MASTER IN HIGH PERFORMANCE COMPUTING

Data Management Tools for NFFA-EUROPE Project

Supervisor:

Prof. Stefano COZZINI,

Candidate:

Ahmed Mohamed Saleh Hassanin KHALIL

5th EDITION
2018–2019

Abstract

This thesis discusses and presents some developments toward new data services within the EU NFFA-EUROPE project. The work performed originates by the need to rationalize and organize large scientific data-sets using a FAIR approach. The activity leverages on results obtained in previous MHPC work and tackle some of the issues about FAIR principle that are coming out due to an increase in size of variety of the original datasets.

More specifically the overall goal of the thesis is to setup well organized data services to manage all the SEM images coming from different sources and partner within the NFFA-EUROPE project.

The specific goals within this thesis are the following;

- Creation of python application to collect and enrich metadata for SEM images coming from different sources.
- Develop a massive parallel processing approach to be able to reduce time in collecting metadata on a large amount of images.
- Plan and develop of an easy to setup and portable computational ecosystem to accomplish the above goal based on Kubernetes and Spark, with the idea to easily deploy in on different computational infrastructure.
- Measure performance on different computational infrastructure of the massive data processing.

Keywords: Kubernetes, Docker, Apache Spark

Dedicated to my parents

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor Prof. Stefano Cozzini for his help, support, expertise, and giving me the opportunity to work on this challenging HPC project. The work led to the implementation of this thesis within the NFFA-EUROPE project is funded by CNR-IOM. I would like to thank CNR-IOM for sponsoring this thesis project. I acknowledge the whole HPC lecturers and tutors for their nice explanation and help. My sincere thanks to Layla Martin-Samos, Nicolas Salles, and Alberto Sartori for their help, nice explanation, and encouragement. I am thankful to Rossella Aversa for the discussions and useful comments. My thanks to my MHPC classmates for the long discussions and exchanging ideas. I acknowledge Stefano Cozzini Team and his students for the discussions. I would like to thank the eXact Lab members Giuseppe Piero Brandino, Eugenio Naselli, Marco Cesaratto, Alberto Chiusole, and Francesco De Giorgi for the discussions and help in resolving technical problems. I would also like to thank Claudia Parma and Mila Bottegal for their help in sorting out bureaucratic paperwork. I am grateful to my family for their continuous support and reassurance during this MHPC study.

Contents

Chapter 1

The NFFA-EUROPE Project

This thesis work is embedded within the scientific data management activities of NFFA-EUROPE project [1]. The project provides open access to large and distribute European scientific infrastructure that allows the implementation of integrating multidisciplinary research at the nanoscale, extending from synthesis to nanocharacterization including theory and numerical simulation. One of the crucial goals within NFFA-EUROPE project was to set up an overarching information and Data management Repository Platform (IDRP) for nanoscience multidisciplinary research and associated data management tools.

In this chapter, we will briefly illustrate the Data infrastructure and some data services recently built on the top of it. We will focus in particular on the services to manage a large amount of Scanning Electron Microscope (SEM) images coming from the CNR-IOM instruments and, more recently from the other instruments coming from the wider NFFA scientific community.

1.1 The NFFA data infrastructure

NFFA-EUROPE data platform is dedicated for serving nanoscience research, and within such distributed research infrastructure the first overarching Information and Data management Repository Platform (IDRP) was developed for nanoscience. Within the same efforts the project defined as well a nanoscience the scientific metadata standards for data sharing among the NFFA-EUROPE community and also beyond. The standard is now a component of an open collaborative initiative within the framework of the Research Data Alliance (RDA) [2]. The NFFA-EUROPE data infrastructure is composed by the following key elements:

1.1.1 NFFA Datashare

NFFA-EUROPE Datashare service provides cloud platform and secure storage for all the NFFA-EUROPE researchers. Every researcher has an automated access to NFFA Datashare under NFFA credentials, whereas a file share and the collaboration platform are hosted in the local server at the CNR-IOM [3] under its custody. The produced research data are retained by the NFFA-EUROPE rules, and hence all researchers have efficient control on their produced data, so that they can access, process, or retrieve their data at any time.

1.1.2 Information and Data Repository Platform (IDRP.nffa.ue)

NFFA-EUROPE established the first integrated Information and Data Repository Platform (IDRP) for the nanoscience community. The goal of such IDRP repository is to record all relevant metadata for a given nanoscience project/experiment, by means of a structured on-line data catalogue where users can access, share, and widely publish their data to larger research groups. This facilitates the process of re-producibility and re-preparations of scientific experiments. Additionally, it allows the scientists, from the nanoscience community, to perform comparison besides validation studies on the results obtained, from the shared data, using different methods.

1.1.3 SEM images data services: sem-classifier.nffa.eu

In addition to data storage and metadata registration, the data infrastructure aims to offer also advanced analysis tools for raw data. The first of these services is a SEM image classification, which has been developed by CNR-IOM. This tool was developed due to the fact that CNR-IOM SEM facility stores over a 10 year services period a lot of images collected by different research groups, for a total of about 15000, unique images in the Tagged Image File Format format (TIFF). Among them, 18577 have been manually labelled to create a full dataset, called dataset1, composed by ten categories. Such dataset has been then used to train a neural network to automatically identifies images. A first data services has then developed to offer a way to identify and classify any image accordingly to this ten categories.

1.2 The SEM images problem

The on-going scientific activities of this thesis is a prosecution of started work in 2016 and is focused on the data produced by a single instrument, the Scanning Electron Microscope (SEM) located at CNR-IOM. This is an extremely versatile instrument, routinely used in nanoscience and nanotechnology to explore the structure of materials with spatial resolution down to 1 nanometer. Almost 150,000 images were collected and such large amount of data were the basis for several interesting MHPC research theses. We remark that such data-set was actually collected in June 2016 and span over a period of 8/10 years of activity in the TASC laboratory at CNR-IOM in Trieste. From June 2016 on the SEM images are constantly increasing and for this reason now the storage associated to instruments in synchronized constantly on the Datashare facility of the institute. A further set of images coming from another instruments located at CNR-IOM has been recently shared and synchronized on the Datashare, making thus the total number of images larger and more variegate. Moreover, the results so far obtained on the classification procedure have been presented to other NFFA-EUROPE partners and some of the SEM instruments within the project showed interest and started sharing some initial images on the Datashare. This means that new sources of SEM data are foreseen to arrive on the the Datashare. We thus face the problem to classify and store them in a FAIR [4] way, keeping track of all the embedded metadata. The aim of this thesis is therefore to develop some tools to collect and well organize all metadata pertaining to the images we are collecting. The elements of such tools are described in Chapter 2, while in Chapter 3 we presented the scalable infrastructure that can allow us to process a massive amount of data.

Chapter 2

Scientific Image Analyses

As we have mentioned in Chapter 1, the increasingly number of images obtained by means of SEM instruments within the NFFA-EUROPE project requires us to arrange a clear and unique procedure to be handled in a proper way to make the dataset FAIR. Such procedure includes the following steps:

1. extracting the standard metadata within the TIFF format coming from different SEM instruments within NFFA-EUROPE project,
2. measuring the SEM image scales and performing an Object Character Recognition (OCR) to finally calculate the image pixel size,
3. running SEM linear regression/classification (SEM inference problem) and
4. committing all the resultant metadata to a dedicated MySQL database.

This chapter is aimed to describe the SEM dataset and Python script developed in this work to implement the steps sketched above.

2.1 The SEM dataset

The input datasets analyzed in this work are scientific nanoscale images produced by a SEM instrument in a TIFF format, at different nanoscience research groups within the NFFA-EUROPE project. The SEM images are available at the cloud storage platform provided by NFFA-EUROPE Datashare service as well as the C3HPC cluster in a *nested* structure. Figure 2.1 presents an example of SEM images from three different research groups (CNR-IOM, TASC, and ZEISS) within the NFFA-EUROPE project.

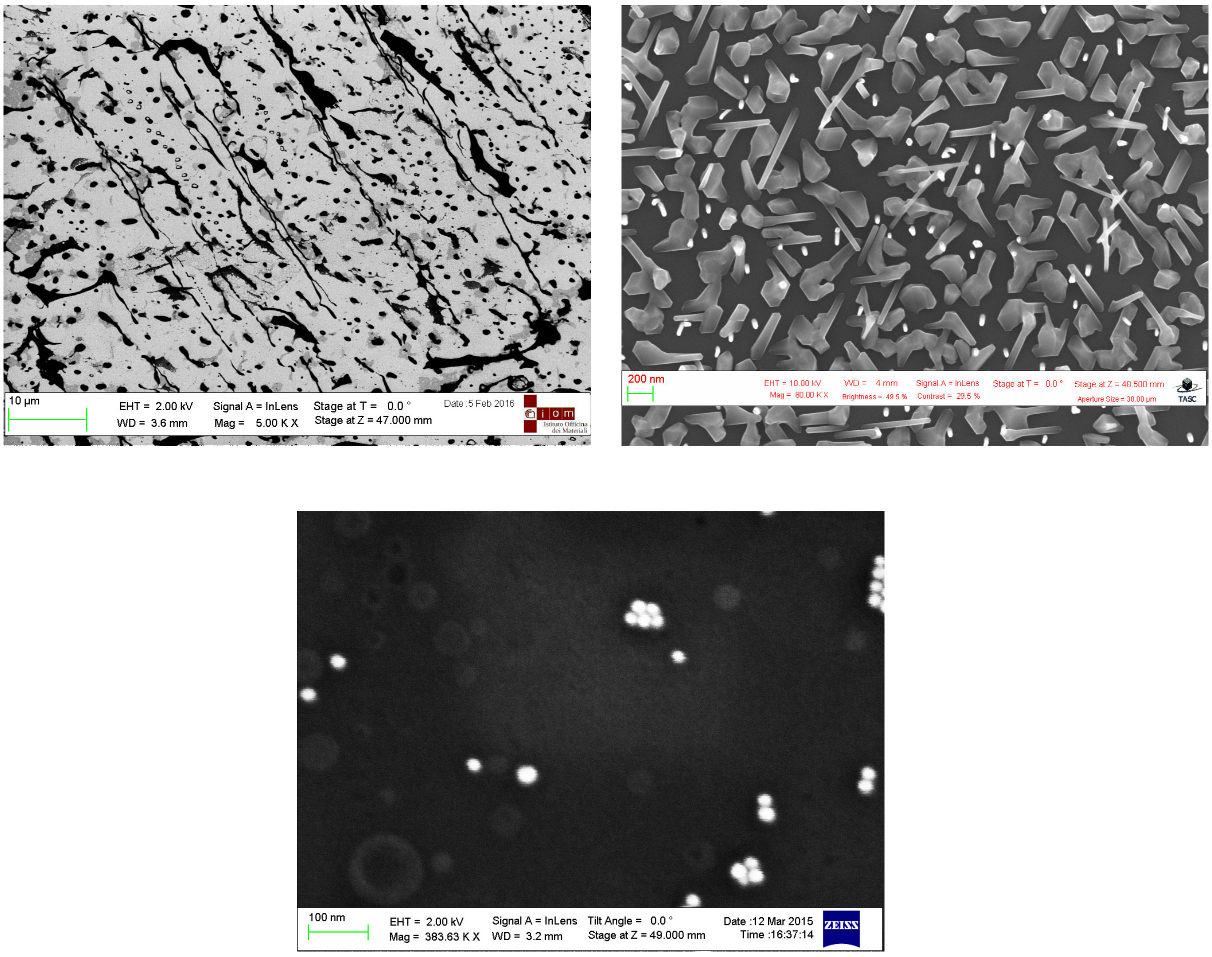


FIGURE 2.1: An example of three images produced by the SEM instrument at different nanoscience research groups - CNR-IOM (top left), TASC (top right), and ZEISS (bottom middle) - within the NFFA-EUROPE project.

In the following sections, we present the main four parts of SEM analyses executed by the Python code.

2.2 Extracting the standard metadata within TIFF format

Metadata is data that describes other data. Namely, it provides descriptive information about the main dataset (SEM images) contents, such as metadata of image width, length, resolution unit, brightness, creation date, electron beam time, noise reduction, and pixel size. Metadata can describe individual files, single objects, or complete collections and has the criteria of being descriptive, structural, and administrative or technical. The metadata of SEM images goes into two main categories:

1. Exchangeable image file format (Exif) metadata;
2. Instrument metadata.

Both of these two types are recorded and stored in the actual image file by means of SEM instrument. The SEM metadata has been accessed by Pillow/PIL library [5] which allows us to fetch the typical Exif metadata, for every single SEM image overall the dataset, by means of `SEMEXIF` function defined in Listing 2.1.

```python
@property
def SEMEXIF(self):
    self.exif_dict = dict([(k, v) for v, k in ExifTags.TAGS.items()])
    # make a list of all available keys
    self.exif_keys = [key for key in self.exif_dict]
    # then create a list of the corresponding tag numbers
    self.exif_numbers = [self.exif_dict[k] for k in self.exif_keys]
    return self.exif_dict, self.exif_keys, self.exif_numbers
```

LISTING 2.1: An auxiliary function of `SEMEXIF` defined for extracting the standard Exif metadata.

The `SEMEXIF` function returns three objects of a dictionary and two lists shown in snippet 2.2. The Exif dictionary `self.exif_dict` includes all Exif keys together with corresponding tag numbers given as values. The lists `self.exif_keys` and `self.exif_numbers` contain the keys and corresponding values (Exif tags/tag numbers), respectively.

```python
self.exif_dict = {'ProcessingSoftware': 11, 'NewSubfileType': 254, 'SubfileType': 255, '
    ImageWidth': 256, 'ImageLength': 257, 'BitsPerSample': 258, 'Compression': 259, ...}
self.exif_keys = ['ProcessingSoftware', 'NewSubfileType', 'SubfileType', 'ImageWidth', '
    ImageLength', 'BitsPerSample', 'Compression', ...]
self.exif_numbers = [11, 254, 255, 256, 257, 258, 259, ...]
```

LISTING 2.2: Contents of the standard Exif dictionary along with lists of keys and values.

The returned Exif tags besides `ImageMetadata` function shown in Listing 2.3 enable us to obtain all SEM Exif metadata available in the image.

```python
def ImageMetadata(self, imm):
    self.image_metadata = imm.tag
    self.image_tags = np.array(self.image_metadata)
    return self.image_metadata, self.image_tags
```

LISTING 2.3: The metadata function `ImageMetadata` defined for extracting the SEM images metadata and corresponding available tags in the image. The function takes one argument `imm`, which is the image object returned from opening the image by means of Pillow library.

The structure of the SEM Exif metadata for a single image from the SEM dataset is presented in Listing 2.4.

```
{'NewSubfileType': 0, 'ImageWidth': 1024, 'ImageLength': 768, 'BitsPerSample': 8, '
    Compression': 1, ..., 'NoiseProfile': 'Not found'}
```

LISTING 2.4: An example of structure of the Exif metadata for one single SEM image. Exif keys that are not existing in the image are assigned to “Not found" value.

All SEM images have Exif metadata either in a partial or a full format. A “Not found" value is assigned for those Exif keys that are not available in the image.

Extracting the Instrument metadata has been proceeded, as the Exif one, by means of SEM tags and image object, whereas all Instrument metadata of the SEM images are attributed to a specific tag number based on the characteristics of SEM instrument. In our analysis the Instrument tag was assigned to a value of 34118. This tag value does exist in all SEM images, particularly those belong to CNR-IOM research groups within the NFFA-EUROPE project, and it differs from one SEM instrument to another. The structure of an obtainable Instrument metadata is shown in Listing 2.5.

```
{'DP_VENT_INVALID_REASON': 'Vent inhibit = Beam Present', 'DP_OPTIMODE': 'OptiBeam Mode =
     Resolution', 'DP_FIXED_APERTURE': 'VP Aperture = No', 'DP_INPUT_LUT_MODE': 'Input
    LUT Mode = Transparent', 'DP_BSD_AUTOLEVEL_MODE': 'BSD Autolevel Mode = Normal', ..,
    'AP_AR_GAS_FLOW_ACTUAL': 'Argon Gas Flow is =  0.0 %', ..., 'AP_GAMMA': 'Gamma =
    1.0000', ..., 'AP_PIXEL_SIZE': 'Pixel Size = 4.218 nm', ...}
```

LISTING 2.5: An example of structure of the Instrument metadata for one single SEM image. Exif keys that do not exist in the image are assigned to “Not found" value.

We found that the Instrument metadata of SEM images is distributed over two main classes: (a) images with Instrument metadata; (b) images without Instrument metadata. Nevertheless, the Instrument tag is available in the entire SEM dataset. In Fig. 2.2, we show statistics for the two classes of complete and partial Instrument metadata over a dataset of 150,000 SEM images.

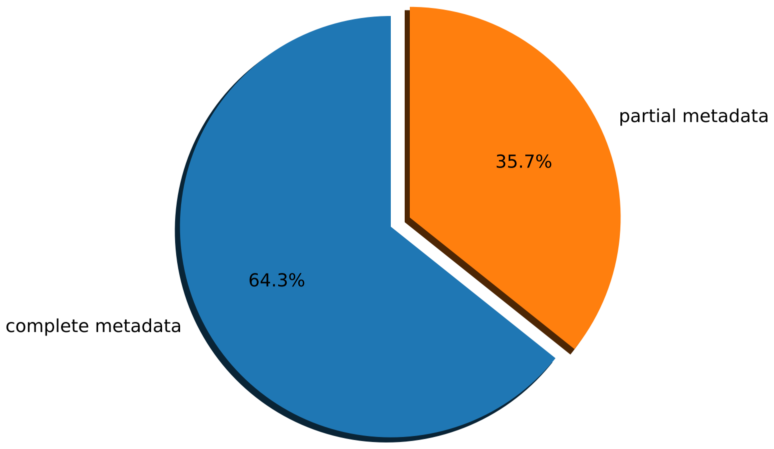


FIGURE 2.2: Percentage of SEM images of complete (64.3%) and partial (35.7%) Instrument metadata. Images of complete metadata contain both Exif and Instrument parameters, while images of partial metadata have Exif parameters only. Namely, the value of pixel size is not available. The SEM dataset used to establish this pie chart is available on the C3HPC high performance computing cluster [6, 7] under `/lustre/exact/SEM-images/new_dati_SEM/` directory.

2.3 Measuring SEM scales and performing OCR

Scale problem analysis of SEM images has been investigated previously in MHPC work done by Coronica [8]. This problem is originated from the fact that SEM images are distributed over a broad nanoscale band ranging from 1 pm to 300 µm (see Fig. 2.3). Therefore, it has been thought that splitting SEM images based on their scale could improve the unsupervised machine learning techniques being developed [8–11] for SEM images classification.

Similar to what has been performed in [8], we used in this work `OpenCV` library [12] for image segmentation and contour detection, in addition to, `Tesseract` engine for Optical Character Recognition (OCR). The algorithm can be described in the following clauses:

- opening the image using OpenCV library and extracting the information bar from the image body,
- cleaning the information bar by means of thresholding methods, such as `THRESH_BINARY`, `THRESH_BINARY_INV`, and `THRESH_OTSU`,
- selecting the scale segment and measuring its width,
- performing OCR for the purpose of reading the digit as well as unit and

- calculating the pixel size for every image.

Figure 2.3 shows the ration of pixel size range, over which a SEM dataset from CNR-IOM hosted on NFFA Datashare platform (datashare.iom.cnr.it) is distributed.

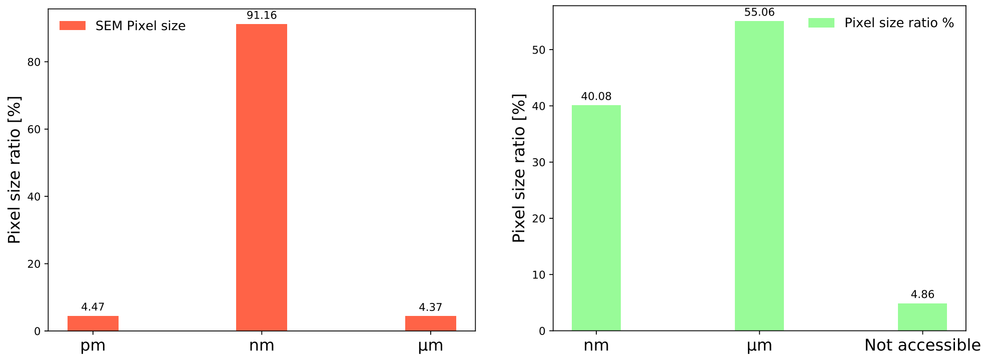


FIGURE 2.3: Range of pixel size ratio of CNR-IOM dataset hosted on NFFA Datashare platform (datashare.iom.cnr.it). *Left panel*: Range of pixel size ratio extracted from the original metadata saved in the image. *Right panel*: Range of pixel size ratio resulted from our analysis using OpenCV library and OCR engine. A fraction of ∼5% of the data is *not accessible*. Namely, the scale, digit, and unit are not readable by the SEM Python code, and then the quantity of pixel size is not measured.

The third part of the SEM Python code (inference) is presented in § 2.4.

