Tampere University

Ashish Garg

EFFECTIVE PARALLELIZATION FOR TELECOMMUNICATION NETWORK ENTITIES CORRELATION

Faculty of Computing Sciences
Master's thesis
September 2022

Abstract

Ashish Garg: Effective parallelization for telecommunication network entities correlation
Master's thesis
Tampere University
Master's Degree Programme in Machine Learning
September 2022

Background. In telecommunications, a cellular network is a radio network distributed over land through cells. These cells are network elements. To perform network management operations it is vital to calculate relationships between them and represent such relationships over knowledge graph. Due to large scale of data in real-time, calculating such relationship require potential computation power.
Objective. We aim to leverage the capability of parallel processing frameworks to calculate relationships between network elements and perform parallelized interaction with knowledge graphs to help reduce overall execution time.
Method. An empirical study conducted where ten days of performance metric data was used to calculate relationships. Ray Core and Apache Spark were used as parallel processing frameworks where their efficiency to parallelize relationship calculation was compared with a normal sequential execution. A similar study designed to check the interaction with orientDB graph databases to perform parallelized relationship creation and updates.
Results. Frameworks were evaluated based on the growth of problem size i.e efficiency of parallelizing from one to several days of data. Ray Core showed better throughput compared to Apache Spark and normal execution for relationship calculation. Around 80% reduction in time observed compared to sequential execution. Relationship calculation and updates in the knowledge graph can also be parallelized using Ray, where efficiency reduces on increase in amount of data.
Conclusion. Current work involved the use of OrientDB as a graph database which is considered a sub-optimal choice to perform parallel edge creation and updates. Future work might investigate the use of other databases like Neo4j and evaluate its interaction with Ray Core and Apache Spark.
Keywords: Ray Core, Apache Spark, Correlation, Knowledge Graph, Network Elements.

Contents

1 Introduction 1
1.1 Problem statement 1
1.2 Research objectives 1
1.3 Research Questions 1
1.4 Research Method 2
1.5 Benefits (for academia or industry) 2
1.6 Structure of the paper 2
2 Background 3
2.1 The Cellular network 3
2.2 Context 3
2.3 Parallel Processing Framework 5
2.3.1 Python Multiprocessing 5
2.3.2 Ray 6
2.3.3 Apache Spark 7
3 Related Work 8
4 Research Method 10
4.1 Goal & Hypothesis 10
4.2 Data Collection 11
4.3 Data Analysis 13
4.3.1 Data Prepossessing 13
4.3.2 Data Analysis 14
4.3.3 Scalability Comparison metrics 16
4.4 Replicability 16
5 Solutions 18
5.1 General Evaluation: 18
5.1.1 Ray: 18
5.1.2 Apache Spark: 21
5.2 Parallelization using Ray: 24
5.2.1 Parallelized File Loading: 24
5.3 Correlation Calculation: 24
5.4 Knowledge Graph Interaction: 25
5.5 Parallelization using Apache Spark: 27
5.5.1 Parallelized File Loading: 27
5.5.2 Correlation Calculation: 27
5.5.3 Knowledge Graph Interaction: 28

6 Evaluation 29

7 Conclusions 33

8 Abbreviation 34

References 36

1 Introduction

In telecommunications, network management system is essential to manage the underlying mobile network. Network element cells called as LNCELLS forms such a network which facilitates the exchange of messages and signals. Several daily network management operations like identifying group of best or worst performing LNCELLS or group of LNCELLS on which software upgrade needs to be done on same time to minimize the overall impact needs to be performed. In order to perform such operation it is essential to calculate relationships, to obtain the group of cells having similar behavior and represent them over a knowledge graph.

1.1 Problem statement

In real-time huge volume of network traffic leads to a massive generation of data. Processing such a volume is computationally expensive and non-trivial. Hence it is essential to optimize multiple bottlenecks to reduce the processing time significantly, such that each day of data processes within a few hours.

1.2 Research objectives

Parallel computing architectures provides considerable computational performance to solve such data scalability problem. In V., Venkata, and Preethi 2017 it is demonstrated that parallel processing improves the overall processing time as CPUs are not ideal for such large-scale data processing because of their sequential processing nature. In this thesis, we aim to study multiple parallel processing frameworks and advancing them to optimize multiple bottlenecks present to calculate LNCELL relationships and their representation over a knowledge graph.

1.3 Research Questions

The following research questions (RQs) are explored in this study.
RQ1 What is the usability of different modern parallel processing frameworks when optimizing bottlenecks?
RQ2 How we can perform optimal parallel computation?
RQ3 Can we perform parallel computation while working with graph database?
RQ4 Which are the best frameworks/tools to parallelize the bottlenecks?

1.4 Research Method

The research methodology of this work includes:

- Stating the goal and establishing a hypothesis.
- Describing data collection strategies.
- Data analysis strategy including the data prepossessing and scalability comparison metrics.
- Replicability.

1.5 Benefits (for academia or industry)

The study reveals significant findings, where the parallel processing framework provided 60 percent better throughput compared to sequential execution. Frameworks/tools analyzed within the study are quite responsive that can scale dynamically with robust handling of machine failures. The contribution of this work involves:

- Study modern parallel frameworks Apache Spark and Ray to understand their implementation, responsiveness, ease of use, availability, cluster support, and handling of machine failures.
- A comparative study between Ray and Apache Spark to optimize relationship calculation between network elements.
- Test the capability of Ray and Apache Spark to parallelize edge creation, updates, and deletion in the knowledge graph.

1.6 Structure of the paper

The rest of thesis is structures as follows. In second section, we introduce the background in this work, the cellular network, the context, and the discussion about parallel processing frameworks. Third section discuss the related work, fourth section describes the Research Methodology, and fifth section presents the solutions. Sixth section evaluates the solution, while seventh section states the conclusion.

2 Background

In this section, we illustrate the background of this work, introducing the cellular network, context, and the parallel processing tools used in this study.

2.1 The Cellular network

In telecommunications, a cellular network is a radio network distributed over land through cells. These cells are network elements. Network elements provide network coverage over diverse locations. Mobile phone can connect even if it is moving through cells during transmission. In order to enable communication and passage of traffic cells are connected as a group to exchange messages and signals. Configuration of each network element is set based on its position within the network and tweaked by the network operator to configure the behavior of the cell. To corroborate the quality and stability of the network cell performance management metrics are also monitored for each based on several key performance indicators. As stated in Martinez-Mosquera, Navarrete, and Luján-Mora 2020 such data is collected from network elements to a centralized system, which is analysed for monitoring and reporting network performance. Performance management files contain the metrics and named counters used to quantify performance of the network.

A network management system (NMS) enables engineers to manage a network's independent components inside the network management framework and perform several vital operations. NMS has data associated with network elements. The data can be CM (Configuration Management) and PM (Performance Management). By performing analysis on this data we can get Insights about the behavior of the network elements.

2.2 Context

Network elements interconnect with relationships present between them. For example, the KPIs forecast of network elements present strong correlations where some network cells behave similarly based on their performance. Networks cells have the same set of values over configuration parameters that can be related. These relationships between network elements are represented as a graph, where network elements are nodes associated with knowledge attributes and the relationships between the attributes as edges.

Figure 2.1 demonstrate the knowledge graphs created from different input sources and stored in the knowledge store. Knowledge Store shows the knowledge graphs and Services triggering the operations to perform over-represented knowledge.

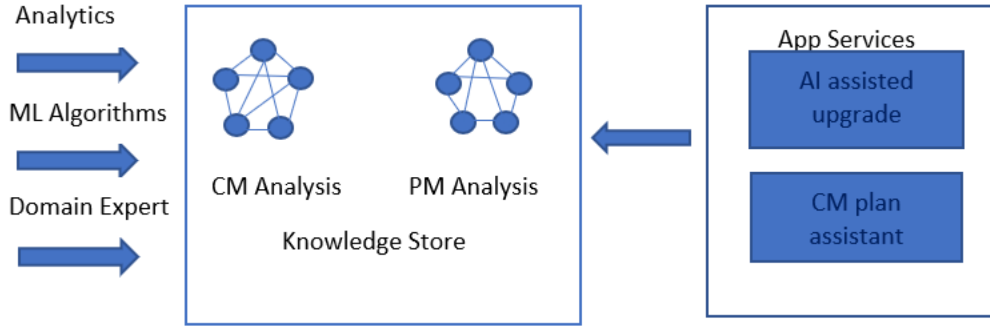

Figure 2.1 *Knowledge Representation*

Upon discussion with Network Management Team, it was identified that network cells are highly correlated to performance management metrics. To identify such groups of entities based on several KPIs measuring KPI-based cell correlation, querying, and updating knowledge graph in real-time is essential. Performance management data is a time series with new performance metrics available at every 15 minutes interval. Real-time data scales up where processing a massive amount of data is computationally expensive and non-trivial. Some relevant bottlenecks identified as follows:

- Parallel loading of excel files.
- Calculating Correlations.
- Generating Nodes (Network Cells) and KPIs in the graph.
- Edge creation and updates within the nodes.

