UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Efficient power, performance and thermal aware strategies over heterogeneous platforms

A Master's Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya by

Ignacio Penas Fernandez

Advisor: Marina Zapater
UPC Advisor:Josep Pegueroles

Barcelona, July 2019

Contents

List of Figures

Listings

List of Tables

Abbreviations

EPFL École Polytechnique Fédérale de Lausanne

GN General purpose Node

GRM Global Resource Manager

HN Heterogeneous Node

HPC High Performance Computing

LRM Local Resource Manager

MANGO Manycore Architectures for Next-Generation HPC computing

ML Machine Learning

PoliMi Politecnico di Milano

SLURM Simple Linux Utility for Resource Management

UPV Universitat Politècnica de València

Abstract

Resource Management is a widely studied field in computer science and of utmost importance for the adequate operation of data center infrastructures. Efficient resource management policies enable to improve the energy consumption of these facilities, thus reducing operational costs. Furthermore, in High Performance Computing (HPC) environment, as is the case of the MANGO H2020 project, allow to improve performance and execution time of applications.

The main objective of this project is the design, implementation and test of a resource manager able to allocate incoming applications to the different servers of the data center, while providing the necessary tools to deploy power, performance and thermal aware policies over an heterogeneous cluster. This cluster, will be composed by regular Intel based servers and FPGA based accelerators. The resource manager will work as a single entry point for all the applications involved in MANGO project.

By the end of the project, we have shown how applying simple yet effective allocation policies without controlling fine grain accelerators and with and overview of the system it is possible to improve performance by 10% by lowering power and temperature and reducing the above mentioned operational costs. The resource management tool developed in this MSc thesis has been deployed in a real prototype infrastructure composed by 8 and 128 FPGAs.

1 Introduction

This MSc thesis consists in the development of a resource manager for the MANGO H2020 European project and the research of the most suitable policies for the resources available. Adequate resource management is a requirement in any type of cluster, even more important in our particular case as we are working with heterogeneous resources. This work consists on the development and deployment of a resource manager that allocates and the executes applications in different nodes considering optimal policies. Also, for this project, the resource manager works as single entry point.

Due to limitations of the project, such as data availability, for this work four policies are tested and implemented, fully distributed, greedy, temperature aware distributed and temperature aware greedy.

For our project resource management consists in a dynamic combination of a scheduler/workload manager and a data provider that retrieves data from the nodes. The workload manager allocates applications considering the data retrieved from the nodes and a given policy, accepts applications as entry point and schedules the applications for its execution or queues them depending on the resource availability and the policy constraints.

The resource manager can be described as an integration among multiple different services aiming to create a Global Resource Manager (GRM) able to handle and oversee an heterogeneous data center. As main Manager tool we have implemented a dockerized version of SLURM. SLURM is a well known open source software that works in a manager-slave manner.

1.1 Motivation

Data center energy consumption is dramatically increasing, world wide is expected to reach 20% by 2025. In addition, the next step of High Performance Computing (HPC) is about to reach exascale. Exascale computing aims at developing computer systems capable of supplying exaflops. This implies a huge increase in the energy consumption (around 20-30MW per computer system [1]).

The Horizon 2020 MANGO project [2] aims at exploring deeply heterogeneous accelerators for use in High-Performance Computing systems running multiple applications with different Quality of Service (QoS) levels. The main goal of the project is to exploit customization to adapt computing resources to reach the desired QoS" [3]

MANGO project implies a group of coordinated software, the necessary data required by the policies we are aiming to implement is not accessible in a common way.

As explained in next sections, apart from the data provided by the servers (as CPU usage, temperature, etc.) accurate information of the hardware nodes is provided by the tool called BarbequeRTRM via subscription server.

1.2 Objectives

The objectives of this project are listed as follows:

- Design and implementation of a GRM that works as entry point of the heterogeneous data center and facilitates the implementation of temperature, power and performance-aware policies.
- Integration of the GRM within the MANGO components including the deployment and testing on the MANGO prototype containing eight servers and sixteen motherboards equiped with 4 FPGAs that will be used as accelerators.
- Development, testing and integration of four policies to the thermal behaviour and performance of the prototype cluster and will serve to showcase the capabilities of the MANGO final demonstrator.

1.3 Previous Work

We can find examples of previous heterogeneous clustering on High Performance Computing as could be the combination of common servers with graphic cards for Machine Learning training or crypto currency mining for example. Other examples could be the clusters which combine edge devices (as Raspberry pi) with HPC servers.

There are also running data centers which combine and offers FPGA and CortexV7 processors as an alternative to the normal servers. This offers the customer the possibility to build its own specialized accelerators for and specific application. In this scope is located MANGO project aiming on new technologies such us custom application accelerators.

1.4 Thesis organization

The rest of the document is organized as follows. Chapter 2 describes the main features of MANGO project such as the project objectives, architecture and main components relevant to this project. Subsequently, in Chapter 3 we depict the full architecture of the Global Resource Manager. Chapter 4 explains the integration process of the GRM within the MANGO prototype and the features that made it possible. The last part of the GRM that is detailed in chapter 6 involves both the policies applied and the developed tools to allow the execution of those policies. Finally, chapters 8 and 9 close the project by summarizing the results of the projects and aiming new possible work.

2 The MANGO European Project

MANGO is an EU founded project which main objective is the exploration of new HPC architectures always aiming the 3P standards (Power, Performance, Predictability). The project focus on the fact that the most efficient, both power and performance, execution of an application is adapting the hardware to its specific requirements (Image processing hardware requisites are different than code error correction applications).

2.1 Overview

As mentioned before, due to the impulse for exascale one of the most urgent problems in HPC is the performance-power balance. In this scenario, manufacturing a processing unit for each application would provide the most efficient execution in terms of power and performance while in terms of material resource would be absolutely ridiculous. MANGO will focus on exploiting the current unstudied field of specific hardware accelerators aiming QoS and power-performance efficiency.

The project architecture is composed by both HPC servers and hardware accelerators. In MANGO terms, HPC servers are called as GN, General purpose Nodes, and the hardware accelerators, HN (Heterogeneous node).

The main partners of MANGO consortium are a combination of academic institutions and companies. Universidad Politécnica de Valencia, aside from being in charge of the project coordination, develops the architecture target of the hardware nodes. Those hardware nodes are implemented in FPGA's provided by ProDesign Electronics meanwhile the cluster prototype is integrated at EATON data center in *"Le Lieu"*. Software stack, which is in charge of integrating the hardware accelerators and the HPC servers (detailed in Section 2.3), is coordinated by Politecnico di Milano. Finally, in the MANGO architecture point of view, École Polytechnique Fédérale de Lausanne (EPFL) develops the Global Resource Manager of the cluster.

Regarding the hardware accelerators, despite the fact that the project includes three different architectures, for this MSc thesis we have only focused efforts on the one developed by UPV mentioned above called PEAK because it was the more stable accelerator at the time this thesis was taking place.

The GRM provides thermal, power and performance management for both hardware and software while also facilitates the validation and testing of the project giving a single entry point for stress testing and policy application on the higher lever. As for MANGO project purposes,

the policies developed are oriented to test and improve the thermal and power behavior while keeping in mind the performance required for a HPC cluster.

2.2 Objectives

Summarizing what we have previously mentioned, we have the following goal as main MANGO project objective:

- Exploration of alternative architectures guided by 3P (Power, Performance and Predictability optimization).

For the development of this MSc Thesis, we focus on the Software goals of the project:

- Adapt programming models and compiler support to the new architectures
- Develop the right resource manager to deal with the system

2.3 Software stack

Mango software architecture works in a hierarchical manner, from fine grain (hardware controllers and drivers) to gross grain (cluster overview power and temperature aware allocators).

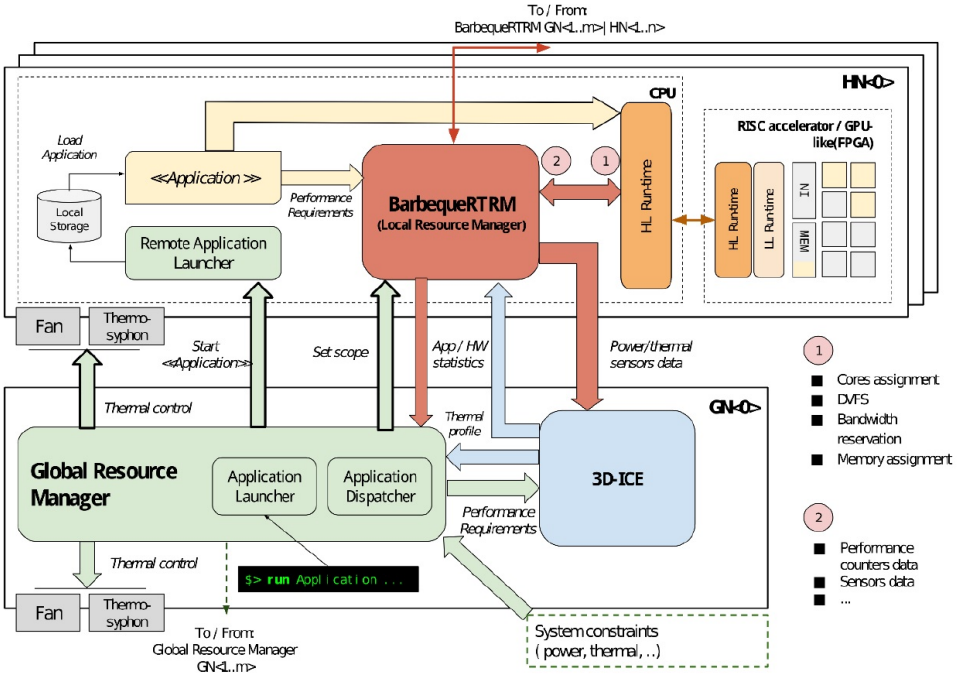


Figure 1: Resource management strategy aimed by MANGO.

Figure 1 shows the software architecture pursued by the project. In this architecture we have two big differentiated layers, the Local Resource Manager (LRM) and the Global Resource Manager (GRM). Moreover, another downer layer is connected to the LRM: MangoLibs.

MangoLibs are the libraries, functions and tools developed on the project that BarbequeRTRM, the Local Resource Manager invokes to communicate with the hardware accelerators.

Back in the big picture, the LRM is installed in every GN and its main function is to perform the resource allocation in each node. The Local Resource Manager makes allocation based on the QoS required by the applications. In short, the Local Resource Manager is in charge of assigning the available resources at node level.

The Local Resource Manager (LRM, also called BarbaqueRTRM), in charge of allocating resources to incoming applications at node level, grants a QoS allocation policy. Thus, each application, besides the binaries needs to be executed, must be parsed with a file (called *recipe*) in which it specifies its requirements. The main requirements possibilities are shown in Appendix B

Finally, at the higher level stands the Global Resource Manager. As this component and its connection with the LRM are the main development work of this MSc thesis, are widely explain in Section 3 and 4. As brief introduction, the GRM main objectives, which were also mentioned above, are: providing MANGO prototype a single entry point and manage the global available resources of the cluster in other to apply policies constraint to 3P standards.

All the data provided in this section show how the integration process of this stack is a challenging task because the components are not standardized at all. Thus, the last four months have consisted on weekly integration meeting among the partners involving this part of the project (Most times, EPFL, UPV and PoliMi).

2.4 Validation applications

The following adapted applications apart from being part of the MANGO project, are the key point for the validation process. These applications have been migrated from its original compilation process to run in these specific accelerators.

Besides the architecture development, MANGO lays its validation in three main applications. These applications consistently focus its implementation on guarantee QoS and versatility to offer the different accelators a way to test and validate its development.

- Online video transcoding
- Volume rendering for medical imaging
- Error correcting codes in communications

For this thesis porpoises the only functional application was the one performing volume rendering for medical imaging. This application is generated by Philips bio-engineering section. The main use of this application is for the processing of the images obtained by the different equipment available as MRI (Magnetic Resonance Image), CT (Computed Tomography), X-Ray, etc.

Before, it was mentioned the application recipe (Example in Appendix B.1) as the tool for the LRM to understand the requirements of the each specific application. If we compare recipes

from other applications, as for example Appendix B we can see the different requirements. Philips application is prepared to be executed on CPU, PEAK(HN) or Nu+(HN) while online video transcoding is only oriented to be executed on CPU or Tesla(HN) accelerators.

2.5 The MANGO prototype

This section presents the detailed hardware available for prototyping while also describes the hardware (not only hardware accelerators of the projects but all the components) architecture of the system.

Although the initial project proposal included sixteen common HPC servers at the end, due to space and budget limitations the prototype in which this project is developed and tested contained eight servers.

The main components of the prototype are depicted in table 1u:

			# of component per motherboard		
Setup	#	#Servers per setup	#Motherboards	#FPGA	#Others
1	4	1	4	12	4
2	2	1	4	8	8
3	2	1	4	6	10
Total		8	32	26	22

Table 1: Setup provided for the final validation of MANGO project

Each ProDesign motherboard has space for four hardware components with a specific shocket, FPGA or ARM accelerators. The motherboards are the component that interconnect the hardware accelerators with the servers. The composition of the many core architecture will be 16 hardware components per server which means that every servers must be capable of executing applications and parsing data from two different motherboards.

Figure 2 shows part of the real prototype which is located in *"Le Lieu"*, Switzerland[1]. Figure 2a on the left represents a cluster of 4 ProFPGA motherboards each one containing 4 FPGAs. On the right, Figure 2b is the rack of servers connected to the motherboards via PCI.

[1]Images from MANGO deliverable 5.4

(a) Cluster of 4 motherboards representing an HN node placed on a rack

(b) Rack of servers with the corresponding PCI connections for the motherboards, each server is a GN node

Figure 2: Example of the real cluster infrastructure in *"Le Lieu"*

3 Global Resource Manager (GRM) Architecture

3.1 Overview & Objectives

As the global resource manager is the higher-level software component of MANGO software stack, its main objective is to perform the assignment of incoming workload to the different node of the cluster. Therefore, it needs to be granted complete full access to the cluster resources in MANGO. This means that the global resource manager will not decide in which specific FPGA(HN) or CPU(GN) the program will be run but the node.

The first objective of the global resource manager would be to choose in which node the requested application must be executed. This decision is made considering the overall image of the total resources and their current availability. To this end, the GRM interacts with each of the instances of the LRM (BBQ) running in one of the Intel cores (at the server-side) of each of the clusters. Because of its hierarchical construction, as the global resource manager is the top-level service of the cluster, it will need to have visibility and full connectivity with all the GNs (and in particular the instances of BBQ RTRM running on the GNs) of the whole MANGO platform. However, it does not require access to the HNs. Only knowing their availability and load will suffice, as the LRM will be the one in charge or running the application on the HNs. Also, attached to the application we could find the recipe of its requirements which will also be necessary for the allocation policy.