2.4 Inference of SEM images

Classification of SEM images has been presented in MHPC work in 2016 by Aversa [10] through supervised machine learning technique using TensorFlow [13]. A deep neural network has been trained over large SEM dataset, which automatically assigned for that purpose. Similar to what has been performed in Refs. [10] and [11], we employed the pre-trained SEM model [14] in order to measure the SEM image inference on our distributed software infrastructure. The essential operations used in measuring the SEM inference are given by the following functions:

- reading the pre-trained model using `tf.gfile.GFile` TensorFlow module;

```
def ReadModel(self, semmodel):
    with tf.gfile.GFile(semmodel, 'rb') as f:
        model_data = f.read()
    return model_data
```

- running the model with saving the score rates into a dictionary;

```python
def RunModel(self, model_data_bc_value, image_array):
    graph = tf.Graph()
    with graph.as_default() as g:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(model_data_bc_value)
        tf.import_graph_def(graph_def, name='')
        sem_human_string, sem_score = [], []
        with tf.Session() as sess:
            softmax_tensor = sess.graph.get_tensor_by_name('final_result:0')
            predictions = sess.run(softmax_tensor, {'DecodeJpeg:0': image_array})
            top_k = predictions[0].argsort()[-len(predictions[0]):][::-1]
            for node_id in top_k:
                human_string = sem_labels[node_id]
                score = predictions[0][node_id]
                score = ('%.5f') % score
                sem_human_string = (*sem_human_string, human_string)
                sem_score = (*sem_score, score)
            self.inference  = dict(zip(sem_human_string, sem_score))
            return self.inference

```

In the above function, two nodes are essentially required for running the model and performing the image inference: (a) final result node `final_result`; (b) decode image node `DecodeJpeg`. Accessing these information is carried out using `TensorBoard` model visualization tool [15], and hence the SEM model graph is presented in Fig 2.4.

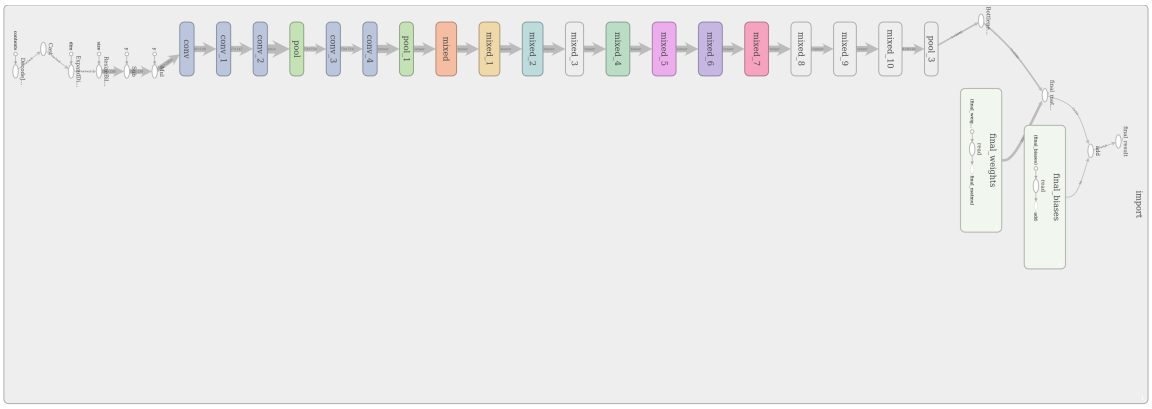


FIGURE 2.4: SEM model graph obtained using `TensorBoard`.

As a consequence of running the SEM inference over a dataset of 16000 SEM images from `datashare.iom.cnr.it`, the statistics of inference high score ($\geqslant 0.8$) for the SEM 10 categories is presented in Fig. 2.5.

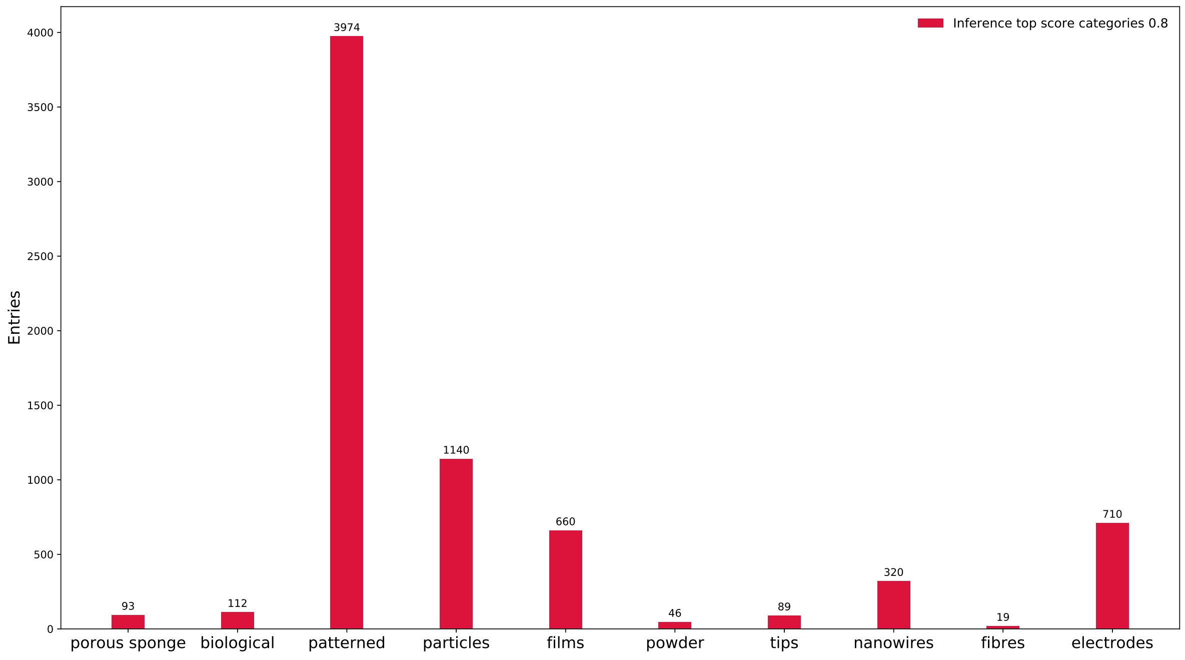


FIGURE 2.5: Inference top score categories $\geqslant 0.8$.

As aforementioned in the beginning of this Chapter, the analyses have been performed on SEM images of TIFF format. On the contrary, the SEM model was pre-trained over JPG SEM images. Therefore, the returned TIFF object has been converted to JPG one, so that the resultant image content (`image_array`) and the SEM model became compatible for running SEM inference analysis:

```
get_jpg = imm.convert('RGB')
image_array = np.squeeze(np.array(get_jpg)[:,:,0:3])
```

We describe in § 2.5 the database of SEM images.

2.5 SEM Database

Our SEM analyses resulted in significant amount of metadata that provides comprehensive description of the scientific images. Such SEM metadata has been managed into relational database by means of Structured Query Language (SQL). The database can be accessed using MySQL [16], which is a portable, high performance, and scalable database platform widely used in Data Management and Data Science disciplines.

We built up the SEM database on OpenStack of the CNR-IOM computing system [17] where MySQL is installed on the base layer, while phpMyAdmin [18] is set up on the top one. phpMyAdmin is a web-interface tool written in PHP language and made for handling and administrating MySQL database. These data management efficient tools have enabled us to structure and organize our analyses output and easily share it with a wide community of nanoscience research from the NFFA-EUROPE project.

Prior to the use of MySQL and phpMyAdmin, the following configuration parameters have been set in `mysql.cnf` file, see Listing 2.6.

```
[mysqld]
max_connections = 300
innodb_strict_mode = 0
max_allowed_packet = 1G
innodb_log_file_size = 2G
innodb_log_buffer_size = 512M
innodb_file_format = barracuda
innodb_file_per_table = 1
sql-mode="NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION"
```

LISTING 2.6: Database configurations.

The above configuration parameters set under `[mysqld]` section are aimed to optimize MySQL database for receiving long query buffers of SEM metadata. Such configurations are followed by restarting MySQL and phpMyAdmin services:

```
systemctl restart mysql.service
systemctl restart apache2
```

The SEM database is an organized collection of scientific images metadata spanned over four main tables; Exif, Instrument, Pixel size, and Inference metadata. We created our database by means of Python via importing `mysql.connector` and the `DatabaseConnection` function defined in Listing 2.7.

```
@property
def DatabaseConnection(self):
    self.db_connection= mysql.connector.connect(
    host= "host",
    user= "user",
    passwd= "passwd",
    database=None,
    charset='utf8',
    use_unicode=True)
    # creating 'db_cursor' instance class to execute the 'SQL' statements
    self.db_cursor = self.db_connection.cursor()
    # setting mysql to global mode for very long tables
    self.db_cursor.execute("SET @@global.sql_mode= '';")
    return self.db_connection, self.db_cursor
```

LISTING 2.7: A function of `DatabaseConnection` defined in the SEM Python code.

After setting the communication with MySQL, the SEM database has been created using the function shown in Listing 2.8.

```
def CreateDatabase(self, dbname=None):
    self.db_cursor.execute("CREATE DATABASE IF NOT EXISTS %s" %(dbname))
    self.db_cursor.execute("USE %s" %(dbname))
    return
```

LISTING 2.8: A function of `CreateDatabase` defined in the SEM Python code.

The SEM database structure is shown in Fig. 2.6.

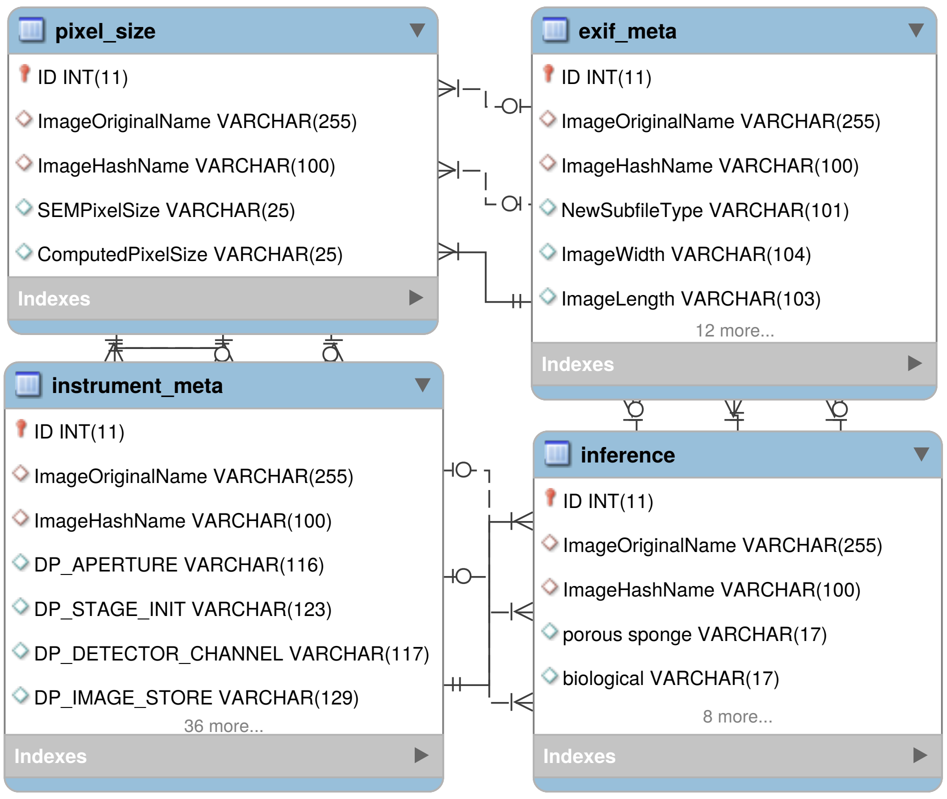

pixel_size
ID INT(11)
ImageOriginalName VARCHAR(255)
ImageHashName VARCHAR(100)
SEMPixelSize VARCHAR(25)
ComputedPixelSize VARCHAR(25)
Indexes
exif_meta
ID INT(11)
ImageOriginalName VARCHAR(255)
ImageHashName VARCHAR(100)
NewSubfileType VARCHAR(101)
ImageWidth VARCHAR(104)
ImageLength VARCHAR(103)
12 more...
Indexes
instrument_meta
ID INT(11)
ImageOriginalName VARCHAR(255)
ImageHashName VARCHAR(100)
DP_APERTURE VARCHAR(116)
DP_STAGE_INIT VARCHAR(123)
DP_DETECTOR_CHANNEL VARCHAR(117)
DP_IMAGE_STORE VARCHAR(129)
36 more...
Indexes
inference
ID INT(11)
ImageOriginalName VARCHAR(255)
ImageHashName VARCHAR(100)
porous sponge VARCHAR(17)
biological VARCHAR(17)
8 more...
Indexes

FIGURE 2.6: A general structure of SEM database obtained using MySQL Workbench [19] visualizing database architects tool. Four main tables contain all the resultant metadata of Exif, Instrument, Pixel size, and inference analyses.

Creating the four main tables of the SEM database has been implemented by means of `CreateTables` function presented in Listing 2.9.

```python
def CreateTables(self, tables_list, dictname, dict):
    # get metadata values and escape strings to avoid sql injection
    values = [self.EscapeStrings(str(y)) for y in dict.values()]
    # get metadata keys and set VARCHAR size
    meta_keys = ', '.join("`" + str(x) + "` VARCHAR(%d) DEFAULT NULL" % int(len(y)+100)
    for x, y in zip(dict.keys(), values))
    self.db_cursor.execute("CREATE TABLE IF NOT EXISTS %s ( %s );" % (dictname, meta_keys
    ))
    self.db_cursor.execute("SHOW COLUMNS FROM %s" %(dictname))
    # make a list of all tables
    list_columns = self.ListMeta
    if 'ID' and 'ImageOriginalName' and 'ImageHashName' in list_columns:
        print('ID, ImageOriginalName, and ImageHashName columns do exist!')
    else:
        self.db_cursor.execute("ALTER TABLE %s ADD ID INT(11) NOT NULL
        AUTO_INCREMENT PRIMARY KEY FIRST, \
        ADD ImageOriginalName VARCHAR(255) DEFAULT NULL AFTER ID, \
        ADD ImageHashName VARCHAR(100) DEFAULT NULL AFTER ImageOriginalName"% (dictname))
    return
```

LISTING 2.9: A function of `CreateTables` defined in the SEM Python code.

The `CreateTables` function sets all tables for hosting SEM buffers with creating three additional columns of ID, ImageOriginalName, and ImageHashName which represent the order along with image original and hash names, respectively. The function calls two auxiliary functions needed for the following purposes:

- avoiding SQL injection [20] attack[1] by means of `EscapeStrings` function.
- listing all column objects of the database using `ListMeta` function, which is made to avoid re-adding the entries of ID, ImageHashName, and ImageOriginalName into the database.

Every individual SEM image has a unique hash name returned by the following function.

```python
def ImageHash(self, imm):
    image_hash_name = hashlib.md5(imm.tobytes()).hexdigest()
    return image_hash_name
```

LISTING 2.10: A function of `ImageHashName` defined in the SEM code where `imm` is the image object.

The above `ImageHash` function calls message digest algorithm (MD5) module [21] imported from `hashlib` library [22] and returns a unique hash name, which is given based on the

[1]SQL injection is a code injection technique used for malicious SQL statements which can thereby demolish the database.

structure of pixels within the image. Such hash name enables us to identify the database main entry and connect all tables together, and hence specify, e.g., duplicated images over the entire dataset (see Fig. 2.7).

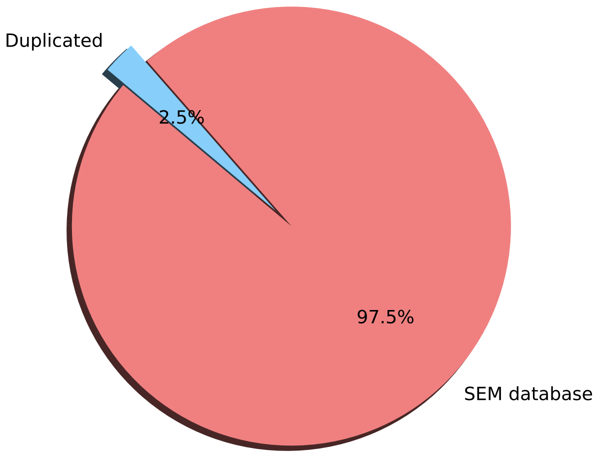


FIGURE 2.7: Ratio of duplicated images within a SEM database of a size of 150 GB available on the C3HPC high performance computing cluster. The corresponding database named `c3e_sem_db` is available on the SEM MySQL.

The SEM analyses code is parallelized by means of Apache Spark framework [23] in *standalone* mode within containerized environment, see Chapter 3 for details. Therefore, MySQL connector `MySQL Connector/J` is used in order to allow communication with the SEM database, see Listing 2.11.

```
def SparkMySQL(self, sem_schema, dbtable=None):
    sem_schema.write.format('jdbc').options(
    url='jdbc:mysql://host:port',
    # to avoide Java exception of encoding problem
    useUnicode='true&characterEncoding=utf8',
    driver='MySQL Connector/J',
    dbtable = dbtable,
    user='user',
    password='password').mode('append').save("append")
    return
```

LISTING 2.11: A function of `SparkMySQL` defined in the parallel SEM code.

The `SparkMySQL` function open communication with MySQL by means of JDBC DriverManager Interface [24], in order to manage resultant metadata into the SEM database.

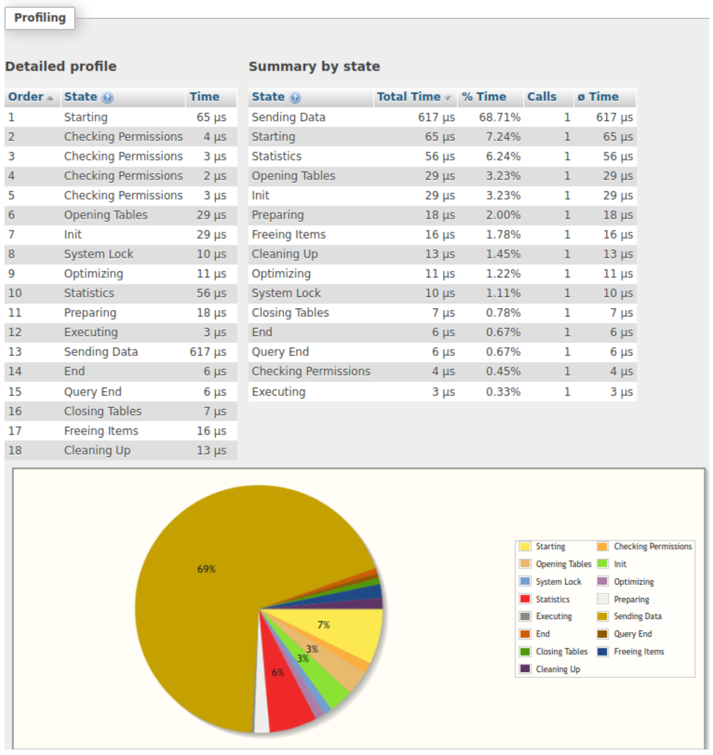
2.5.1 Example of usage and profiling

Profiling the SEM SQL database in finding relevant information/metadata is a crucial matter in evaluating performance. Therefore, we present in this section detailed profile result of an example of SEM metadata usage: fetching a list of SEM images defining inference rates of patterned surface category along with DP_TILTED, ImageWidth, and SEMPixelSize metadata via joining the four SEM metadata tables (*exif_meta, instrument_meta, inference,* and *pixel_size*). This has been performed by means of the SQL query presented in Listing 2.12.

```sql
SELECT inference.ID, inference.ImageOriginalName, inference.ImageHashName,
       inference.`patterned surface`, instrument_meta.DP_TILTED,
       exif_meta.ImageWidth, pixel_size.SEMPixelSize
FROM inference JOIN instrument_meta ON inference.ID = instrument_meta.ID
               JOIN exif_meta ON exif_meta.ID = inference.ID
               JOIN pixel_size ON pixel_size.ID = inference.ID;
```

LISTING 2.12: SQL query of fetching a list of images defining ID, ImageOriginalName, ImageHashName, patterned surface inference rates, DP_TILTED parameter, ImageWidth, and SEMPixelSize metadata. The four tables of SEM SQL database are joined in executing this query.

Detailed profile of the query is shown in Fig. 2.8.