Time utilized in processing one day worth of performance metrics are stated below within Figure 2.2. Based on the time consumed over each bottleneck it was identified that Correlation Calculation and Edge Creation within knowledge graphs are the major bottlenecks that need to be optimized.

Direct use of CPU is not ideal for such large-scale data processing. Parallel computing architectures can provide substantial computational performance to solve such data scalability problems. Though the idea of parallelization is attractive but evaluation of the best alternative among available frameworks is not trivial and depends on the architecture of the machine. In the current work optimizing these bottlenecks is of importance. Parallel computing can break input data into batches and parallelize file reading, data transformation, correlation calculation, and data querying with feeding. Interweaving these steps in such a way can massively optimize resource utilization. Python provides a native library known as multiprocessing

Operation	Time Taken	Decision
Correlation Calculation	0:11:22.8675	Optimization needed.
Database Connection	0:00:01.138600	No Optimization needed.
Generating Nodes	0:00:00.470364	No Optimization needed.
Generating KPI's	0:01:08.276279	No Optimization needed.
Edge Creation	1-2 Days	Optimization needed.

Figure 2.2 *Bottlenecks time consumption*

to parallelize jobs where modern parallel processing frameworks like Ray as demonstrated by Moritz et al. 2017 and Spark as shown in Zaharia et al. 2016 can also help to achieve such parallelism.

2.3 Parallel Processing Framework

In this section, we will discuss python workload parallelization and the frameworks used for parallelization such as Ray and Apache Spark.

2.3.1 Python Multiprocessing

Python provides Multiprocessing library, which helps to parallelize the workload by creating child processes. The creation of child processes involves sub classing the multiprocessing process, which in turn creates the process that can run independently. A simple code snippet below demonstrates the implementation.

```
import multiprocessing
import time
class Process(multiprocessing.Process)
    def __init__(self, id):
        super(Process, self).__init__()
        self.id = id
    def run(self):
        time.sleep(1)
        print("I'm the process with id: {}".format(self.id))

if __name__=='__main__':
p = Process(0)
```

```
p.start()
p = Process(1)
p.start()
```

Hereafter initiating the process with ID 0, process one will start immediately and hence will not wait for process 0 to finish. However, there are multiple drawbacks identified with multiprocessing:

- There is Input/Output time complexity involved when data shuffles between processes.
- Entire memory copies into each sub-process, which builds a lot of overtime.
- Difficulty in scaling existing code to a cluster of computers.

Hence to overcome such limitations, modern parallel processing frameworks are designed, which are discussed in next sections.

2.3.2 Ray

Ray, an open-source project, was initiated in the U.C Berkeley RISE lab in 2018. Designing and developing distributed systems need extensive deep programming and infrastructure knowledge. The understanding behind this work is to overcome this challenge and enable the user to build a distributed application over a laptop and even scale the same to the cluster without any changes. The python code is scalable using Ray Core functionalities.

Ray Core is a distributed execution framework that runs the python code as a distributed application with minimal code changes. The function to be parallelized is assigned as a task using the decorator @ray.remote over the function definition. Function calls will run in parallel on separate CPU cores. Ray also comes with a dashboard that lets to understand memory utilization, per actor resource usage, executed tasks, logs, and more. Some advantages of using Ray as a parallel processing engine are:

- Possible to scale without code rewriting.
- A multi-cloud solution.
- Dynamic Scaling.
- Open-source.
- Capability to run the same code on more than one machine.
- Users can specify required resources to allocate for a particular function call.

```
@ray.remote (num_cpus=4, num_gpus=2)
def my_function():
    return 1
```

2.3.3 Apache Spark

Apache Spark is an open-source project developed by U.C Berkeley AMP Lab. It is a unified analytics engine to scale up computationally expensive tasks. It is designed over top of RDD (Resilient distributed database) using which multiple operations over multiple clusters are done on the same data frame. Spark comes up in various flavors.

- Spark Core: It is a base execution engine over which other top functionalities are developed.
- Spark SQL: Built over the top of spark core, provides SQL language support, with command line interface and several database connection drivers.
- Spark Streaming: Provide usability and optimizations for streaming analytics.
- Machine Learning Library: Provide spark-specific implementations of native machine learning libraries.
- GraphX: Parallelized graph-processing functionality built over the top of spark. Advantages:
  - Parallel Processing possible.
  - Open-Source.
  - Dynamic Scaling.

  Disadvantages:
  - Rewriting of code required.

3 Related Work

Parallel Programming is one of the most active research areas in data engineering. An increase in the data to process and availability of multi-core architectures led to parallelizing the existing systems to reduce process time and generate faster results.

In recent years, parallelization strategies developed for specific use-cases as in Asgari et al. 2022 parallel computing frameworks reviewed for calibrating watershed hydro-logic models. Xu, H. Liu, and Long 2020 demonstrated the use of hybrid distributed computing framework to perform wind speed big-data forecasting, the wind speeds were divided into Resilient Distributed database groups and operated in parallel. Husain et al. 2022 discussed use of multi-core parallel algorithm to compute fractal dimensions to understand the geometric irregularity present in Indian coastline. Yang et al. 2020 implemented parallel computing accelerated path independent DIC method using C++ language for two type of devices (GPU and multi-core CPU). Feature matching done with multi-core CPU resulted in 3.4x faster speed than sequential execution. All such reviewed studies reported a speedup gain of at least 60-70 percent in comparison to sequential execution.

Researchers also showed a great interest in parallelizing correlation calculation for various use-cases. For instance, Mu, X. Liu, and Wang 2018 state Pearson-based correlation decision tree (PCC-Tree) with its parallel implementation developed in Map-Reduce (MR-PCC-Tree). To parallelize computationally expensive tasks proposed MR-PCC-Tree algorithm performs on the cluster of processors for 8 datasets. The performance was discussed on three factors: Speedup, Scaleup, and Size up. The processor numbers are grown from 1 to 7. On larger datasets the speedup tends to be approximately linear as the proportion of communication cost becomes smaller. Scaleup had a downward trend with an increase in data and processors, whereas the algorithm displays good size-up performance in the given system. In Joubert et al. 2019 researcher is calculating the custom coefficient correlation between Single Nucleotide Polymorphisms (SNPs) using genome sequencing data, genomic data is huge where a parallel processing algorithm can scale up to the thousands of computer nodes. Native python multiprocessing libraries.

Pool have also gained sufficient traction in recent years. Xu, H. Liu, and Long 2020 showed the implementation of a hybrid wind speed big data forecasting framework proposed on Apache Spark using the distributed computing strategy. The framework can divide the wind speed big data into RDD groups and operate them in parallel. The experimental results indicated that proposed distributed computing framework on Spark forecast wind-speed big data in multi-steps. It is proved that the proposed distributed computing framework has a faster computation speed

when processing big data compared to stand-alone method.

Al-Saqqa, Al-Naymat, and Awajan 2018 demonstrated a relatively new open-source platform Dask developed by Continuum Analytics for training machine learning models faster whereby providing the capability to scale NumPy and Pandas workflows and flexible in implementing custom workloads. Rocklin 2015 showed the use of dask for parallel computation with blocked algorithms and Task Scheduling. The constraint in all these works with these processing frameworks is the need to write the whole code from scratch based on the particular framework to achieve complete parallelism. To solve the constraint and parallelize code fast Moritz et al. 2017 introduces a relatively new parallel programming framework Ray as a distributed execution framework that runs the python code as a distributed application with minimal code changes. Python functions can directly be assigned as a task using @ray.remote over the function definition. Each task as a particular function call can run parallel on separate CPU cores.

Though understanding parallel processing frameworks is vital, researchers are also interested in identifying the suitable method to evaluate such frameworks for the given problem. Zhang, Yan, and He 1994 shows the use of latency metric to evaluate the parallel program and architecture stability. Latency here involves the delay caused by communication between processors and memory modules over the network within a parallel system. It can form the major obstacle to improving parallel computing performance and stability. Jogalekar and Woodside 2000 explained a way to measure the Speedup, Efficiency, and Scalability of such frameworks. It also stated the use of throughput as metric to check scalability of a system based on the cores.

4 Research Method

In this section, we describe the research methodology, including the goal and the hypothesis, the data collection, the data analysis, and the replicability.

4.1 Goal & Hypothesis

In this work the capability of modern computer systems having multiple cores is leveraged to explore relevant areas of interest such as:

1. Studying modern parallel processing frameworks and understanding their ease of use.
2. Implementing parallel file loading functionality.
3. Parallelizing the functionality to calculate the relationship between network elements.
4. Understanding the compatibility to use the respective framework with graph database.
5. Parallelizing edge creation and updating functionality within knowledge graph.
6. Evaluating and selecting the best framework to parallelize the bottlenecks.