The second objective, as stated for MANGO project, is to have a single entry-point for the cluster which would be capable of scheduling applications. However, having a single entry-point means facing a potential single failure point and, therefore, reliability and replicability must be granted in order to prevent the system from fatal failures which will prevent all the cluster for working.

Overview

Resource management at the global level is a challenging task, especially for a scenario with the heterogeneity degree and the power/performance/predictability requirements of MANGO. The role of the GRM in MANGO will be to act as a single-entry point for all the MANGO applications, and to decide on runtime the most appropriate node, among the 8 MANGO servers, to execute each new incoming application. This decision is made by considering the power/thermal status, performance capabilities and utilization of the overall MANGO platform, as well as of the individual clusters. Moreover, the Global Resource Manager is in charge of interacting

with each of the instances of the Local Resource Manager (BarbequeRTRM), which will be running in one of the Intel cores (at the server-side) of each of the clusters.

From a functionality-wise perspective, the GRM is be composed of the following main components:

- The resource manager software, which is in charge of managing incoming workloads, scheduling (i.e., queuing them) and allocating them to the nodes. In our case, this workload allocator is the SLURM resource manager. SLURM is complemented with the rest of the services developed in the MANGO GRM to adapt its functionality to the heterogeneity of the MANGO prototype.
- Data manager and data retrieval services composing all the chain from the connection with the local resource manager to the injection on the data base which is deeply explained on section 4.
- The workload allocation policies (in what follows, ”GRM Allocator”), which take decisions on the specific allocation of tasks to nodes. The GRM allocator contains the power/ performance/thermal- aware policies, which are in charge of improving the energy efficiency and performance of the system, described in section 5.
- The 3D-ICE thermal simulator, in charge of predicting the thermal behaviour of the system

From a purely implementation perspective, the GRM consists on a bunch of services working together in a coordinated way. Figure 3 shows an overview of the GRM global structure. The core functionality of the GRM is the ”GRM Allocator”, which takes decisions upon how and where to allocate a workload (i.e., contains policies and algorihtms), and the ”Slurm Controller” block, performs the allocation (i.e., contains the software in charge of managing the workload).

Therefore, when a new application is launched in the cluster via the entry point, the GRM Allocator will apply one upon the various policies developed ad-hoc for the MANGO project, which will decide the node where the application will be executed, and will finally perform the allocation, by executing the SLURM Controller (in particular the slurmctld), which will, in its turn, using the slurmd of the selected node, assign the task to a node and pass its control to BBQ. This flow is depicted in Figure 5.

3.2 Global Resource Manager Software Architecture

To facilitate deployment and in order to achieve the objective of replicability and robustness every service on the GRM will be running under docker containers[2]. By using Docker containers we are also reducing the overhead of software installation in the host machines that are needed in the Global Resource Manager. In all the Figures that follow in the present document, Docker container will be represented as boxed with rounded corners, such as the ones depicted in Figure 3, whereas native software components are represented as squared boxes.

[2]https://www.docker.com/

We modify SLURM to equip it with a number of features specifically targeted to the MANGO platform. As a base installation, we used an already dockerized version of SLURM, which is available on [4]. In this base docker installation, we can find a simulated cluster prepared to be executed in only one host. However, for the MANGO deployment we need to manage a real cluster with multiple nodes. Therefore, the following modifications were applied in order to adapt the base dockerized SLURM to the MANGO requirements:

- Updating SLURM to the latest available version
- Changing SLURM compilation to force it use an external allocator that will be capable of managing the MANGO heterogeneous cluster.
- Changing the deployment file named "docker-compose.yaml" as follows:
  - SLURMCTL container running on GN0
  - SLURMDBD container running on GN0
  - MySQL container running on GN0
  - SLURMD container running on GN0-15

The "Kafka server" was taken from Apache Licensed Spotify docker repository[3].

[3] https://hub.docker.com/r/spotify/kafka/

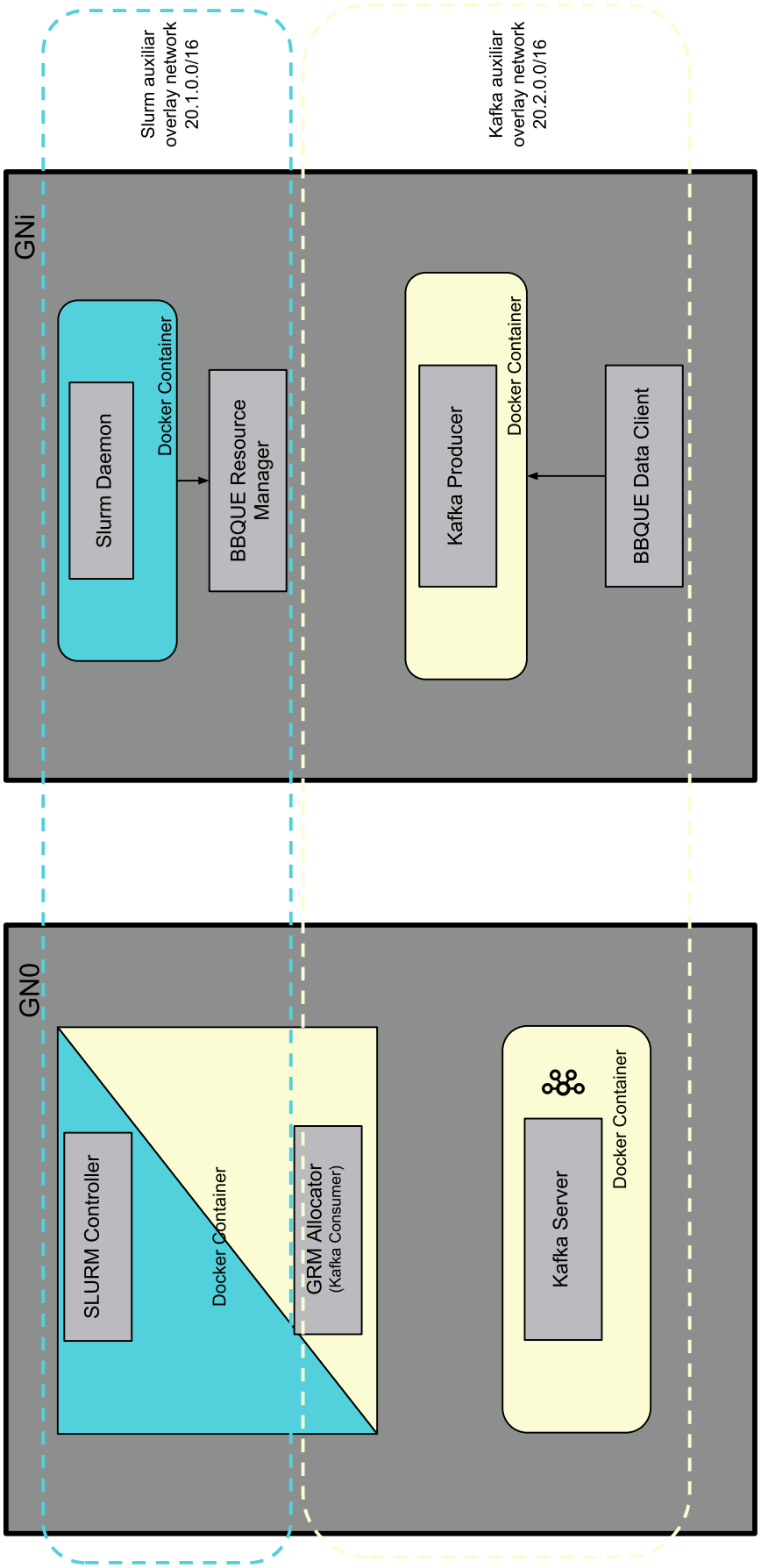


Figure 3: Global Resource Manager Overview

3.3 The SLURM Resource Manager tool

As described in the previous section one of the components of the GRM is the workload scheduler and allocator. Indeed, this is the most important component as it will allow to achieve the objective of having a single entry-point, and will also enable full control on the application execution status. SLURM was selected to accomplish this purpose as it is a highly scalable cluster management tool. However, because it is a general-purpose HPC cluster management tool, it does not provide the heterogeneity-awareness required by MANGO. However, by implementing the interaction between SLURM and BBQ, we can enable the hierarchical approach and integration between GRM and LRM. Furthermore, because of its open-source nature, SLURM can be easily modified to suit the requirements of MANGO, without requiring any kernel modification on the host system. Moreover, it enables replication and fault-tolerance, therefore being very suitable for single-entry point systems.

Regarding the architecture, SLURM works in a centralized way having a central manager called "slurmctld" (SLURM controller) responsible of monitoring the status of applications and the resource availability. On the other hand, each node will be running a SLURM daemon, or "slurmd", which supervises applications running on each node and reports its status. In short, SLURM daemons work as a remote shell for the controller. Moreover, all the information gathered by SLURM, can be stored in a MySQL database, managed by the "slurmdbd" daemon, which runs in the controller and records accounting information, historical data and current status of the nodes, among others.

An overview of how SLURM works is depicted in Figure 4, taken from the SLURM official documentation.

Regarding allocation, SLURM comes with a very basic set of scheduling and allocation policies, which consist on round-robin and "best fit / first fit" algorithms. However, as mentioned before, MANGO requires more sophisticated allocation policies (capable of being aware of the heterogeneous resources) than the ones included in SLURM. Therefore, there is a need to include new policies in SLURM.

Due to the complexity of the MANGO project the fully integrated plugin has not been possible to develop. There are two main reasons; the first is that in order to retrieve data from the LRM it is necessary a client permanently connected to it; and the second, the diversity of accelerators made impossible to integrate a plugin containing all these features. In consequence, we needed an external allocator which was capable of taking into account the diversity at the same time it receives updates from the LRMs. Thus, SLURM will work as log parser and executioner tool as the external allocator will order SLURM to execute an specific application in an specific node. From a purely implementation perspective, we are going to create two different instances, one running SLURM and other one running the external allocator daemon.

Therefore, when a new application is launched in the cluster via the entry point, the external allocator will call the new policy developed ad-hoc for the MANGO project, which will decide the node where the application will be executed. Then, the external allocator will call the SLURM Controller (slurmctld) asigning in the command the selected node where SLURM must execute the parsed application. This flow is depicted in Figure 5.

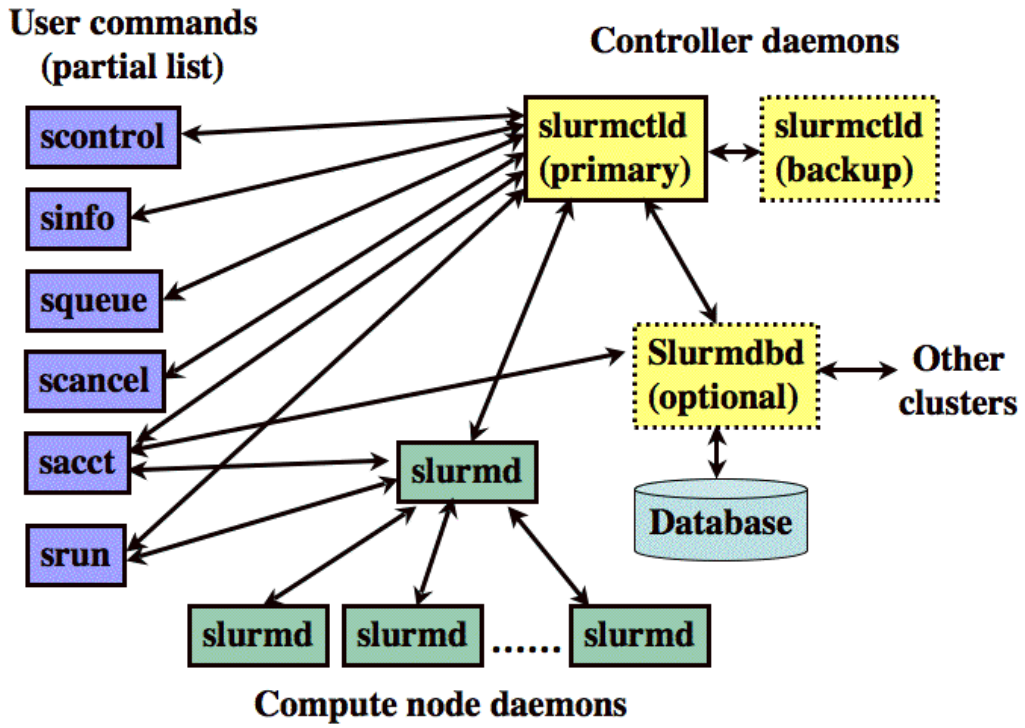


Figure 4: Overview of SLURM basic blocks

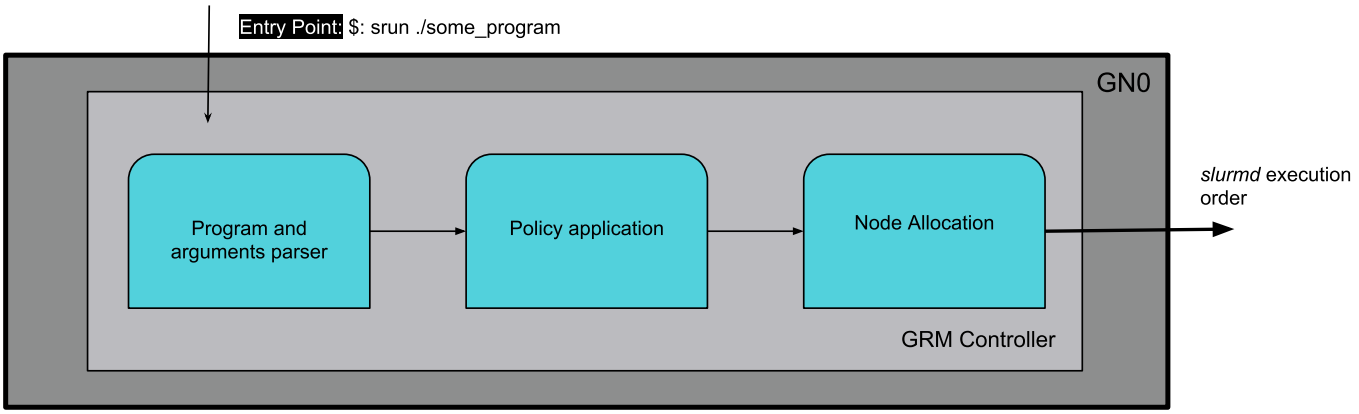


Figure 5: SLURM flow, application execution, policy and node allocation

Therefore, the installation of SLURM will require the installation of the external allocator service adapted to the MANGO architecture. The external allocator will be running in the same container as slurmctl because they need direct access in order to exchange information.