Profiling

Detailed profile

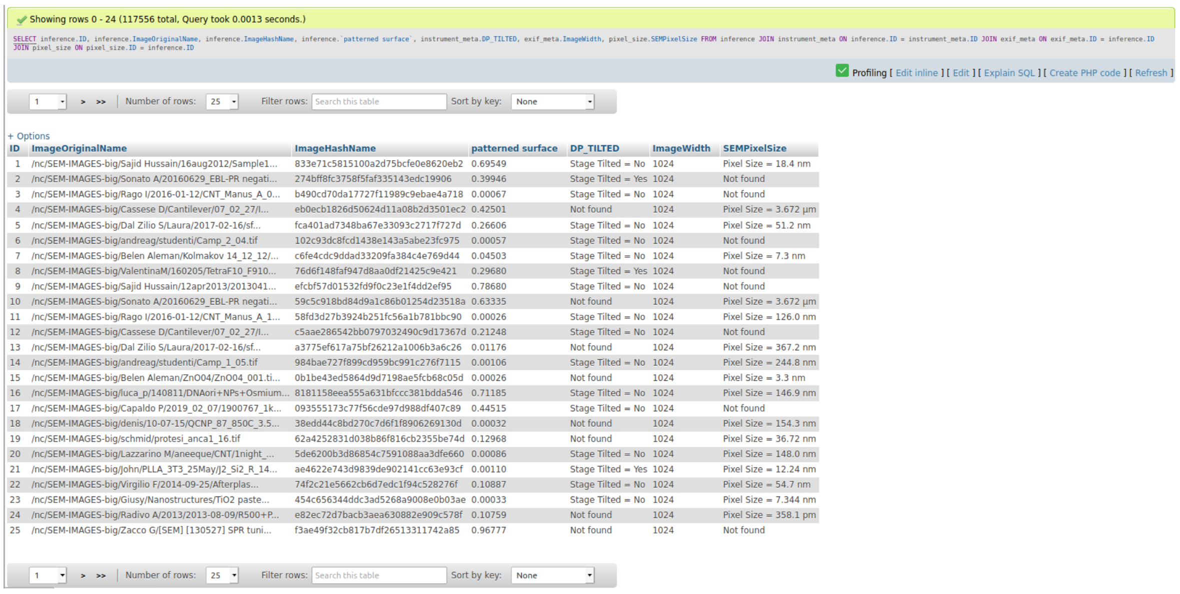
Order	State	Time
1	Starting	65 µs
2	Checking Permissions	4 µs
3	Checking Permissions	3 µs
4	Checking Permissions	2 µs
5	Checking Permissions	3 µs
6	Opening Tables	29 µs
7	Init	29 µs
8	System Lock	10 µs
9	Optimizing	11 µs
10	Statistics	56 µs
11	Preparing	18 µs
12	Executing	3 µs
13	Sending Data	617 µs
14	End	6 µs
15	Query End	6 µs
16	Closing Tables	7 µs
17	Freeing Items	16 µs
18	Cleaning Up	13 µs

Summary by state

State	Total Time	% Time	Calls	ø Time
Sending Data	617 µs	68.71%	1	617 µs
Starting	65 µs	7.24%	1	65 µs
Statistics	56 µs	6.24%	1	56 µs
Opening Tables	29 µs	3.23%	1	29 µs
Init	29 µs	3.23%	1	29 µs
Preparing	18 µs	2.00%	1	18 µs
Freeing Items	16 µs	1.78%	1	16 µs
Cleaning Up	13 µs	1.45%	1	13 µs
Optimizing	11 µs	1.22%	1	11 µs
System Lock	10 µs	1.11%	1	10 µs
Closing Tables	7 µs	0.78%	1	7 µs
End	6 µs	0.67%	1	6 µs
Query End	6 µs	0.67%	1	6 µs
Checking Permissions	4 µs	0.45%	1	4 µs
Executing	3 µs	0.33%	1	3 µs

FIGURE 2.8: Profiling the SQL query shown in Listing 2.12.

The Total execution time of the above SQL query is in the order of µs. The query, its output, and time taken are presented in Fig.2.9.



Showing rows 0 - 24 (117556 total, Query took 0.0013 seconds.)

```
SELECT inference.ID, inference.ImageOriginalName, inference.ImageHashName, inference.`patterned surface`, instrument_meta.DP_TILTED, exif_meta.ImageWidth, pixel_size.SEMPixelSize FROM inference JOIN instrument_meta ON inference.ID = instrument_meta.ID JOIN exif_meta ON exif_meta.ID = inference.ID
JOIN pixel_size ON pixel_size.ID = inference.ID
```

Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]
1 > >> Number of rows: 25 Filter rows: Search this table Sort by key: None
+ Options

ID	ImageOriginalName	ImageHashName	patterned surface	DP_TILTED	ImageWidth	SEMPixelSize
1	/nc/SEM-IMAGES-big/Sajid Hussain/16aug2012/Sample1...	833e71c5815100a2d75bcfe0e8620eb2	0.69549	Stage Tilted = No	1024	Pixel Size = 18.4 nm
2	/nc/SEM-IMAGES-big/Sonato A/20160629_EBL-PR negati...	274bff8fc3758f5faf335143edc19906	0.39946	Stage Tilted = Yes	1024	Not found
3	/nc/SEM-IMAGES-big/Rago I/2016-01-12/CNT_Manus_A_0...	b490cd70da17727f11989c9ebae4a718	0.00067	Stage Tilted = No	1024	Not found
4	/nc/SEM-IMAGES-big/Cassese D/Cantilever/07_02_27/I...	eb0ecb1826d50624d11a08b2d3501ec2	0.42501	Not found	1024	Pixel Size = 3.672 µm
5	/nc/SEM-IMAGES-big/Dal Zilio S/Laura/2017-02-16/sf...	fca401ad7348ba67e33093c2717f727d	0.26606	Stage Tilted = No	1024	Pixel Size = 51.2 nm
6	/nc/SEM-IMAGES-big/andreag/studenti/Camp_2_04.tif	102c93dc8fcd1438e143a5abe23fc975	0.00057	Stage Tilted = No	1024	Not found
7	/nc/SEM-IMAGES-big/Belen Aleman/Kolmakov 14_12_12/...	c6fe4cdc9ddad33209fa384c4e769d44	0.04503	Stage Tilted = No	1024	Pixel Size = 7.3 nm
8	/nc/SEM-IMAGES-big/ValentinaM/160205/TetraF10_F910...	76d6f148faf947d8aa0df21425c9e421	0.29680	Stage Tilted = Yes	1024	Not found
9	/nc/SEM-IMAGES-big/Sajid Hussain/12apr2013/2013041...	efcbf57d01532fd9f0c23e1f4dd2ef95	0.78680	Stage Tilted = No	1024	Not found
10	/nc/SEM-IMAGES-big/Sonato A/20160629_EBL-PR negati...	59c5c918bd84d9a1c86b01254d23518a	0.63335	Not found	1024	Pixel Size = 3.672 µm
11	/nc/SEM-IMAGES-big/Rago I/2016-01-12/CNT_Manus_A_1...	58fd3d27b3924b251fc56a1b781bbc90	0.00026	Stage Tilted = No	1024	Pixel Size = 126.0 nm
12	/nc/SEM-IMAGES-big/Cassese D/Cantilever/07_02_27/I...	c5aae286542bb0797032490c9d17367d	0.21248	Stage Tilted = No	1024	Not found
13	/nc/SEM-IMAGES-big/Dal Zilio S/Laura/2017-02-16/sf...	a3775ef617a75bf26212a1006b3a6c26	0.01176	Not found	1024	Pixel Size = 367.2 nm
14	/nc/SEM-IMAGES-big/andreag/studenti/Camp_1_05.tif	984bae727f899cd959bc991c276f7115	0.00106	Stage Tilted = No	1024	Pixel Size = 244.8 nm
15	/nc/SEM-IMAGES-big/Belen Aleman/ZnO04/ZnO04_001.ti...	0b1be43ed5864d9d7198ae5fcb68c05d	0.00026	Not found	1024	Pixel Size = 3.3 nm
16	/nc/SEM-IMAGES-big/luca_p/140811/DNAori+NPs+Osmium...	8181158eea555a631bfccc381bdda546	0.71185	Stage Tilted = No	1024	Pixel Size = 146.9 nm
17	/nc/SEM-IMAGES-big/Capaldo P/2019_02_07/1900767_1k...	093555173c77f56cde97d988df407c89	0.44515	Stage Tilted = No	1024	Not found
18	/nc/SEM-IMAGES-big/denis/10-07-15/QCNP_87_850C_3.5...	38edd44c8bd270c7d6f1f8906269130d	0.00032	Not found	1024	Pixel Size = 154.3 nm
19	/nc/SEM-IMAGES-big/schmid/protesi_anca1_16.tif	62a4252831d038b86f816cb2355be74d	0.12968	Not found	1024	Pixel Size = 36.72 nm
20	/nc/SEM-IMAGES-big/Lazzarino M/aneeque/CNT/1night_...	5de6200b3d86854c7591088aa3dfe660	0.00086	Stage Tilted = No	1024	Pixel Size = 148.0 nm
21	/nc/SEM-IMAGES-big/John/PLLA_3T3_25May/J2_Si2_R_14...	ae4622e743d9839de902141cc63e93cf	0.00110	Stage Tilted = Yes	1024	Pixel Size = 12.24 nm
22	/nc/SEM-IMAGES-big/Virgilio F/2014-09-25/Afterplas...	74f2c21e5662cb6d7edc1f94c528276f	0.10887	Stage Tilted = No	1024	Pixel Size = 54.7 nm
23	/nc/SEM-IMAGES-big/Giusy/Nanostructures/TiO2 paste...	454c656344ddc3ad5268a9008e0b03ae	0.00033	Stage Tilted = No	1024	Pixel Size = 7.344 nm
24	/nc/SEM-IMAGES-big/Radivo A/2013/2013-08-09/R500+P...	e82ec72d7bacb3aea630882e909c578f	0.10759	Not found	1024	Pixel Size = 358.1 pm
25	/nc/SEM-IMAGES-big/Zacco G/[SEM] [130527] SPR tuni...	f3ae49f32cb817b7df26513311742a85	0.96777	Not found	1024	Not found

1 > >> Number of rows: 25 Filter rows: Search this table Sort by key: None

FIGURE 2.9: SQL query of joining the four SEM metadata tables (*exif_meta, instrument_meta, inference,* and *pixel_size*). The query is executed in the SQL shell of phpMyAdmin. The total execution time of the query is in the order of µs.

Chapter 3

Kubernets and Apache Spark Containers

In this chapter we present and discuss our approach of developing a distributed and scalable software infrastructure in order to process massive amounts of images; this has been accomplished via containerizing the application, i.e., the script described in the previous chapter by means of Docker. The isolation property of containers is a great advantage in developing applications that can run on multiple platforms, but unfortunately so far this isolating ability of Docker containers limited the possibility of exploiting a large number of physical hosts. To overcome this problem we use Kubernetes described in the following sections.

3.1 Kubernetes

Kubernetes [25] is a container-orchestration engine and portable open source platform. It is originally developed by Google for managing containerized services and applications, such as Docker technology [26], across a cluster of computing nodes. Kubernetes is widely used in data science and aimed to provide better handling and automation for applications within containerized environments. In our work, we have used Kubernetes to orchestrate Spark containers installed on our infrastructures. In § 3.1.1 we present Kubernetes deployment process, while in § 3.2 Apache Spark architecture and installation are discussed.

3.1.1 Kubernetes deployment

Deployment of Kubernetes on the top of set of nodes allows us to have a Kubernetes cluster, which brings individual Docker daemons into communication and orchestrate the workload. In this work, we set up a cluster that consists of two fundamental modalities:

- Standalone cluster configuration that allows the nodes membership to be recognized and transparent to every single node of the cluster;
- Kubernetes cluster for performing advanced operations such as managing communication among nodes in a containerized environment, along with balancing the workload.

We performed the SEM analyses utilizing the containerization technology, so that Docker and Kubernetes have been installed on the OpenStack from CNR-IOM cloud computing system [17], IRON machine from CNR-IOM infrastructure as well as C3HPC high performance computing cluster. Kubernetes has been set up on the lowest layer of our software infrastructure on top of it Docker has been installed. The software deployment has been implemented by means of Ansible's configuration deployment and orchestration language [27] that sets all the necessary components, dependencies, and configurations for our software infrastructure on all available nodes in an automatic way.

In the following, we summarize the automation deployment process of Kubernetes cluster, on the CNR-IOM **OpenStack**, by means of Ansible's playbooks (`YAML` files):

- setting up Kubernetes via adding its repository (kubernetes apt-key) and installing its dependencies (kubelet, kubeadm, and kubectl) using the following YAML file;

```
---
 - name: install kubernets dependencies
   hosts: all
   become: yes
   tasks:
   - name: add Kubernetes apt-key
     apt_key:
       url: https://packages.cloud.google.com/apt/doc/apt-key.gpg
       state: present
   - name: add Kubernetes APT repo
     apt_repository:
       repo: deb http://apt.kubernetes.io/ kubernetes-xenial main
       state: present
```

```
        filename: 'kubernetes'
    - name: install kubelet kubeadm with apt
      apt:
        name: "{{ packages }}"
        state: latest
      vars:
        packages:
        - kubelet
        - kubeadm
  - name: install kubectl on the master node
    hosts: master
    become: yes
    tasks:
    - name: install kubectl
      apt:
        name: kubectl
        state: present
        force: yes
    - name: disabling swappiness
      hosts: all
      become: yes
      tasks:
      shell: |
        swapoff -a
      args:
        executable: /bin/bash
...
```

- initializing Kubernetes cluster by means of the following YAML file;

```
---
- name: kubernetes cluster initialization
  hosts: master
  become: yes
  tasks:
    - name: initialize kubernetes cluster on the master pod
      shell: kubeadm init --pod-network-cidr=10.244.0.0/16 --token-ttl=0 >> initialize_kubernetes.txt
      args:
      chdir: $HOME
      creates: initialize_kubernetes.txt
    - name: create .kube directory
      become: yes
      file:
        path: $HOME/.kube
        state: directory
        mode: 0755
    - name: copy admin.conf file to kube config
      copy:
        src: /etc/kubernetes/admin.conf
        dest: $HOME/.kube/config
        remote_src: yes
        owner: root
    - name: set kernel bridge module sysctl bridge-nf-call-iptables to 1
      shell: sysctl net.bridge.bridge-nf-call-iptables=1
      args:
        executable: /bin/bash
    - name: deploy flannel pod network
      shell: kubectl apply -f    https://raw.githubusercontent.com/coreos/flannel/2140
  ac876ef134e0ed5af15c65e414cf26827915/Documentation/kube-flannel.yml >> pod_net.txt
      args:
        chdir: $HOME
        creates: pod_net.txt
...
```

In order to verify the initialization of Kubernetes cluster, we execute the following `kubectl` command-line [28] on the master node:

```
# get status of the master node
(base) root@spark-master:~# kubectl get nodes
NAME           STATUS    ROLES     AGE    VERSION
spark-master   Ready     master    118s   v1.16.3
```

- joining worker nodes to Kubernetes cluster using the YAML file listed below;

```
---
  - name: build up kubernetes cluster
    hosts: master
    become: yes
    gather_facts: false
    tasks:
      - name: fetch join cluster command and register it
        shell: kubeadm token create --print-join-command
        register: join_command
      - name: set join cluster command
        set_fact:
          join_command: "{{ join_command.stdout_lines[0] }}"
  - name: join workers to the cluster
    hosts: compute
    become: yes
    tasks:
      - name: finally, set up worker nodes
        shell: "{{ hostvars['localhost'].join_command }}"
        args:
          executable: /bin/bash
...
```

After executing this Ansible playbook the two worker nodes became members of our Kubernetes cluster, see Listing 3.1.

```
(base) root@spark-master:~# kubectl get nodes
NAME            STATUS   ROLES    AGE   VERSION
spark-master    Ready    master   7d    v1.17.0
spark-slave01   Ready    <none>   7d    v1.17.0
spark-slave02   Ready    <none>   7d    v1.17.0
```

LISTING 3.1: Kubernetes cluster deployed on OpenStack of CNR-IOM computing system.

The cluster shows *Ready* for *STATUS*, and hence we installed pod network which allows communication among nodes within the cluster. The `flannel` virtual networking [29] has been installed for such target.

In the next sections we discuss the main features, architecture, and deployment of the Apache Spark framework.

3.2 Apache Spark

The architecture of Kubernetes cluster discussed in the previous section serves as the lowest layer of the software infrastructure exploited in our SEM analyses. On the top of such layer we deployed the Spark cluster. Apache Spark [23] is an open source, cluster-computing framework widely used in the realm of Big Data analysis. Spark is written in Scala and developed to process large workloads along with datasets, being MapReduce-like engine made for low latency iterative jobs. Spark has achieved big success in data engineering and analytics, since Spark overcomes the limitations of its ancestor frameworks, such as Hadoop [30] and MapReduce [31], due to Spark's speed and efficiency in handling complicated analytics. Moreover, Spark can achieve, in processing big dataset workloads, low latency on a short timescale of sub-second. This backs to the fact that Spark has the ability to utilize data locality and work in-memory not only for computation, but also for objects storage. Therefore, once a given dataset loaded by Spark it turns into an immutable *resilient distributed dataset* (RDD) abstraction. Namely, data are split and collected as partitions across several nodes of distributed infrastructure. Such abstraction makes the data to be processed as a whole, even though they are distributed over several machines. RDDs in Spark are Scala immutable distributed collection of objects cached in RAM, and can be coded using *maps* in creating new RDDs of user-defined functions or *reduce* in collecting the result from distributed nodes. These features allow much faster data processing compared to MapReduce and Hadoop, in which data can be mapped or reduced if only stored on disk. MapReduce is a distributed system aimed to orchestrate operations on large volume workloads executed within a cluster worker nodes, where the master node mange the job through remote calls for reducing and collecting data. In MapReduce, memory is mainly used for the program computation, i.e., not for objects storage.

In Spark, we perform computation through various parallel operations on RDDs cashed in the memory. Operations on RDDs are divided into two main types: (a) *transformations*; (b) *actions*. Transformations are those operations on RDDs, e.g., `map` and `filter` that return a new RDD, while actions are operations that return results to the driver node, where the spark application is launched, such as `collect` and `count`. Computation of transformed RDD is performed *lazily*, i.e., computation will not be performed until the results is called by the driver program through an action.

There are other frameworks designed to tackle with processing large data volume in different ways, such as distributed stream processing computation (Apache Storm [32]), interactive processing (Apache Tez [33]), and graph processing (Neo4j [34]). Apache Spark allows instead a unified processing pipelines to several sources of big data storage, e.g, HDFS, Cassandra, HBase, and S3. Spark enable higher-level languages interpreter, such as Java, Scala, R, and Python for different computation tasks. Additionally, it provides wide set of efficient libraries for different use in big data analyses, suchlike machine learning, SQL, and graph processing.

In the next sections we discuss the Spark architecture besides our methodology in deploying the Spark framework on HPC systems by means of Kubernetes container-orchestration engine.

3.2.1 Spark architecture

A Spark cluster has only one *master* and *workers* that can be expanded over any number of nodes. Spark copes with jobs on its cluster architecture through the manager node by means of Java virtual machine (JVMs) processes. These processes are spanned over two types: *driver processes*, in which data and tasks are distributed to *executor processes*, which are distributed across workers. The master-worker architecture is shown in Fig. 3.1.

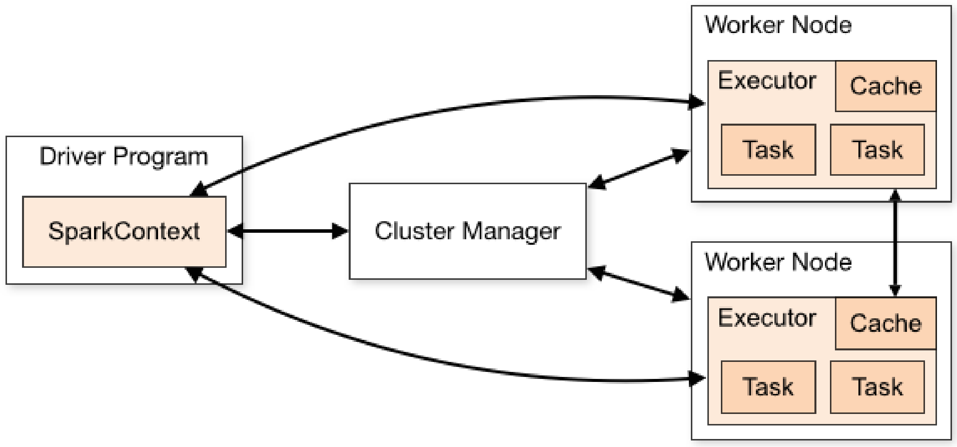


FIGURE 3.1: Spark cluster architecture diagram. Taken from [35].

In the above Spark master-worker architecture the driver program communicate with the cluster manger by means of *SparkContext*. Hence, Spark manger interacts with its workers, which are in charge in computation, in order to send the application code and allocate resources for the job in terms of, e.g., memory, number of executors, number of cores per each worker node. A related point to consider is SparkContext enables several communications with different types of cluster managers, such as Standalone, Apache Mesos, and Hadoop YARN. In our work, we set up the Spark framework in *Standalone* cluster manager.

After setting all necessary Spark configuration parameters, jobs are submitted to the cluster using `spark-submit` command. Consequently, the driver splits jobs into tasks which are assigned to the workers by SparkContext to be executed.

In the following sections, we present and discuss Spark platform automatic deployment mechanism by means of Kubernetes container-orchestration engine.