In the first area, the aim is to investigate modern parallel processing frameworks like Ray Core and Apache Spark. The goal is to understand the scalability they bring using multiple cores in parallel, their availability as a licensed version or open-source, programming libraries support, responsiveness, dynamic scalability, handling of machine failures and pre-emption, and cluster support. In terms of ease of use, the usability aspect of the framework investigated as the need for code rewriting, custom resource allocation, configuration setting, and framework flexibility. In the second area, parallelized file loading functionality is implemented to load multiple files simultaneously. In the third area, the calculated combinations of available network cells are clustered into partitions equal to available cores i.e each CPU node is assigned a particular cluster of combinations, correlations are then calculated in parallel among CPU nodes. In the fourth area, respective availability of drivers and parallel processing framework/tool connection capability with orient DB graph database is explored. In the fifth area, in order to represent relationships between network cells as knowledge graph the process of querying and edge creation is parallelized to optimize the interaction with graph database. In stheixth area, evaluation of performance of framework/tool is accessed.

Hypothesis testing helps to predict the relationship between two variables. Hence, a testable working hypothesis is essential for this work. We designed our hypothesis based on guidelines defined by Misra et al. 2021 for this comparative study. Related work over parallel processing platforms demonstrated that Spark gained relevance as a parallel processing framework over the years due to its scalability, wider use, support, and availability whereas Ray is a relatively new framework in comparison. Evaluation of Ray for parallelizing bottlenecks is required to understand the framework better. Hence, a null (H0) and alternative hypothesis (H1) for this thesis work is:

- H0: Ray is equivalent to Spark in terms of usability and scalability.
- H1: Ray is better than Spark in terms of usability and scalability.

4.2 Data Collection

The network management team closely monitors the mobile network based on three core data metrics: Configuration, Fault & Performance Management Data.

- **Configuration Management Data (CM):** The Network cell has an array of parameters set as a configuration based on its position on the mobile network, stated as configuration management parameters. These configurations are tweaked by the network operator to alter the functionality of the network cell. Such information of an array of network cells is aggregated in the form of Configuration Management Data.

- **Performance Management Data (PM):** Performance of each network cell based on various key performance indicators (KPIs) are collected as Performance Management Data.

Recent reports from Network management team identified that network cells are highly co-related for their behavior based on various performance metrics. Hence, evaluating the correlation between them using performance management data is essential compared to other metrics. The performance data collected from the Nokia network management team which manages network cells for 4G network and 5G network based on various performance metrics known as key performance indicators. 29 days of 4G and 5G network cells data was collected from a region in Southern Italy and stored in excel files. One days' worth of performance metrics contained in each excel file.

The sample mock data as shown in Figure 4.1 and Figure 4.2 is a time-series data where performance metrics of network cells are captured every 15 minutes. The data contains the timestamp, Network Cell, and various KPIs as columns. The format of the dataset is entirely numerical (floating point). The Network Cell is called LNCELLS in the case of 4G networks and NRCELL in the case of 5G networks.

- 4G Data: 50 LNCELLS were reported and their Performance metrics as time-series data based on 167 Key performance indicators were captured every 15 minutes.
- 5G Data: 60 NRCELLS reported their Performance metrics as time-series data based on 116 Key performance indicators captured every 15 minutes.

Timestamp	LNCEL name	KPI 1	KPI 2	KPI 3	...	KPI 267
00:00:00	Cell 1	2.99	4.68	8.76	...	3.1
00:00:00	Cell 2	4.55	3.21	3.99	...	2.4
00:00:00	Cell 3	8.99	1.24	1.73	...	2.3
00:00:00	Cell 4	3	2.98	4.88	...	6.7
...	...	...	...	...	...	4.5
...	...	...	...	...	...	3.6
00:00:00	Cell 13	7.99	5.77	8.99	...	2.4
00:00:15	Cell 1	2.66	3.11	2.68	...	4.55
00:00:15	Cell 2	3.12	2.54	1.32	...	2.37
00:00:15	Cell 3	2.87	3.76	3.21	...	8.43
00:00:15	Cell 4	2.34	3.16	1.22	...	3.21
...	...	...	...	...	...	1.28
...	...	...	...	...	...	3.22
00:00:15	Cell 13	8.32	2.54	3.66	...	1.27
...	...	...	...	...	...	...

Figure 4.1 *4G Network Performance Data*

Timestamp	NRCEL name	KPI 1	KPI 2	KPI 3	...	KPI 117
00:00:00	Cell 1	1.66	1.88	1.87	...	2.13
00:00:00	Cell 2	2.33	2.76	2.43	...	3.21
00:00:00	Cell 3	1.29	5.43	3.22	...	6.54
00:00:15	Cell 1	3.21	1.32	1.98	...	3.22
00:00:15	Cell 2	6.43	2.09	1.45	...	6.44
00:00:15	Cell 3	2.34	3.17	2.34	...	1.76
...	...	...	...	...	...	...

Figure 4.2 *5G Network Performance Data*

Access to the data is restricted and not publicly available, hence the mock data is shown as part of this thesis. Ethical considerations are taken into account and permissions were in place during data collection and presentation. For instance, data is sensitive and collected directly from the product architect of the network management team hence no external tools, API, or scarping technologies were used. Consent was taken to use the data for this thesis work.

4.3 Data Analysis

In this section, we report the data analysis protocol adopted in this study including data prepossessing, data analysis, and scalability comparison metrics.

4.3.1 Data Prepossessing

Performance management data for both 4G and 5G is available in form of separate excel files, as shown in Figure 4.3. Each excel file contains 24-hour performance metrics. It is essential to perform data prepossessing before parallelizing correlation calculation as excel files cannot be fed directly, hence converting the data into the required format is crucial before proceeding with correlation calculation. The preprocessing was composed of three operations:

- Conversion into data-frame & appending.
- Finding Combinations.
- Grouping data-set based on network cell name.

RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:29	Microsoft Excel Worksheet	66,692 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:29	Microsoft Excel Worksheet	47,913 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:29	Microsoft Excel Worksheet	47,751 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:29	Microsoft Excel Worksheet	39,216 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:29	Microsoft Excel Worksheet	39,302 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:29	Microsoft Excel Worksheet	39,360 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:29	Microsoft Excel Worksheet	39,372 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:29	Microsoft Excel Worksheet	47,994 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:30	Microsoft Excel Worksheet	39,561 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:30	Microsoft Excel Worksheet	47,709 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:30	Microsoft Excel Worksheet	39,211 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:30	Microsoft Excel Worksheet	39,153 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:30	Microsoft Excel Worksheet	39,178 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:30	Microsoft Excel Worksheet	38,970 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:30	Microsoft Excel Worksheet	39,241 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:30	Microsoft Excel Worksheet	39,291 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:31	Microsoft Excel Worksheet	39,051 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:31	Microsoft Excel Worksheet	38,755 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:31	Microsoft Excel Worksheet	38,047 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:31	Microsoft Excel Worksheet	39,032 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:31	Microsoft Excel Worksheet	39,162 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:31	Microsoft Excel Worksheet	39,322 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:31	Microsoft Excel Worksheet	39,266 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:31	Microsoft Excel Worksheet	39,106 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:32	Microsoft Excel Worksheet	47,452 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:32	Microsoft Excel Worksheet	38,970 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:32	Microsoft Excel Worksheet	39,076 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:32	Microsoft Excel Worksheet	39,063 KB
RSLTE001_SystemReport_VFIT_ML_BOOMC16_LNCEL_RAW-10465-2021...	18-03-2022 09:32	Microsoft Excel Worksheet	39,144 KB

Figure 4.3 *Data files (4G)*

Conversion into dataframe & appending: Usage of performance data within python functions and implementing mathematical operations on it required conversion of it into a respective data frame. One of the main goals of this thesis work is

to check the scalability of the frameworks, multiple days of data files combined into a single data frame for the purpose.

Finding Combinations: Correlation Calculation will be done between the pair of network cells. Hence, all possible combinations of two network cells within a group are formed from the list of unique network cells present inside the data frame.

Grouping data-set based on network cell name: Data of a single network cell is distributed in an entire data frame based on different timestamps, as after every 15 mins a new metric is available. Hence it is crucial to do grouping of the entire data frame based on the network cell name, to ease the extraction of single network cell values while correlation calculation.

4.3.2 Data Analysis

We first analyzed the time taken by standalone CPU execution for doing correlation calculations. Code was then enhanced to test the execution with parallel processing frameworks. Lastly, all results compared to choose the best platform. Analysis is implemented in the remote lab with 64 GB RAM and 16 CPU cores configuration.

Standalone CPU execution: For standalone CPU execution, the data was fed directly. As performance data scales up in execution, the data was tested based on the growth of problem size i.e, time is taken to process 1 day of data, then 2 days of data, and so on until 29 days of data.

Parallel Processing Framework: Ray Core and Apache Spark have been used as parallel processing frameworks. The functionality is implemented to accommodate parallel processing in a way that once several network cell combinations were identified they were divided into an equal set of combinations based on the number of CPU cores. Figure 4.4 demonstrates 80 combinations divided based on 8 cores present in the system. The frameworks were tested and compared based on the growth of the problem with an increase in the number of files based on days as done in Standalone CPU execution. In addition behavior of both parallel processing frameworks based on the increase in the number of cores in the system is tested and compared.

Interaction with Knowledge Graph: For generating business value from the whole work, it is important to represent relationships over a knowledge graph. Figure 4.5 shows the basic representation of the structure, wherein there is a relationship present between each node (Node 1, 2, and 3) based on KPI 1 and KPI 2. During the interaction with the knowledge graph, two major operations can take place as stated:

- Creation of new edge: Whenever a new relationship is identified a new edge will be created to show the relationship.

