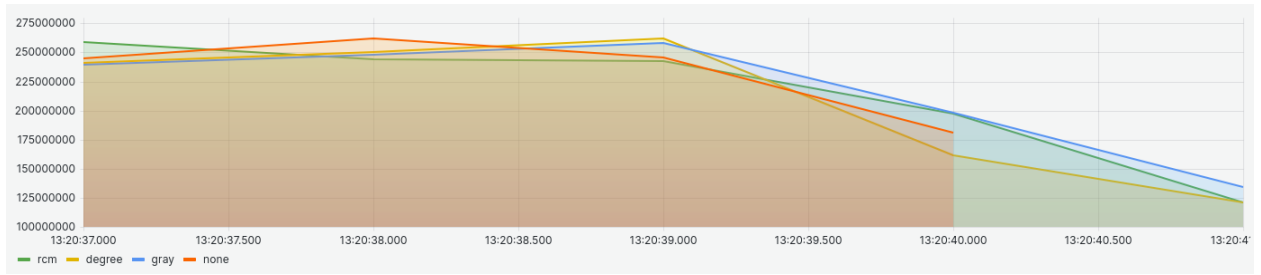


Figure 2.4: MEM_UOPS_RETIRED:ALL_STORES

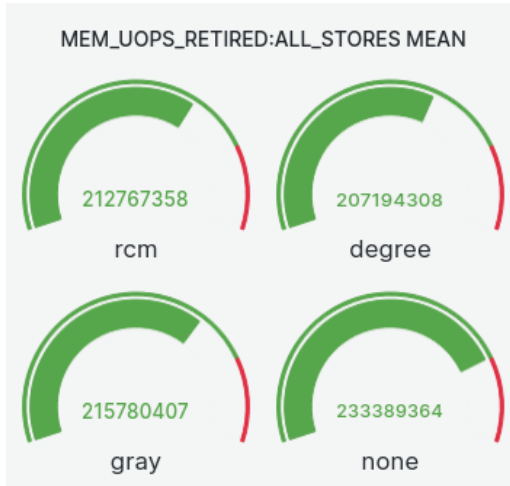


Figure 2.4.1: MEM_UOPS_RETIRED:ALL_STORES Mean

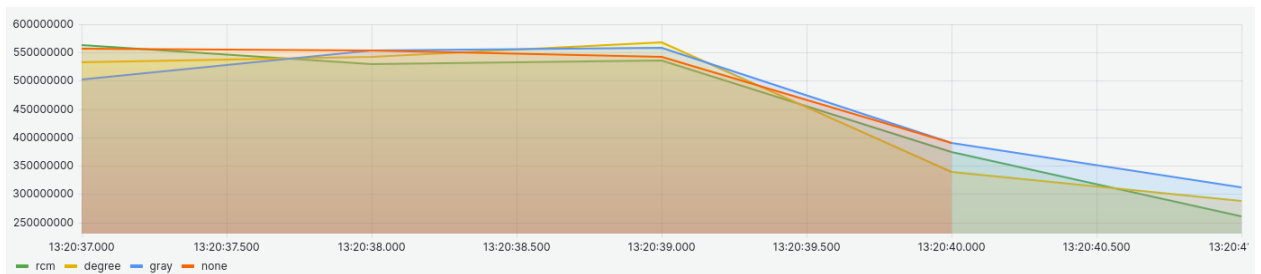


Figure 2.5: MEM_LOAD_RETIRED:LI_HIT

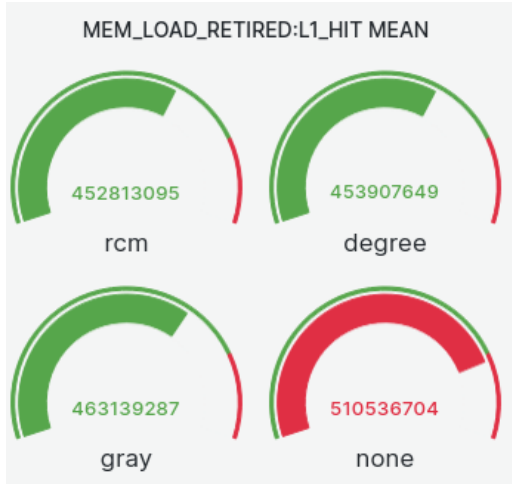


Figure 2.5.1: MEM_LOAD_RETIRED:L1_HIT Mean

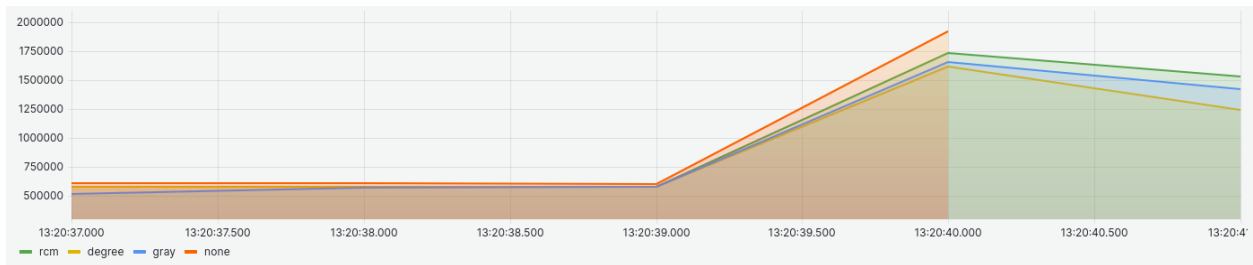


Figure 2.6: MEM_LOAD_RETIRED:L2_HIT

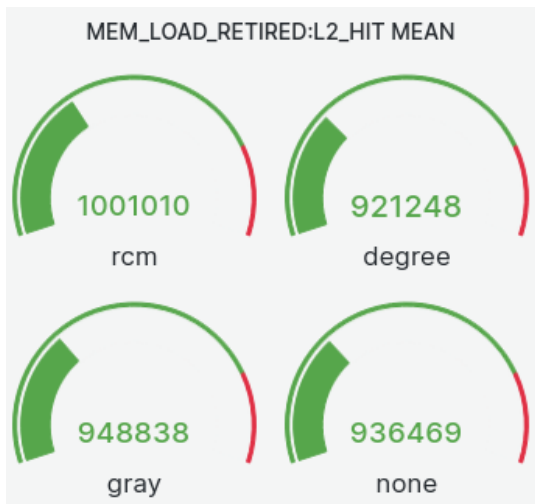


Figure 2.6.1: MEM_LOAD_RETIRED:L2_HIT Mean

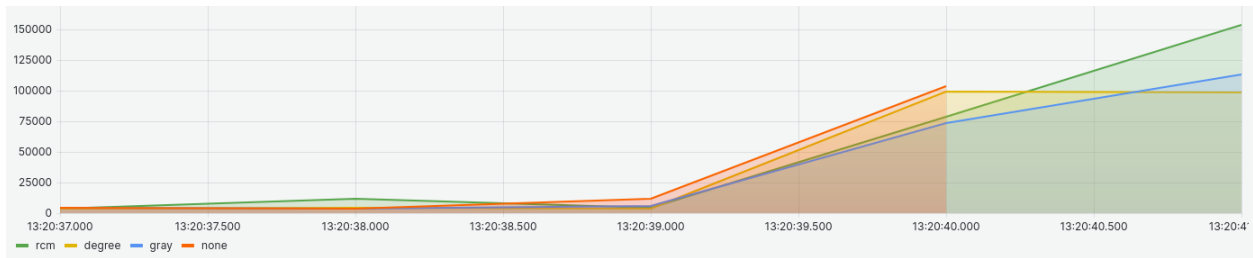


Figure 2.7: MEM_LOAD_RETIRED_L3_HIT

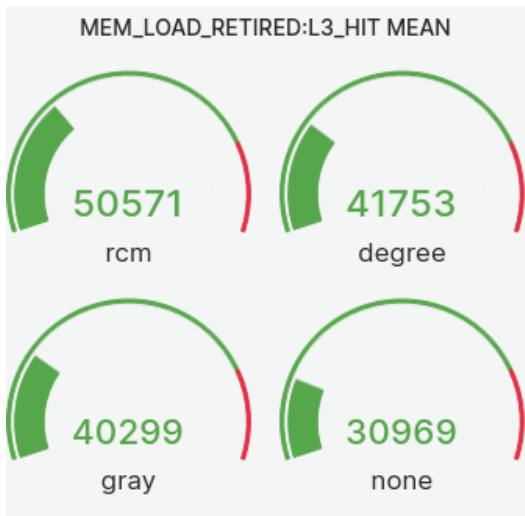


Figure 2.7.1: MEM_LOAD_RETIRED_L3_HIT Mean

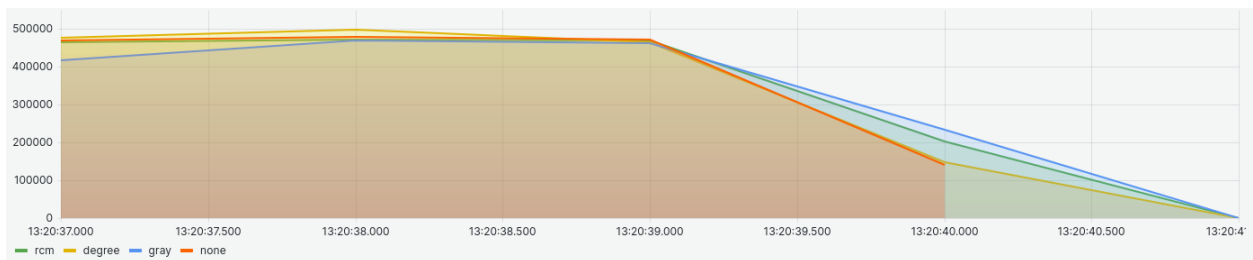


