

A Parallel Implementation of the Pandas Framework

by
Saba Hafeez Khan

A thesis submitted to the Department of Computer Science,
College of Natural Sciences and Mathematics
in partial fulfillment of the requirements for the degree of

Master of Science

in Computer Science

Chair of Committee: Edgar Gabriel

Committee Member: Thamar Solorio

Committee Member: Peggy Lindner

University of Houston
May 2020

Copyright 2020, Saba Hafeez Khan

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor, Dr. Edgar Gabriel, for introducing me to this project idea. His valuable advice, help, and guidance through every stage of the project was instrumental in defining my path for this research and played an enormous role in the successful completion of this project. I am extremely grateful to Dr. Peggy Lindner for enriching my study by providing a real-world dataset from her work.

I would also like to thank Dr. Tamar Solorio for extending her help in completing my thesis. I extend my gratitude to Dr. Venkat Subramaniam, whose interactive and rigorous feedback during the Software Design course was instrumental in teaching me good design principles. I would like to recognize the assistance provided by John Rodgers, his willingness to help and provide practical suggestions is much appreciated. My grateful thanks are also extended to Alamzeb Khan for his guidance with regards to analysis of my results.

Finally, I wish to acknowledge the support and great love of all my family members. I am especially thankful to my parents and my son; the completion of my master's degree would not have been possible without their patience and support.

ABSTRACT

High-performance is a highly desirable trait for applications today. Companies large and small are migrating their serial applications to parallel versions to reduce execution time and increase efficiency. However, preparing serial applications for parallel processing is not a simple process. Pandas, which is a Python library containing rich data structures and tools, is used abundantly in Data Science applications. However, the Pandas framework is built for single-core processing and is unable to fully utilize multi-core processors or cluster technology. Because of this limitation, Pandas users are forced to look for other frameworks when working with large quantities of data.

This thesis introduces a Parallel-Pandas library which makes the process of parallelizing serial Pandas applications easy and transparent. The Parallel-Pandas library provides Pandas users the ability to upgrade existing applications transparently, by using only a library import. This thesis contains details about the design decisions and implementation of the Parallel-Pandas library. The Parallel-Pandas library is evaluated with unit testing, microbenchmarks, and a real-world application with different datasets. Parallel-Pandas has also been compared with PySpark, a framework that provides parallelism by following the MapReduce structure. The results presented in this paper show that the Parallel-Pandas library has promising potential and delivers performance close to manually parallelized and tuned applications.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	iii
ABSTRACT	iv
TABLE OF CONTENTS.....	v
LIST OF TABLES	vii
LIST OF FIGURES	viii
1 INTRODUCTION.....	1
1.1 INTRODUCTION TO THE PROBLEM DOMAIN.....	1
1.2 MOTIVATION	2
1.3 GOALS OF THESIS	3
2 BACKGROUND	4
2.1 PYTHON	4
2.1.1 Use of Python for High-performance	4
2.2 MESSAGE PASSING INTERFACE.....	5
2.2.1 MPI for Python.....	5
2.3 NUMPY	6
2.4 PANDAS	7
2.5 NATURAL LANGUAGE TOOLKIT (NLTK).....	7
3 RELATED WORK	8
3.1 DASK	8
3.2 SPARK – PYSPARK	9
3.3 RAY	10
3.4 MODIN	10
4 CONTRIBUTIONS.....	12
4.1 THE PARALLEL-PANDAS LIBRARY	12
4.1.1 Data Distribution	12
4.1.2 Functions Implemented	14
4.1.3 Other Features Developed	15
4.2 IMPLEMENTATION DETAILS	15

4.2.1	Constructor of the Parallel Dataframe Subclass	16
4.2.2	‘from_dict’ Function	17
4.2.3	‘corr’ Function.....	19
4.2.4	‘drop’ Function.....	20
4.2.5	‘apply’ Function	22
4.2.6	‘div’ Function	23
4.2.7	Constructor of the Parallel Series Subclass	24
4.2.8	‘value_counts’ Function	25
4.2.9	‘collect’ Function	27
4.2.10	‘loc’ Property	27
4.2.11	Global Properties	29
5	EVALUATION.....	31
5.1	PERFORMANCE ANALYSIS METRICS	31
5.2	HARDWARE AND SOFTWARE CONFIGURATION	32
5.3	MICROBENCHMARKS	32
5.4	USE CASE – DUPLICATE DETECTION	38
5.4.1	Datasets	39
5.4.2	Algorithm	39
5.4.3	Application I/O (Input/Output)	42
5.4.4	Different Applications Developed and Compared	43
5.4.5	Results and Discussion	51
6	CONCLUSIONS	62
7	FUTURE WORK	64
8	BIBLIOGRAPHY	66