SLURM Basic Commands

This subsection shows the basic commands needed to execute, cancel and monitorize SLURM status.

- Running applications:
  - srun: run a parallel job on the cluster. After running this command, SLURM calls the custom allocator mentioned before where the optimal allocation (regarding temperature, power, performance, etc) is performed. Example:
    - Example: “srun mango-philips.app –node GN1”, would run the MANGO Philips application on GN1
  - Sbatch: subscribes a batch script to SLURM. When the allocation is granted SLURM runs the script once.
- Management:
  - scontrol: monitorizes and allows the modification of SLURM configuration as job, nodes, partition.
  - scancel: signals and stops jobs under SLURM control
- Monitorization:
  - sinfo: displays the current status information of the cluster
  - squeue: shows information of the scheduled jobs

3.4 Docker as connectivity and deployment tool

Docker is a tool we can compare with a virtual machine. But, unlike a virtual machine docker shares the kernel of the host computer making the containers lighter and the deployment much faster. Containers are packages that envelops the code and all its dependencies which can be easily and quickly migrated among computers.

Docker Swarm is a cluster manager tool. In a Docker environment, a Swarm is a cluster composed by Docker daemons which can be virtualized or in actual machines. In our case, each General Purpose node will act as a Node(how Docker Swarm calls the components of the cluster) meanwhile GN0 is the Swarm Master.

Docker Swarm is a tool provided by Docker Engine which is specifically designed for cluster management, fast deployments, decentralized designs and scalability. Even more important for MANGO GRM is the networking capabilities of Docker Swarm as it permits to create overlay networks and then attach the required container to that network. Also, all the containers attached to the same overlay network have full connectivity which means that the port bidding for connections inside the network is not necessary and solves many connectivity issues. This way we are parsing all the traffic that would go through the ports mentioned on 3.4.2 with respect

to SLURM and Kafka on the host machines converge over the ports opened between hosts also described in that section.

As we can see in Figure 3 there is a color differentiation between Docker containers and the dashed boxes. They represent two different overlay networks, one is dedicated to Slurm communications and the other is dedicated to Kafka messaging.

3.4.1 Overlays and IP description

In figure 3 we have a networking overview of both networks. As far as is possible the IP addressing will be static since Kafka server container requires a notification IP which should point its actual IP on the overlay. However, this is not supported yet by the command docker stack deploy we are using to deploy all out services, so Kafka server is the only container which will be manually deployed.

- *slurm_overlay* network is represented by 20.1.0.0/16 where:
  - *slurmctl*: 20.1.0.5
  - *slurmdbd*: 20.1.0.6
  - *slurmd*: 20.1.0.(7-15) nodes from 0-7 respectively
- *kafka_overlay* network is represented by 20.2.0.0/16
  - Kafka server: 20.2.0.100
  - Kafka producers and consumer: auto-assigned

3.4.2 Port Requirements

In this subsection we describe the necessary port connections to establish the connections between docker containers.

- SLURM
  - *slurmctl*: 6817, 6818, 6819, 7199
  - *slurmdbd*: 6819
  - *slurmd*: 6818
- Kafka
  - Producer + BBQUE data client: 9092, 10001, 30100, 30200
  - Consumer: 9092, 7199

Next, we can find the ports that must be opened (only from connections from the 16 nodes) in **All Hosts** to create the Docker Swarm cluster:

- TCP 2377 for cluster deployment

- TCP/UDP 7964 for communication among nodes
- UPD 4789 for overlay network traffic

As stated before, Docker Swarm allows full visibility among containers which means that containers connected to the same overlay network can be accessed by any free port. Thus, the first part of the section is mostly informative, while the second one shows the ports of each server that must be accessible from the master node in order to create a Docker Swarm stack.

3.4.3 Deployment

The deployment of the GRM consists on the following steps:

1. Establish connectivity between hosts by mean of opening the ports mentioned in the section 3.4.2 above under “All Hosts”.
2. Create Swarm stack on the node which will work as master. Remark that is possible to have more than one master node in Swarm. For MANGO project, GN0 is the master. Swarm network is initialized using the following command:

```
$: docker swarm init
```

3. After executing the previous command the following one will prompt which is the command to join the rest all the nodes to Swarm

```
$: docker swarm join --token XXXXXXXX
   192.168.99.121:2377
```

4. Once Swarm is deployed, the next step is creating the overlay netwokrs:

```
$: docker network create -d overlay --attachable --
   subnet=20.1.0.0/16 slurm_overlay

$: docker network create -d overlay --attachable --
   subnet=20.2.0.0/16 kafka_overlay
```

5. Join al the GN’s to the Network File System folder located on the master node.
6. Run under the folder DockerizedSlurm/compose_deploy the following command. This command will deploy all the containers required by the GRM and will connect them as specified in the configuration file *docker-compose.yml* to the networks created before and will attach the corresponding containers to the volume containing the applications and the necessary data.

```
$: docker stack deploy -c docker-compose.yml
   mango_slurm
```

4 Integration of the GRM in the MANGO prototype

This chapter describes research and implementation performed for the integration of the Global Resource Manager within the MANGO software stack and the prototype. There are two will differentiated sections, one in which we treat the problem of executing the MANGO applications from the Master Node (GN0) and a second one where the data parsing from the LRM to the allocator is depicted.

4.1 BarbaqueRTRM-SLURM communication

As SLURM is going to work as Global Resource Manager it will need the complete data of the entire cluster including: mapping and resource usage for each application deployed in the node, utilization and load, power and temperature of the available resources on the HN and GN. Data need to be updated on real time(almost) so the allocator can be more precise otherwise unnecessary queues can be created at the Local Resource Manager.

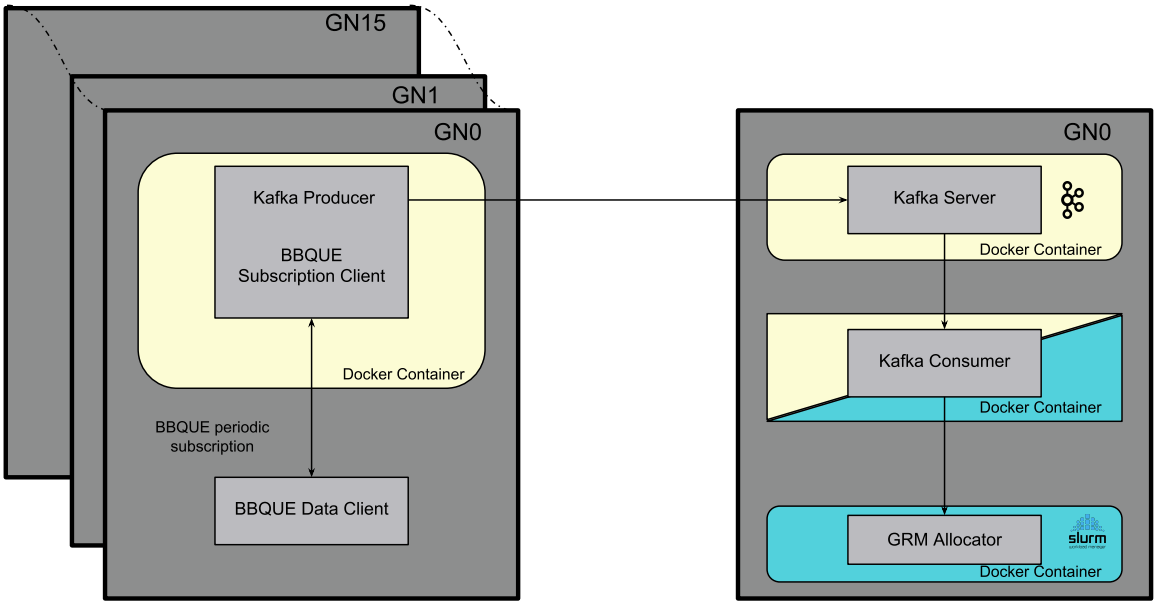


Figure 6: Flow overview, from BarbaqueRTRM to SLURM

Figure 6 illustrates all the components that performs the data gathering service from BarbqueRTRM to SLURM allocator. In this figure, there are two clear differentiations: server and

daemons. Pretty similar to SLURM working architecture. In the server side, which is always allocated in GN0 node we can find a broker and a database (Kafka and the Allocator). This two components work as the main features to retrieve the status of the heterogeneous cluster and present the data to SLURM which having an overview of the system will perform the appropriate allocation.

As BarbequeRTRM data server is already widely explained there is no need to enter in details of its architecture or mechanisms. Only mention that it implements a subscription type server. This server provides the information of the node in which is installed. Due to this fact, was not possible to have only one instance on the main node GN0 as it would need to handle 16 subscriptions at the same time completely preventing the scalability of the cluster and creating a single point of failure. The final adopted solution was to distribute the subscription servers to each node.

The Kafka Broker was chosen as it can handle millions of messages per second and provides a high reliability. Kafka is defined as a real time data Streaming publish/subscribe message broker redesigned as a distributed commit log. The main components of Kafka are messages, topics, Publishers and Subscribers. Topics works as distributors in the broker, so it can be shared among different applications. Messages compose the information transmitted through the broker being the publisher the instance that sends messages into the broker and the consumer the service that is subscribed to a specific topic. Basically, the roles were changed, so the daemons publish messages and the servers subscribe to the topic where those messages are published. In case of necessity Kafka offers an easy replicability.

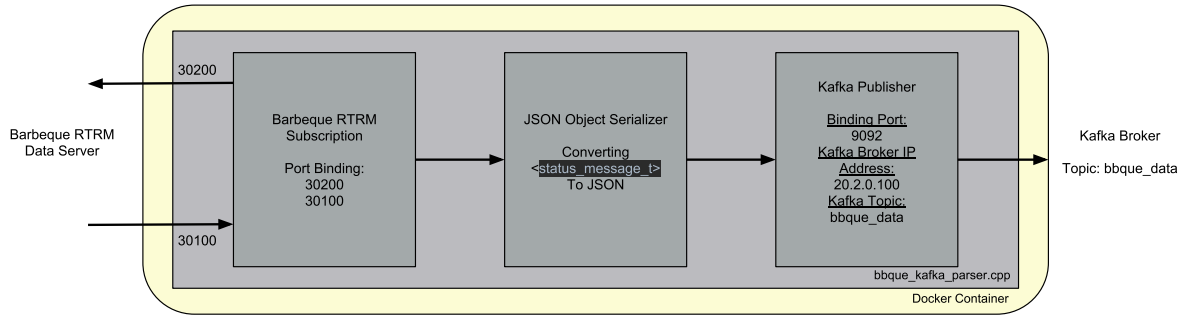


Figure 7: Data Parser work-flow from the Local Resource Manager to Kafka

The daemonized part, illustrated on left GN0 of Figure 6, consists in a container that runs a C program composed by a periodic subscription to BBQUE data client that retrieves the corresponding data of that node, a Kafka Producer in charge of transmitting the information to the server and an object serialization function which translates the object created on the subscription to the client with the status of the node to a more suitable JSON format. The flow is more precisely described in Figure 7

Regarding the Kafka Producer, every time a new message is received from the subscription(5 seconds period), a new publication is pushed by this instance to Kafka Server. As shown in figure 6 both subscriptions are running in the same service avoiding redundancy and the need of developing other service that connects both services.

The last step of this communication flow is the Kafka-consumer inside the external allocator. The external allocator will update the cluster status every time it consumes a message.

4.2 SLURM-BarbequeRTRM communication

This section shows how the global resource manager can send an execution order to the LRM. In short, we need to execute and application on GN0 (single entry-point) where the GRM controller is running, and SLURM with the allocation decision made by the external allocator (depending on the objective of the allocation policy) will be in charge of executing this application.

In order for this flow to work correctly, two issues had to be overcome: (i) how to execute a program in the host computer from a docker container exporting all the libraries required by MANGO apps, and (ii) how to share that program/script to be run among the cluster so that the GRM controller does not need to handle sending the application to the actual node which will run the program.

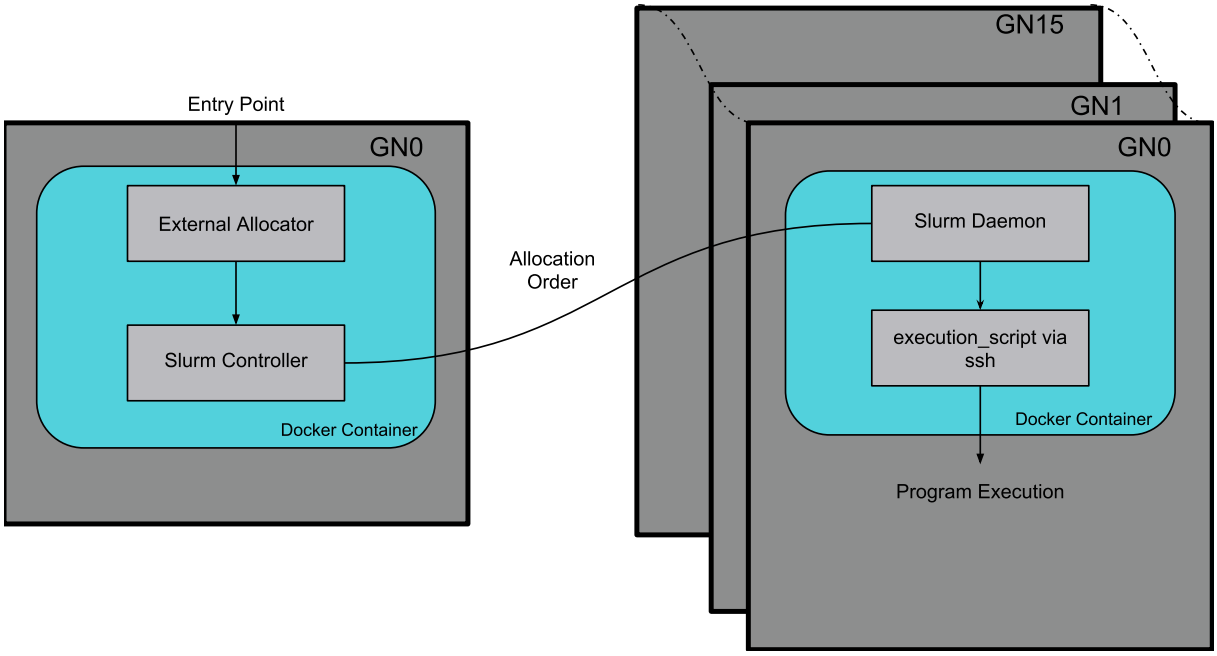


Figure 8: Application execution flow from the single entry-point on GN0 to any Local Resource Manager

In order to accomplish the first issue the architecture illustrated on Figure 8 was developed. Then, with the next command, we can execute an script located in the docker container in the host machinde:

Listing 1: Command for SSH remote script executions

```
$: ssh user@172.17.0.1 'bash -h' < execute_app.sh
```

Thus, this *execute_app.sh* script can contain any export required by the applications.