3.2.2 Spark platform deployment

The mechanism of Spark cluster auto-deployment is performed using YAML files and can be summarized in the items listed below:

- installing Docker on the master and worker nodes using the following YAML file;

```
---
 - hosts: all
   become: yes
   tasks:
   - name: install docker
     shell: |
       # update the apt package index
       apt-get update
       # install packages to allow apt to use a repository over HTTPS
       apt-get -y install apt-transport-https ca-certificates \
       curl gnupg-agent software-properties-common
       # adding Docker's official GPG key
       curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
       apt-key fingerprint 0EBFCD88
       add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
                           $(lsb_release -cs) stable"
       # then re-update
       apt-get update
       # install the latest version of Docker CE
       apt-get -y install docker-ce docker-ce-cli containerd.io
       # list the versions available in your repo
       apt-cache madison docker-ce
       # install version 5:18.09.5~3-0~ubuntu-bionic
       apt-get -y install docker-ce=5:18.09.5~3-0~ubuntu-bionic \
       docker-ce-cli=5:18.09.5~3-0~ubuntu-bionic containerd.io
       # enable and start service
       systemctl start docker
       systemctl enable docker
       # to configure docker and run it as a normal user (non-root) user
       usermod -aG docker ubuntu
     args:
       executable: /bin/bash
...
```

- creating Docker-Spark image, in standalone mode, by means of `Dockerfile` which contains the Spark optimal configurations along with all packages needed for executing the Spark code over SEM images coming from different sources;

```
# base image
FROM ubuntu:18.04

# define spark & hadoop versions
ENV SPARK_VERSION=2.4.4
ENV HADOOP_VERSION=2.7.1

# expose the UI Port 8080
Expose 8080-8081

# set args for spark, hadoop, java paths
ARG SPARK_URL=https://archive.apache.org/dist/spark/spark-${SPARK_VERSION}/spark-${
    SPARK_VERSION}-bin-hadoop2.7.tgz
```

```
ARG HADOOP_URL=https://archive.apache.org/dist/hadoop/common/hadoop-${HADOOP_VERSION}/hadoop-${HADOOP_VERSION}.tar.gz
ARG JAVA_URL=https://download.java.net/java/GA/jdk10/10.0.2/19aef61b38124481863b1413dce1855f/13/openjdk-10.0.2_linux-x64_bin.tar.gz
ARG ENV_PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/bin/scala:/usr/local/spark/bin:/usr/local/spark/sbin:/usr/bin/scala

# install aux. packages
RUN apt-get update -qq && \
    apt-get install -qq -y wget net-tools iputils-ping libgl1-mesa-glx scala python-pip python3-pip vim time

# download and install spark
RUN wget --no-verbose ${SPARK_URL} && tar -xzf spark-${SPARK_VERSION}-bin-hadoop2.7.tgz \
    && mv spark-${SPARK_VERSION}-bin-hadoop2.7 /usr/local/spark \
    && rm spark-${SPARK_VERSION}-bin-hadoop2.7.tgz

# download and install hadoop
RUN wget --no-verbose ${HADOOP_URL} && tar -xzf hadoop-${HADOOP_VERSION}.tar.gz \
    && mv hadoop-${HADOOP_VERSION} /usr/local/hadoop \
    && rm hadoop-${HADOOP_VERSION}.tar.gz

# download and install java
RUN wget --no-verbose ${JAVA_URL} && \
    tar -xvf openjdk-10.0.2_linux-x64_bin.tar.gz && \
    mkdir -p /usr/lib/jdk && mkdir -p /usr/local/oracle-java-10 && \
    cp -r jdk-10.0.2/* /usr/local/oracle-java-10 && \
    mv jdk-10.0.2 /usr/lib/jdk && \
    rm /openjdk-10.0.2_linux-x64_bin.tar.gz

# download and install anaconda
RUN wget https://repo.anaconda.com/archive/Anaconda3-2019.10-Linux-x86_64.sh \
    && bash Anaconda3-2019.10-Linux-x86_64.sh -b -p /root/anaconda3 \
    && rm Anaconda3-2019.10-Linux-x86_64.sh \
    && echo "source activate base" > ~/.bashrc
ENV PATH /root/anaconda3/bin:$PATH

# create conda environment
ADD conda_envs_final.yml /root/
RUN conda env create -f /root/conda_envs_final.yml \
    && echo "source activate sem" > ~/.bashrc
ENV PATH /root/anaconda3/envs/sem/bin:$PATH

# download and install mysql connector
RUN wget https://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-java_8.0.16-1ubuntu18.04_all.deb
RUN dpkg -i mysql-connector-java_8.0.16-1ubuntu18.04_all.deb \
    && rm /mysql-connector-java_8.0.16-1ubuntu18.04_all.deb

# install sparkmeasure package
RUN pip install sparkmeasure

# install tesseract-ocr engine
# and tesseract-ocr files for Greek language
```

```
RUN apt-get update -qq && \
    apt-get install -qq -y tesseract-ocr libtesseract-dev tesseract-ocr-ell

# add spark conf files
ADD spark-env.sh /usr/local/spark/conf/spark-env.sh
ADD spark-defaults.conf /usr/local/spark/conf/spark-defaults.conf

# export paths
RUN echo "export PATH=${ENV_PATH}:/usr/local/spark/bin" >> ~/.bashrc && \
    echo "export PATH=${ENV_PATH}:/usr/local/spark/sbin" >> ~/.bashrc && \
    echo "export PATH=${ENV_PATH}:/usr/bin/scala" >> ~/.bashrc

# add init master & worker files
ADD init-worker.sh init-master.sh /
RUN chmod +x /init-master.sh /init-worker.sh

# set spark, scala, and java envs
ENV PATH $PATH:/usr/local/spark/bin
ENV PATH $PATH:/usr/local/spark/sbin
ENV PATH $PATH:/usr/bin/scala
ENV PATH $PATH:/usr/local/oracle-java-10

CMD ["/bin/bash && source activate sem"]
```

LISTING 3.2: A `Dockerfile` of Spark image. The necessary packages (SEM conda environment) along with Spark optimal configurations contained in `spark-defaults.conf` file are set for running the SEM Spark code within containerized environment.

The Docker-Spark image shown in Listing 3.2 has been built up and tagged on the master node using Docker command-lines:

```
 # build up spark image
 $ docker build -t IMAGE_NAME:TAG .
 # tag spark image
 $ docker tag IMAGE_NAME:TAG MASTER_IP:PORT/IMAGE_NAME:TAG
```

Consequently, we show in the following snippet our Spark and its tagged images alongside the Ubuntu base image:

```
REPOSITORY                          TAG      IMAGE ID       CREATED              SIZE
192.168.10.21:30003/spark-hadoop    2.4.4    6a2068bec6b2   About a minute ago   12.5GB
spark-hadoop                        2.4.4    6a2068bec6b2   About a minute ago   12.5GB
ubuntu                              18.04    549b9b86cb8d   2 weeks ago          64.2MB
```

- deploying a docker local registry that serves as an image repository. To do so, we developed a registry YAML file [36] shown in Listing 3.3;

```
---
kind: PersistentVolumeClaim
apiVersion: v1
```

```
metadata:
  name: dockerregistry-pvclaim
  labels:
    app: dockerregistry
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
       storage: 25Gi
...
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: dockerregistry-deploy
  labels:
    app: dockerregistry
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dockerregistry
  template:
    metadata:
      labels:
        app: dockerregistry
    spec:
      nodeName: spark-master
      containers:
      - image: registry
        name: dockerregistry-pod
        imagePullPolicy: Always
        env:
        - name: registry-rootdir-storage
          value: /root/kubernetes/images
        volumeMounts:
        - name: spark-image-store
          mountPath: /root/kubernetes/images
      volumes:
      - name: spark-image-store
        hostPath:
          path: /root/kubernetes/images
          type: DirectoryOrCreate
...
---
kind: Service
apiVersion: v1
metadata:
  name: dockerregistry-srv
  labels:
    app: dockerregistry
spec:
  selector:
    app: dockerregistry
  type: NodePort
  ports:
    - port: 5000
      targetPort: 5000
      nodePort: 30003
      protocol: TCP
...
```

LISTING 3.3: A Docker registry YAML file used in deploying a Docker local registry on our Kubernetes cluster to host the Spark image. The registry YAML file has three main sections: (a) `PersistentVolumeClaim` section to claim and save storage resources for the Spark image; (b) `Deployment` section that allows us to deploy the Docker local registry on the Kubernetes cluster; (c) `Service` section which enables us to expose the registry service on each Kubernetes node.

In order to check the status of registry deployment, the following `kubectl` command-lines is executed in the terminal:

```
# get pods
(base) root@spark-master:~# kubectl get pods
NAME                                   READY   STATUS    RESTARTS   AGE
dockerregistry-deploy-746cfbccd4-gwqsj  1/1    Running   0          103m
# get deployments
(base) root@spark-master:~# kubectl get deployments
NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
dockerregistry-deploy    1/1     1            1           103m
```

- declaring the local image repository via editing `/etc/docker/daemon.json` file on all Kubernetes nodes with the contents of:

```
'{ "insecure-registries":["REGISTRY\_NAME:TARGET\_PORT", "IP:NODE\_PORT"] }'
```

Hence, all cluster nodes get an access to the Spark image. This step is automatically executed on the master and compute nodes by means of the following YAML file;

```
---
 - hosts: all
   become: yes
   tasks:
    - name: edit docker daemon, reload daemon, and restart docker service
      shell: |
        echo '{ "insecure-registries":["REGISTRY_NAME:TARGET_PORT", "IP:NODE_PORT"] }' >>
      /etc/docker/daemon.json
        systemctl daemon-reload
        systemctl restart docker.service
      args:
        executable: /bin/bash
...
```

As it is shown in the above YAML file, editing Docker `daemon.json` file is followed by reloading the daemon and restarting Docker service, respectively:

```
systemctl daemon-reload
systemctl restart docker.service
```

- pushing our Spark image to the local registry by means of Docker command-line executed in the terminal of master node;

```
$ docker push IP:PORT/IMAGE_NAME:TAG
# with an output:
The push refers to repository [IP:PORT/IMAGE_NAME]
a5f3f773576c: Pushed
bbf5f4d0d5a5: Pushed
64d1ce7c0ac6: Pushed
ed840b5ee2fe: Pushed
99e785337496: Pushed
d41886438118: Pushed
e4f114c705ec: Pushed
8be938d350a0: Pushed
9211bd989187: Pushed
19750959e936: Pushed
06f109d446b4: Pushed
1356e76f7cab: Pushed
8e2016793e6f: Pushed
17abcd11b353: Pushed
965c5708673f: Pushed
3cdd13f6c042: Pushed
918efb8f161b: Pushed
27dd43ea46a8: Pushed
9f3bfcc4a1a8: Pushed
2dc9f76fb25b: Pushed
2.4.4: digest: sha256:1381650c94751bd317eaaf3d86f3b9e5d0404da74a8e2f11add9e567c5b85bf7
    size: 4516
```

After the implementation of above steps, the cluster is ready for master and worker Spark Pods deployment via using `kubectl` Kubernetes command-line over relevant YAML files presented in next sections.

Spark-Master deployment

In order to get Spark-Master deployment with all optimal configurations, we developed a YAML file available at [37]. Our Spark-Master YAML file has two main parts of kind of *Deployment* and *Service* shown in Listing 3.4.

```
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: spark-master
  labels:
    app: spark-master
spec:
  replicas: 1
  selector:
    matchLabels:
      component: spark-master
  template:
    metadata:
      labels:
        component: spark-master
    spec:
      nodeName: spark-master
      volumes:
        - name: nextcloud
          hostPath:
            path: /nextcloud/data
            type: Directory
      containers:
        - name: spark-master
          image: localhost:30003/spark-hadoop:2.4.4
          imagePullPolicy: "IfNotPresent"

          name: spark-master
          ports:
            - containerPort: 7077
            - containerPort: 8080
              protocol: TCP
          volumeMounts:
            - mountPath: "/nextcloud"
              name: nextcloud
          command: [ "/bin/bash" ]
          args: [ "-c", "./init-master.sh; sleep infinity" ]
      hostNetwork: true
      dnsPolicy: Default
...
---
kind: Service
apiVersion: v1
metadata:
  name: spark-master-srv
  labels:
    app: spark-master
spec:
  selector:
    app: spark-master
  ports:
    - name: webui
      port: 8080
      targetPort: 8080
      protocol: TCP
    - name: spark
      port: 7077
      targetPort: 7077
      protocol: TCP
...
```

LISTING 3.4: Spark-Master deployment YAML file with all optimal configurations pertained to `volumes`, `containers`, `ports`, `volumeMounts`. The YAML file has two main parts of Spark *Deployment* and *Service*. The deployment was executed on the master node using `kubectl` command-line over the YAML script.

The Spark-Master YAML file sets indispensable configurations of Spark cluster deployment by means of Kubernetes. Three essential sections are set in the part of Deployment, such as volumes, containers, and mount point (volumeMounts). The section of *volumes* contains the input parameters (hostPath, path) for SEM dataset host path. Under *containers* section, we set the crucial information of Docker-Spark image (name and tag), while *volumeMounts* section hosts the information of mount point of the SEM dataset input. The Spark-Master Service part contains the parameters for configuring Spark ports and network communication protocol - Transmission Control Protocol (TCP). To check the status of Spark-Master deployment, we execute in terminal `kubectl` command-line as shown in the following snippet.

```
(base) root@spark-master:~# kubectl get pods
NAME                                   READY   STATUS    RESTARTS   AGE
dockerregistry-deploy-746cfbccd4-rzmtm 1/1     Running   0          18h
spark-master-587bc5579f-9snw7          1/1     Running   0          18h
```

Our Spark cluster shows two Pods of Docker local registry and Spark-Master. Both Pods are *Ready* with a single replica and register *Running* as a status. The Spark cluster is therefore well prepared to spawn worker nodes.

Spark-Workers deployment

Similar to the Spark-Master deployment, we developed the Spark-Workers YAML file [37] with only one replica shown in Listing 3.2.2.

```
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: spark-worker
spec:
  replicas: 1
  selector:
    matchLabels:
      component: spark-worker
  template:
    metadata:
      labels:
        component: spark-worker
    spec:
      volumes:
        - name: nextcloud
          hostPath:
            path: /nextcloud/data
            type: Directory
      containers:
        - name: spark-worker
          image: localhost:30003/spark-hadoop:2.4.4
          imagePullPolicy: "IfNotPresent"
          name: spark-worker
          ports:
            - containerPort: 7078
            - containerPort: 8081
              protocol: TCP
          volumeMounts:
            - mountPath: "/nextcloud"
              name: nextcloud
          command: [ "/bin/bash" ]
          args: [ "-c", "./init-worker.sh; sleep infinity" ]
      hostNetwork: true
      dnsPolicy: Default
...
```

The Spark-Worker YAML file has one main part, of kind of *Deployment*, which contains the three main sections of volumes, containers, and mount point (volumeMounts). Such Spark-Worker deployment with one replica was performed using `kubectl` command-line over the YAML file. Hence, our cluster has one master and worker nodes, along with the Docker local registry:

```
(base) root@spark-master:~# kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
dockerregistry-deploy-746cfbccd4-rzmtm   1/1     Running   0          18h
spark-master-587bc5579f-9snw7          1/1     Running   0          18h
spark-worker-7764b7c74f-btlpw          1/1     Running   0          18h
```

Scaling the Spark platform over all remaining nodes was the last step in such Kubernetes-Spark auto-deployment process. Such step is executed using `kubectl` command-line over number of available compute nodes *n*:

```
kubectl scale -n default deployment spark-worker --replicas=n
```

After executing all the above procedures, the Spark cluster is ready for performing our SEM analysis. In § 3.2.3, we list all Kubernetes-Spark Pods deployed on OpenStack infrastructure with results obtained.

3.2.3 OpenStack Kubernetes-Spark cluster

The auto-deployment procedures implemented on **OpenStack** of CNR-IOM computing system results in a fully functional Kubernetes-Spark cluster composed of one master and two worker nodes and/or Pods of 4 cores each (see Listing 3.5).

```
(base) root@spark-master:~# kubectl get pods
NAME                                      READY   STATUS    RESTARTS   AGE
dockerregistry-deploy-746cfbccd4-rzmtm    1/1     Running   0          2d20h
spark-master-587bc5579f-9snw7             1/1     Running   0          2d20h
spark-worker-7764b7c74f-btlpw             1/1     Running   0          2d20h
spark-worker-7764b7c74f-h89n7             1/1     Running   0          2d20h
```

LISTING 3.5: OpenStack Kubernetes-Spark cluster. The cluster is running with one master and two worker nodes/Pods via employing 4 cores each. The input SEM images are accessible by all nodes/Pods by means of Ceph filesystem.

The cluster is tested and used in processing a *reference dataset* of 2000 SEM images mounted on every Kubernetes-Spark node/Pod by means of Ceph filesystem (FS) [38]. Figure 3.2 shows OpenStack Kubernetes-Spark cluster architecture consisted of one master and two worker nodes/Pods. The SEM data input is accessible by every node/Pod via employing Ceph FS. The SEM metadata output is committed to MySQL database and shared with a nanoscience community through phpMyAdmin web-interface.

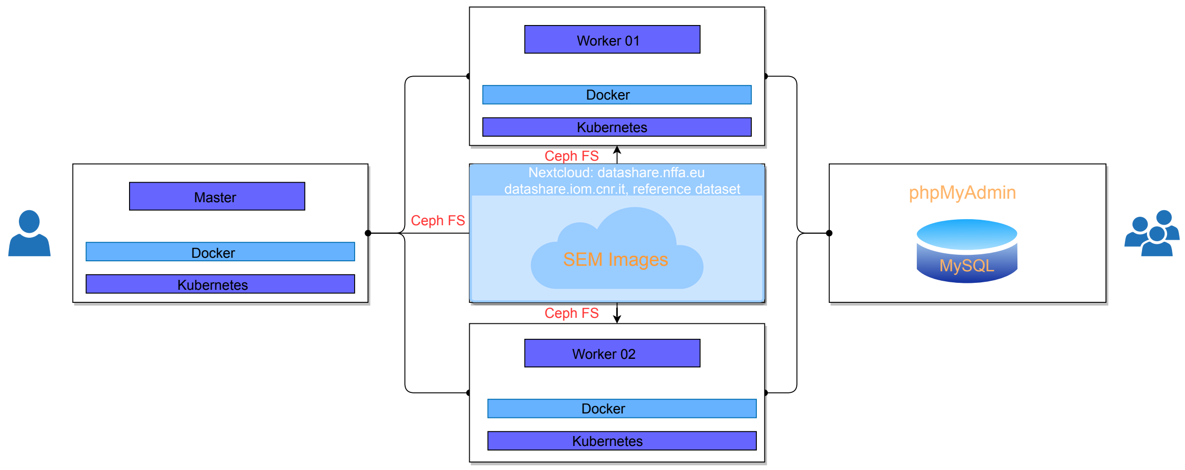


FIGURE 3.2: OpenStack Kubernetes-Spark cluster architecture diagram. The cluster has in total three nodes/Pods (one master and two workers with 4 cores each). The SEM dataset is accessible by all Pods using Ceph FS. The analyses output is shared with the community of nanoscience research and visualized through phpMyAdmin web-interface.

In this work, we intend to process a large number of SEM images coming from different sources within the NFFA EUROPE project via employing two Spark executor approaches [39] known as *Fat* and *Tiny* within containerized environment on different infrastructure and filesystems. In Spark *Fat executors approach*, we assign a single executor instance and memory per worker and/or Pod while in Spark *Tiny executors approach*, a single executor instance and memory are allocated for every single core involved in computation. In both Spark approaches, a driver memory instance has to be allocated based on the available resources and workload. Snippets 3.6 and 3.7 show an example of Spark configuration parameters generated in an automatic way, for the Fat and Tiny executor approaches, once the job is launched:

```
spark.kubernetes.task.cpus 1
spark.kubernetes.driver.memory 5G
spark.kubernetes.executor.memory 5G
spark.kubernetes.executor.instances 2
spark.kubernetes.executor.cores 4
spark.kubernetes.cores.max        8
spark.kubernetes.total.executor.cores 8
```

LISTING 3.6: An example of configuration parameters of the Spark *Fat executors approach* for a distributed job of 8 cores on OpenStack Kubernetes-Spark cluster. Two executor instances are assigned for the two Spark workers/Pods. A memory of 5 G is allocated for the Spark driver (master Pod) and every single Spark worker/Pod. Only 4 cores per Spark executor (worker Pod) are in charge in computation.

The configuration parameters of Spark Tiny executors approach utilized in executing a parallel job of 8 cores are listed in the following Snippet:

```
spark.kubernetes.task.cpus 1
spark.kubernetes.driver.memory 5G
spark.kubernetes.executor.memory 1G
spark.kubernetes.executor.instances 8
spark.kubernetes.executor.cores 1
spark.kubernetes.cores.max        8
spark.kubernetes.total.executor.cores 8
```

LISTING 3.7: An example of configuration parameters of the Spark *Tiny executors approach* for a distributed job of 8 cores on OpenStack Kubernetes-Spark cluster. One executor instance is assigned for every core, i.e., we employ in total 8 executor instances. Similar to Fat approach, 5 G of memory is allocated for Spark driver (master Pod), while 1 G of memory is assigned for every core involved in computation.

As a test run, the OpenStack architecture presented in Fig. 3.2 has been used in processing a reference dataset of 2000 SEM images. The elapsed time and strong scalability, of this test run, as a function of number of cores is shown in Fig. 3.3.