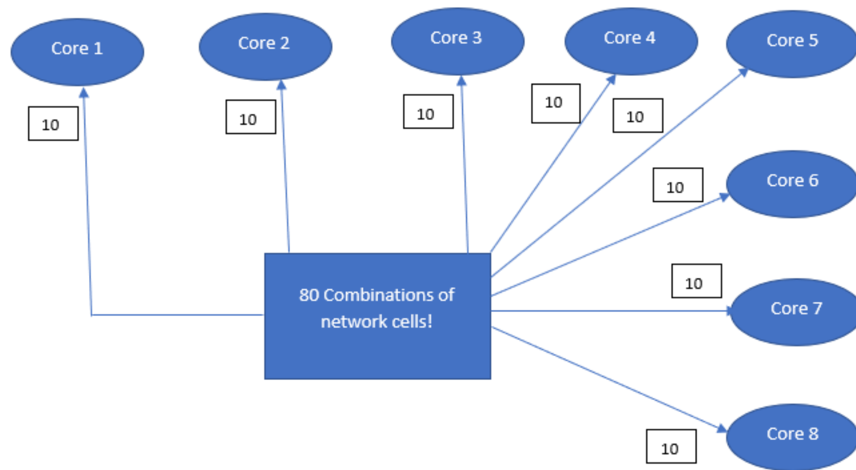

Figure 4.4 *Distribution based on cores*

- Updating existing edge: If there is a change in relationship value between existing two nodes on the graph and the change is above or below some threshold the relationship value will be updated accordingly. The threshold value provided by the user within settings.

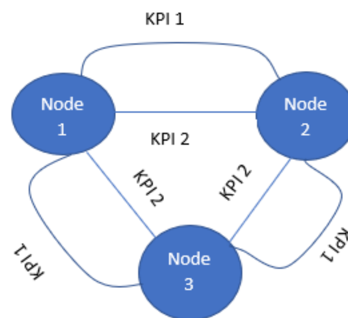

Figure 4.5 *Relationship between nodes*

As stated in Parallel Processing Framework section, available cell combinations divided into clusters equal to number of available cores and edge creation with updates done in parallel among CPU nodes. For instance as shown in Figure 4.6 if we have 300 combinations and 3 cores in the system, 100 combinations per core can be considered for new edge creation and updates in parallel.

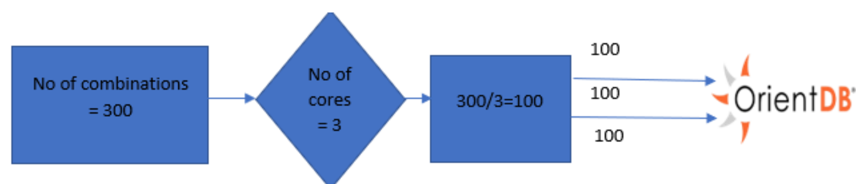

Figure 4.6 *Distribution based on cores*

4.3.3 Scalability Comparison metrics

Scalability metrics for parallel system as demonstrated in P. Jogalekar and M. Woodside 2000 used to choose comparison metrics for the current work. In the case of distributed or parallel processing systems, it is crucial to model their behavior as a steady-state, as there are always fixed jobs running in parallel based on the number of cores present in the system. The productivity of the framework will be evaluated through throughput.

Throughput: Throughput is an inverse of the time taken to complete the job. In a parallel processing system with N jobs running in parallel the throughput is equal to N/ (job time). Number of jobs N is stated as the degree of freedom in the analysis, whereas job time is the average time processors take to finish the job.

The current system does not involve any communication between the parallel running jobs, hence no network communication overhead is considered during this analysis. To check the behavior of the framework based on growth in problem size, a throughout obtained versus number of input files graph will be plotted as shown in Figure 4.7. Each file contains the performance data for a single day. In such a scenario, the number of the CPUs used in parallel is kept fixed i.e. 16.

In addition to the execution speed, the goal of this thesis work is to explore multiple aspects of framework like responsiveness, dynamic scalability, handling of machine failures, and pre-emption. Quality of Service (QOS) will also be calculated to include a measure of the goodness of a service. Factors included under quality of service are:

- Responsiveness.
- Dynamic Scalability.
- Handling of machine failures and pre-emption.
- Cluster Support.

In addition to evaluating the solution based on the stated metrics productizing the framework requires evaluation of additional costs like the cost of software licenses, and perhaps the cost of operation such as management and help desks.

4.4 Replicability

Access to data is restricted within the company, hence results cannot be replicated directly.

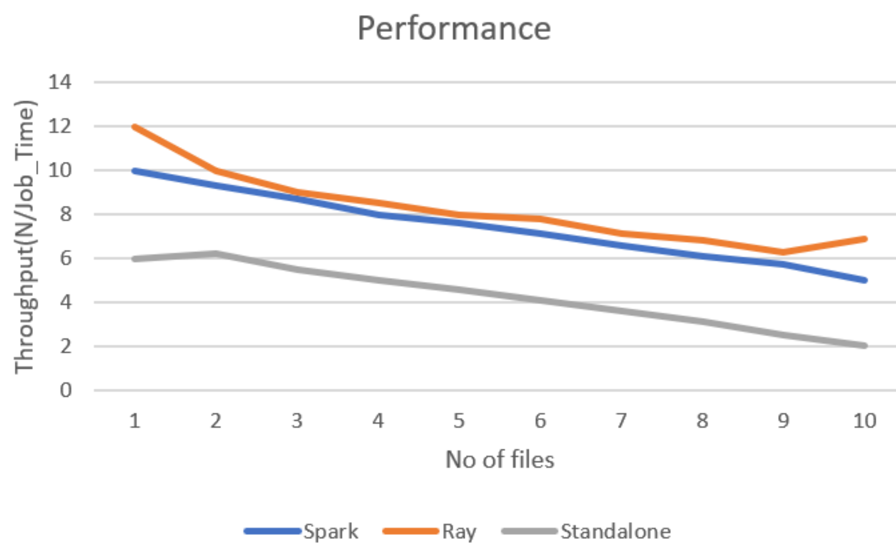

Figure 4.7 *Performance comparison based on growth of files*

5 Solutions

In this section, we will discuss the parallel processing frameworks used and the results obtained by their application to answer the goals established in the Research Methodology section.

5.1 General Evaluation:

5.1.1 Ray:

Ray is a flexible python first distributed computing framework. Python on its own is ineffective for distributed computing. Its interpreter is effectively single threaded that makes it difficult to, for example, leverage multiple CPU's on the same machine, let alone a whole cluster of machine, using plain python. Using Ray as a framework, the user can parallelize python programs on a laptop and run the code tested locally on the cluster practically without any changes. Ray provides extensive high-level libraries that are easy to configure, for instance Ray's Reinforcement learning library. Ray is useful when we combine several of its modules to facilitate custom machine learning heavy workloads, by doing this user can design flexible distributed python. Programming distributed systems is not trivial and using currently available frameworks to get clusters of computers to do what we want is considerably difficult. Ray is built over top three layers:

- Low-level distributed computing framework for python with concise core API.
- A set of high-level libraries.
- Integration and partnerships with notable projects.

Ray is capable of setting up and managing a cluster of computers that can run distributed tasks, a Ray cluster nodes are connected via a network. A driver programmed called a head node that can run jobs as a collection of tasks on the nodes in a cluster. The basic structure of the Ray configuration can be found in Figure 5.1. Within a python session user can easily import and initialize Ray as follows:

```
import ray
ray.init()
```

These two lines of code will start a Ray Cluster on a local machine. This cluster can utilize all the cores available on a computer as workers, and Ray will be initialised as shown in Figure 5.2, which indicates that Ray cluster is up and running, Ray comes with its own pre-packaged dashboard which can be check out at

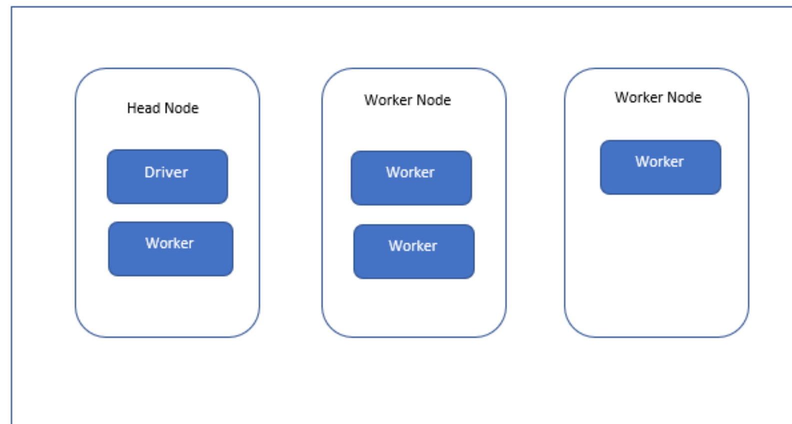

Figure 5.1 *The basic components of a Ray cluster*

```
... INFO services.py:1263 -- View the Ray dashboard at http://127.0.0.1:8265
{'node_ip_address': '192.168.1.41',
 'raylet_ip_address': '192.168.1.41',
 'redis_address': '192.168.1.41:6379',
 'object_store_address': '.../sockets/plasma_store',
 'raylet_socket_name': '.../sockets/raylet',
 'webui_url': '127.0.0.1:8265',
 'session_dir': '...',
 'metrics_export_port': 61794,
 'node_id': '...'}
```