LIST OF TABLES

Table 5-1: Execution time in seconds for the Serial-Pandas application using the book dataset.....	52
Table 5-2: Performance statistics for the Parallel-Pandas application using the book dataset.....	53
Table 5-3: Execution time in seconds for the MPI-Pandas application using the book dataset.....	54
Table 5-4: Execution time in seconds for the PySpark application using the book dataset.....	55
Table 5-5: Execution time in seconds for the Serial-Pandas application using the news articles dataset	57
Table 5-6: Performance statistics for the Parallel-Pandas application using the news articles dataset	57
Table 5-7: Execution time in seconds for the MPI-Pandas application using the news articles dataset	59
Table 5-8: Execution time in seconds for the PySpark application using the news articles dataset	59

LIST OF FIGURES

Figure 4-1: Depiction of a Pandas dataframe and a Parallel-Pandas distributed dataframe with ‘columns’ orientation	13
Figure 4-2: Function signature of the parallel dataframe constructor	16
Figure 4-3: Function signature of the 'from_dict' method	18
Figure 4-4: Function signature of the 'corr' method	20
Figure 4-5: Function signature of the 'drop' function	20
Figure 4-6: Function signature of the ‘apply’ function	22
Figure 4-7: Function signature of the 'div' function	23
Figure 4-8: Function signature for the constructor of the parallel series	24
Figure 4-9: Function signature for the 'value_counts' function	26
Figure 5-1: Microbenchmark results for 'apply' function	34
Figure 5-2: Microbenchmark results for 'from_dict' function	35
Figure 5-3: Microbenchmark results for 'value_counts' function	36
Figure 5-4: Microbenchmark results for 'corr' function	38
Figure 5-5: Steps of the duplicate detection algorithm, the steps that are contained in boxes with dotted outlines can be done in parallel	41
Figure 5-6: Initial column-wise data distribution of the dataset in the Parallel-Pandas application	44
Figure 5-7: A subset of the distributed inverted index in the Parallel-Pandas application	45
Figure 5-8: A subset of the distributed similarity matrix in the Parallel-Pandas application	46
Figure 5-9: Initial dataframe distribution (row-wise) in the MPI-Pandas application	49
Figure 5-10: Column-wise data distribution in the MPI-Pandas application	50
Figure 5-11: Speed-up of the Parallel-Pandas duplicate detection application while using the Gutenberg dataset	54
Figure 5-12: Performance comparison of duplicate detection application for the dataset containing 519 Gutenberg books	56
Figure 5-13: Speed-up of the Parallel-Pandas application, while using the news articles dataset	58

Figure 5-14: Performance comparison of duplicate detection application for the dataset containing 18,698 news articles	60
Figure 5-15: Closer look - Performance comparison of duplicate detection application for the dataset containing 18,698 news articles.....	61

1 INTRODUCTION

1.1 INTRODUCTION TO THE PROBLEM DOMAIN

As we have been moving into a technologically advanced world, two areas of scientific computing, High-Performance Computing (HPC) and Big Data have emerged out of different necessities. The HPC software stack comes from the need of being able to use the large-scale compute resources efficiently whereas the Big Data Software stack comes from the need of handling large scale data efficiently. HPC programming models such as Message Passing Interface (MPI) [14], OpenMP [19] and CUDA [18] are known to require more learning and can be considered expert territory whereas Big Data program models such as Hadoop MapReduce [7], Spark [27], and Pandas [13] are considered simpler but may have limitations when it comes to full utilization of the hardware. Attempts are being made to bring these two programming models closer to each other so that maximum applications can benefit from the availability of the high-end hardware resources.

From the programming language standpoint, the area of High-Performance Computing has been using the more traditional statically typed languages of C/C++ and Fortran which are known to make better use of the hardware. Python, which is a dynamically typed language, is gaining more and more popularity mainly because of its easy and clear syntax, quick learning time, and the availability of the immense libraries and modules, because of which development can move rapidly. This language is very popular in the Big Data community as well, since it has libraries such as NumPy [17], and Pandas to aid scientific computing.