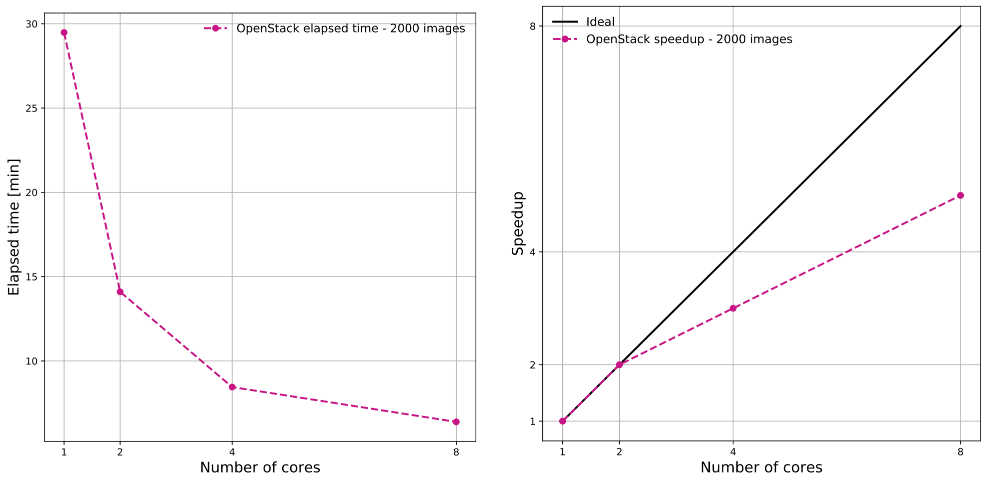


FIGURE 3.3: Elapsed time and strong scalability of a workload of 2000 images processed on OpenStack. Left panel presents total elapsed time as a function of number of cores. Right panel presents strong scalability with maximum speedup 5 times a serial process.

With respect to Fig. 3.3, the total elapsed time (left plot) decreases with increasing number of cores and strong scalability (right plot) reached 5 times a serial process with using 8 cores. Nevertheless, the OpenStack architecture did not reach a plateau speedup feature. Namely, employing more resources with increasing SEM images workload will lead to better scalability.

This test run result motivated us to expand and deploy our Kubernetes-Spark cluster on larger and scalable infrastructure. Therefore, the auto-deployment steps of Kubernetes-Spark platform presented in § 3.1.1 and § 3.2.2 have been implemented on IRON node from CNR-IOM infrastructure and C3HPC cluster. In next sections, we present the architecture of Kubernetes-Spark platform deployed on both IRON node and C3HPC cluster.

3.2.4 IRON machine Kubernetes-Spark cluster

Similar to OpenStack of CNR-IOM cloud computing system, Kubernetes-Spark cluster has been installed on the IRON node, which belongs to CNR-IOM infrastructure. The IRON node has 20 cores in total of a type of `Intel Broadwell`, and then three Kubernetes-Spark Pods shown in Listing 3.8 have been deployed in the form of one master and two worker Pods.

```
[root@iron ~]# kubectl get pods
NAME                                          READY   STATUS    RESTARTS   AGE
csi-cephfsplugin-2knrz                        3/3     Running   0          18d
csi-cephfsplugin-provisioner-7966f9c69c-n6wbn 4/4     Running   0          18d
csi-cephfsplugin-provisioner-7966f9c69c-vbpwx 4/4     Running   0          18d
csi-cephfsplugin-provisioner-7966f9c69c-vszpr 4/4     Running   0          18d
spark-master-5dc6bf4d59-qmcjm                 1/1     Running   0          8d
spark-worker-77c88ddb94-djh8r                 1/1     Running   0          8d
spark-worker-77c88ddb94-wps8t                 1/1     Running   0          8d
```

LISTING 3.8: IRON machine Kubernetes-Spark cluster. The cluster is running and used in processing SEM images coming from different sources within the NFFA-EUROPE project with one master and two worker Pods via employing 20 cores in total. The input SEM images are accessible by the machine and all Pods using Ceph filesystem.

The IRON node Kubernetes-Spark platform architecture is shown in Fig. 3.4.

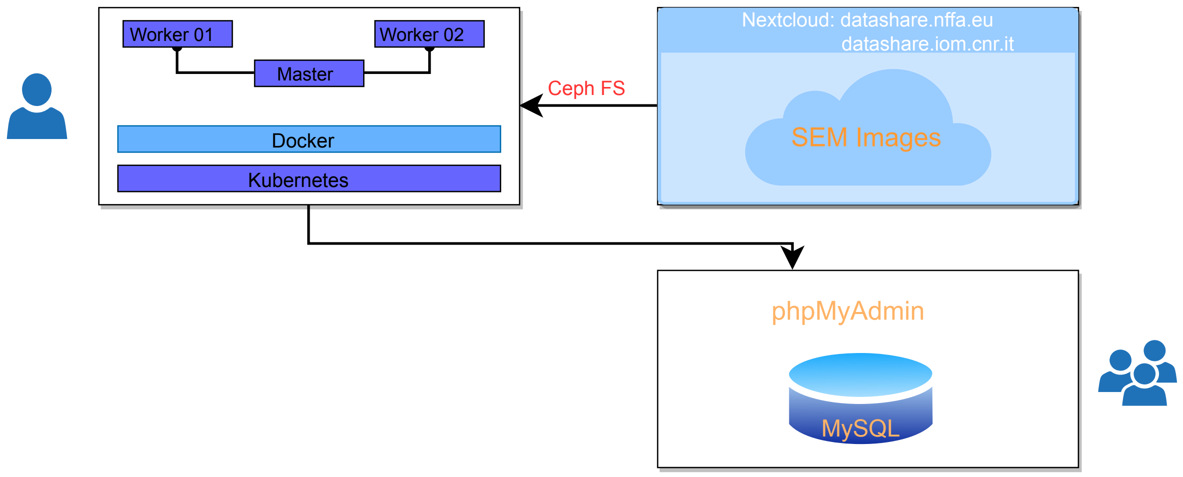


FIGURE 3.4: IRON Kubernetes-Spark cluster architecture diagram. The cluster employs three Pods (one master and two workers) with 20 cores in total. The SEM dataset from different sources is accessible by every Pod via using Ceph FS. The resultant output of the SEM analyses is visualized through phpMyAdmin web-interface tool and shared with wide community of nanoscience research.

The above architecture presented in Fig. 3.4 has been used in processing large number of SEM images coming from different sources within the NFFA-EUROPE project. The SEM input dataset is mounted on the IRON node and every Pod by means of Ceph FS. All Pods are connected with MySQL database which hosts the resultant output of our SEM analyses. The results obtained are presented and discussed in Chapter 4.

3.2.5 C3HPC Kubernetes-Spark cluster

Analogous to CNR-IOM OpenStack, the auto-deployment steps of Kubernetes-Spark platform have been performed on C3HPC cluster with employing 9 nodes in total. One master along with 8 worker pods are forming the Kubernetes-Spark architecture, shown in Listing 3.9 and depicted in Fig. 3.5. On C3HPC, every single node/Pod has 24 cores of a type of `Intel Ivybridge`.

```
[akhalil@login ~]$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
dockerregistry-01-dep-5c77fd7b8-qhrmv   1/1     Running   0          14d
spark-master-c7dc46ccd-krcrp            1/1     Running   0          8d
spark-worker-5757b9cbcc-2rcgm           1/1     Running   0          8d
spark-worker-5757b9cbcc-665b9           1/1     Running   0          8d
spark-worker-5757b9cbcc-7qmrn           1/1     Running   0          8d
spark-worker-5757b9cbcc-clqhl           1/1     Running   0          8d
```

```
spark-worker-5757b9cbcc-kk28t          1/1     Running   0          8d
spark-worker-5757b9cbcc-nkjng          1/1     Running   0          8d
spark-worker-5757b9cbcc-p9ff4          1/1     Running   0          8d
spark-worker-5757b9cbcc-r45sk          1/1     Running   0          8d
```

LISTING 3.9: C3HPC Kubernetes-Spark cluster. The cluster is running and used in processing SEM images coming from different research groups within the NFFA-EUROPE project with one master and 8 worker nodes/Pods via employing 24 cores each. The input SEM images are accessible by all nodes and/or Pods using Lustre filesystem.

The C3HPC Spark-Kubernetes cluster presented in Fig. 3.5 has been used in processing large and different SEM datasets produced from several experiments within NFFA-EUROPE project.

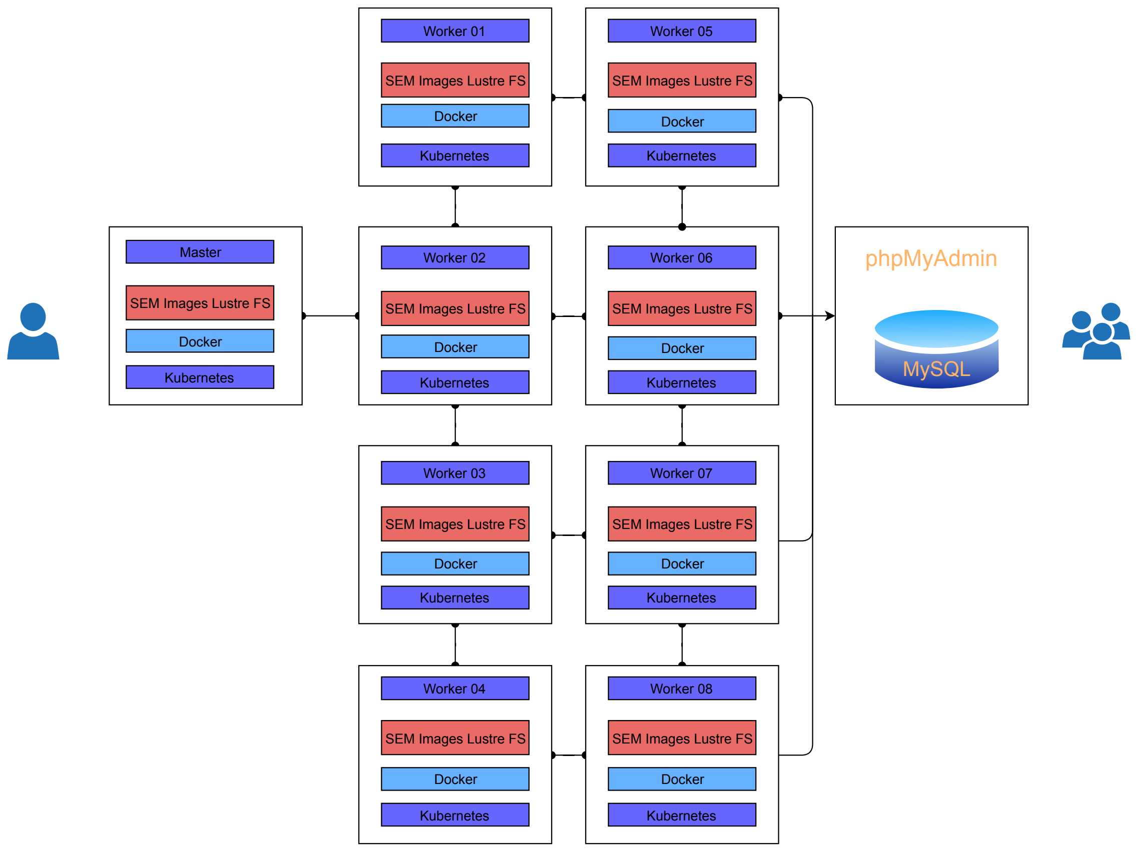


FIGURE 3.5: C3HPC Kubernetes-Spark cluster architecture diagram. The cluster has 9 nodes/Pods in total (one master and 8 workers) with 24 cores each of a type of `Intel Ivybridge`. The SEM dataset from different sources is accessible by every node and/or Pod by means of Lustre FS. All Pods are connected to MySQL, where SEM analyses output is committed. The output is therefore visualized through a web-interface tool (phpMyAdmin) and shared with a wide community of nanoscience research within the NFFA-EUROPE project.

The SEM images input data are hosted on the cluster and accessible by every node and/or Kubernetes-Spark Pod by means of Lustre filesystem [40]. The SEM analyses output is committed to MySQL database, and hence visualized through phpMyAdmin web-interface tool to be shared with the nanoscience community within the NFFA-EUROPE project. The results obtained are presented and discussed in Chapter 4.

Chapter 4

Results

The infrastructure described in Chapter 3 is used in processing large number of scientific images coming from SEM instruments within the NFFA-EUROPE project with a goal of obtaining all the metadata. Therefore, the SEM analyses have been performed on a distributed infrastructures of a single and multiple nodes by means of Apache Spark big data cluster-computing framework. The Spark has been used in different SEM analyses presented in MHPC work in 2016 and 2018 by Rossella Aversa [10] and Luca Ciuffreda [11], respectively. The results presented there do not show a strong tendency of Spark to scale under different conditions. Aversa found out that the filesystem choice is an important input in processing SEM images. In this context, Lustre was scaling much better than the local filesystem.

As we have discussed in Chapter 2, the SEM analyses, on our distributed infrastructure, tackle with four main parts of:

1. scientific metadata saved in the image;
2. image pixel size by means of OpenCV library and OCR engine;
3. image classification with calculating top predictions of the image over ten labels and
4. managing all the analyses output into MySQL database.

This Chapter is aimed to report the results obtained from detailed benchmark study for the SEM Spark code on different architectures of a single and multiple nodes: the IRON node from CNR-IOM infrastructure alongside C3HPC of the Carnia Industrial Park [6] high performance computing cluster provided by the eXact-lab srl [7]. The IRON node has 20 cores in total of `Intel Broadwell` type, while a single node of C3HPC has 24 cores of a type `Intel Ivybridge`. Two different SEM workloads are used in this benchmark analysis: (a) small dataset of 16000 SEM images; (b) large dataset of 118000 SEM images. Both datasets are spanned over a nested structure and accessible by means of Ceph and Lustre filesystems on the IRON node and C3HPC cluster, respectively.

As already mentioned in Chapter 3, we have deployed the Spark platform on top of Kubernetes cluster within a containerized environment on the IRON node and C3HPC cluster. This distributed architecture is tested and in a ready status to perform our massive SEM analyses using two different Spark executor approaches [39] called *Fat* and *Tiny*. It is therefore crucial to understand the Spark application performance on different filesystems and identify the code's bottleneck. However, measuring the Spark performance is not an easy task, since lot of components and configuration parameters are involved during the code execution process.

4.1 Measuring Spark performance

As we discussed in Chapter 2, Spark is a cluster computing engine and at a high level parallelism Spark creates RDDs from the input data (SEM images). The application then starts to do lazy transformation among processes/nodes and finally carries out actions of reducing and collecting data to the manager node. Such Spark operations of transformation and action, along with the several instances that have to be set for getting an Spark application running can give us an idea about the bottlenecks of Spark code which can be challenging to resolve.

We are interested in measuring the Spark performance, on different architectures and filesystems, with the two Spark approaches of Fat and Tiny executors by means of `SparkMeasure` [41, 42] library, which serves as an efficient tool for monitoring [43] Spark performance workloads. The library returns necessary information for workload benchmark analysis known as *Spark Executor Task Metric* and/or *aggregated Spark performance metrics* shown in Listing 4.1.

```
Scheduling mode = FIFO
Spark Context default degree of parallelism = 20
Aggregated Spark stage metrics:
numStages => 9
sum(numTasks) => 104
elapsedTime => 4282053 (1.2 h)
sum(stageDuration) => 4279030 (1.2 h)
sum(executorRunTime) => 58748591 (16.3 h)
sum(executorCpuTime) => 18815 (19 s)
sum(executorDeserializeTime) => 12224 (12 s)
sum(executorDeserializeCpuTime) => 1527 (2 s)
sum(resultSerializationTime) => 38 (38 ms)
sum(jvmGCTime) => 4700 (5 s)
sum(shuffleFetchWaitTime) => 0 (0 ms)
sum(shuffleWriteTime) => 0 (0 ms)
max(resultSize) => 23131416 (22.0 MB)
sum(numUpdatedBlockStatuses) => 92
sum(diskBytesSpilled) => 0 (0 Bytes)
sum(memoryBytesSpilled) => 0 (0 Bytes)
max(peakExecutionMemory) => 0
sum(recordsRead) => 0
sum(bytesRead) => 0 (0 Bytes)
sum(recordsWritten) => 0
sum(bytesWritten) => 0 (0 Bytes)
sum(shuffleTotalBytesRead) => 0 (0 Bytes)
sum(shuffleTotalBlocksFetched) => 0
sum(shuffleLocalBlocksFetched) => 0
sum(shuffleRemoteBlocksFetched) => 0
sum(shuffleBytesWritten) => 0 (0 Bytes)
sum(shuffleRecordsWritten) => 0
```

LISTING 4.1: An example of aggregated Spark stage metrics of a SEM job executed on the IRON node, from the CNR-IOM infrastructure, via using 20 cores.

Based on the above output parameters returned by the Spark stage metrics, we present benchmark analysis for two different SEM workloads as a function of number of cores and nodes/Pods. The measurements have been performed over the small and large SEM datasets on the IRON node and C3HPC cluster. On the IRON node, we processed the small SEM input of 16000 images from datashare.iom.cnr.it, where the data are stored in the cloud platform provided by NFFA-EUROPE Datashare service, and hence all input images are accessible by every container of the IRON node (one Kubernetes master along with two worker Pods) utilizing Ceph filesystem. The data storage is different on the C3HPC cluster, where the large and small SEM datasets are stored on the Lustre parallel filesystem.

In this benchmark analysis we collected the aggregated output of Spark Task Metric class of the following parameters:

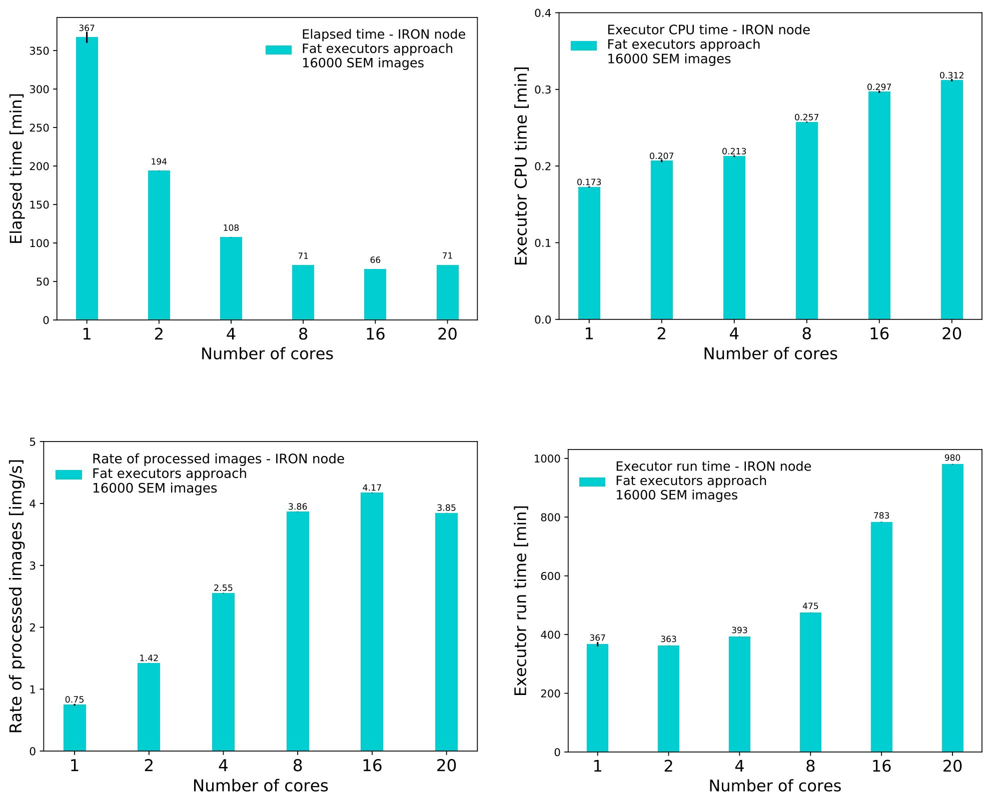
- *elapsedTime*: total execution time of the spark application;
- *executorCpuTime*: total CPU time spent by the Spark executor in executing the processes including fetching-shuffle data time;

- *executorRunTime*: total elapsed time spent by the executor in running the processes including fetching-shuffle data time;
- *executorDeserializeTime*: total time taken by the executor in deserializing the task;
- *jvmGCTime*: total time taken by the Java Virtual Machine (JVM) in garbage collection resulted from memory allocation during the task execution.

In the next sections we discuss the results of both Fat and Tiny executor approaches on the IRON node (Ceph filesystem) and C3HPC cluster (Lustre filesystem).

4.1.1 Fat executors approach

On the IRON node: We show in Fig. 4.1 the performance results for a workload of 16000 using Fat executors approach, in which the number of executors is fixed to one executor per Pod with changing number of cores.