Given the previous architecture, each application will require two execution scripts which contains all the library exports required by that application, its name, location on the shared folder and some more parameters described below.

In short, the execution process requires two different scripts, one which will run the actual application and another which specifies the SLURM parameters required by that application. Initially, all the applications are executed under the same circumstances for testing and validation purposes.

As mentioned before, the second script, imports all the required MANGO(and external) libraries and simply executes the application as it would be manually done in the command line and after this step is LRM job to locally allocate the application. On the other hand, the first script is in charge of executing the second one via SSH while fixing the execution parameters in SLURM. For example, in Listing 2 we are defining the time limit for the execution of the specific application (predefined to 10 min) and also the location and name of the report of the execution.

The second issue will initially be solved with a shared folder among the cluster, so we then grant every node access to every program avoiding any issue regarding access permissions. To this goal SSHFS (Secure Shell File System) and docker volumes are combined, in consequence, every host which docker daemon is attached to the swarm network and all docker containers with the volume mounted will be able to see and execute that program.

Listing 2: Example *sbatch* execution script

```
#!/bin/bash
#SBATCH --output=/opt/mango-apps/slurm_logs/slurm-%A.out
#SBATCH --time=10:00

if [ "$1" = "matrix" ]
then
    if [ -z "$2" ]
    then
        echo "No matrix size provided, assigning 200"
        matrixsize=200
    else
        if [ $(($2%2)) -eq 0 ]
        then
            matrixsize=$2
        else
            matrixsize=200
        fi
    echo "Running Matrix Multiplication $matrixsize"
    fi
```

```
    cd /opt/mango-apps/matrix
    ssh -oStrictHostKeyChecking=no ipenas@172.17.0.1 'cat |
        bash /dev/stdin' "$matrixsize" < execute_app.sh
fi
```

5 Global Resource Manager Allocator

As previously stated, for the GRM to accomplish the goals of the project we need to apply and test different power/performance/temperature aware workload allocation policies. The ”GRM Allocator” module is in charge of this. Furthermore, all policies need to take into consideration the current usage of the cluster, which requires a close to real time synchronization between local and global RM.

This is of utmost importance in MANGO since, given that the prototype counts with a wide range of heterogeneous resources (regular CPUs, PEAK, Nu+ and custom accelerators such as DCT and Tesla) a node could be unavailable for one application (e.g. that requires one PEAK and one Tesla) but available for another with different requisites (e.g., which requests one Nu+ and a regular CPU). Thus, the global resource manager requires an underlying flexible system that can provide a complete overview of the cluster which, at the same time, can be updated with the current status for each application requirements. To express these dependencies, the GRM Allocator will use a graph network which contains all the available nodes. This support will be implemented by using the Networkx [5] Python library. Networkx provides all the necessary tools to manipulate, study and distribute for complex networks graphs

To perform the above mentioned tasks, the GRM Allocator consists on the following modules, which will be next explained:

- Cluster architecture builder
- Cluster architecture update
- Workload allocation algorithms

5.1 Cluster Architecture Builder

This component is composed by a JSON parser which is configured to read the following architecture configuration file (i.e Listing 3).

The main function of this component is to provide a default network architecture that can be used by other modules. The architecture of the platform managed by the GRM should be provided following the following JSON format so that the json_reader block is able to successfully build the architecture. Then the JSON is converted to python dictionary which facilitates iterating over it.

Listing 3: Example of initial architecture input file

```
{
  "name": "Mango Cluster",
  "version": "1.0.1",
  "master": "gn0",
  "architecture": {
    "gn0": {
      "id": "gn0",
      "master": 1,
      "components": {
        "peak": 0,
        "nu+": 0,
        "tesla": 0
      }
    },
    "gn1": {
      "id": "gn1",
      "master": 0,
      "components": {
        "peak": 0,
        "nu+": 0,
        "tesla": 0
      }
    },
    "gn2": {
      "id": "gn2",
      "master": 0,
      "components": {
        "peak": 4,
        "nu+": 0,
        "tesla": 0
      }
    }
  }
}
```

Listing 3 is a key point of the process. With this initial configuration we provide to the allocator the number of accelerators contained in each node. Also, providing the number of nodes and the name by which they need to be identified. This last feature is critical as both, the name of the GN and the name of the accelerators must match with the names by which the BBQUE-Slurm parser codes the information injected into Kafka. The architecture defined in Listing 3 is transformed into the network represented in Figure 9.

To ensure that the allocation is perform correctly, the *Networkx* object that is instantiated in this step has to be a DiGraph, i.e., a directional graph. *Networkx* networks are composed by two items: nodes and edges. As shown in Figure 9 there are paths joining together the different nodes. These paths (called edges in the *Networkx* documentation) could be forced to have only

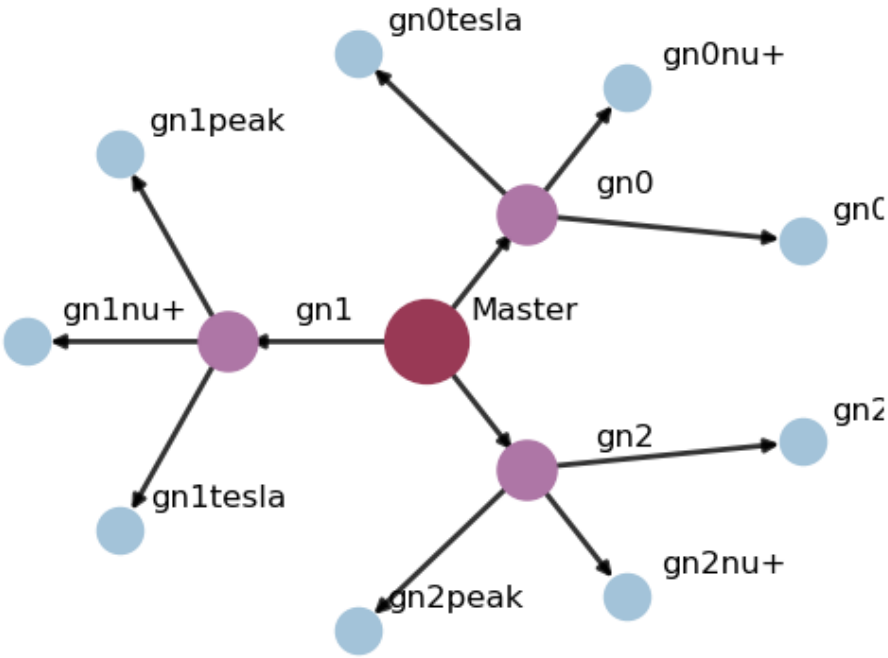


Figure 9: Network obtained after creating the initial network architecture given the configuration file shown in Listing 3

one direction by declaring the NetworkX object itself as DiGraph or DiMultiGraph. *NetworkX* offers four different types of Graphs depending on the edges characteristics: Graph, DiGraph, MultiGraph and MultiDiGraph. For the GRM Allocator current design we use a DiGraph type, strict single directional path. In the case of re-creating an edge between nodes, this new edge will substitute the previous which makes easier the process of updating the status of the cluster. On the other hand, to find the suitable allocation the shortest path is used as described bellow. Thus, directional paths assure that the allocation found is correct and avoid errors such as loops.

The edges described above, are the main component that would let us apply an arbitrary policy. Both edges and nodes can be assigned attributes. These attributes can be considered as a dictionary as each attribute is assigned a name and a value. However, one of them is mandatory, a predefined called weight, which can only be assigned a numeric value. This parameter is used by the path finding algorithms included in the library to find the shortest path between points. For simplification purposes, the Network architecture will be built like the one shown in Figure 9, then despite having more than one accelerator of each kind, every type will be represented by a single node. Then, this node contains two more attributes which are *#node* and *#node in use* representing the total number of hardware accelerators of the corresponding type node contains and the number of those currently in use respectively.

As illustrated in Figure 10 we build a network where the main node is the controller of the global resource manager (Called Master). This node is followed by the General Purpose nodes and then, attached to them the Heterogeneous Nodes. In this last layer, unlike the previous where the 64 cores of each node are represented by the same bubble, each accelerator core will have its own node. By default, weights of 1 are assigned to each edge. Also mention that the weighting system will be normalized to 1 being 1 the max possible weight and 0 the minimum.

Figure 10 represents a possible final network configuration which includes all the GN available for the project. Moreover, only changing from the configuration file in Listing 9 to the one in Annex C the system is able to scale without further modifications.

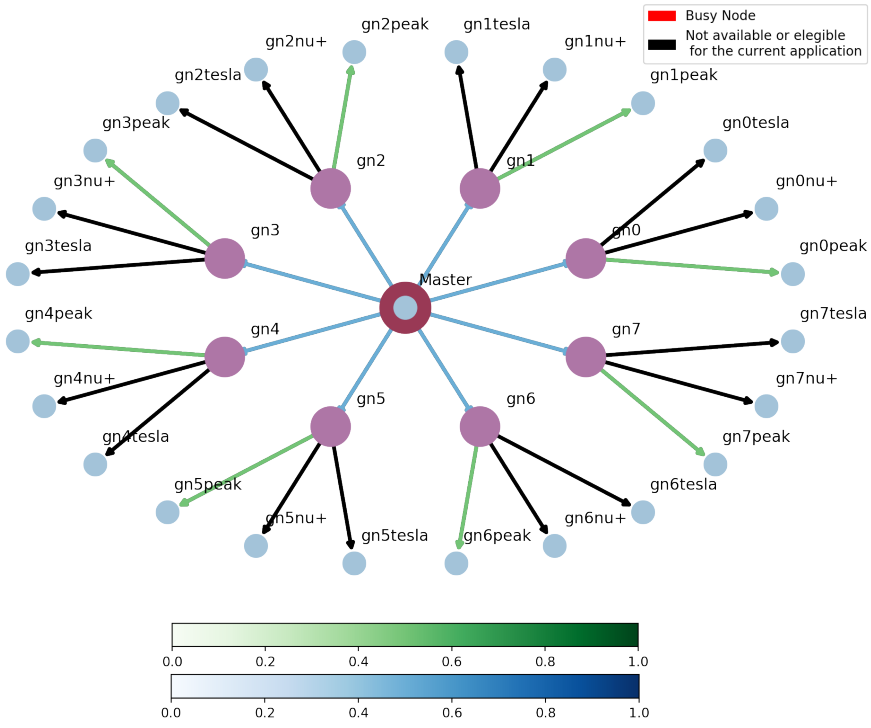


Figure 10: Larger network automatically scaled changing the configuration file

5.2 Cluster Availability Update

In order for the GRM Allocator to perform allocation decisions as closely to real-time as possible, it is necessary to have a real time ”cluster overview”. To accomplish this task, we use the Kafka Server (which connects to BBQ and gathers data, as previously described) and a Kafka Broker which consumes those messages, described in this section.

The ”cluster overview” contains many data: HN availability, HN accelerator type, HN temperature, HN power consumption and HN errors report. Data from the GN is gathered via Docker. Thanks to the Docker Swarm network and the fact that Docker share kernel with the host computer, server usage data is accessible inside Docker containers. However, this data is only relevant in case that all the CPU's are busy (a case only achieved when exercising corner cases about capacity computing).

Apart from consuming Kafka messages, this block is also in charge of updating the DiGraph object. Firstly, a client which is registered as consumer in the Kafka topic *bbque_data* waits until a new message is received, then parses the data and calls the update function. Then, the algorithm weights the edges depending on the node availability. For instance, a node whose HNs are fully busy is assigned a 1, while a HN with free resources is assigned a 0. The same technique is applied at a local scale, meaning that the path joining a GN and its HNs is set to 0 or 1 depending on their availability.

5.3 Algorithms

This final block is called whenever a new allocation request is received and it is split into two sections. One in which the actual policy is applied and a second one where an auxiliary node is created.

Policies implemented for this project can be selected when launching an application, thus in run-time, which is very suitable on testing scenarios like ours. In consequence, if no policy is specified in the input command, the allocator will use whether the default one (Greedy) or other if it was previously modified. In order to make this situation possible the allocation is performed with a copy of the *NetworkX* object made at the time the new allocation request is received, then, depending on the algorithm selected the paths are weighted in different ways (Section 6 describes the available policy algorithms and how they weight the different edges).

The copy of the main cluster overview object performed before, is in the second section used to include the auxiliary node thereby we keep the occupancy cluster clean so the availability update process is performed as fast as possible preventing thread block in the iterative daemon. The allocation decision is made automatically after the previous described process has been successfully executed. Afterwards we execute one of the most powerful tools of the library, the path finding algorithms. For this purpose, we chose Dijkstra algorithm which is widely used as it is one of the most efficient.

Dijkstra algorithm is an algorithm used for shortest path finding when the nodes (or as in our case, the path joining the nodes) are weighted. Thus, after the weighting process, we call this algorithm with the Master node as origin and the Auxiliary node as destination. Then, the algorithm finds the shortest path which corresponds with the allocation. Also, it provides the next two shortest paths so if after that process a problem is found, the next allocation is used.

5.4 Entry Point

The fact of having an external allocator which will interact independently from SLURM implies the necessity of developing a custom entry point with high availability and reliability. In order to do so, a new Kafka topic has been added to the server. This Kafka topic will have 24 hours persistence to grant reliability and a historic of the previous hours requests. Availability is granted via multi-threading method where a new thread per request is created. Creating a new thread per request means we are able to handle a max of 64 per second (Safety time to generate the allocation and parsing the job to SLURM), good performance in terms of availability.

It is necessary to take into consideration the collision between the updater and the entry point. As both threads require to access the cluster object it was necessary to establish dependencies among them so every time an instance needs to access the main object must call the mutex (MUTual EXclusion) [6] object related to the cluster. Thus only one thread is accessing it at a time.

Besides of creating a new thread on demand, the entry point is located inside the cluster status update loop thus, assuring the last allocation policy is applied with updated information already parsed. Also adding a dependency between the entry point and the updater tool ease

6 Efficient power-, performance- and thermal-aware strategies

In this section we describe the policies applied on the MANGO cluster, together with the specifications of the components over which the testing have been done.