Figure 5.2 *Ray Initialisation*

http://127.0.0.1:8265. In terms of ease of use, the best and most successful frameworks are the ones that integrate well with existing solutions and ideas. Ray is a compute-first framework whereas other modern frameworks are data-first where it achieves the functionality to distribute computing over nodes using tasks and actors. In the python program, a Ray API uses the concept of decorators. The function can be made as a remote task by assigning a remote decorator over the definition of it. As shown in the below code syntax user can also specify required resources for a particular function call.

```
@ray.remote(num_cpus=4)
def my_function():
    return 1
```

These remote tasks are stored as objects in memory storage and referred to as futures. The results after execution are retrieved from these futures by using `ray.get(future)`. Figure 5.3 shows the functionality where a compute-intensive function parallelized based on the available cores within a CPU environment. Such capability can be enhanced over multi-nodes if used within a cluster environment. Ray's API designed for simplicity and generality where its system architecture is designed for performance and scalability. As shown in Figure 5.3, the functionality

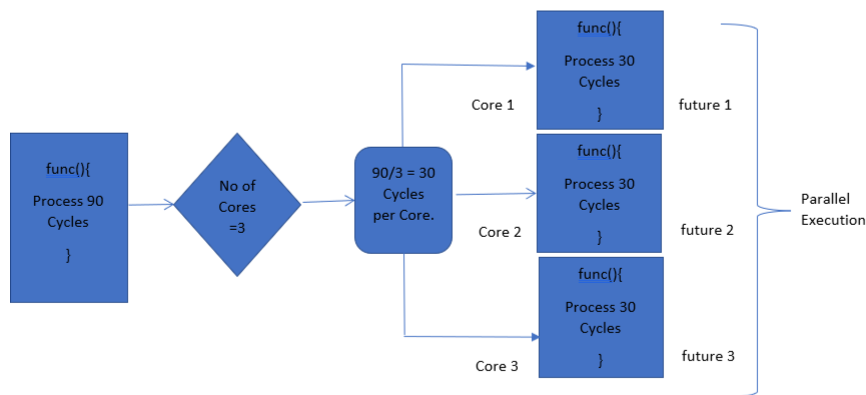

Figure 5.3 *Distribution based on cores*

to divide execution over code needs just a few lines of change of code. Ray manages task distribution and coordination under the hood. Ray is flexible when it comes to the heterogeneity of computations. For instance, a complex simulation decomposed into several tasks or steps. Tasks can be related in a way that subsequent tasks may depend on the outcome of an upstream task, Ray as a framework allows for dynamic execution that deals well with task dependencies. Flexible resource usage is also mandatory in task execution, as some tasks might require a GPU while other tasks can simply run on a CPU. Ray provides the flexibility to use the type by mentioning it in the remote call.

As discussed, a Ray cluster consists of several worker nodes, each worker node consists of several worker processes or simply workers. Each worker has a unique ID, an IP (Internet Protocol) address, and a port by which they can be referenced. To manage the allocation of tasks, and resources, and facilitate coordination between them. Ray has a component known as Raylet integrated with workers. These are the smart components that manage the worker processes. Raylets are shared between jobs and consist of two components namely a task scheduler and an object store. Figure 5.4 shows the basic structure where the object store takes care of memory management and ultimately makes sure workers have access to the data they need. The second component i.e scheduler takes care of resource management, among other things. For instance, if a task requires 4 CPUs, the scheduler needs to make sure it can find a free worker process that can grant access to said resources.

Coming to robustness, in distributed system there is a potential chance for things to go wrong in a way that machine might have an outage, abort a task or go up in flames. Ray has the capacity of fault tolerance when worker dies unexpectedly, the Ray will rerun the task until either the task succeeds or minimum tries exceeded. In overall execution, Ray is quite responsive system with server getting created quickly and remote functions parallelized the instant objects are created.

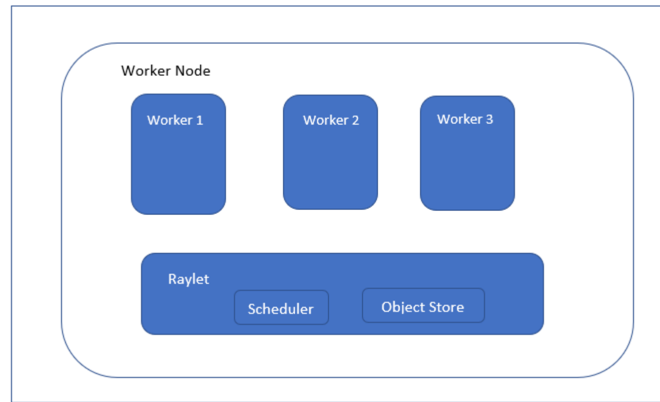

***Figure* 5.4** *System Components of Ray Worker Node*

5.1.2 Apache Spark:

Apache Spark identified as a unified analytics engine for large-scale data processing. By being unified it combines data processing with AI technologies. Originally written in Scala, it provides the Python functionality in the form of high-level API (Application Programming Interface). For instance, pandas API on Spark for pandas workloads. Fundamentally Spark was built using the concept of RDD's (Resilient Distributed Database), which is immutable distributed collection of objects. Each data-set in an RDD divided into logical partitions, which computes over different nodes of cluster. It supports in-memory processing computation i.e it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Figure 5.5, shows the iterative operations in Spark RDD, where it stores intermediate results in a distributed memory instead of stable storage and make the system faster. Here MR1, MR2, MR3 represents parallel process running over the chunk of data, and after two such iterations final result is stored under Stable storage. As data is not transferred over clusters, this increase the overall rate of execution, User can define the number of iterations as per the need. Python im-

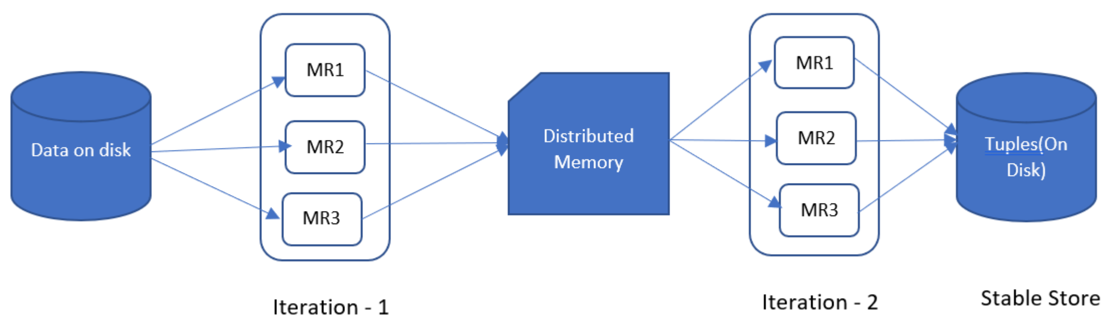

***Figure* 5.5** *Parallelization using RDD(Resilient Distributed Database)*

plementation of Apache Spark is provided by Pyspark. It allows to write Spark applications using python APIs and provides a Pyspark shell for interactively analyzing data in a distributed environment. An entry point to any Spark application is to create a Spark session. Spark session provides a way to interact with various Spark functionality such as RDDs, data frames, and datasets with a lesser number of constructs. In Pyspark, such a session can be created by following a code snippet.

```
import pyspark
from pyspark.sql import SparkSession
spark=SparkSession.builder.master("local[1]").appName("Test").
      getOrCreate()
```

Using the above code snippet a Spark-Shell is initialized and a dashboard with UI (User Interface) is created to view all submitted Spark jobs and their status. Apache Spark also provides quite a good cluster support in a way that Spark applications can run as an independent set of processes on a cluster. All of them are coordinated by SparkContext in the main program. To run on a cluster Spark connects to the cluster manager. As shown in Figure 5.6, once connected Spark acquires executors on nodes, which act as a process to run computation and store data of the application. Spark Context sends tasks to the executors to run. Cluster managers can act as an external service for acquiring resources in the cluster. The driver program listen to incoming connections from executors throughout its lifetime. Coming to dynamic scalability,

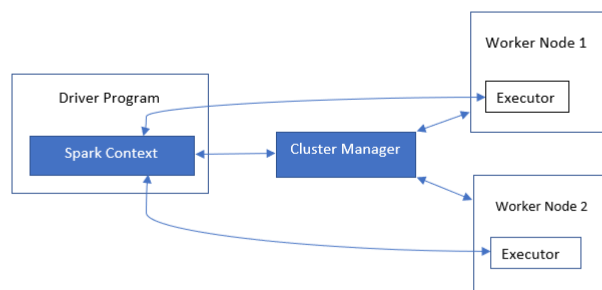

Figure 5.6 *Spark Cluster*

Spark gives control over resource allocation both across applications and within applications. Spark provides a mechanism to dynamically adjust the resources the application occupies based on workload, wherein the application can give resources back to the cluster if they are no longer used and request them again later when there is demand. Such a feature is useful when multiple applications share resources in a Spark cluster.