Hence, a new trend of using the HPC technologies such as MPI with Python, which is more common in Big Data applications, is gaining traction. This trend attempts to bring the HPC technologies within easy reach of the Big Data applications.

1.2 MOTIVATION

High-performance is a highly desirable trait for almost all applications nowadays. Companies large and small are trying to migrate their serial code to parallel versions to increase efficiency. However, it is not always easy to update a serial application to a fast, parallel version. One must fully understand the underlying hardware resources that are available and the unique requirements of the application before taking on the task of migrating an application. Pandas is a library that is heavily used in data science for handling different types and sizes of data. It is liked for its ease-of-use, flexibility and the rich data manipulation functions that it provides. However, Pandas cannot run efficiently on multi-core hardware, let alone clusters.

This project aims at making the task of migrating serial applications to parallel versions simpler, for the users of the Pandas framework. In this project, we begin at building the framework that will improve performance of Pandas applications and parallelize the application. It will provide transparency and can be used just with a library import, without needing any code-changes, and without understanding all the details of what is happening under-the-hood.

The aim of this project is to combine the technologies of MPI, Pandas and Python. MPI has great performance and has the ability to use cluster-computing efficiently, Pandas is widely used for big-data analytics but is based on sequential processing, and Python is an easy to use language. Hence this project will make parallel and efficient processing more accessible to a variety of different users.

1.3 GOALS OF THESIS

The goal of this project is to create a parallel version of Pandas (Python Data Analysis Library) functions using MPI (Message Passing Interface) technology. The aim is to create an interface identical to the serial Pandas to create a seamless transition, so that upgrading existing applications to the parallel version is just a matter of importing the Parallel-Pandas library. Currently, such a library does not exist.

As a use case, and for narrowing down the Parallel-Pandas library functions that are developed during the thesis, optimization of the speed and performance of detecting duplicate documents will be focused upon.

This thesis begins by introducing the application domain and the various technologies of Python, NumPy, Pandas, and MPI to familiarize the reader. This is followed by a discussion of related work. After that, implementation, and experimental setup will be discussed in detail. The thesis will be concluded with an evaluation of the contribution and ideas for future work.

2 BACKGROUND

This section will introduce the reader to the technologies that have been used during this research.

2.1 PYTHON

Python [22] is a portable, interpreted, object-oriented language which has a very clear syntax. It is a high-level general-purpose language, which can be extended with functions implemented in C/C++, and it can also be used as an extension language to provide programmable interfaces to applications. It is a powerful language with a community of users that is constantly increasing.

2.1.1 Use of Python for High-performance

Traditionally HPC has been dominated by compiled, low-level languages such as C/C++, since it is thought that compiled languages are able to use the hardware more efficiently. However, for most of the HPC applications, only a small portion of the code is time critical enough to need compiled languages, the rest of the code is input/output, memory management, error handling etc. where interpreted high-level languages can swoop in to speed up the development and debugging processes [4]. Hence, Python which is an interpreted language has been gaining popularity for being used in high-performance applications. The proponents of Python advertise its readability, succinctness, and the ability of writing computationally expensive parts of code in compiled languages and accessing them through Python modules [25].

2.2 MESSAGE PASSING INTERFACE

Message passing has proved to be very effective in parallel computing and distributed systems. Message Passing Interface (MPI) [14] is a language independent communications API (Application Programming Interface). It is the specification produced by the MPI Forum to standardize message passing in cluster computing. MPI provides a portable message passing system, hides the networking and memory management details to ease development and does not sacrifice performance in the process. The standard does not provide an implementation but defines syntax and semantics. In addition to point-to-point blocking and non-blocking send and receive operations MPI also standardizes collective operations such as ‘gatherAll’, ‘allReduce’ etc. It has an object-oriented feel and uses communicators to specify the communication domain. Most implementations of MPI support communication over InfiniBand (using native InfiniBand protocol) and TCP (Transmission Control Protocol) which is typically on Ethernet, as well as other network interconnects.

2.2.1 MPI for Python

Implementations of the MPI standard have traditionally been developed in scientific languages like C, C++ and Fortran, the two major ones are MPICH [9] and OpenMPI [2]. Due to the productivity gain that is associated with interpreted languages, attempts have been made to enable the use of Python with parallel computing frameworks such as MPI.