Figure 2.9: FP_ARITH:SCALAR_DOUBLE

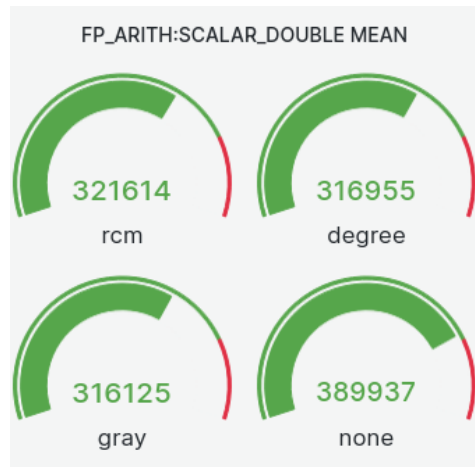


Figure 2.9.1: FP_ARITH:SCALAR_DOUBLE Mean

10. REFERENCES

- H. Bian, J. Huang, L. Liu, D. Huang, X. Wang, ALBUS: A method for efficiently processing SpMV using SIMD and Load balancing, *Future Generation Computer Systems*, Volume 116, 2021, pp. 371-392.
- Merrill D., Garland M., (2016). Merge-based Parallel Sparse Matrix-Vector Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press
- SparCity (2021-2022). Deliverables. Taken from <https://sparcity.eu/index.php/deliverables/deliverables/> on 08/01/2023.
- Yıldız, A., Durgut, E. C., Çağıltay, B. (2022). *Real-Time Super Computer Monitoring, Progress Report II*.