In terms of ease of use, code needs to be rewritten according to rules and standards defined by Spark, hence Spark functionalities cannot be added directly into

the application code. To provide usability, Apache Spark provides the abstraction library known as Pyspark. Pyspark provides a custom library wrapped around basic python libraries to scale the processing. For instance, Pyspark Dataframe API implemented on top of RDDs, and as explained before it is possible to parallelize operations using RDDs. A code snippet showing the creation of the Pyspark data frame is shown below.

```
import pandas as pd
from pyspark.sql import Row
pandas_df = pd.DataFrame({
        'a': [1,2,3],
        'b': [2.,3.,4.],
        'c': ['string1', 'string2', 'string3'],
        'd': [date(2000, 1, 1), date(2000, 2, 1), date(2000, 3, 1)],
        '3':[datetime(2000, 1, 1, 12, 0), datetime(2000, 1, 2, 12, 0),
        datetime(2000, 1, 3, 12, 0)] })
df = spark.createDataFrame(pandas_df)
```

In this way, a distributed data frame is set-up, which is executed over executors. `df. collect()` operation helps to collect distributed data to the driver side as local data in python. Hence using Pyspark with the available code is not very straightforward and trivial and a significant amount of code-rewriting is required.

In terms of robustness, Apache Spark uses the concept of lineage graphs. When a series of transformations is performed on an RDD, they are not evaluated immediately but lazily. When a new RDD is created from an existing RDD, the new RDD contains a pointer to the parent RDD. Similarly, all dependencies between RDDs are logged in a graph as metadata rather than actual data. This graph is called a lineage graph. Consider the following steps as mentioned below:

- Create a new RDD from a text file.
- Apply map operation on first RDD to get second RDD.
- Apply filter operation on second RDD to get third RDD.
- Apply count operation on third RDD to get fourth RDD.

Figure 5.7, shows the lineage graph of these operations:

***Figure* 5.7** *Lineage Graph*

This lineage graph is useful in case any of the partitions are lost. Spark can replay the transformation on that partition using the lineage graph existing in DAG (Directed Acyclic Graph) to achieve the same computation, rather than replicating the data across different nodes as in HDFS (Hadoop distributed file system).

5.2 Parallelization using Ray:

5.2.1 Parallelized File Loading:

In terms of Performance Management Data, there is a possibility that several performance metric files are available to process at the same time. Each file represents 24 hours of data as stated under Research Methodology. Hence, to make input pipeline fast parallel loading of these files into the application is essential for optimization. This section shows, how this functionality can be acheived using Ray.

Figure 5.8 shows a function where `LoadXLSX()` is designed that takes all the filenames as an input list and transfer to function `calculatingFuturesExcelLoad()` which loops to all the filenames and execute parallel loading by calling the function `loadExcel()`. For instance, if there are three files available to load and three scores in the system. Each file can be loaded in parallel as shown in Figure 5.8 as parallel futures and the result can be aggregated in the output data frame.

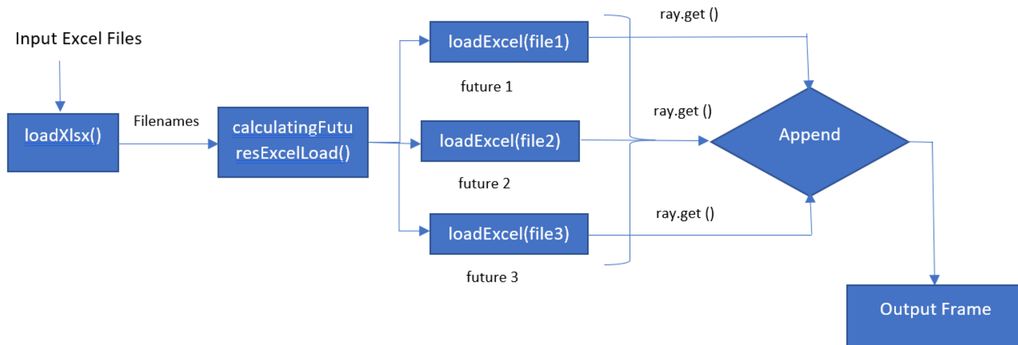

Figure 5.8 *Parallel File loading*

5.3 Correlation Calculation:

Correlation calculation between network elements is essential to establishing relationships. As part of a prepossessing step, possible combinations of two network cells within a group are identified from the available list as shown in Figure 5.9. Within one day itself, there are performance metrics of one network cell on various timestamps. Hence, to simplify the calculation for the correlation they were grouped using a `groupby()` function so that unique LNCELL values are present together.

In order to parallelize the correlation calculation, all the combinations of LNCELLS are now divided based on the number of cores present in the system, As shown in Figure 5.10, for instance of there are 900 combinations of LNCELLS and 3 cores are present in the system, than correlation between 300 combinations per core will be calculated in parallel. Inside `ComputeCorrelation()` a for loop is traversed over the supplied combination list. In each iteration, values of each network cell within a combination is fetched i.e value of entire timestamps from a supplied group list, correlation is calculated between that particular combination based on several KPI's.

Combs	nodeFrom	nodeTo
('1-ALL0119457-071', '1-ALL0119457-072')	1-ALL0119457-071	1-ALL0119457-072
('1-ALL0119457-071', '1-ALL0119457-073')	1-ALL0119457-071	1-ALL0119457-073
('1-ALL0119457-071', '1-AOL0119779-071')	1-ALL0119457-071	1-AOL0119779-071
('1-ALL0119457-071', '1-AOL0119779-072')	1-ALL0119457-071	1-AOL0119779-072

Figure 5.9 *Groups*

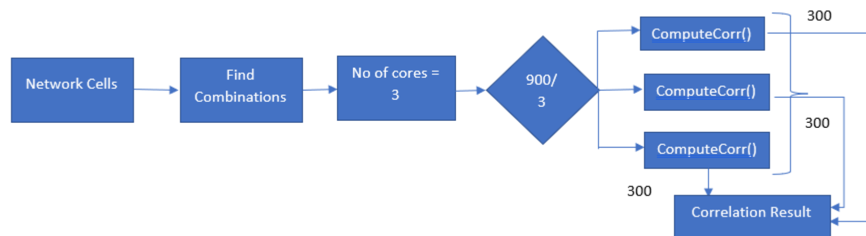

Figure 5.10 *Correlation Parallelization*

5.4 Knowledge Graph Interaction:

OrientDB is an open-source NoSQL graph database that was used to store and generate knowledge graphs. Pyorient is an open-source library identified as a medium to connect existing python code with the knowledge graph. Pyorient is composed of two layers. At the foundation, it is the python wrapper around the orientDB binary protocol. Built upon that is orientDB's SQL language called object graph mapper. A client can be created with the code shown below to connect existing python code with the knowledge graph.

```
import pyorient
client = pyorient.OrientDB(<<orient_db_server_url")
client.set_session_token(True)
session_id = client.connect("Username", "Password")
return client
```

It will be identified whether the database exists or not. If no, then a new database is created and a client is returned. Figure 5.11, shows the general flow of knowledge graph interactions in which first all the available network cells (nodes) are identified and inserted into the knowledge graph, post that all the relevant KPIs are identified and pushed within the knowledge graph. Edge creation or updation process initiated based on available nodes or combinations. Table 5.1 shows the time taken by

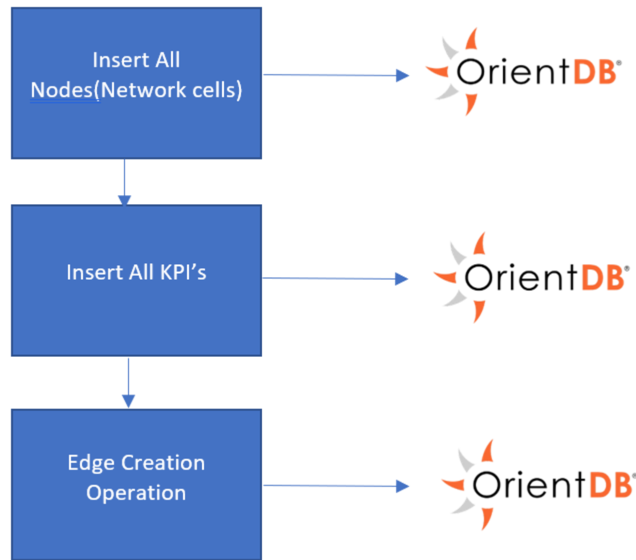

Figure 5.11 *Graph Operations*

different operations for loading 1 day of performance metric edge creation operation was identified as a major bottleneck , hence need to be optimized.