The algorithms that follow this section need some prior explanation to follow the syllables:

- GN_i: General Purpose node (i.e. Regular Intel-based HPC servers).
- HN_j: Heterogeneous Node. Every different hardware accelerator. Any of these are controller by a general porpoise node, thus $HN_{i,j}$ means the heterogeneous node j of the general porpoise node i.
- $\omega(GN_i - HN_{i,j}) \leftarrow$ represents the weight of the **edge** joining the corresponding nodes. For example, $\omega(GN_0 - HN_{0,1})$ may represent the weight of the edge joining the *PEAK* accelerators node of HN_3 (which is the HN attached to GN_0 in the experiments) with the node GN_0. As we have a total number of different accelerators of 3 $\longrightarrow j\varepsilon[1,3]$
- $policy_\omega \leftarrow$ Variable that temporally stores the weight to be assign to a certain edge. Before assigning this weight to the edge some test are required to check that obtained value is within the normal margins (Usually 0-1)
- $temperature_impact \leftarrow$ This is a predefined variable that allos us modify on run time the relevance of the temperature in the allocation decisions.
- $find\_shortest\_path \leftarrow$ A function included in *Networkx* library that uses the path finding algorithm *Djistra* seeks the fastest way to go from the Master node to the Aux_node. Aux_node is an auxliary node which is added on allocation runtime to a copy of the graph. This node is connected with zero weight edges to any accelerator in the system to facilitate the allocation process. The returned value of this function is a list with the complete shortest path (considering weighted edges) from *Master* to *Aux_node*

One important point of the policies described bellow is that they are executed considering an overview of the accelerators. To provide the data we do not make any difference between the same kind of accelerator. For example, usually we would have the three different accelerators running on each GN thus, the policies will count the number of FPGA containing each accelerator. In appendix C in section *GN4* of the configuration file Listing 6 where it describes the accelerator kernels available on *gn*4 there are 3 *PEAK*, 3 *Nu+* and 2 *Tesla*. If there are 2 *PEAK* busy the availability (in case that the incoming application prior request is *PEAK*) will be 1 *PEAK*

6.1 Greedy policy

This allocation policy is used as first step for testing purposes. This policy is based on a first in first out system where the applications are executed in the first empty slot found. Idle consumption of the nodes is significant as the only possible

Thus, this policy gives less weight to the nodes which are already executing any application, mathematically:

Algorithm 1 Greedy algorithm

procedure GREEDY(GN_i, HN_j) ▷ Priority to already busy nodes
  $ThresHold \leftarrow 0.5$
  for GN_i in $i = 1,8$ **do** ▷ Iterate every hardware node on the given GN
    for $HN_{i,j}$ in $j = 1,n$ **do**
      if $available\_accel_{i,j} > busy\_accel_{i,j}$ **then**
        $policy_\omega = -\frac{AvailableHW}{TotalHW} * ThresHold$
        $\omega(GN_i - HN_{i,j}) \leftarrow ThresHold + policy_\omega$
      else
        $\omega(GN_i - HN_{i,j}) \leftarrow 1$
  $node \leftarrow find\_djistra\_shortest\_path(Master\_Node, aux\_node)$
  return node ▷ The gcd is b

6.2 Fully distributed policy

Algorithm 2 Fully distributed algorithm

procedure DISTRIBUTED($GN_i, HN_j, temperatures$) ▷ Priority to free nodes
  $ThresHold \leftarrow 0.5$
  for GN_i in $i = 1,8$ **do** ▷ Iterate every hardware node on the given GN
    for $HN_{i,j}$ in $j = 1,n$ **do**
      if $available\_accel_{i,j} > busy\_accel_{i,j}$ **then**
        $policy_\omega = -\frac{AvailableHW}{TotalHW} * ThresHold$
        $\omega(GN_i - HN_{i,j}) \leftarrow ThresHold + policy_\omega$
      else
        $\omega(GN_i - HN_{i,j}) \leftarrow 1$
  $node \leftarrow find\_djistra\_shortest\_path(Master\_Node, aux\_node)$
  return node ▷ The gcd is b

6.3 Temperature/Power aware policy

As the name shows, this policy prioritizes nodes with the lower temperature. There are two temperatures to take into consideration, the HN temperature and the GN temperature. HN temperature is somehow stable considering that applications only affects in one or two degrees

when executed on a given HN. GNs, on the other hand, raises its temperature because the LRM assigns a core for each application which means that the nodes can have up to 32 cores busy handling each HN board. Thus, a single application is not important in temperature node terms meanwhile 20 applications simultaneously executed on the same node will raise the temperature.

Considering that this policy considers both HN and GN temperature in different ways. HN temperatures per node are averaged because as show in Figure 14 FPGAs on the same motherboard temperatures are almost equal and we can extrapolate the same to motherboards per node. In consequence and as the GRM is not choosing in which FPGA the application is going to be executed, this policy performs a big picture temperature resource allocation.

Greedy-Temperature policy

Algorithm 3 Temperature-Greedy algorithm

procedure TEMP-GREEDY($GN_i, HN_j, temperatures$) ▷ Lowest temperature priority
  $temperature\_list \leftarrow short(temperatures)$
  $max_temperature \leftarrow max(temperature_list)$
  $ThresHold \leftarrow 0.5$
  $temperature\_impact \leftarrow 0.2$
  for GN_i in $i = 1,8$ **do** ▷ Iterate every hardware node on the given GN
    $it\_temperature \leftarrow temperature\_list(GN_i)$
    $temp\_weight \leftarrow \frac{max\_temperature - it\_temperature}{max\_temperature} * temperature\_impact$
    $\omega(Master - GN_i) \leftarrow ThresHold * temp\_weight$
    for HN_j in $j = 1,n$ **do**
      if $available\_accel_{i,j} > busy\_accel_{i,j}$ **then**
        $policy_\omega = -\frac{AvailableHW}{TotalHW} * ThresHold$
        $\omega(GN_i - HN_{i,j}) \leftarrow ThresHold + policy_\omega$
      else
        $\omega(GN_i - HN_{i,j}) \leftarrow 1$
  $node \leftarrow find\_dijstra\_shortest\_path(Master\_Node, aux\_node)$
  $return\ node$ ▷ The gcd is b

Distributed-Temperature policy

Algorithm 4 Temperature-Distributed algorithm

procedure TEMP-SPREAD($GN_i, HN_j, temperatures$) ▷ Lowest temperature priority
  $temperature\_list \leftarrow short(temperatures)$
  $max_temperature \leftarrow max(temperature_list)$
  $ThresHold \leftarrow 0.5$
  $temperature\_impact \leftarrow 0.2$
  for GN_i in $i = 1,8$ **do** ▷ Iterate every hardware node on the given GN
    $it\_temperature \leftarrow temperature\_list(GN_i)$
    $temp\_weight \leftarrow \frac{max\_temperature - it\_temperature}{max\_temperature} * temperature\_impact$
    $\omega(Master - GN_i) \leftarrow ThresHold * temp\_weight$
    for HN_j in $j = 1,n$ **do**
      if $available\_accel_{i,j} > busy\_accel_{i,j}$ **then**
        $policy_\omega = \frac{AvailableHW}{TotalHW} * ThresHold$
        $\omega(GN_i - HN_{i,j}) \leftarrow ThresHold + policy_\omega$
      else
        $\omega(GN_i - HN_{i,j}) \leftarrow 1$
  $node \leftarrow find\_djistra\_shortest\_path(Master\_Node, aux\_node)$
  $returnnode\ b$ ▷ The gcd is b

7 Experiment setup and results

7.1 Global Resource Manager Validation

This section describes the different approaches taken into consideration in order to partially validate MANGO project against the reviews coming on May and to test the different policies described in section 6. To this purpose we first need to describe the different scenarios in which we ran our tests.

Firstly, we created a reduced target system in which we tested the correct functionality of the features described all along this document. This includes the dockerization of SLURM, the connection with the other MANGO software, the allocator, the entry-point and the policies. The hardware targets of this scenario are described in Table 2. Thus, Figure 11 presents the corresponding network generated by the allocator after parsing the architecture just mentioned. Henceforth the legend of any graph that describes the representation of cluster network status, refer to Figure 11 legend as the same colors will be used in any representation.

GN	HN	SLURM Docker Version	BBQUE Connector Docker Version	Mango Repo Branch	Mango Repo Commit
5	8	mango_latest_v1.2	latest	mango_upv_20190208	54ce1c0ad3d4
6	11	mango_latest_v1.2	latest	mango_upv_20190208	54ce1c0ad3d4

Table 2: Target Architecture of the initial functionality test

Next, we describe the steps followed to validate the complete Global Resource Manager architecture described above, considering the deploy configuration shown in Annex A.1 and summarized in Table 3

GN#	HN#	#PEAK	#Nu+	#Tesla
GN5	HN8	3	0	0
GN6	HN11	3	0	0

Table 3: This table summarized the configuration file used during the validation process

Deployment

```
$ docker stack deploy -c compose-mango.yml mango
```

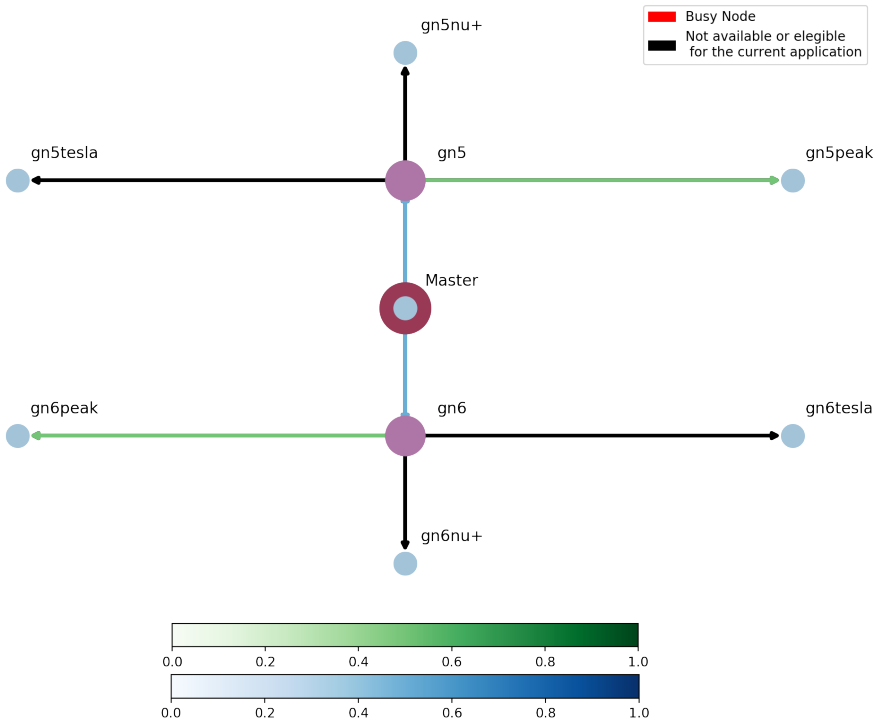


Figure 11: Network architecture for the functionality tests

Figure12 shows how the Swarm successfully deployed all the services along the configured server. The services reported by Docker Swarm will be marked as 0/1 if there would have been any problem on the process. Apart from the SLURM daemon and the BBQUE-Kafka connector services, this configuration deploys in the node marked as master the main SLURM Controller container, the databases and the Kafka Server required by the GRM.

BBQUE connectivity

Figures 13a and 13b represent the logs of the kafka-bbque docker connectors of both nodes GN5 and GN6 where we can see successful subscription between the container and the BBQUE data server and also the visibility between the Kafka producers and the server.

This step is validated by means of an auxiliary script which consumes messages from the Kafka Server to which the daemons are pushing the data received from the BBQUE data client. A common output of this testing script is represented in Figure 14. This messages contain the summarized information provided by BBQUE which is pushed towards kafka server. In short, the number of each accelerators currently busy in the board and its temperature which is common to the four FPGA's placed per motherboard.

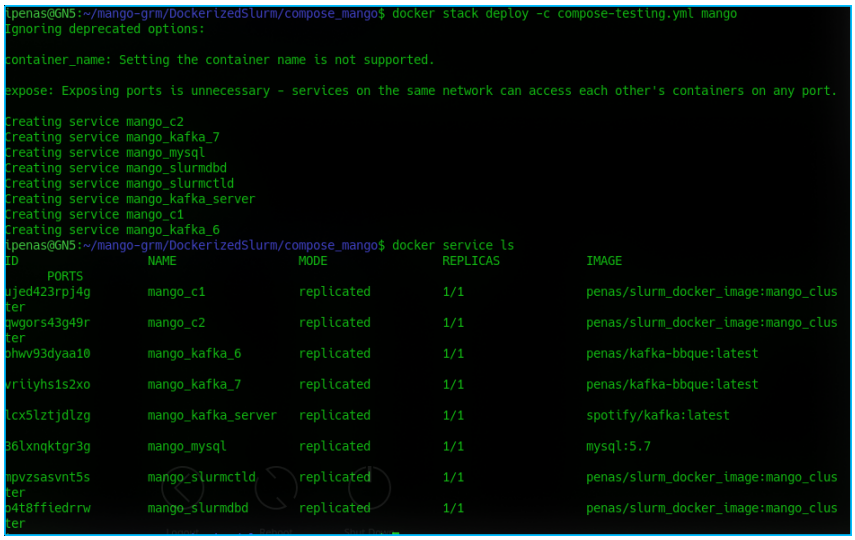
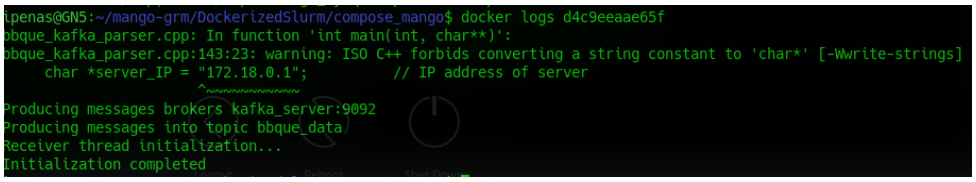


Figure 12: Docker Swarm Successful deployment of services



(a) Kafka-BBQUE on GN5

(b) Kafka-BBQUE on GN6

Figure 13: This figure shows the correct deployment of the docker images that subscribe to the BBQUE data server and posts its results to the Kafka Server of the GRM

SLURM

Here, we demonstrate how an application can be executed in any node contained in the SLURM environment (GN 5 and GN 6 in this particular case). Thus, running the following commands inside the *slurmctld* container will prove the point:

Listing 4: Example of initial architecture input file

```
$ sbatch -w c6 /opt/mango-apps/run.sh matrix 200
```

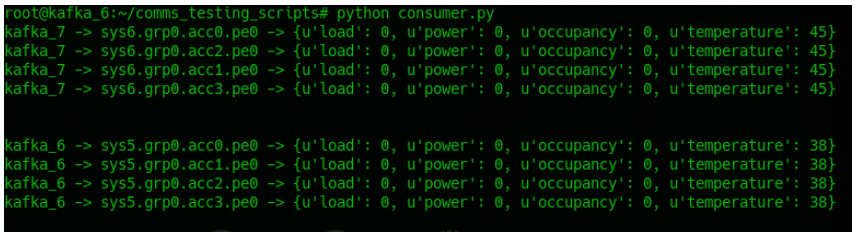


Figure 14: Sample of filtered BBQUE messages pushed bu the Kafka producers here kafka_6 represents GN5 and kafka_7, GN6

```
$ sbatch -w c6 /opt/mango-apps/run.sh matrix 200
$ sbatch -w c7 /opt/mango-apps/run.sh matrix 200
```

Commands provided in Listing 4 schedule three works into SLURM, targeting GN5 the first two and GN6 the second one. Thus, Figure 15 shows the SLURM log of the jobs completion and Figures X and X the reports of each BBQUE instance (GN5 and GN6 instances) where we can see that both applications were correctly executed.