Bindings in Python have been developed so that Python applications can also benefit and make use of the parallel architectures. Mpi4py [4], which builds on MPI C-bindings, is a

notable project among the Python implementations. It is written in Cython, which is a C library wrapping language. It integrates with NumPy, uses the array class with minimal overhead, and has a clear and readable syntax. It can work with any of the underlying MPI implementation that is based on the MPI specifications.

Mpi4py not only provides implementation for communication by passing Python objects, but it also provides communication of memory buffers and means of defining a user defined MPI datatype. Using mpi4py in conjunction with memory buffer communication is as good as using compiled languages like C/C++, since it avoids the overhead created by the pickling/de-pickling of the objects that is required in Python object communication [5]. The functions for sending and receiving memory buffers are accessed by ‘Send’ and ‘Receive’ whereas the Python object functions are called by their lowercase ‘send’ and ‘receive’ counterparts.

2.3 NUMPY

NumPy [17] is a Python package created to aid scientific computing. NumPy provides support for handling large multi-dimensional arrays and matrices by providing powerful n-dimensional array structures and functions for dealing with these structures. It also provides tools for integrating C/C++ and Fortran code and contains functions for linear algebra, Fourier transforms and random numbers. NumPy’s array container can be used in conjunction with mpi4py’s buffer communication to get results comparable to the C/C++ implementations [5]. This makes Python more appealing to the performance critical applications as well.

2.4 PANDAS

Pandas [13] is a Python library containing rich data structures and tools. It has been created to improve the adoption of Python in the scientific programming community by bridging the gap and enriching Python for data analytics and data manipulation applications. Specifically, some of the features it provides are: label-based data access, advanced pivoting and reshaping, grouping and aggregating of data, combining and joining of datasets, hierarchical indexing, data alignment, and dealing with missing data etc.

Traditionally, Pandas works with data collections in a relatively sequential manner. The Pandas ‘read_table’ function now offers chunk size to read huge files in chunks to make the processing more memory manageable. When dealing with huge data collections, faster processing times can be achieved by parallelizing the process and making use of the many processors that are now available in laptops, consequently further performance boosts can be attained by using distributed systems and clusters [1].

2.5 NATURAL LANGUAGE TOOLKIT (NLTK)

NLTK [3] is a Python library that provides language processing tasks such as stop-word removal, stemming, tokenization, speech tagging, and text parsing etc. The aim of this library is to design an intuitive framework for providing natural language processing functionality so that the user does not have to worry about the details associated with the processing annotated language data. The NLTK package can be used by Python with the following two statements:

```
import nltk
nltk.download()
```


3 RELATED WORK

Some related work in this area is described in this section.

3.1 DASK

Currently, there is a Python framework, Dask [23], which provides a parallel implementation of some of the Pandas functions. From a user perspective it requires a setup of the application and is not transparent for the user coming from the serial version of the code. From an implementation perspective, it does not use MPI for internode communication. [6]

Dask is a framework that uses blocked algorithms and task scheduling to achieve parallel execution. It creates an acyclic graph of tasks for computation and represents it as a Dask graph, which is a Python dictionary. It has a scheduler and worker pattern and does lazy evaluation, which means that it waits to implement a task until it must. Because of this setup an extra ‘compute’ function needs to be called to force computation and hence converting codes from serial to parallel would require more than just a library import. Dask uses TCP (Transmission Control Protocol), which typically uses ethernet, for internode communication. It can be setup to use different interconnect, such as InfiniBand, in case of a specialized capability of a system. However, since it is TCP-based, it will be using IPoIB (Internet protocol over InfiniBand) which does not provide the same throughput and latency as using the native InfiniBand protocol. Dask defines an array structure, a bag structure and a dataframe structure. The dataframe structure is partitioned row-wise and distributed across the

cluster. The array structure uses Dask graphs to create a NumPy like structure but uses all the cores available in a system. [26]

3.2 SPARK – PYSPARK

Spark [27] is a cluster computing framework, written in Scala. Spark provides a general programming interface to be used interactively to process large datasets on clusters. It supports other programming languages such as Python, Java and R. PySpark is a programming interface enabling access to Spark from Python. The core of Spark is the implementation of Resilient Distributed Dataset (RDD) [28], which are a fault-tolerant, immutable, distributed memory abstraction. In addition to fault-tolerance and efficiency, Spark also addresses the limitations of the Hadoop MapReduce framework [7] in the area of iterative and interactive applications.