Operation	Time Taken	Decision
Generating Nodes	0:00:00.470364	No Optimization needed
Generating KPI's	0:01:08.276279	No Optimization needed
Edge Creation	1-2 Days	Optimization needed

Table 5.1 *Knowledge graph operation times*

A `created()` function was designed to take care of general knowledge graph interactions. It loops over the number of combinations present and executes two operations either create an edge if a new correlation value is identified or update the edge if there is a change in the previous value more than the previously defined threshold. The queries used to perform such operations are:

- **To add a node:** ”insert into Cells set Name = ’Node Name’”
- **To add a edge:** ”create edge ’KPI Name’ from(select from Cells where Name=’Node Name’) to (select from Cells where Name = ’Node Name’) content value: ’Value to insert’”

Hence, to increase the speed of execution `createEdge()` function can be parallelized. Parallelization performed in Ray by making `createEdge()` as a remote function. As shown in Figure 5.12, available network cell combinations were divided based on the available cores. A separate client connection initiated by each remote function. As orient DB does not allow multiple client connections at same time, a 20-second gap scheduled between each client connection.

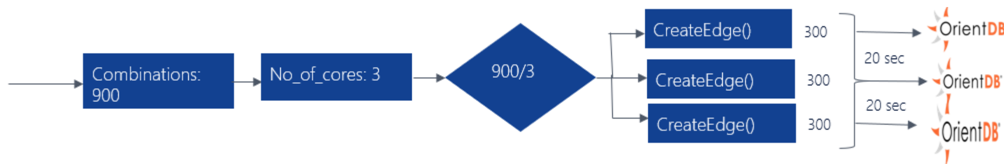

Figure 5.12 *Edge Creation*

5.5 Parallelization using Apache Spark:

5.5.1 Parallelized File Loading:

As data is available in form of an excel file, Apache Spark doesn’t support parallel loading of such files directly, hence parallel file loading was not possible within it respectively.

5.5.2 Correlation Calculation:

As existing code was not directly parallelizable using Apache Spark hence an implementation using Pyspark was done, and the code rewritten accordingly. Pyspark has its own set of native libraries to do correlation calculations, hence the data needs to be fed in a particular format. To do that separate data frame for each combination set and each KPI set is created as shown below. Figure 5.13, shows a data frame for Cell C1 and C2 concerning KPI 1. The index represents different timestamps of a day. Such a step needs to be repeated for all available parameters. After calculating each data frame it is important to convert them into a vectorized format using a vector assembler, as the correlation calculation function takes input into the vectorized format. Figure 5.14 shows the whole sequence of the process explained above.

Index	KPI 1 (C1)	KPI 1(C2)
0	100.0	67
1	107	71
2	103	69
3	108	91
4	111	98
...	...	...

Figure 5.13 *Dataframe*

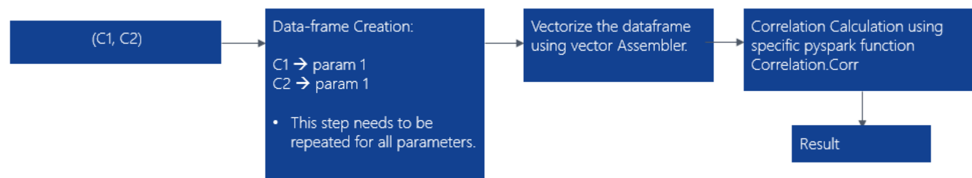

Figure 5.14 *Dataframe*

5.5.3 Knowledge Graph Interaction:

As code is rewritten into the Pyspark format, it is not possible directly to connect python with orientDB using Pyorient as was in the case of Ray. Hence a specific driver is needed to connect Pyspark with orientDB. Pyspark does not support any such driver to provide integration with knowledge graph, hence such implementation is not possible using Apache Spark.

6 Evaluation

In this section, we will present and discuss results obtained after implementing parallel processing frameworks to answer the established goal questions. The experimentation achieved enough insights that may lead to a concrete comparison between frameworks and evaluate the best to solve established problems.

General Evaluation and ease of use: General evaluation of the framework involved examining multiple aspects of the framework like their scalability, availability, programming libraries support, responsiveness, dynamic scalability, cluster support, handling of machine failures, and preemption. The below table shows the comparison of these aspects between Ray and Apache Spark.

Measure	Ray	Apache Spark
Responsiveness	Responsive	Responsive
Dynamic Scalability	Possible	Possible
Cluster Support	Possible	Possible
Machine Failures	Fault tolerance capacity	Possible through lineage graphs

Table 6.1 *Comparison*

From the comparison table above, it is noted that Ray and Apache Spark quite equates each other in terms of responsiveness as both can process parallelization instantly. In terms of dynamic scalability and cluster support, Ray is better in terms of usability as local code can be deployed over a cluster directly with few changes where proper YARN(Yet Another Resource Negotiator) configuration setup and code-rewriting required for Apache Spark. In handling machine failures and preemption both provide good support where Ray has a capacity for fault tolerance and Apache Spark has the concept of lineage graph. In terms of availability, both are open-source. In terms of programmability, both support python where code rewriting is required in Apache Spark hence Ray is better in terms of usability here. Quality of Service as a metric was identified as stated in the research methodology section for comparison which includes responsiveness, dynamic scalability, handling of machine failures, and preemption and cluster support. Both platforms are equal in all aspects but Ray performs better in terms of usability for dynamic scalability. In general, Ray is better for the use case in terms of programmability.

Parallelized file loading: As stated in the solution section, Ray provides the capability to load relevant excel files in parallel, by making `LoadExcel()` as remote and executing it in parallel whereas Apache Spark does not provide specific libraries to load excel files in parallel. Hence, Ray founds to be better here.

Parallelized Correlation Calculation: As discussed in the Research Methodology section, throughout obtained by application of framework based on the growth of problem size will be compared to evaluate the best framework. Table 6.2 shows the throughput values obtained by normal execution or execution of Ray over a set of files for correlation calculation.

No of files	Results(Normal)	Results(Ray)
1	0:11:22	0:02:50
2	0:18:19	0:03:39
3	0:24:21	0:04:30
4	0:29:37	0:05:24
5	0:36:54	0:07:25
6	0:41:21	0:08:37
7	0:47:87	0:09:06
8	0:53:22	0:10:10
9	0:59:09	0:13:33
10	1:7:10	0:13:35

Table 6.2 *Throughput(Normal v/s Ray*

As shown in Figure 6.1, based on the growth of problem size Ray showed a pretty good progress compared to normal sequential execution. Though throughput decreases on growth of problem size, the performance is better compared to sequential execution. Table 6.3 shows the throughput values obtained by execution

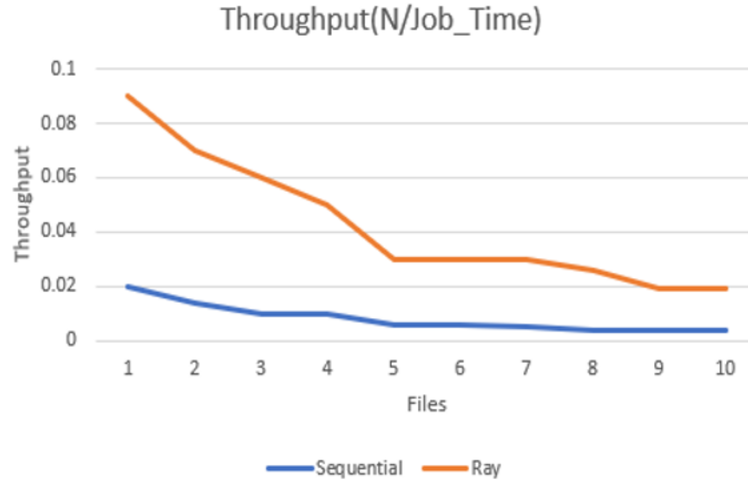

Figure 6.1 *Performance plot Ray v/s Sequential*

of Apache Spark over a set of files as compared to normal execution. As shown in Figure 6.2, the performance over throughput obtained using Apache Spark is much less than normal execution. This is because there is a lot of overhead involved to convert input data into a respective data frame that can be processed by Apache

No of files	Results(Normal)	Results(Ray)
1	0:11:22	0:18:11
2	0:18:19	0:24:02
3	0:24:21	0:31:22
4	0:29:37	0:38:11
5	0:36:54	0:45:33
6	0:41:21	0:55:21
7	0:47:87	1:05:22
8	0:53:22	0:15:12
9	0:59:09	1:25:14
10	1:7:10	1:36:22

Table 6.3 *Throughput(Normal v/s Spark)*

Spark libraries for calculating correlation as described under the solution section. From the analysis done above, it can be concluded that Spark shows no performance

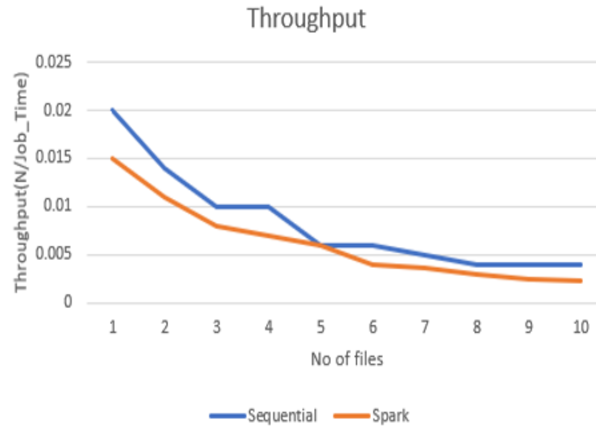

Figure 6.2 *Throughput obtained based on growth of files*

improvement compared to normal execution, where Ray contributed significantly to increasing the speed and getting results fast.