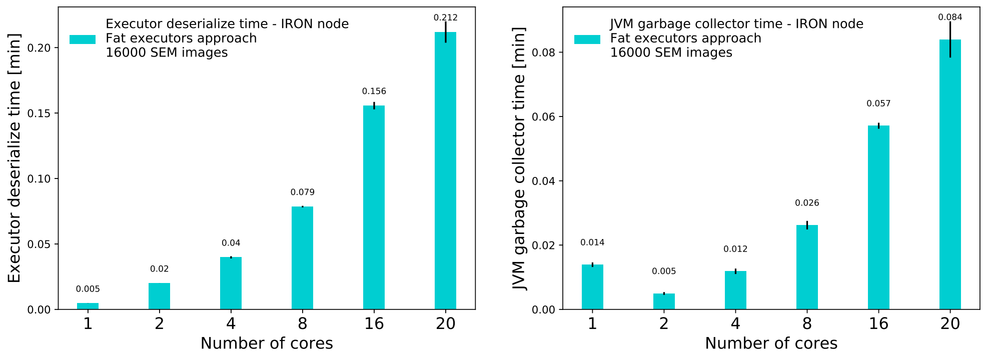
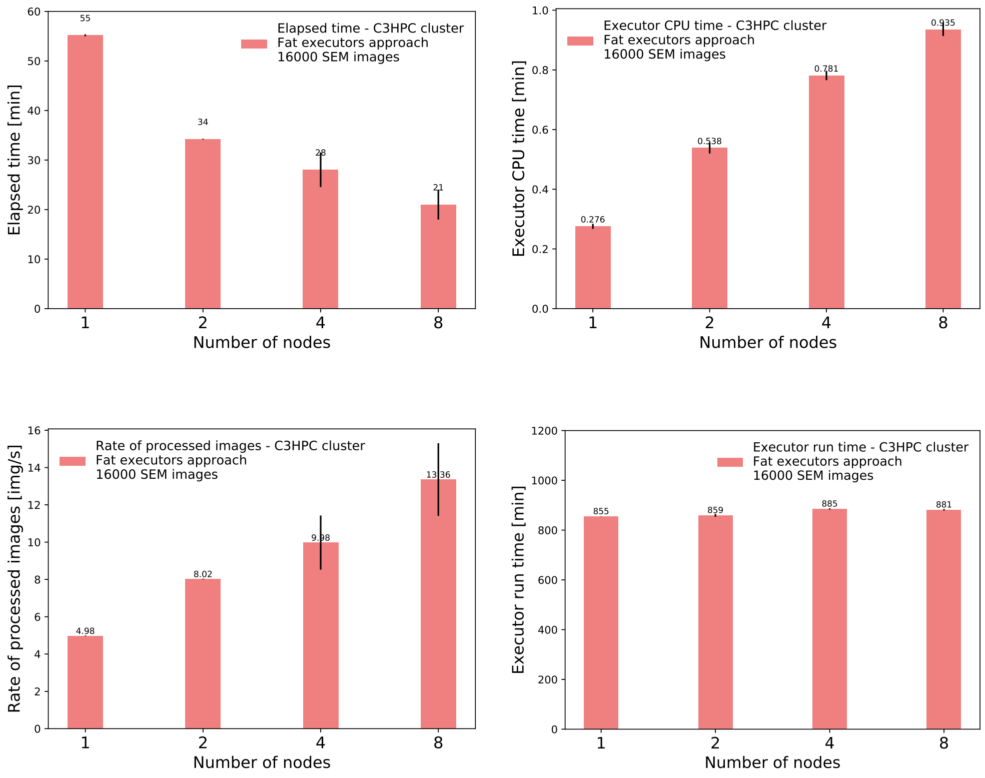


FIGURE 4.1: Spark Measure Metric of *Fat executors approach* for a workload of 16000 images processed on IRON node. The measurements of this Spark stage metric are averaged over two iterations.

We can see clearly significant increase in the number of images processed at higher level parallelism $N_{cores} = (8, 16, 20)$ compared to the low one, with elapsed time around 70 minutes for a workload of 16000 SEM images. However, there is a drop in performance mainly correlated to the time taken by two Task Metric parameters; (a) executorDeserializeTime, (b) jvmGCTime at $N_{cores} = (16, 20)$.

On the C3HPC cluster: The same workload of 16000 SEM images stored on the Lustre parallel FS has been analyzed, and hence the Spark Measure Metric is shown in Fig. 4.2.



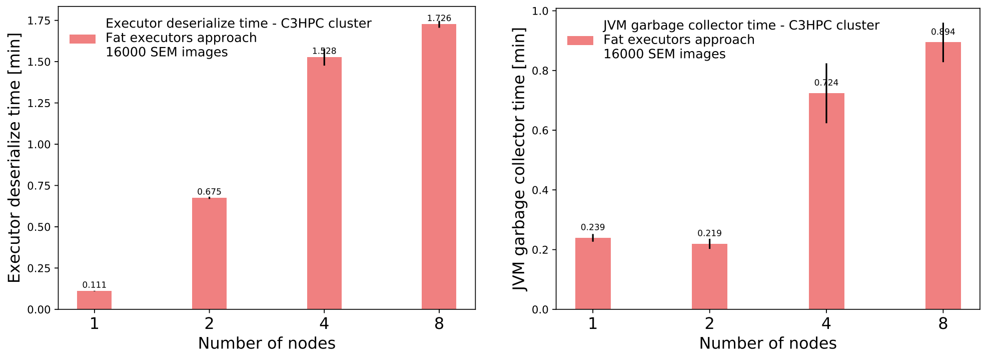


FIGURE 4.2: Spark Measure Metric of *Fat executors approach* for a workload of 16000 images processed on C3HPC cluster. The measurements of this Spark stage metric are averaged over four iterations.

Despite of the difference in FS on IRON (Ceph) and C3HPC (Lustre), the above results from Spark stage metrics, on the C3HPC, are in agreement with the ones obtained on the IRON node for same workload. Compared to a single node process, an increase in the number of images processed per second is observed with utilizing the fourth and eighth node. Similar to the IRON node, the performance obtained with including the fourth and eighth nodes in computation is mainly affected by the deserialization and JVM garbage collector times. Namely, time spent by deserialization processes, in which input images are converted into an object instances, along with JVMs garbage collectors for data stored in the memory is increasing with processing large number of images and nodes.

In this regard, Ciuffreda in his study [11] found out that the time spent by JVMs garbage collection can be insignificant parameter in Spark performance with decreasing the workload, whereas no clear correlation was found between JVMs garbage collection time and the system performance with a smaller workload of 12000 SEM images.

Large SEM dataset: We performed using the approach of Fat executors the analyses of the large SEM dataset of 118000 images, corresponding to 155 GB on the C3HPC cluster. This dataset is stored in C3HPC cluster under Lustre parallel FS. We allocated in this Fat executors approach analysis one driver node with a `driver.memory` of 16 GB and 8 nodes with `executor.memory` of 16 GB per (executor instance/Pod). The Spark Measure Metric for such large workload is presented in Fig. 4.3.

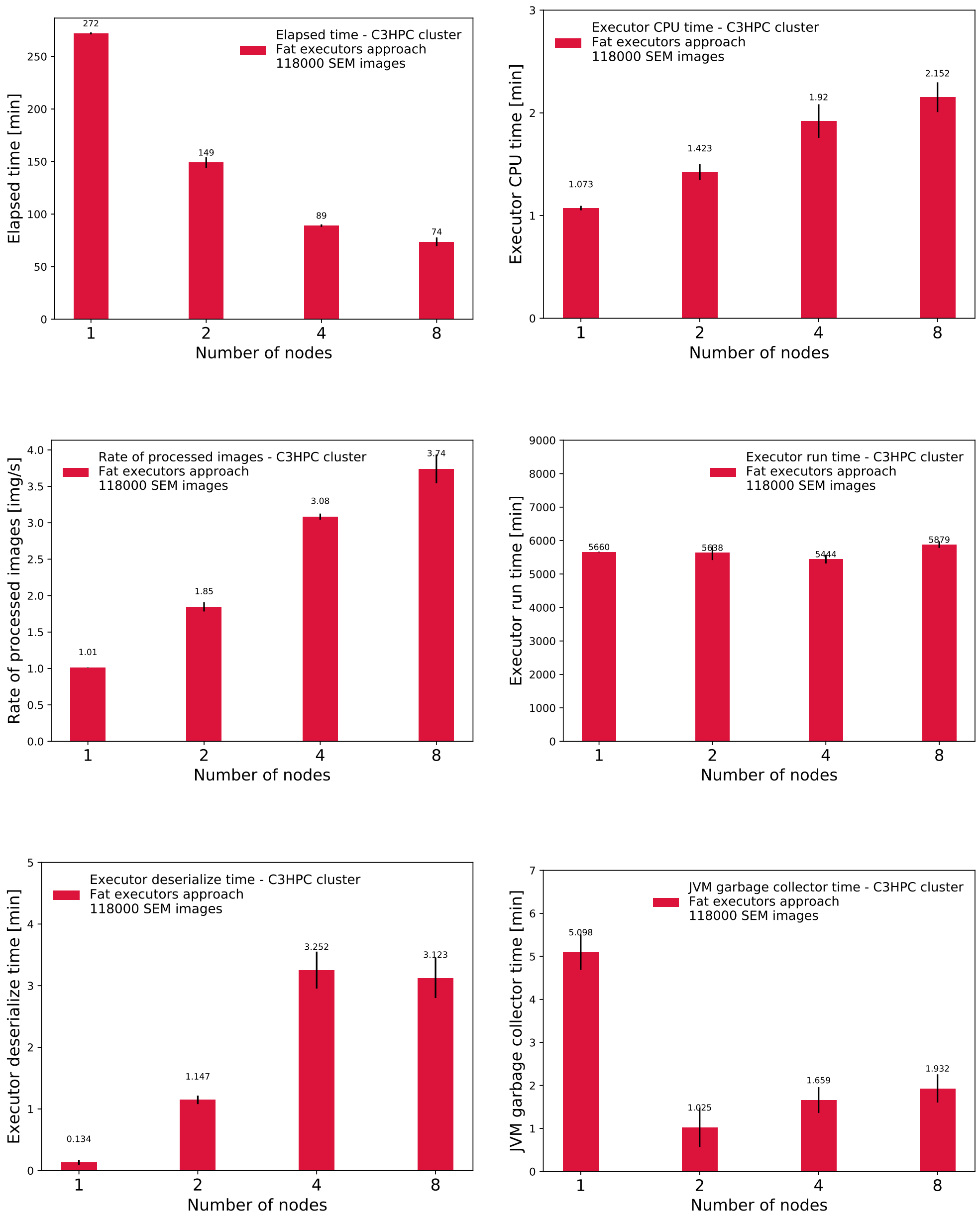


FIGURE 4.3: Spark Measure Metric of *Fat executors approach* for a workload of 118000 images processed on C3HPC cluster. The measurements of this Spark stage metric are averaged over four iterations.

The IRON node is excluded from such large dataset processing, since the total elapsed time summed over all processes would take so long (in total ∼10 days).

4.1.2 Tiny executors approach

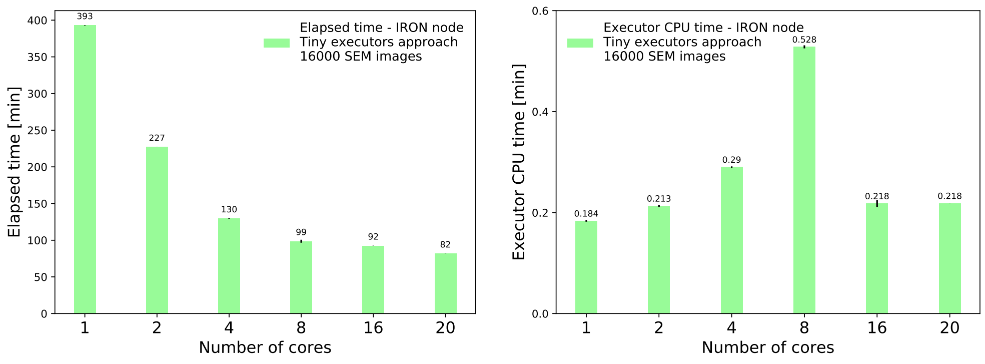
Contrary to the Fat executors approach presented in § 4.1.1, the number of executors is changing for each run per node/Pod. In this Spark Tiny approach, one driver node is allocated with `driver.memory` of 16 GB and one executor instance is assigned per core with `executor.memory` of 1 GB. In the following, we present and discuss the results obtained using the Tiny executors approach on both IRON node (small SEM dataset) and C3HPC cluster (small and large SEM datasets).

On the IRON node: We executed the SEM Spark code with assigning one executor for each core involved in the computation. Namely, for a single process only one executor is employed for the single core, while 20 executors, as an example, are employed for a process of 20 cores. The resources allocated using this approach on the IRON node are shown in Tab. 4.1.

Number of cores	1	2	4	8	16	20
Number of executors	1	2	4	8	16	20
Total executor memory [G]	1	2	4	8	16	20

TABLE 4.1: Resources of Tiny executors approach used on IRON node. In this executors approach we assigned a single executor with 1 G for every single core and the `driver.memory` equals 16 G.

The results of Spark Measure Metric obtained using this executors approach on the IRON node is shown in Figs. 4.4.



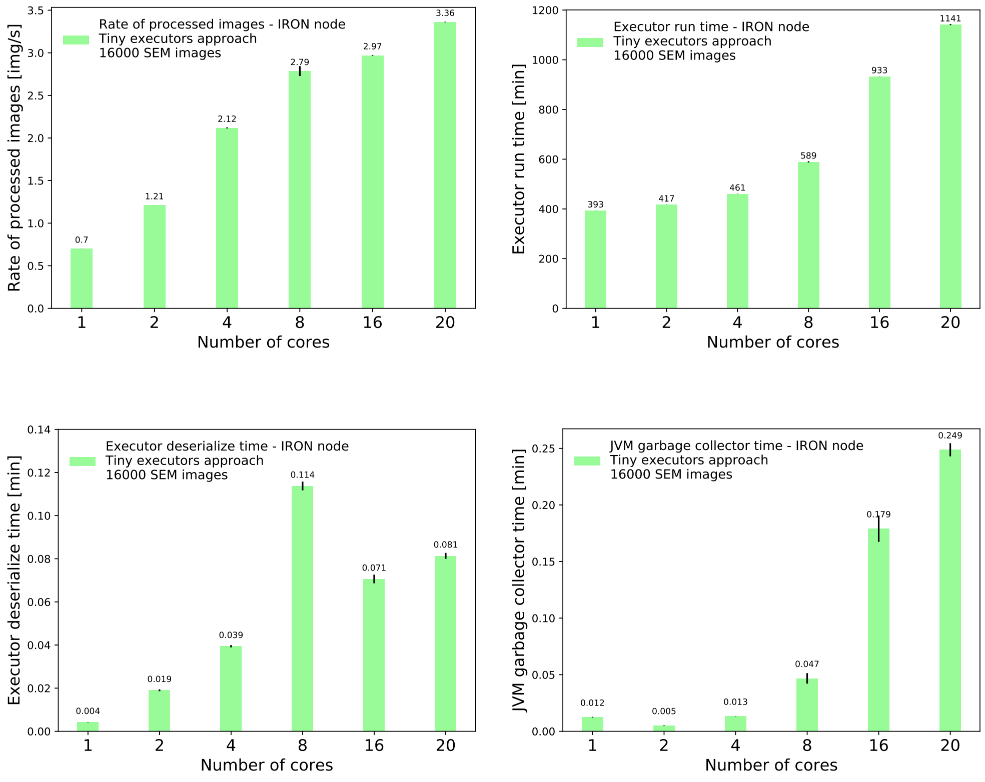


FIGURE 4.4: Spark Measure Metric of *Tiny executors approach* for a workload of 16000 images processed on IRON node. The measurements of this Spark stage metric are averaged over two iterations.

In this Tiny executors approach executed on a single node, the total elapsed time declines, while the rate of processed images increases as a function of number of cores/executors. This result is in agreement with the one of Fat executors approach, and then we can conclude that both Spark executor approaches show a good performance on the scale of a single node.

On the C3HPC cluster: The SEM Spark code was executed over 8 nodes, i.e., in total 192 cores are in charge with computation. Hence, a number of 24 executors per node/Pod have been assigned. Table 4.2 summarizes the resources allocated on the C3HPC cluster for the Spark Tiny executor approach.

Number of nodes	1	2	4	8
Number of executors	24	48	96	192
Total executor memory [G]	24	48	96	192

TABLE 4.2: Resources of Tiny executors approach used on C3HPC cluster. In this approach we employ a single executor instance with `executor.memory` of 1 G for every single core, i.e., in total 24 G of memory has been allocated per node/Pod and the `driver.memory` equals 16 G.

The resultant Spark Measure Metric is presented in Fig. 4.5.

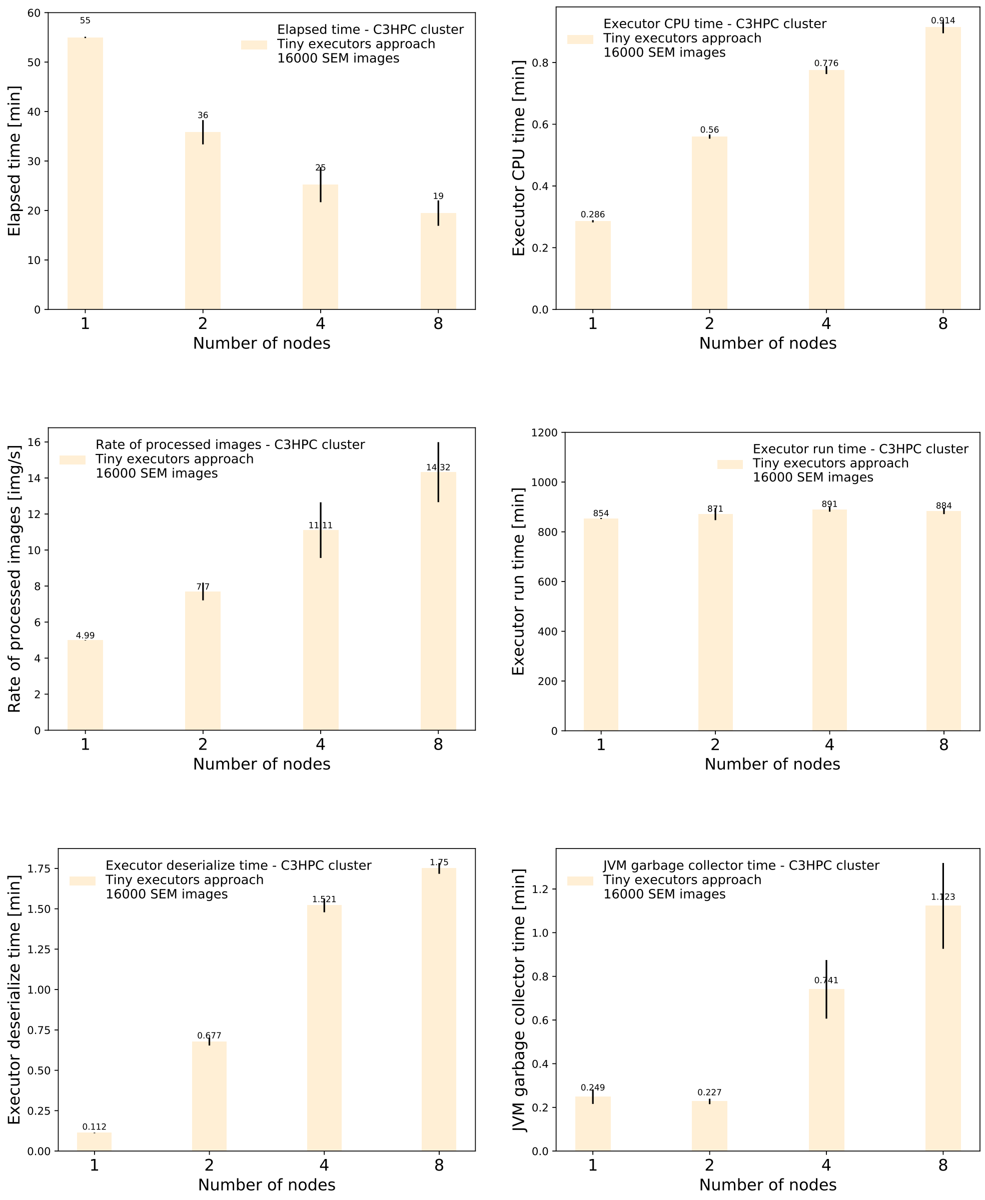


FIGURE 4.5: Spark Measure Metric of *Tiny executors approach* for a workload of 16000 images processed on C3HPC cluster. The measurements of this Spark stage metric are averaged over four iterations.

Large SEM dataset: In order to complete our detailed benchmark study, on the C3HPC cluster, we implemented the Spark Tiny executors approach over the large SEM dataset of a volume of 118000 images located under Lustre FS. Here, we allocated same resources (driver.memory, executor.memory, executor.instances/cores) as the small SEM dataset. The Spark Measure Metric for such large workload is presented in Fig. 4.6.