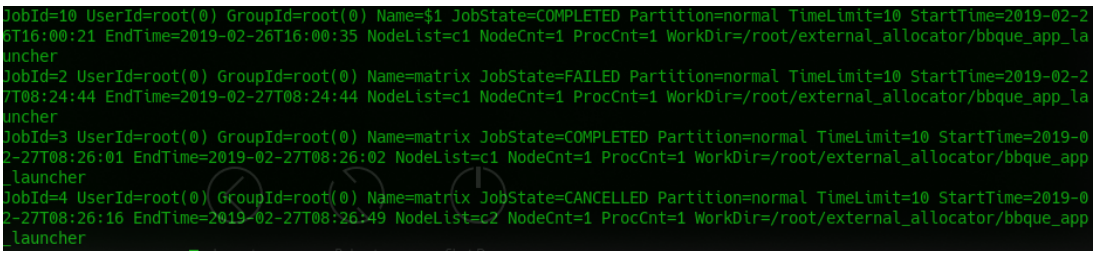


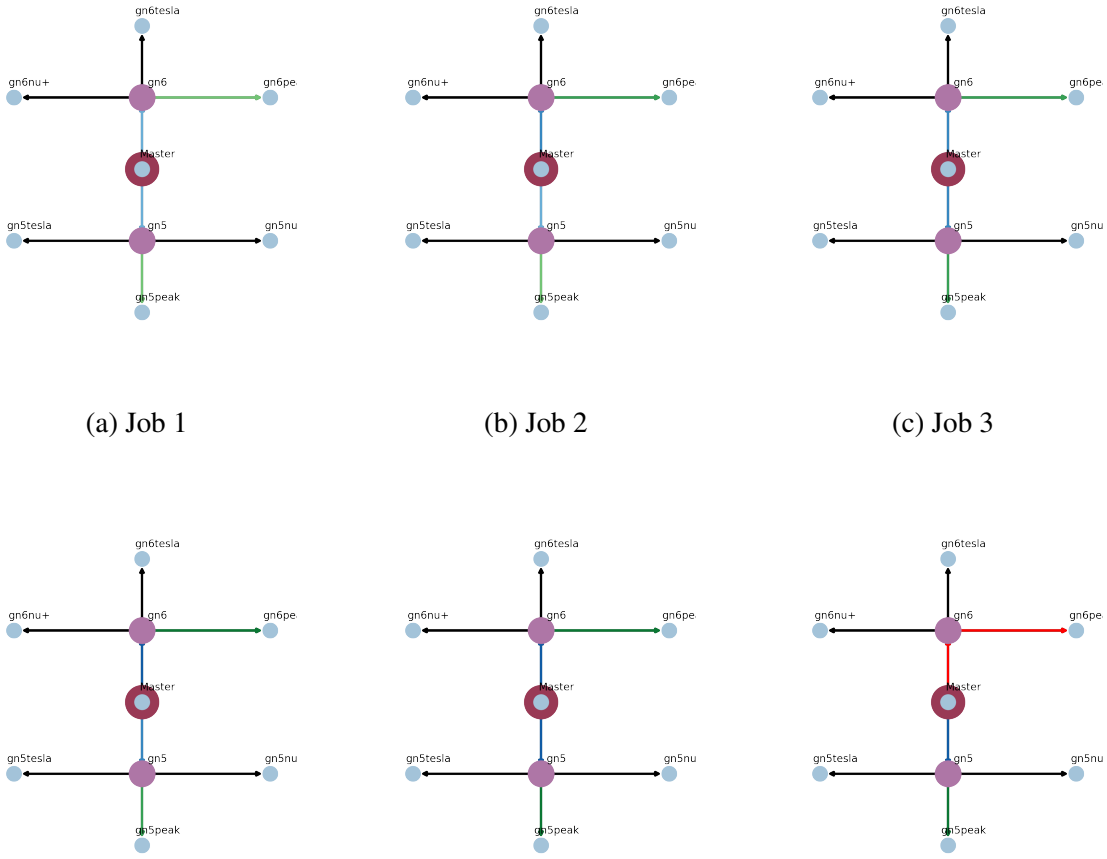
Figure 15: Example of SLURM completion job report in which the three different variants are shown

Allocator

This step aims to validate each of the policies' functionality described in Section 6. In consequence, we should support the application results with a picture of the status of the network each time an allocation is performed to help understand how the decision is made.

Given the configuration stated above for this test, the maximum number of accelerators available is 6, three *PEAK's* on HN8 and three *PEAK's* on HN11. As we want to validate the functionality of the allocator subsystem we configured the GRM only to execute sequentially 6 applications. With that setup we should be able to observe the evolution while it has applications running, also how it takes into consideration the occupancy reported by BBQUE updating the graph and lastly validate the policies.

Figure 16 shows the update network after the execution order is given by the allocator and the application is running in both resource managers, SLURM and BBQUE. Besides, this image shows a good point with respect to Figure 11 as the real architecture under which the validation



(a) Job 1 (b) Job 2 (c) Job 3

(d) Job 4 (e) Job 5 (f) Job 6

Figure 16: Evolution of cluster *Networkx* graph status throughout the first 6 allocated jobs applying the distributed policy

is performed corresponds to 3 *Peak* accelerators per node. Then, the edges connected to different accelerators are represented in darker colours.

In order to validate the policies concerning temperature, it is necessary to perform more complex tests, in which more nodes are involved because the temperature policy is executed with the relative differences of the nodes as explained in the section above. Thus, those policies will be validated and tested at the same time in section 7.2

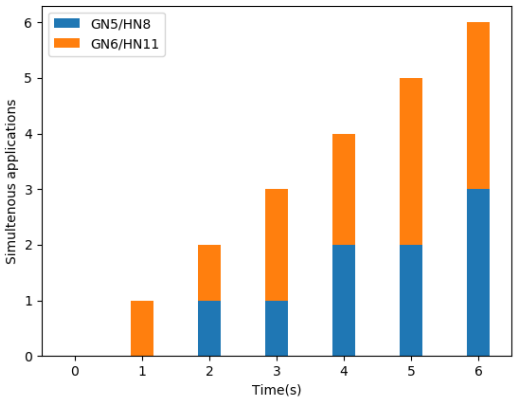


Figure 17: Example of SLURM completion job report in which the three different variants are shown

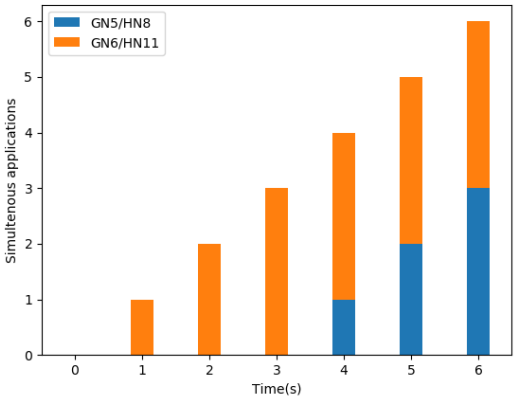


Figure 18: Example of SLURM completion job report in which the three different variants are shown

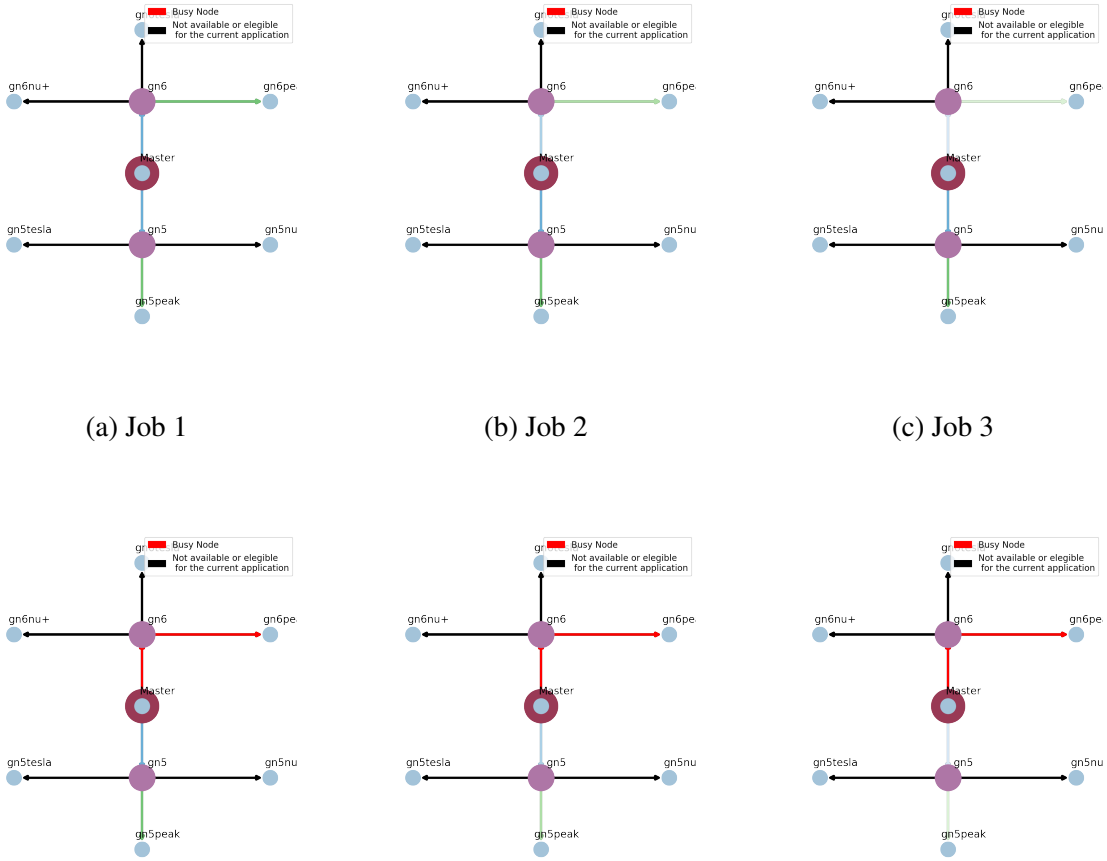
Entry Point

To grant correct usage of the GRM we provide an entry point system. This system is composed by two main blocks, a script and a system signal interruption.

The script, described in Listing 5, reads the running docker containers and extracts the container Id. Afterwards, executes the Unix signal *USR2* interrupt within the docker container of the SLURM controller. Assuming the allocator is running and listening to that signal, it reads the file just created (Named */opt/mango-apps/next_app.log*) with the name and the location of the application to run and proceeds with the allocation and execution of the scheduled application.

Listing 5: Entrypoint script

```
#!/bin/bash
```



(a) Job 1 (b) Job 2 (c) Job 3

(d) Job 4 (e) Job 5 (f) Job 6

Figure 19: Evolution of cluster *Networkx* graph status throughout the first 6 allocated jobs applying the distributed policy

```
container_name=$(docker ps --format "{{.Names}}" | grep ctld)

echo "App name is $1"
echo "Location is $2"
echo "Controler container name: $(echo $container_name)"
echo "Total number of arguments is $#"

if [[ -z "$container_name" ]]; then
    echo "SLURM controller docker seems no to be running"
    exit 1
else
     echo "Slurm Docker  controller found: "
     echo "$container_name"
```

```
fi

echo $1 > /opt/mango-apps/next_app.log
echo $2 > /opt/mango-apps/next_app.log

docker exec -it $container_name kill -USR2 python
```

The interruption handler of the allocator is configured to listen to the Unix User Signal 2 (USR2). When this interruption is received the daemon reads the corresponding recipe of the application and parses the accelerator requirements to the allocation process described in previous sections.

7.2 Policies results and validation

It has been checked during the project that the execution of matrix multiplications over *PEAK* accelerators doesn't change the temperature of the hardware at all. As in the previous section wasn't possible to show temperature aware policies, in the following section we will compare the performance of the results considering the four policies in a wider experiment in order to show the advantages of aiming lower temperature nodes trying to reduce the power consumption related to refrigeration.

The experiment configuration for this section will follow the occupancy trend represented in Figure 20 and the experiments wont be based on number of total applications executed but total execution time in order to have a base to compare the results of the different policies.

.

Following the structure described in the previous experiment the cluster setup is depicted in Table 4 and illustrated in Figure 21 as a cluster graph.

GN	HN	SLURM Docker Version	BBQUE Kafka Docker Version	Mango Stack Version	GRM Commit	#PEAK
0	3	mango_latest_v1.3	latest	mango_v0.3	52b6e8f8443c	3
1	4	mango_latest_v1.3	latest	mango_v0.3	52b6e8f8443c	3
5	8	mango_latest_v1.3	latest	mango_upv_20190208	54ce1c0ad3d4	3
6	9	mango_latest_v1.3	latest	mango_upv_20190208	54ce1c0ad3d4	3

Table 4: MANGO stack status during the 4 nodes test

Once again, due to the instability and lack of availability of the different accelerators, we are focusing this experiment only in PEAK accelerators, thus the edges connected to other accelerators are opened and coloured in black. Also, PEAK accelerators are the main testing hardware design of the project as the implementation of applications over them is much faster.

As in previous graphs, Figure 21 shows the architecture of a four node heterogenous cluster where the accelerators Nu+ and Tesla have been disabled creating a composition of GN's and PEAK accelerators.