Parallelized Edge Creation: As described under the solution section, parallel edge creation is possible using Ray by parallelizing `CreateEdge()` function which created multiple parallel clients on the gap of 20 seconds. Table 6.4 compares time taken to do edge creation between normal execution and Ray based on no of edges. Figure 6.3 shows throughput comparison between the two. As can be seen from both demonstrations Ray performs significantly well in parallelizing edge creation when the number of edges is low where as number of edges increases performance of Ray becomes comparable to the normal execution. This degradation in performance is possible due to the sub-optimal behaviour of orientDB. .

Coming to implementation using Apache Spark, orientDB doesn't provide any driver support to connect with Pyspark hence as mentioned in solution section such

No of Edges	Results(Normal)	Results(Ray)
300	0:11:08	0:18:11
600	00:26:17	0:24:02
900	0:34:78	0:31:22
1200	0:44:46	0:38:11
1500	0:51:58	0:45:33

Table 6.4 *Edge Creation(Normal v/s Ray)*

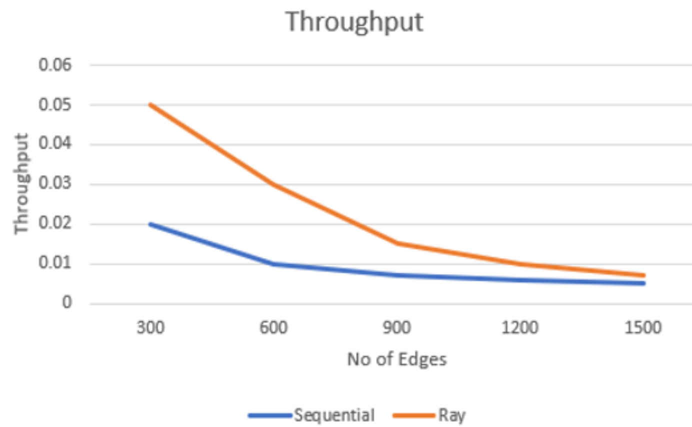

Figure 6.3 *Throughput Ray Edge Creation*

implementation is not possible. From the analysis done above it can be evaluated that Ray performs better than sequential for edge creation but only for limited number of edges and Apache Spark cannot be used.

7 Conclusions

In this thesis, we investigated application of the parallel processing frameworks Ray and Apache Spark to parallelize the correlation calculation between network elements and interaction with knowledge graph.

General Evaluation with ease of use. Based on the comparison done in the evaluation section it can be concluded that Ray and Spark quite equates each other in terms of Quality of service i.e Responsiveness, Dynamic Scalability, Cluster Support and Machine Failures. Ray performs relatively well in terms of usability due to minimal code changes needed to scale. Hence, Ray is a better alternative for achieving the goals established under this section.

Parallelized Correlation Calculation. Efficiency of Framework is evaluated based on the growth of problem size. Hence, in terms of throughout discussed under Evaluation section, Ray performs relatively well where it is able to optimize correlation calculation by 70-80 % compared to sequential execution. Apache Spark relatively under performs due to the need to convert existing code in specific format to be used by Pyspark, which brings an extra overhead to overall execution time. Hence, Ray is the better alternative to achieve goals established under this section.

Parallelized Edge Creation. Parallel Edge Creation and updates is possible using Ray as a parallel processing framework, where in terms of growth of problem size it performs relatively well for fewer files but as data chunk increases the performance becomes relatively equal with sequential execution. Spark cannot be used because of non-availability of drivers to connect with orientDB. Hence, Ray performs relatively well under this section for fewer chunks of data.

From the discussion done above, it can be concluded that Ray performs relatively well for the use-case established under this thesis. Hence the alternative hypothesis established for this work holds. Future work might consider the test with different database such as neo4j instead of orientDB. As orientDB identified as an sup-optimal database, it is vital to test if Ray efficiency be improved to interact with knowledge graphs based on a optimal database.

8 Abbreviation

Abbreviation	Meaning
NE	Network Element
KPI	Key Performance Indicator
CM	Configuration Management
PM	Performance Management
RAM	Random Access Memory
RDD	Resilient Distributed Database
API	Application Programming Interface
YARN	Yet Another resource negotiator
HDFS	Hadoop Distributed File System

References

Asgari, Marjan et al. (2022). "A review of parallel computing applications in calibrating watershed hydrologic models". In: *Environmental Modelling Software* 151, p. 105370. ISSN: 1364-8152. DOI: https://doi.org/10.1016/j.envsoft.2022.105370. URL: https://www.sciencedirect.com/science/article/pii/S1364815222000767.

Husain, Akhlaq et al. (2022). "Fractal dimension of India using multicore parallel processing". In: *Computers Geosciences* 159, p. 104989. ISSN: 0098-3004. DOI: https://doi.org/10.1016/j.cageo.2021.104989. URL: https://www.sciencedirect.com/science/article/pii/S0098300421002739.

Jogalekar, P and M Woodside (2000). "Evaluating the scalability of distributed systems". eng. In: *IEEE transactions on parallel and distributed systems* 11.6, pp. 589–603. ISSN: 1045-9219.

— (2000). "Evaluating the scalability of distributed systems". In: *IEEE Transactions on Parallel and Distributed Systems* 11.6, pp. 589–603. DOI: 10.1109/71.862209.

Joubert, Wayne et al. (2019). "Parallel accelerated Custom Correlation Coefficient calculations for genomics applications". In: *Parallel Computing* 84, pp. 15–23. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2019.02.003. URL: https://www.sciencedirect.com/science/article/pii/S0167819118301431.

Martinez-Mosquera, Diana, Rosa Navarrete, and Sergio Luján-Mora (2020). "Development and Evaluation of a Big Data Framework for Performance Management in Mobile Networks". In: *IEEE Access* 8, pp. 226380–226396. DOI: 10.1109/ACCESS.2020.3045175.

Misra, Durga Prasanna et al. (2021). "Formulating Hypotheses for Different Study Designs". eng. In: *Journal of Korean medical science* 36.50, e338–e338. ISSN: 1011-8934.

Moritz, Philipp et al. (2017). *Ray: A Distributed Framework for Emerging AI Applications*. DOI: 10.48550/ARXIV.1712.05889. URL: https://arxiv.org/abs/1712.05889.

Mu, Yashuang, Xiaodong Liu, and Lidong Wang (2018). "A Pearson's correlation coefficient based decision tree and its parallel implementation". In: *Information Sciences* 435, pp. 40–58. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2017.12.059. URL: https://www.sciencedirect.com/science/article/pii/S0020025517311763.

Rocklin, Matthew (Jan. 2015). "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: pp. 126–132. DOI: 10.25080/Majora-7b98e3ed-013.

Al-Saqqa, Samar, Ghazi Al-Naymat, and Arafat Awajan (2018). “A Large-Scale Sentiment Data Classification for Online Reviews Under Apache Spark”. In: *Procedia Computer Science* 141. The 9th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2018) / The 8th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2018) / Affiliated Workshops, pp. 183–189. ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2018.10.166`. URL: `https://www.sciencedirect.com/science/article/pii/S1877050918318167`.

V., M., Ch Venkata, and B. Preethi (2017). “A Comparative Study on the Implementation of Matrix Addition in Sequential and Parallel Computing Paradigms”. eng. In: *International journal of computer applications* 157.4, pp. 32–36. ISSN: 0975-8887.

Xu, Yinan, Hui Liu, and Zhihao Long (2020). “A distributed computing framework for wind speed big data forecasting on Apache Spark”. In: *Sustainable Energy Technologies and Assessments* 37, p. 100582. ISSN: 2213-1388. DOI: `https://doi.org/10.1016/j.seta.2019.100582`. URL: `https://www.sciencedirect.com/science/article/pii/S2213138819306551`.

Yang, Junrong et al. (2020). “SIFT-aided path-independent digital image correlation accelerated by parallel computing”. In: *Optics and Lasers in Engineering* 127, p. 105964. ISSN: 0143-8166. DOI: `https://doi.org/10.1016/j.optlaseng.2019.105964`. URL: `https://www.sciencedirect.com/science/article/pii/S0143816619312217`.

Zaharia, Matei et al. (Oct. 2016). “Apache Spark: A Unified Engine for Big Data Processing”. In: *Commun. ACM* 59.11, pp. 56–65. ISSN: 0001-0782. DOI: `10.1145/2934664`. URL: `https://doi.org/10.1145/2934664`.

Zhang, X.D., Y. Yan, and K.Q. He (1994). “Latency Metric: An Experimental Method for Measuring and Evaluating Parallel Program and Architecture Scalability”. In: *Journal of Parallel and Distributed Computing* 22.3, pp. 392–410. ISSN: 0743-7315. DOI: `https://doi.org/10.1006/jpdc.1994.1100`. URL: `https://www.sciencedirect.com/science/article/pii/S0743731584711002`.