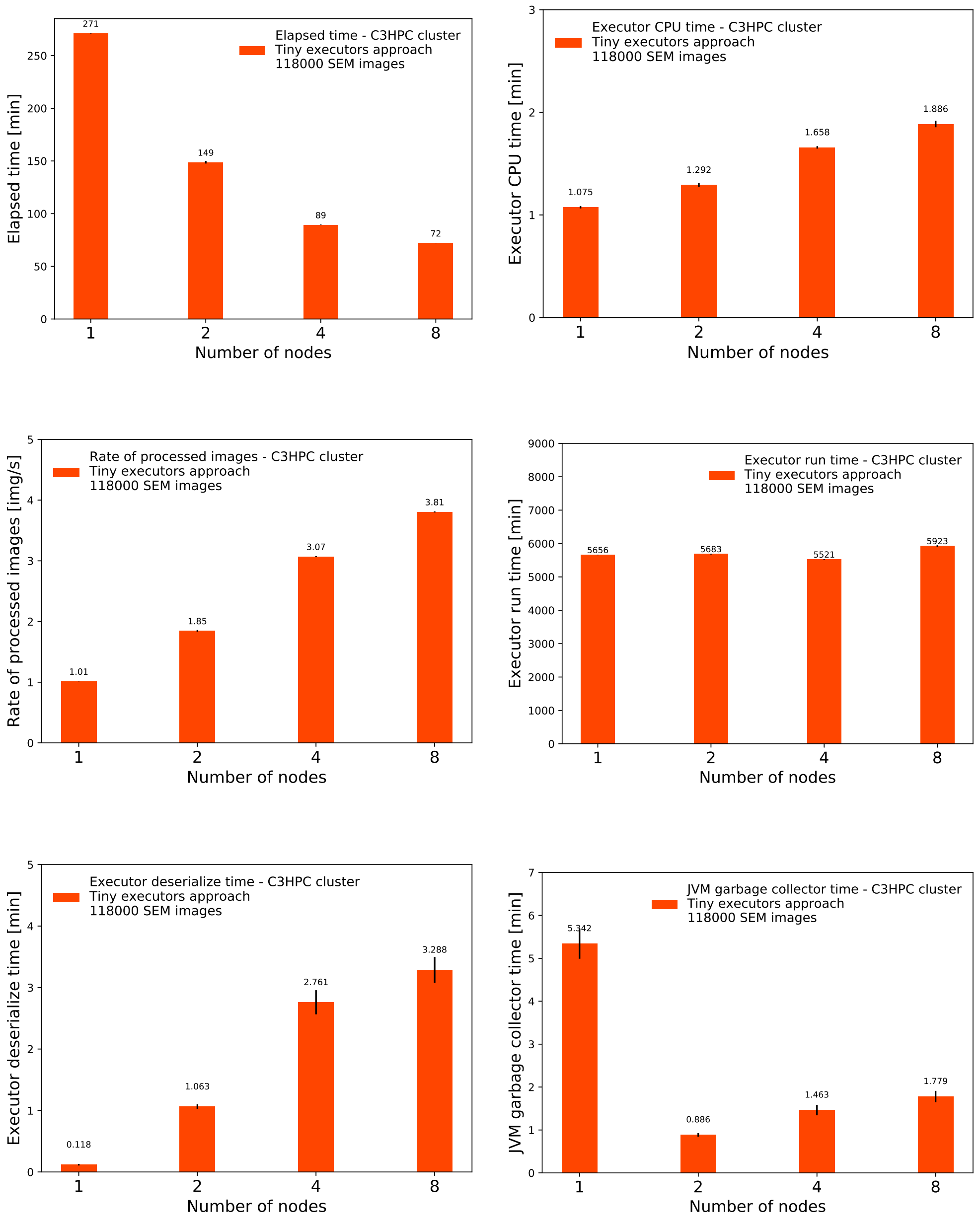


FIGURE 4.6: Spark Measure Metric of *Tiny executors approach* for a workload of 118000 images processed on C3HPC cluster. The measurements of this Spark stage metric are averaged over four iterations.

4.2 Strong scalability analysis

In this section, we investigate the scalability of the infrastructure (single and multiple nodes) after setting up the optimal configuration parameters of the Spark cluster deployed within containerized environment on the IRON node and C3HPC cluster. We are also interested here in measuring the effect of Ceph and Lustre filesystems, by which the input SEM dataset is accessible on each Pod.

As already mentioned, there are two Spark executor approaches involved in this benchmark study on both IRON node and C3HPC cluster. On IRON machine we have in total three Pods; one master and two workers. On the C3HPC we have large number of nodes/Pods (one master node/Pod and 8 worker nodes/Pods) over which the Spark cluster is deployed by means of Kubernetes container-orchestration engine.

IRON node speedup is calculated as a relative gain in total elapsed time as a function of number of cores. As a consequence of the above results from Spark Measure Metric, we present in Fig. 4.7 total elapsed time and corresponding scalability of the IRON node.

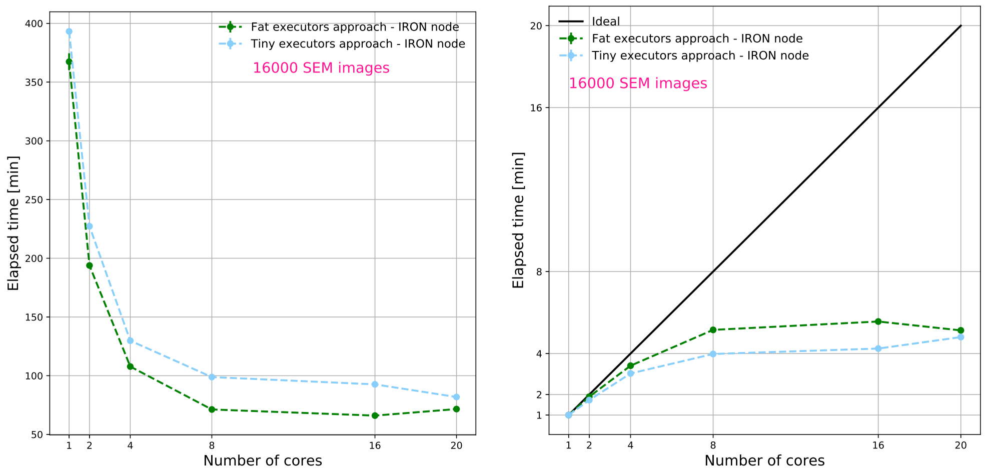


FIGURE 4.7: Elapsed time and strong scalability of IRON node with *Fat* and *Tiny* executor approaches. Left panel presents total elapsed time as a function of number of cores for both Fat and Tiny approaches. Right panel presents strong scalability of the IRON node with speedup gain of 5 for Fat executors approach, while it is around 4 for the Tiny one. The measurements of elapsed time and speedup are averaged over two iterations for a workload of 16000 SEM images.

The IRON machine achieved speedup gain at 5 with Spark Fat executors approach, while speedup around 4 is obtained using the Tiny one. A workload size of 16000 SEM images is used in this analysis for two iterations. In the following, we present and discuss the speedup analysis with both Spark executor approaches on multiple nodes – C3HPC cluster.

C3HPC cluster total elapsed time and speedup are presented in Fig. 4.8 as a function of number of nodes with Fat and Tiny executors approach for a workload of 16000 images.

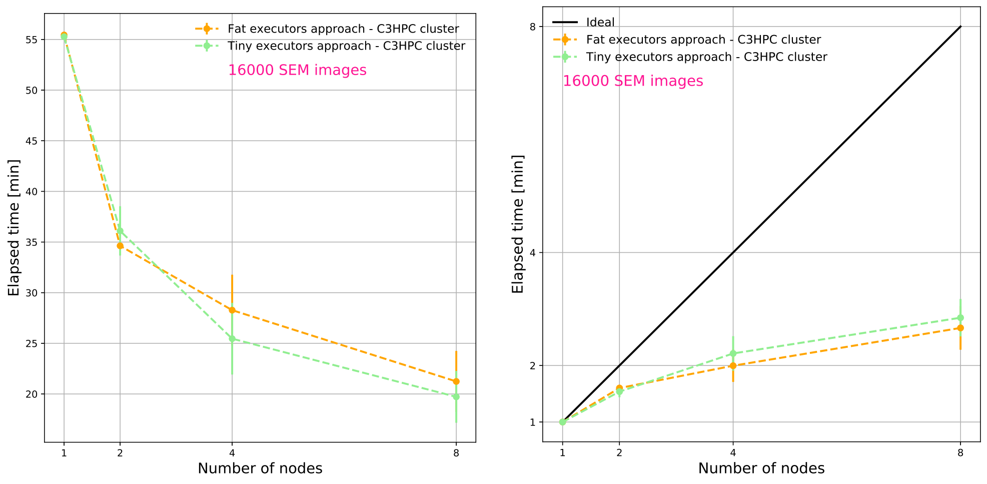


FIGURE 4.8: Elapsed time and strong scalability of C3HPC cluster with *Fat* and *Tiny* executor approaches. Left panel shows total elapsed time as a function of number of nodes. Right panel shows strong scalability of C3HPC cluster with both Spark executor approaches. Maximum attainable speedup gain approaches 3, with both Fat and tiny executor approaches. The measurements of elapsed time and speedup are averaged over four iterations for a workload of 16000 SEM images.

The large SEM dataset total elapsed time and speedup on C3HPC cluster is introduced in Fig. 4.9 in terms of the two Spark executor approaches used in the above analysis.

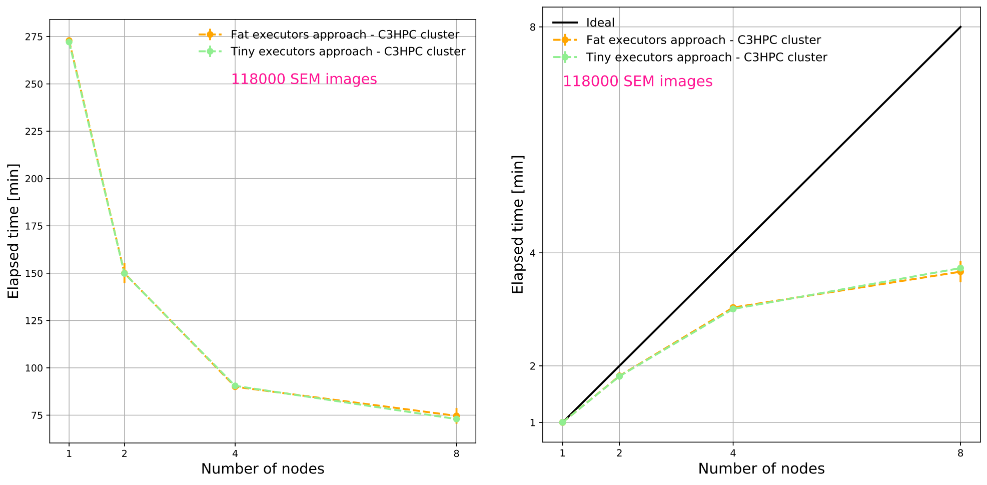


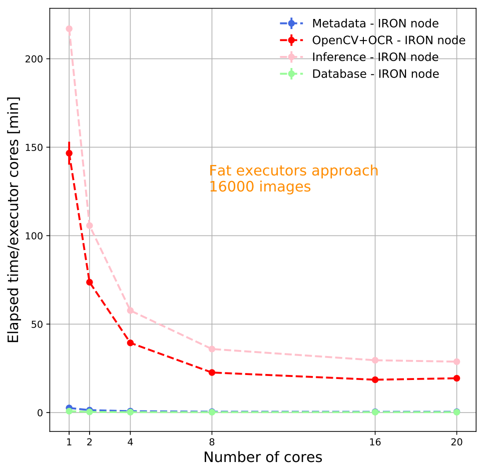
FIGURE 4.9: Elapsed time and strong scalability of C3HPC cluster with *Fat* and *Tiny* executor approaches. Left panel shows total elapsed time as a function of number of nodes. Right panel shows strong scalability of C3HPC cluster with both Spark executor approaches. Both Spark executor approaches show same performance with speedup gain below 4. The measurements of elapsed time and speedup are averaged over four iterations for a workload of 118000 SEM images.

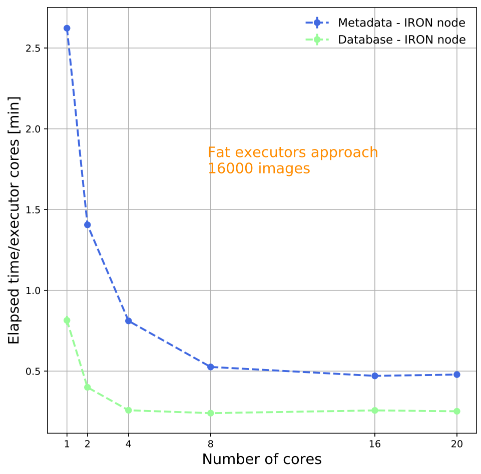
Due to the above speed up analyses implemented on the IRON node (Ceph FS) and C3HPC cluster (Lustre FS) over different workload volumes, we can conclude that Spark shows same performances with employing *Fat* and *Tiny* executor approaches, in particular, with increasing the workload (large SEM dataset). For better understanding the code bottlenecks, we profile in § 4.2.1 the four main parts of the SEM Spark code on both architectures of IRON and C3HPC cluster.

4.2.1 Profiling the code's four parts

We present in Figs. 4.10, 4.11, 4.12, 4.13, 4.14, and 4.15 the elapsed time taken by the four parts (Metadata, OpenCV+OCR, Inference, Database) of the SEM Spark code and corresponding speedup as a function of number of cores (IRON node) and/or number of nodes/Pods (C3HPC cluster). The two Spark executor approaches (Fat and tiny) are utilized in this profiling study, which is essential in specifying the part that negatively affects the code's performance.

On the IRON node: We processed only the SEM dataset of a size of 16000. It is obvious that the Database part drops the performance with low speedup of 3, while the Metadata along with OpenCV+OCR and Inference achieved speed up around 5 and 7, respectively. Figures 4.10 and 4.11 shows IRON node profiling study.





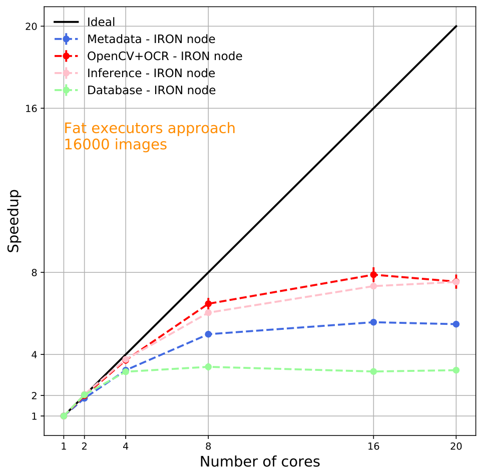
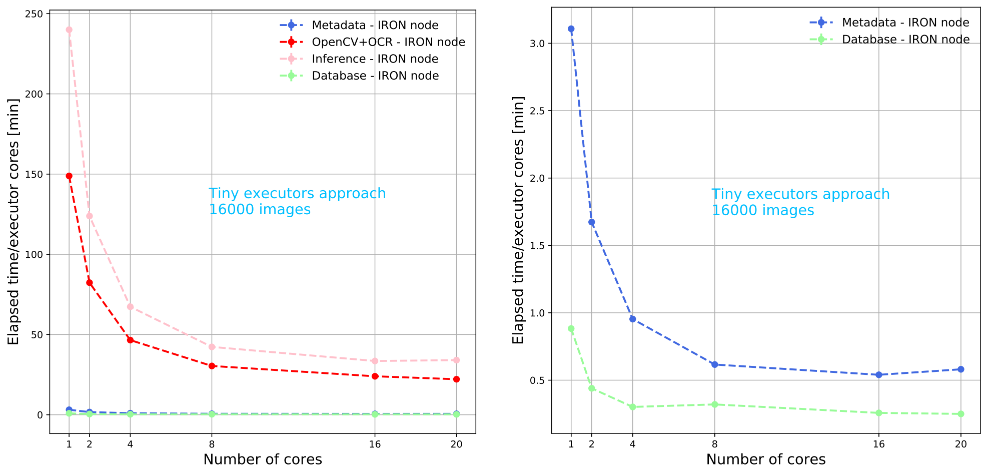


FIGURE 4.10: Elapsed time per executor core and strong scalability of IRON node for the code's four parts with Spark Fat executors approach. *Top panel*: Elapsed time of the code's four parts (left plot) on IRON node as a function of number of cores. Additionally, a zoom of the two parts of Metadata and Database (right plot) is presented. The two parts of Database and Metadata are faster than the OpenCV+OCR and Inference ones. *Bottom panel*: Strong scalability of IRON node with maximum speedup gain around 7 for OpenCV+OCR and Inference parts, while it is 5 for the part of Metadata. The last part of Database gains speedup below 4. The measurements of elapsed time and speedup are averaged over two iterations for a workload of 16000 SEM images.

Similar performance is obtained using the Spark Tiny executors approach over same workload, see Fig. 4.11.



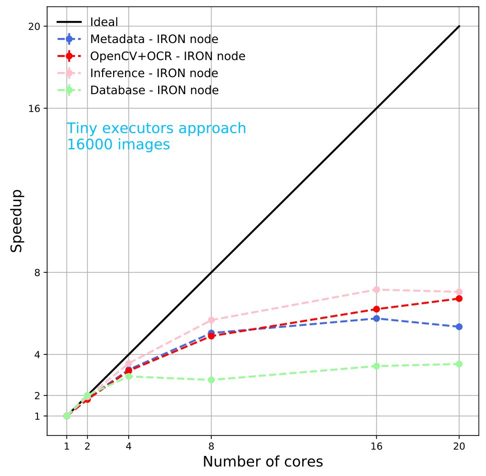
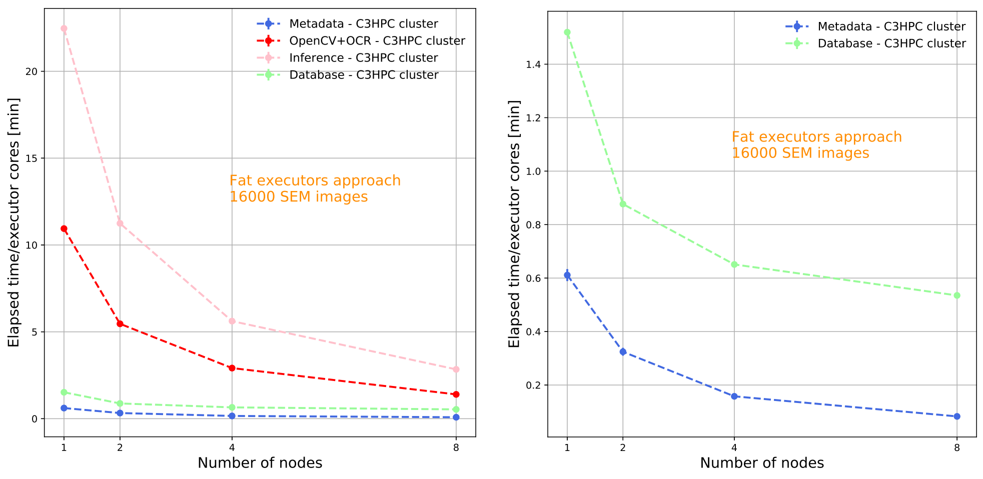


FIGURE 4.11: Elapsed time per executor core and strong scalability of IRON node for the code's four parts with Spark Tiny executors approach. *Top panel*: Elapsed time of the code's four parts (left plot) on IRON node as a function of number of cores. Additionally, a zoom of the two parts of Metadata and Database (right plot) is presented. The two parts of Database and Metadata are faster than the OpenCV+OCR and Inference ones. *Bottom panel*: Similar to Spark Fat executor approach, strong scalability of the IRON node shows speedup gain approaches 7 for OpenCV+OCR and Inference parts, while it is around 5 for the part of Metadata. The last part of Database gains speedup below 4. The measurements of elapsed time and speedup are averaged over two iterations for a workload of 16000 SEM images.

On the C3HPC cluster: Both SEM datasets have been analyzed. The results of elapsed time per executor core and corresponding speedup for the four parts of SEM Spark code are shown in Figs. 4.12, 4.13 (workload of 16000 SEM images) and Figs.4.14, 4.15 (workload of 118000 SEM images) via employing Spark Fat and Tiny executor approaches.



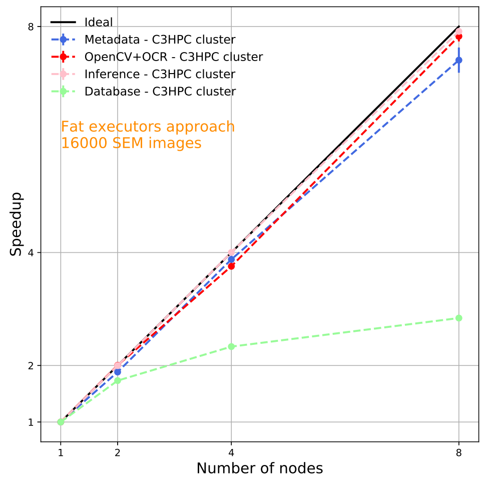
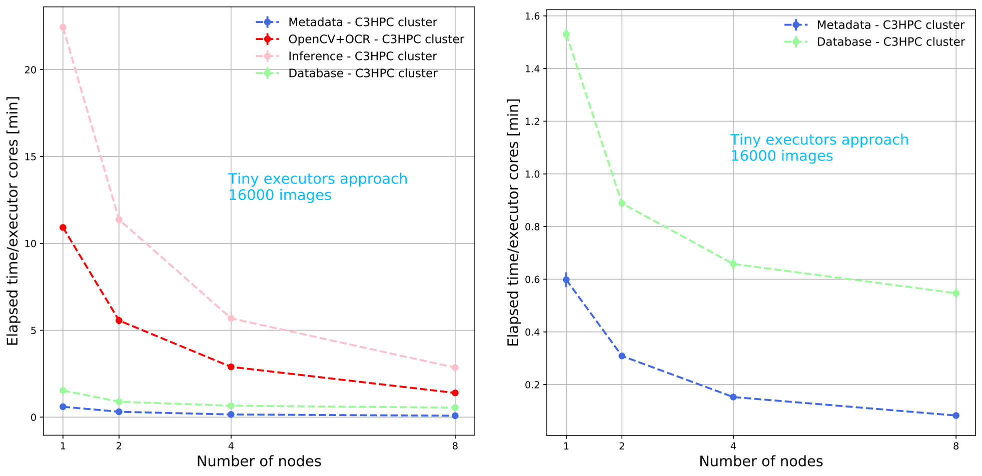


FIGURE 4.12: Elapsed time per executor core and strong scalability of C3HPC cluster for the code's four parts with Spark Fat executors approach. *Top panel*: Elapsed time per executor core of the code's four parts (left plot) on C3HPC cluster as a function of number of nodes/Pods. Moreover, a zoom of the two parts of Metadata and Database (right plot) is presented. The two parts of Metadata and Database are faster than the OpenCV+OCR and Inference ones. *Bottom panel*: Strong scalability of C3HPC cluster with perfectly linear speedup gain approaching the ideal theoretical limit for Metadata, OpenCV+OCR, and Inference parts, while it reaches 3 for the part of Database. The measurements of elapsed time and speedup are averaged over four iterations for a workload of 16000 SEM images.