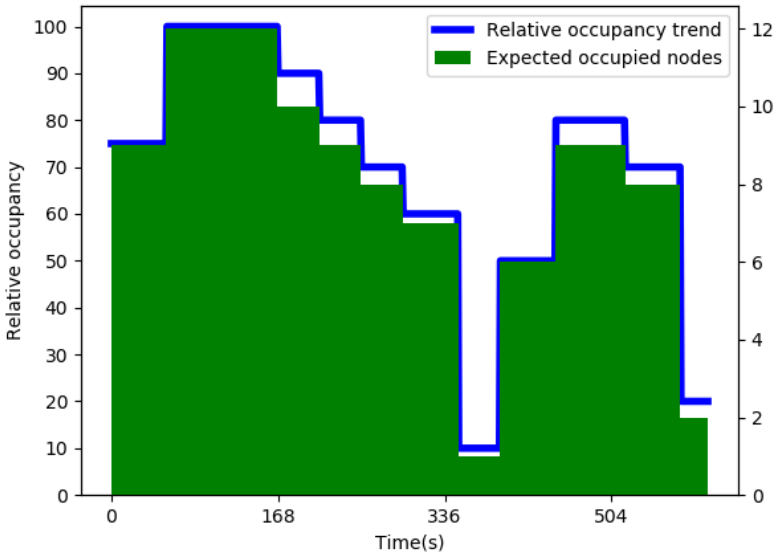


Figure 20: Experiment predefined occupancy trend trying to follow a distribution with two marked peaks of demand in order to show the adaptability of the temperature policies

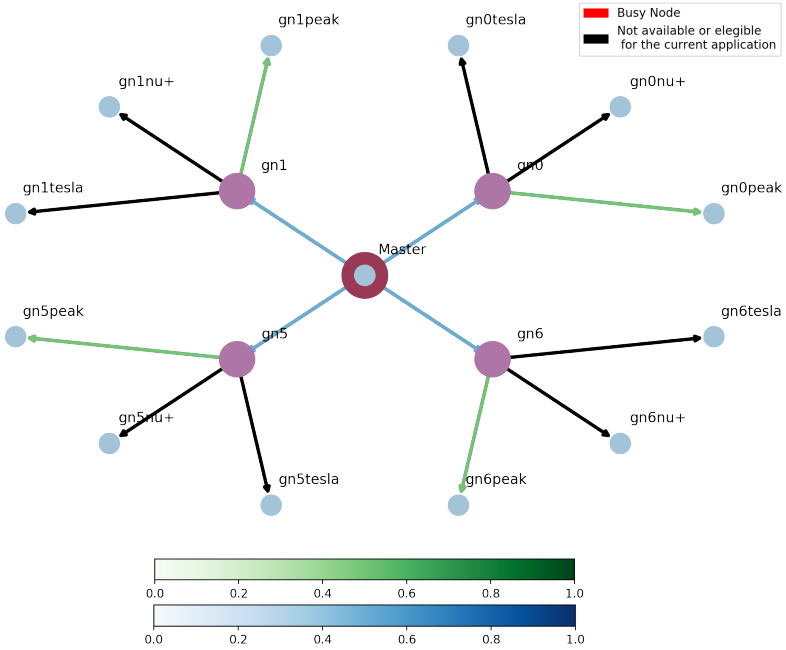


Figure 21: Extension of the previous architecture where we consider two more nodes with its repective hardware

Table 5 shows the performance experiment results. For this experiment, the allocator was programmed to execute a total of 300 matrix multiplications. The average time corresponds to the execution time of a multiplication of a 200x200 random generated matrix by it self. The experiment was performed once with each of the 4 four policies describe in Section 6.

This experiment confirms how the performance of the application improves around a 10% by simply distributing the applications to the emptiest node.

Policy	Application	Average execution time(s)		Successful executions (out of 300)
		GRM(Slurm)	LRM(BBQUE)	
Distributed	Matrix multiplication (200x200)	80,366	79,212	297
Greedy	Matrix multiplication (200x200)	89,491	89,210	265
Temperature Distributed	Matrix multiplication (200x200)	80,109	79,933	296
Temperature Greedy	Matrix multiplication (200x200)	89,391	88,863	270

Table 5: Performance experiment results

8 Conclusions

Despite the integration difficulties of such a large project involving so many tools, we have managed to develop a Resource Manager which is able to run applications relying the execution on the LRM while it implements efficient policies which are aware of the current cluster overview. Relying the data transmission between clusters to a Kafka broker and the allocation execution to Networkx high efficient algorithms have permitted us to create a reliable resource manager that can be easily replicated or extended.

As shown in Section 7 we have managed to achieve a great performance implementing a fully distributed policy. The performance of this policy is better due to the memory requirement of the applications (Philips is a memory eater application as it performs image processing).

Aiming power, greedy policy outperforms the rest since, as explained in 6.1, this policy gives priority to nodes which are already executing applications. Thus, the other nodes can enter in idle state, which significantly reduces the power consumption.

9 Future Work

Thanks to the flexible and high availability graph allocation system of this projects multiple possibilities are opened. The main proposal would be to apply ML strategies to allocation policies [7]. With this system, the allocator would not only be aware of the current status of the cluster but also specific features such as performance for each application, temperature/power response under stress, etc... of the nodes which are not available with plain data.

Regarding the GRM, in order to improve its performance, we propose two new features which based on MANGO project requirements have not been developed:

1. Integrate the allocator within SLURM context
2. Further integration of the GRM into MANGO software stack

The current status of the GRM prevents it from controlling in which specific FPGA (hardware accelerator in general) the application is executed. Thus, a huge improve on performance would be to integrate LRM and GRM in the same instance which will also give the possibility of parallel executions.

The effort of integrating MANGO allocator as a SLURM plugin was too high considering the work required for such a task apart from the develop of the system it self. As next step, whether an integration within the SLURM environment or an adaptation of the current allocator to substitute SLURM is required to prevent the system redundancy which is currently happening in the system.

References

[1] Wenguang Chen. The demands and challenges of exascale computing: an interview with Zuoning Chen. *National Science Review*, 3(1):64–67, 03 2016.

[2] Mango project oficial web page. http://www.mango-project.eu/.

[3] José Flich, Giovanni Agosta, Philipp Ampletzer, David Atienza Alonso, Carlo Brandolese, Etienne Cappe, Alessandro Cilardo, Leon Dragić, Alexandre Dray, Alen Duspara, William Fornaciari, Edoardo Fusella, Mirko Gagliardi, Gerald Guillaume, Daniel Hofman, Ynse Hoornenborg, Arman Iranfar, Mario Kovač, Simone Libutti, Bruno Maitre, José Maria Martínez, Giuseppe Massari, Koen Meinds, Hrvoje Mlinarić, Ermis Papastefanakis, Tomás Picornell, Igor Piljić, Anna Pupykina, Federico Reghenzani, Isabelle Staub, Rafael Tornero, Michele Zanella, Marina Zapater, and Davide Zoni. Exploring manycore architectures for next-generation hpc systems through the mango approach. *Microprocessors and Microsystems*, 61:154 – 170, 2018.

[4] Giovani Torres. Original dockerized version of slurm. https://github.com/giovtorres/slurm-docker-cluster.

[5] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. 01 2008.

[6] Python documentation. Python mutual exclusion support. https://docs.python.org/2/library/mutex.html.

[7] Arman Iranfar. Machine Learning-Based Quality-Aware Power and Thermal Management of Multistream HEVC Encoding on Multicore Servers. *JOURNAL OF IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 2018.

Appendices

A File description for Dockerization and deployment

SLURM DockerFile

```
FROM centos:7

LABEL org.label-schema.docker.cmd="docker-compose up -d" \
      org.label-schema.name="slurm-docker-cluster" \
      org.label-schema.description="Slurm Docker cluster on
         CentOS 7"

ARG SLURM_VERSION=17.02.11
ARG SLURM_DOWNLOAD_MD5=b32f4260a921d335a2d52950593f0a29
ARG SLURM_DOWNLOAD_URL=https://download.schedmd.com/slurm/slurm
    -17.02.11.tar.bz2

ARG GOSU_VERSION=1.10

RUN yum makecache fast \
    && yum update -y \
    && yum -y install epel-release \
    && yum -y install \
         libtool \
         gtk+-devel gtk2-devel \
         autoconf \
         wget \
         bzip2 \
         perl \
         gcc \
         gcc-c++\
         vim-enhanced \
         git \
         make \
         automake \
         munge \
         munge-devel \
         python-devel \
         python-pip \
         python34 \
         python34-devel \
         python34-pip \
         mariadb-server \
```

```
        mariadb-devel \
        psmisc \
        bash-completion \
    && yum clean all \
    && rm -rf /var/cache/yum

RUN pip install Cython nose \
    && pip3 install Cython nose

RUN set -x \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/
       gosu/releases/download/$GOSU_VERSION/gosu-amd64" \
#    && wget -O /usr/local/bin/gosu.asc "https://github.com/
   tianon/gosu/releases/download/$GOSU_VERSION/gosu-amd64" \
#    && export GNUPGHOME="$(mktemp -d)" \
#    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys
   B42F6819007F00F88E364FD4036A9C25BF357DD4 \
#    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local
   /bin/gosu \
#    && rm -rf $GNUPGHOME /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true

COPY plugin /root/plugin

RUN autoconf -V

RUN groupadd -r slurm --gid=995 && useradd -r -g slurm --uid
   =995 slurm

RUN set -x \
    && wget -O slurm.tar.bz2 "$SLURM_DOWNLOAD_URL" \
    && echo "$SLURM_DOWNLOAD_MD5" slurm.tar.bz2 | md5sum -c - \
    && mkdir /usr/local/src/slurm \
    && tar jxf slurm.tar.bz2 -C /usr/local/src/slurm --strip-
       components=1 \
    && rm slurm.tar.bz2 \
    && cd /usr/local/src/slurm \
    && cp -r /root/plugin/cons_res_ext /usr/local/src/slurm/src
       /plugins/select/cons_res_ext \
    && ls /usr/local/src/slurm/src/plugins/select/ \
    && cp /root/plugin/Makefile.am /usr/local/src/slurm/src/
       plugins/select/Makefile.am \
    && cp /root/plugin/configure.ac /usr/local/src/slurm/
       configure.ac \
```

```
    && sh /usr/local/src/slurm/autogen.sh \
    && cp /root/plugin/slurm.spec /usr/local/src/slurm/slurm.
       spec \
    && ./configure --enable-debug --prefix=/usr --sysconfdir=/
       etc/slurm \
        --with-mysql_config=/usr/bin  --libdir=/usr/lib64 \
    && make install \
    && install -D -m644 etc/cgroup.conf.example /etc/slurm/
       cgroup.conf.example \
    && install -D -m644 etc/slurm.conf.example /etc/slurm/slurm
       .conf.example \
    && install -D -m644 etc/slurm.epilog.clean /etc/slurm/slurm
       .epilog.clean \
    && install -D -m644 etc/slurmdbd.conf.example /etc/slurm/
       slurmdbd.conf.example \
    && install -D -m644 contribs/slurm_completion_help/
       slurm_completion.sh /etc/profile.d/slurm_completion.sh \
    && cd \
    && rm -rf /usr/local/src/slurm \
    && mkdir /etc/sysconfig/slurm \
         /var/spool/slurmd \
         /var/run/slurmd \
         /var/run/slurmdbd \
         /var/lib/slurmd \
         /var/log/slurm \
         /data \
    && touch /var/lib/slurmd/node_state \
         /var/lib/slurmd/front_end_state \
         /var/lib/slurmd/job_state \
         /var/lib/slurmd/resv_state \
         /var/lib/slurmd/trigger_state \
         /var/lib/slurmd/assoc_mgr_state \
         /var/lib/slurmd/assoc_usage \
         /var/lib/slurmd/qos_usage \
         /var/lib/slurmd/fed_mgr_state \
    && chown -R slurm:slurm /var/*/slurm* \
    && /sbin/create-munge-key

RUN mkdir /entry_point

## This section is only for graphs
RUN yum install -y graphviz-devel
RUN pip install --upgrade pip
RUN python -mpip install -U matplotlib
RUN pip install pygraphviz
```

```
COPY slurm.conf /etc/slurm/slurm.conf
COPY slurmdbd.conf /etc/slurm/slurmdbd.conf
COPY keys/id_rsa.pub /root/.ssh/id_rsa.pub
COPY keys/id_rsa /root/.ssh/id_rsa
COPY external_allocator /root/external_allocator

##Reseting permisions for private key
RUN chmod 400 ~/.ssh/id_rsa

COPY docker-entrypoint.sh /usr/local/bin/docker-entrypoint.sh
ENTRYPOINT ["/usr/local/bin/docker-entrypoint.sh"]

CMD ["slurmdbd"]
```

Kafka DockerFile

```
FROM ubuntu:latest

LABEL org.label-schema.vcs-url="https://c4science.ch/source/
    mango-grm.git" \
        org.label-schema.name="bbque-slurm-parser" \
        org.label-schema.description="C++ Kafka parser for
            barbaque" \
        maintainer="Ignacio Penas"


ARG RDKAFKA_DOWNLOAD_URL=https://github.com/edenhill/librdkafka
    .git
ARG CPPKAFKA_DOWNLOAD_URL=https://github.com/mfontanini/
    cppkafka.git

ENV LD_LIBRARY_PATH "$LD_LIBRARY_PATH:/usr/local/lib:/opt/mango
    /bosp/lib/bbque/"

RUN apt-get update \
     && apt-get -y install \
            nano \
            net-tools \
            wget \
            bzip2 \
            zlib1g-dev \
            perl \
            gcc \
            g++\
            git \
```

```
            make \
            cmake \
            zlib1g-dev \
            libpthread-stubs0-dev \
            python \
            python-pip \
            psmisc \
            bash-completion \
            libboost-all-dev \
            libstdc++6 \
    && apt-get clean
    #&& rm -rf /var/cache/apt

#This installations are for testing the comunication between
   kafka servers and
RUN pip install kafka-python

RUN set -x \
    && cd /root \
    && git clone "$RDKAFKA_DOWNLOAD_URL" \
    && cd librdkafka\
    && ./configure \
    && make && make install \
    && ls -la /usr/local/include

RUN cd /root/ \
    && ls -la \
    && git clone "$CPPKAFKA_DOWNLOAD_URL" \
    && cd cppkafka/ \
    && mkdir build \
    && cd build \
    && cmake .. && make && make install \
    && ls -la /usr/local/include



COPY src/bbque_kafka_parser.cpp /root/src/bbque_kafka_parser.
   cpp
COPY comms_testing_scripts /root/comms_testing_scripts
COPY include /root/include
COPY lib /root/lib

COPY docker-entrypoint.sh /usr/local/bin/docker-entrypoint.sh
ENTRYPOINT ["/usr/local/bin/docker-entrypoint.sh"]
```