Figure 4.13 presents performance of the code's four parts over same workload, of 16000 SEM images, with Spark Tiny executors approach.



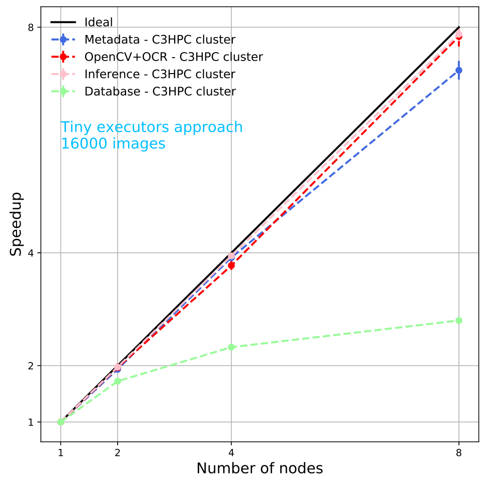
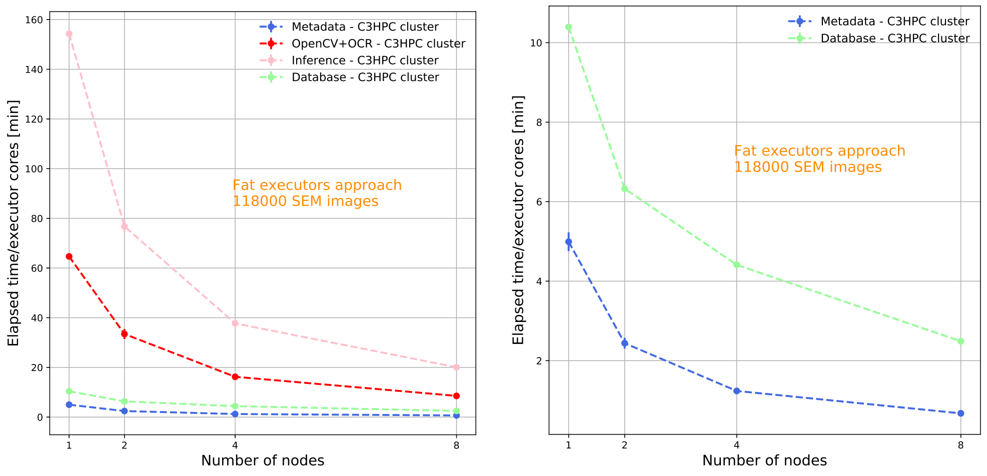


FIGURE 4.13: Elapsed time per executor core and strong scalability of C3HPC cluster for the code's four parts with Spark Tiny executors approach. *Top panel*: Elapsed time per executor core of the code's four parts (left plot) on C3HPC cluster as a function of number of nodes/Pods. Moreover, a zoom of the two parts of Metadata and Database (right plot) is presented. The two parts of Metadata and Database are faster than the OpenCV+OCR and Inference ones. *Bottom panel*: Strong scalability of C3HPC cluster with perfectly linear speedup gain approaching the ideal theoretical limit for Metadata, OpenCV+OCR, and Inference parts, while it is below 4 for the part of Database. The measurements of elapsed time and speedup are averaged over four iterations for a workload of 16000 SEM images.

Profiling the SEM code's four parts has been implemented for the large SEM dataset of 118000 SEM images for four iterations. The elapsed time per executor core besides corresponding scalability are introduced in Figs 4.14 and 4.15.



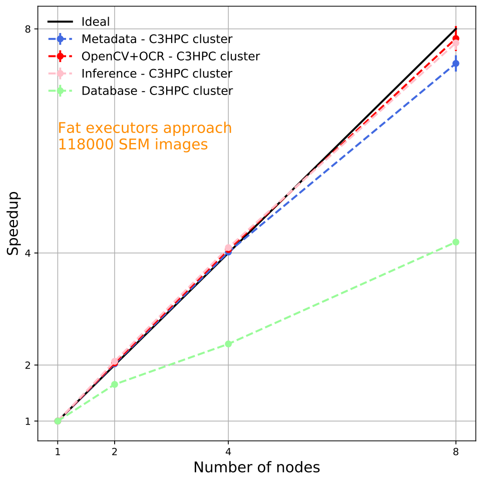
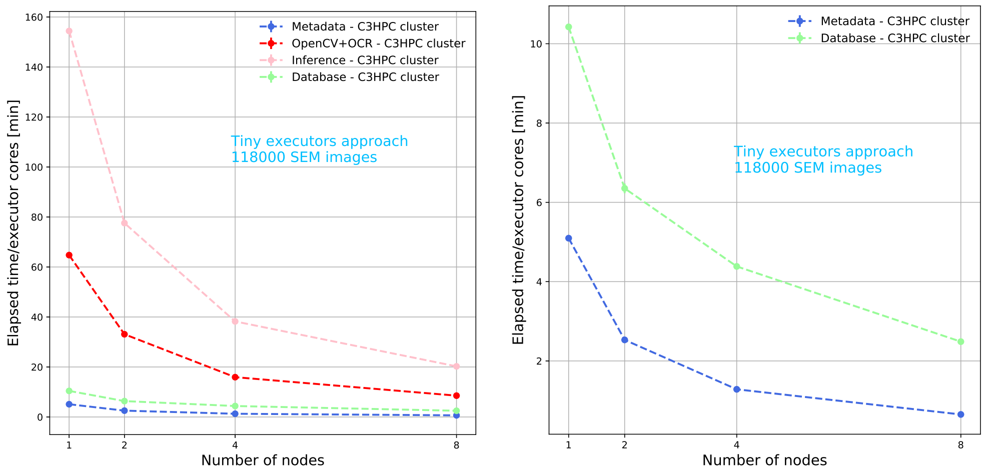


FIGURE 4.14: Elapsed time per executor core and strong scalability of C3HPC cluster for the code's four parts with Spark Fat executors approach. *Top panel*: Elapsed time per executor core of the code's four parts (left plot) on C3HPC cluster as a function of number of nodes/Pods. Additionally, a zoom of the two parts of Metadata and Database (right plot) is presented. The two parts of Metadata and Database are faster than the OpenCV+OCR and Inference ones. *Bottom panel*: Strong scalability of the C3HPC cluster with perfectly linear speedup gain approaching the ideal theoretical limit for the parts of Metadata, OpenCV+OCR, and Inference. The Database part shows less performance with maximum speedup gain around 4. The measurements of elapsed time and speedup are averaged over four iterations for a workload of 118000 SEM images.

The profiling study has been performed over the large SEM workload of 118000 images via using the Spark Tiny executors approach and presented in Fig. 4.15.



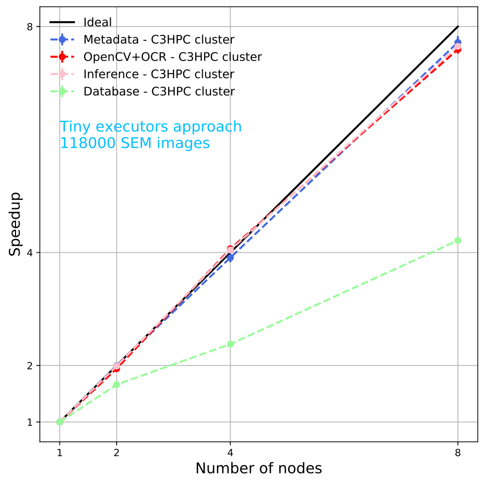


FIGURE 4.15: Elapsed time per executor core and strong scalability of C3HPC cluster for the code's four parts with Spark Tiny executors approach. *Top panel*: Elapsed time per executor core of the code's four parts (left plot) on C3HPC cluster as a function of number of nodes/Pods. Additionally, a zoom of the two parts of Metadata and Database (right plot) is presented. The two parts of Metadata and Database are faster than the OpenCV+OCR and Inference ones. *Bottom panel*: Strong scalability of the C3HPC cluster with perfectly linear speedup gain approaching the ideal theoretical limit for the parts of Metadata, OpenCV+OCR, and Inference. The Database part shows less performance with maximum speedup gain of 4. The measurements of elapsed time and speedup are averaged over four iterations for a workload of 118000 SEM images.

These profiling results from C3HPC cluster over the two SEM datasets of 16000 and 118000 SEM images show a perfectly linear speedup with the parts of Metadata, OpenCV+OCR, and Inference, while the Database part shows less performance with Spark Fat and Tiny executor approaches. This behavior of the Database part is observed in all profiling analyses performed on different architectures of a single (IRON) and multiple nodes (C3HPC cluster), filesystems, and SEM workloads. The low performance of the Database part is expected since, the database has been installed on a remote server belongs to OpenStack of CNR-IOM computing system. Possible optimization for this part is to have the SEM database installed locally, so that the time spent in transferring all metadata buffers to MySQL database will be smaller and better scalability can be attainable. Based on this profiling study, a plateau speedup feature is not observed on a multiple-node architecture (C3HPC cluster). Particularly, overall the parts of Metadata, OpenCV+OCR, and Inference. Therefore, higher speedup can be achieved with increasing resources – number of nodes/Pods in computation.

4.3 Measuring Ceph and Lustre filesystem performances

In this section, we present and discuss performances of the two different filesystems used in our SEM analyses: (a) Ceph FS on the IRON node; (b) Lustre FS on the C3HPC cluster. In these measurements, we execute the SEM Spark code, with the two Spark executor approaches (Fat and Tiny), on a single node of two different infrastructures (IRON – Intel Broadwell and C3HPC – Intel Ivybridge) over the same workload of 16000 SEM images. As already stated in the introduction of this Chapter and Chapter 3, the IRON node has in total 20 cores, while a single node of C3HPC has 24 cores. In order to establish a comparative study between the two different filesystems over equivalent number of cores and workload, the measurements on C3HPC cluster were performed on only one single node via employing in total 20 cores out of 24. The elapsed time and corresponding strong scalability of both Ceph and Lustre filesystems, as a function of number of cores, is shown in Fig. 4.16.

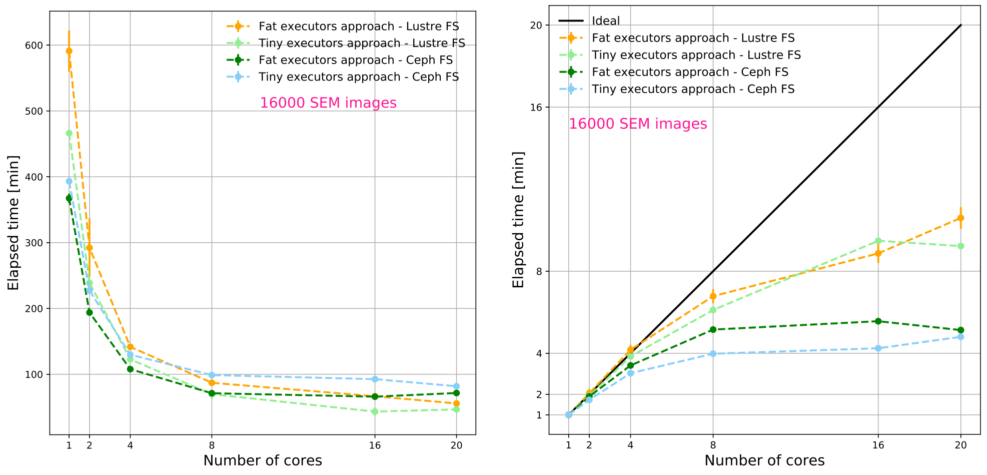


FIGURE 4.16: Elapsed time and strong scalability of IRON node (Ceph FS) versus a single node of C3HPC cluster (Lustre FS) via employing Fat and Tiny Spark executor approaches over a workload of 16000 SEM images. On Ceph FS – IRON node, a speedup gain of 5 is achieved, while on C3HPC node – Lustre FS, maximum speedup gain is above 8 (double Ceph speedup). The measurements of elapsed time and speedup, of every Spark executors approach on both Lustre and Ceph filesystems, are averaged over two iterations for a workload of 16000 SEM images.

According to these performance measurements of both Ceph and Lustre filesystems, we gain higher speedup with the Luster filesystem, with both Spark executor approaches (Fat and Tiny), than the Ceph one by a factor of 2. The Lustre parallel FS performs better and is optimal for the implementation of SEM analyses. This result is in agreement with the results obtained by Aversa in [10].

Chapter 5

Conclusions and Prospects

This thesis contributed to the research on scientific data management within the NFFA-EUROPE project. In the work presented, we have been able to achieve the goal of collecting and defining the standard metadata for a massive volume of Scanning Electron Microscope (SEM) images produced by multiple source within the project. In particular, we achieved the following goals:

- creation of python application to collect and enrich metadata for SEM images coming from different research groups within the NFFA-EUROPE project,
- developing a massive parallel processing approach in collecting metadata on a large amount of images,
- planning and setting up a fully distributed, scalable, and portable infrastructure on the CNR-IOM computing facilities and C3HPC cluster, in order to process massive amounts of SEM images,
- developing a python Spark code able to accomplish the above goal based on Kubernetes and Apache Spark,
- installing a database, on the OpenStack of CNR-IOM computing system, by means of Structured Query Language (SQL) to be populated by many sources,
- organizing all collected metadata from the SEM scientific images on the database, which servers large community of nanoscience within the NFFA-EUROPE project and

- measuring performance on different computational infrastructure of the massive data processing.

Our Kubernetes-Apache Spark software infrastructure (Spark cluster in short), has been deployed within containerized distributed environment on the CNR-IOM computing facilities: OpenStack and IRON node, besides the C3HPC cluster. In such deployment, we benefit from a loosely isolated environment provided by Docker technology besides the automation given by Kubernetes container-orchestration engine that enables us to set the optimal Spark configuration parameters, on all available nodes, in an automatic way. The SEM analyses include:

1. extracting standard metadata from the SEM images of TIFF format,
2. measuring the scale of SEM images with performing an OCR for the purpose of calculating the image pixel size [8],
3. performing image classification by means of the SEM pre-trained model [10, 14] and
4. committing all the resultant metadata, from the above problems, to a dedicated MySQL database that serves large community of nano-scientists within the NFFA-EUROPE project.

The above analyses are mainly performed over two separate SEM datasets from CNR-IOM (datashare.iom.cnr.it of 16000 images) and NFFA-EUROPE project (datashare.nffa.eu of 118000 images). The images there from all researchers are stored in the cloud platform provided by NFFA-EUROPE Datashare service and spanned over a *nested* structure.

It is worth studying the role of different Spark executors and how it can influence the performance. We therefore employed two Spark executor approaches in performing these SEM analyses: (a) *Fat executors approach*; (b) *Tiny executors approach*. We conclude that both Spark executor approaches (Fat and Tiny) show same performances in executing the SEM analyses.

A detailed benchmark study has been performed on a single (IRON) and multiple (C3HPC cluster) nodes, for the two different SEM datasets, taking into account the measurements yielded by the `sparkMeasure` library. At higher level parallelism, the overall performance is affected by *deserialization* and *JVM garbage collector* processes. However, it is difficult to establish a correlation between JVM garbage collector and corresponding performance. In particular, at a large SEM dataset of 118000 SEM images, whereas the time taken by JVM garbage collector is minimal with hiring large number of nodes. This behavior is reversed with the small dataset of a workload of 16000 images.

The speedup analysis has been discussed for both small and large SEM datasets on CNR-IOM infrastructure (IRON node - small SEM dataset) and HPC cluster (C3HPC - small and large SEM datasets).

On the IRON node, we have achieved speedup gain of 5 in processing datashare.iom.cnr.it SEM dataset via employing Fat executors approach, while speedup around 4 is obtained using the Tiny approach.

On the C3HPC cluster, processing the small and large SEM datasets was performed utilizing both Spark executors approach (Fat and Tiny). The maximum obtainable speedup gain is below 4 in processing the two SEM datasets with the Spark Fat and Tiny executor approaches.

Profiling study of the SEM Spark code's four parts is provided for:

- Metadata,
- OpenCV+OCR,
- Inference and finally
- Database on different software infrastructure.

Due to this study, we conclude that Spark benchmark analysis is not an easy task and it essentially requires a highly loosely isolated environment. Additionally, *the overall performance of the SEM Spark code over the two SEM datasets involved in these analyses is negatively affected by the Database part*. This behavior is anticipated, since we are committing the resultant metadata buffers to a remote database outside the range of the local network.

On the IRON node, the small SEM dataset of a workload of 16000 images was processed. The parts of OpenCV+OCR and Inference achieved speedup gain around 7, while the Metadata and Database parts gain speedup above and below 4, respectively.

On the C3HPC cluster, both datasets of workload of 16000 and 118000 images were processed. The parts of Metadata, OpenCV+OCR, and Inference show a perfectly linear speedup close to the ideal theoretical limit in processing both SEM datasets, with increasing number of nodes. A plateau speedup feature is not observed on C3HPC cluster, in particular, overall the parts of Metadata, OpenCV+OCR, and Inference of both SEM datasets. Therefore, better scalability can be achieved with increasing resources – number of nodes/Pods in computation – and SEM images workload.

Measuring the performances of Ceph and Lustre filesystems is performed via employing Fat and Tiny Spark executor approaches on a single node. Both Spark executor approaches show good performance on the Lustre filesystem, with speedup higher than the Ceph one by a factor of 2.

Future prospects

The SEM datasets from different research groups within the NFFA-EUROPE project are stored in the cloud platform provided by NFFA-EUROPE Datashare service. Therefore, we propose the following item for future activities:

- test the portability of Kubernetes solution on public cloud infrastructure.

List of Figures

List of Tables

Bibliography

[1] NFFA-EUROPE project, https://www.nffa.eu/

[2] Research Data Allaiance, https://www.rd-alliance.org/

[3] CNR-IOM, https://www.iom.cnr.it/

[4] Mark D. Wilkinson, et al., *The FAIR Guiding Principles for scientific data management and stewardship*, Scientific Data, **3**, 160018 (2016).

[5] Python Imaging Library, https://pypi.org/project/Pillow/

[6] Carnia Industrial Park, Carnia Industrial Park, www.carniaindustrialpark.it

[7] eXact lab, http://www.exact-lab.it/

[8] Piero Coronica, *Feature Learning and Clustering Analysis for Images Classification*, MHPC thesis, 2017/2018.

[9] Cristiano De Nobili, *Deep Learning for Nanoscience Scanning Electron Microscope Image Recognition*, MHPC thesis, 2016/2017.

[10] Rossella Aversa, *Scientific image processing within the NFFA-EUROPE Data Repository*, MHPC thesis, 2015/2016.

[11] Luca Ciuffreda, *Distributed Systems for Neural Network models*, MHPC thesis, 2017/2018.

[12] OpenCV, https://opencv.org/

[13] TensorFlow, https://www.tensorflow.org/

[14] Rossella Aversa, Mohammad Hadi Modarres, Stefano Cozzini, Regina Ciancio, Alberto Chiusole, *The first annotated set of scanning electron microscopy images for nanoscience*, Scientific Data, **5**, 180172 (2018).

[15] TensorBoard, https://www.tensorflow.org/tensorboard/

[16] MySQL, https://www.mysql.com/

[17] OpenStack of CNR/IOM cloud computing system, https://cloud.iom.cnr.it/

[18] phpMyAdmin, https://www.phpmyadmin.net/

[19] MySQL Workbench, https://www.mysql.com/products/workbench/

[20] https://docs.microsoft.com/

[21] https://docs.python.org/2/library/md5.html/

[22] https://docs.python.org/2/library/hashlib.html/

[23] Apache Spark, https://spark.apache.org/

[24] https://dev.mysql.com/doc/connector-j/5.1/en/

[25] Kubernetes, https://kubernetes.io/

[26] Docker, https://www.docker.com/

[27] Ansible, https://docs.ansible.com/

[28] kubectl command-line, https://kubernetes.io/docs/reference/kubectl/

[29] https://raw.githubusercontent.com/coreos/flannel/

[30] Apache Hadoop, https://hadoop.apache.org/

[31] Jeffrey Dean, SanjayGhemawa, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), pp. 137-150.

[32] Apache Storm, https://storm.apache.org/

[33] Apache Tez, https://tez.apache.org/

[34] Neo4j, https://neo4j.com/

[35] https://spark.apache.org/docs/latest/cluster-overview.html

[36] https://aws-labs.com/private-docker-registry-kubernetes/

[37] Kubernetes-Spark deployment, https://github.com/phatak-dev/kubernetes-spark

[38] Ceph filesystem, https://ceph.io/ceph-storage/file-system/

[39] Beginner's Hadoop, http://beginnershadoop.com/

[40] Lustre filesystem, http://lustre.org/

[41] sparkMeasure tool, https://github.com/LucaCanali/sparkMeasure/

[42] sparkMeasure tool, https://index.scala-lang.org/lucacanali/sparkmeasure/

[43] Spark monitoring, https://spark.apache.org/docs/2.4.0/monitoring.html