A.1 MANGO GRM Docker-compose deployment file

This docker compose configuration file includes the deployment architecture to execute the GRM with both Kafka and SLURM parameters and node specification.

```
version: "3.0"

services:
  mysql:
    image: mysql:5.7
    hostname: mysql
    container_name: mysql
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints:
          - node.hostname == gn5
    environment:
      MYSQL_RANDOM_ROOT_PASSWORD: "yes"
      MYSQL_DATABASE: slurm_acct_db
      MYSQL_USER: slurm
      MYSQL_PASSWORD: password
    networks:
      customOverlay:
        ipv4_address: 20.1.0.7
    volumes:
      - var_lib_mysql:/var/lib/mysql

  slurmdbd:
    image: penas/slurm_docker_image:mango_cluster
    command: ["slurmdbd"]
    container_name: slurmdbd
    hostname: slurmdbd
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints:
          - node.hostname == gn5
    volumes:
      - etc_munge:/etc/munge
      - etc_slurm:/etc/slurm
      - var_log_slurm:/var/log/slurm
    expose:
      - "6819"
```

```
    networks:
      - customOverlay
    depends_on:
      - mysql

  slurmctld:
    image: penas/slurm_docker_image:mango_cluster
    command: ["slurmctld"]
    container_name: slurmctld
    hostname: slurmctld
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints:
          - node.hostname == gn5
    volumes:
      - etc_munge:/etc/munge
      - etc_slurm:/etc/slurm
      - slurm_jobdir:/data
      - var_log_slurm:/var/log/slurm
      - /opt/mango-apps:/opt/mango-apps
    expose:
      - "6817"
    networks:
      - customOverlay
      - dataoverlay
    depends_on:
      - "slurmdbd"

  kafka_server:
    image: spotify/kafka:latest
    container_name: kafka_server
    hostname: kafka_server
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints:
          - node.hostname == gn5
    environment:
      ADVERTISED_HOST: kafka_server
      ADVERTISED_PORT: 9092
    expose:
      - 2181
```

```
      - 9092
    networks:
      - dataoverlay
      - customOverlay
    depends_on:
      - "slurmctld"


  c6:
    image: penas/slurm_docker_image:mango_cluster
    command: ["slurmd"]
    hostname: c6
    container_name: c6
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints:
          - node.hostname == gn5
    volumes:
      - etc_munge:/etc/munge
      - etc_slurm:/etc/slurm
      - slurm_jobdir:/data
      - var_log_slurm:/var/log/slurm
      - /opt/mango-apps:/opt/mango-apps
    expose:
      - "6818"
      - "2222"
    networks:
      - customOverlay
    depends_on:
      - "slurmctld"

  kafka_6:
    image: penas/kafka-bbque:latest
    hostname: kafka_6
    container_name: kafka_6
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints:
          - node.hostname == gn5
    volumes:
      - /opt/mango:/opt/mango
```

```
    networks:
      - dataoverlay
    depends_on:
      - "c1"
      - "kafka_server"

  c7:
    image: penas/slurm_docker_image:mango_cluster
    command: ["slurmd"]
    hostname: c7
    container_name: c7
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints:
          - node.hostname == gn6
    volumes:
      - etc_munge:/etc/munge
      - etc_slurm:/etc/slurm
      - slurm_jobdir:/data
      - var_log_slurm:/var/log/slurm
      - /opt/mango-apps:/opt/mango-apps
    expose:
      - "6818"
    networks:
      - customOverlay
    depends_on:
      - "slurmctld"

  kafka_7:
    image: penas/kafka-bbque:latest
    hostname: kafka_7
    container_name: kafka_7
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints:
          - node.hostname == gn6
    volumes:
      - /opt/mango:/opt/mango
    networks:
      - dataoverlay
    depends_on:
```

```
      - "kafka_server"
      - "c7"


volumes:
  etc_munge:
  etc_slurm:
  slurm_jobdir:
  var_lib_mysql:
  var_log_slurm:

networks:
  dataoverlay:
    external:
      name: kafka_overlay
  customOverlay:
    external:
      name: slurm_overlay
```

B Application recipe requirements

B.1 Generic Sample Recipe

```
<?xml version="1.0"?>
<BarbequeRTRM recipe_version="0.8">
    <application priority="4">
        <platform id="org.linux.cgroup">
            <awms>
                <awm id="0" name="OK" value="100">
                    <resources>
                        <cpu>
                            <pe qty="100"/>
                        </cpu>
                        <mem qty="20" units="M"/>
                    </resources>
                </awm>
            </awms>
            <tg>
                <reqs>
                    <task name="t0" id="0" throughput_cps="2"
                        inbw_kbps="2000" outbw_kbps="2500"
                        hw_prefs="peak,gn,nup"/>
                    <task name="t1" id="1" ctime_ms="2000"
                        hw_prefs="peak,gn,nup"/>
                    <task name="t2" id="1" ctime_ms="2000"
                        hw_prefs="peak,gn,nup"/>
                    <task name="t3" id="1" ctime_ms="1000"
                        hw_prefs="peak,gn,nup"/>
                </reqs>
            </tg>
        </platform>
    </application>
</BarbequeRTRM>
<!-- vim: set tabstop=4 filetype=xml : -->
```

B.2 Philips sample Recipe

```
<?xml version="1.0"?>
<BarbequeRTRM recipe_version="0.8">
    <application priority="4">
        <platform id="org.linux.cgroup">
            <awms>
                <awm id="0" name="OK" value="100">
                    <resources>
```

```
                        <cpu>
                            <pe qty="100"/>
                        </cpu>
                        <mem qty="20" units="M"/>
                    </resources>
                </awm>
            </awms>
            <tasks>
                <task name="task1" id="1" throughput_cps="1"
                    inb w_kbps="2000" outbw_kbps="2500" hw_prefs
                    ="peak,gn,nup"/>
            </tasks>
        </platform>
    </application>
</BarbequeRTRM>
<!-- vim: set tabstop=4 filetype=xml : -->
```

C Extended network configuration file

The file shown bellow is an example of *NetworkX* initial configuration which includes all the GN nodes and random number of accelerators.

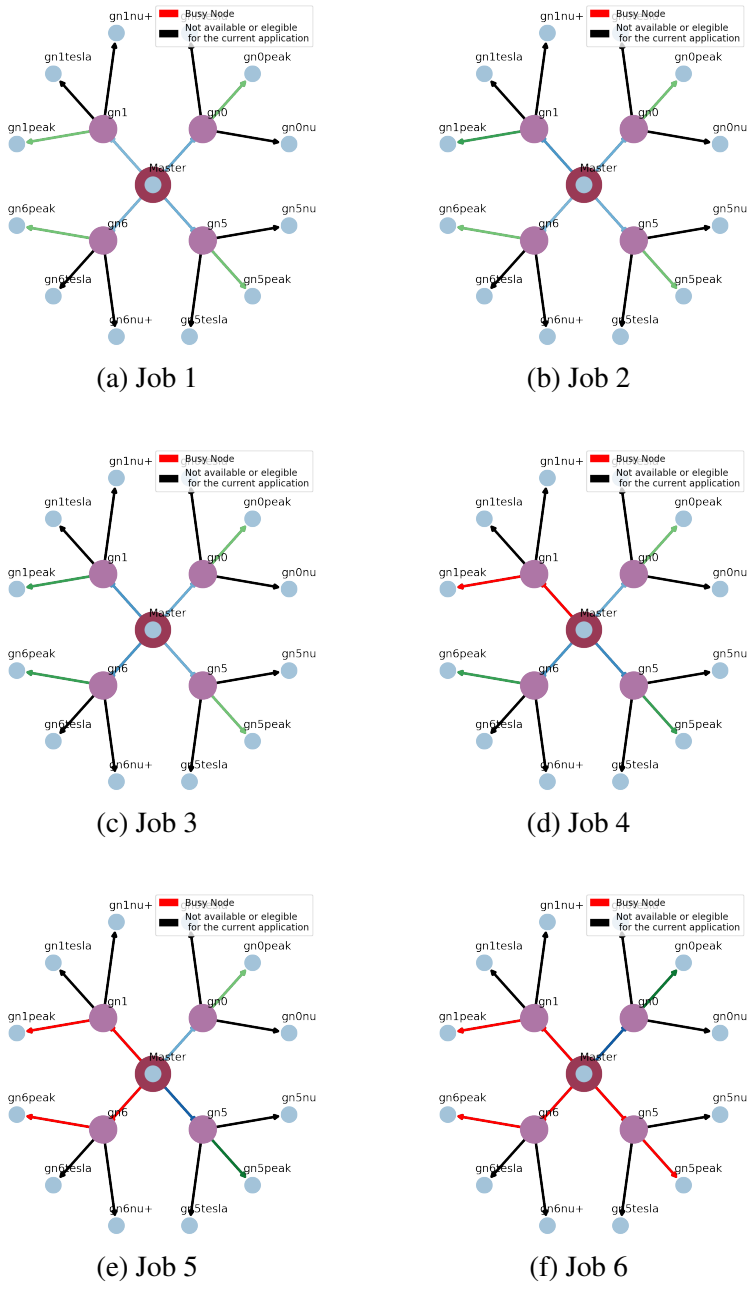
Listing 6: Full availability cluster example

```
{
  "name": "Mango Cluster",
  "version": "1.2.0",
  "master": "gn0",
  "architecture": {
    "gn0": {
      "id": "gn0",
      "master": 1,
      "components": {
        "peak": 4,
        "nu+": 4,
        "tesla": 0
      }
    },
    "gn1": {
      "id": "gn1",
      "master": 0,
      "components": {
        "peak": 2,
        "nu+": 2,
        "tesla": 2
      }
    },
    "gn2": {
      "id": "gn2",
      "master": 0,
      "components": {
        "peak": 4,
        "nu+": 0,
        "tesla": 0
      }
    },
    "gn3": {
      "id": "gn3",
      "master": 0,
      "components": {
        "peak": 4,
        "nu+": 2,
        "tesla": 2
```

```
      }
    },
    "gn4": {
      "id": "gn4",
      "master": 0,
      "components": {
        "peak": 3,
        "nu+": 3,
        "tesla": 2
      }
    },
    "gn5": {
      "id": "gn5",
      "master": 0,
      "components": {
        "peak": 2,
        "nu+": 2,
        "tesla": 0
      }
    },
    "gn6": {
      "id": "gn6",
      "master": 0,
      "components": {
        "peak": 0,
        "nu+": 4,
        "tesla": 0
      }
    },
    "gn7": {
      "id": "gn7",
      "master": 0,
      "components": {
        "peak": 6,
        "nu+": 2,
        "tesla": 0
      }
    }
  }
}
```

D Extended policy graphs

D.1 Temperature distributed



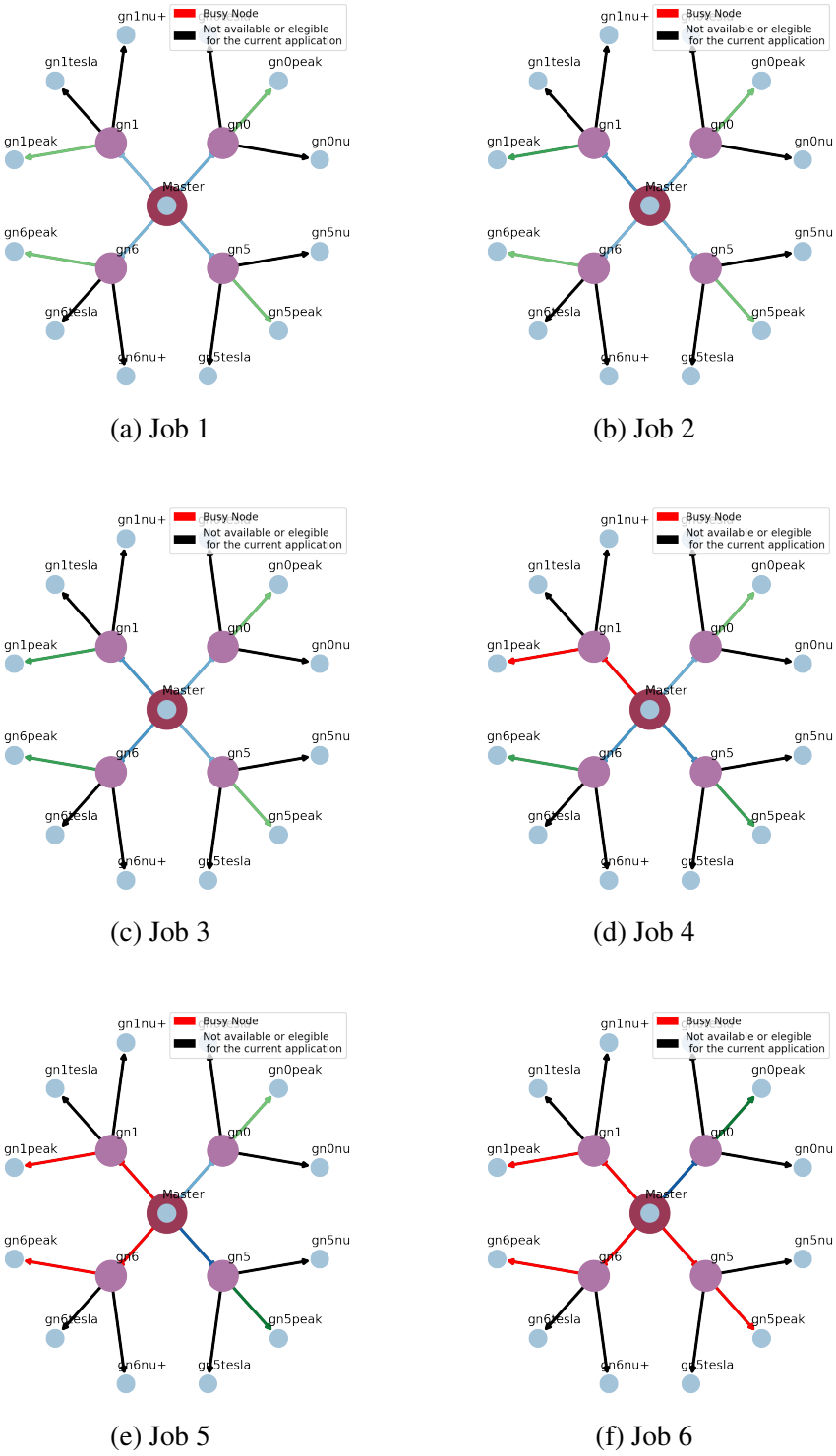
(a) Job 1 (b) Job 2

(c) Job 3 (d) Job 4

(e) Job 5 (f) Job 6

Figure 22: First job scheduling applying the occupancy trend with temperature distributed policy

D.2 Temperature greedy



(a) Job 1

(b) Job 2

(c) Job 3

(d) Job 4

(e) Job 5

(f) Job 6

Figure 23: First job scheduling applying the occupancy trend with temperature greedy policy