The users can perform course-grained transformations to create new RDDs and can perform actions such as ‘count’, ‘collect’ etc. on the RDDs. The user can also control the distribution of the RDDs and can persist them in memory. Fault-tolerance is provided by keeping the lineage of each of the RDD for the entire life of that RDD. Lineage is composed of the details of all the transformations that took place to create the RDD.

Spark runs in a distributed fashion with the combination of a driver and executor pattern. The driver splits the application into tasks and divides them across the worker executors. Some work has been done to assess the benefits of using Spark with high-speed interconnects such as InfiniBand [12], but this work has not been incorporated into the standard Spark

distribution. Hence, similar to Dask, if Spark is used with a system that has InfiniBand, it will use IPoIB and will not get all the performance benefits that can be gained by using the native InfiniBand protocol.

3.3 RAY

Ray [16] is an open-source system for scaling Python applications to clusters. It is a cluster-computing framework targeted towards AI (artificial intelligence) and ML (machine learning) applications. Hence, in addition to the coarse-grained transformations, Ray provides the ability to do fine-grained transformations which are lacking in MapReduce and Spark frameworks.

Similar to Dask, Ray implements the application as a graph of dependent tasks that evolves during execution. Ray provides two different forms of functions: a task, which is a function to be executed on a stateless worker, and an actor, which represents a stateful computation. Ray distributes two components that are currently centralized in other systems: the task scheduler and the metadata containing the lineage. It is not supposed to be a substitute to general-purpose frameworks such as Spark, since it lacks the rich functionality provided by such frameworks.

3.4 MODIN

Modin [20] is an early-stage lightweight open-source multi-process dataframe library prototype, which scales Pandas applications. It provides a scalable dataframe and provides

integration with current Pandas code only with a library import. Modin rewrites Pandas API calls into a sequence of algebraic operators.

It provides distributed processing of dataframes using two execution frameworks: Ray or Dask, the user gets to choose the underlying framework they wish to use. Modin provides distribution of the dataframes by partitioning across both columns and rows. Modin does not use MPI for communication and does not perform lazy evaluation like Spark and Dask. According to its documentation [15], the use of Modin in clusters is experimental and still under development.

4 CONTRIBUTIONS

The main contribution of Pandas is the dataframe object, which is a two-dimensional data structure with integrated indexing, created to make data manipulation fast and easy. The Pandas library also provides a series object, which is a one-dimensional labeled array, to complement the dataframe. This project begins the work of parallelizing the Pandas library, to further improve efficiency, and introduces the Parallel-Pandas Library. This library contains a `ParallelDataframe` object, and a complementing `ParallelSeries` object. In the following, the thesis describes the most relevant architectural features of the Parallel-Pandas library, as well as the functionality currently supported by the library.

4.1 THE PARALLEL-PANDAS LIBRARY

The following section will discuss the design decisions made while developing this library.

4.1.1 Data Distribution

A parallel dataframe or series can either be of type ‘distributed’ or ‘replicated’. ‘Distributed’ type means that data is uniformly distributed among the nodes of the cluster and each node has a portion of the dataframe or series whereas, ‘replicated’ type is when the same data is repeated across all the nodes.

A parallel distributed dataframe can be distributed column-wise or row-wise, represented by an ‘orient’ property, amongst the nodes in the cluster. Each node contains a local dataframe which represents a portion of the global distributed dataframe. `ParallelDataframe` contains an

attribute called ‘global_to_local’, calculated on an as-needed basis, which specifies the mapping of the columns or rows to the nodes (depending on the orientation of the distribution). The column labels and the index (Pandas name for row) labels in a Pandas dataframe are obtained with ‘df.columns’ and ‘df.index’ similarly the global column and index labels in the Parallel-Pandas distributed dataframe can be obtained by ‘df.globalColumns’ and ‘df.globalIndex’ respectively.

For column-wise distribution, requesting a row of the distributed dataframe gives a distributed series. Figure 4-1 depicts the Pandas dataframe and the Parallel-Pandas distributed dataframe, with column distribution and hence global column labels.

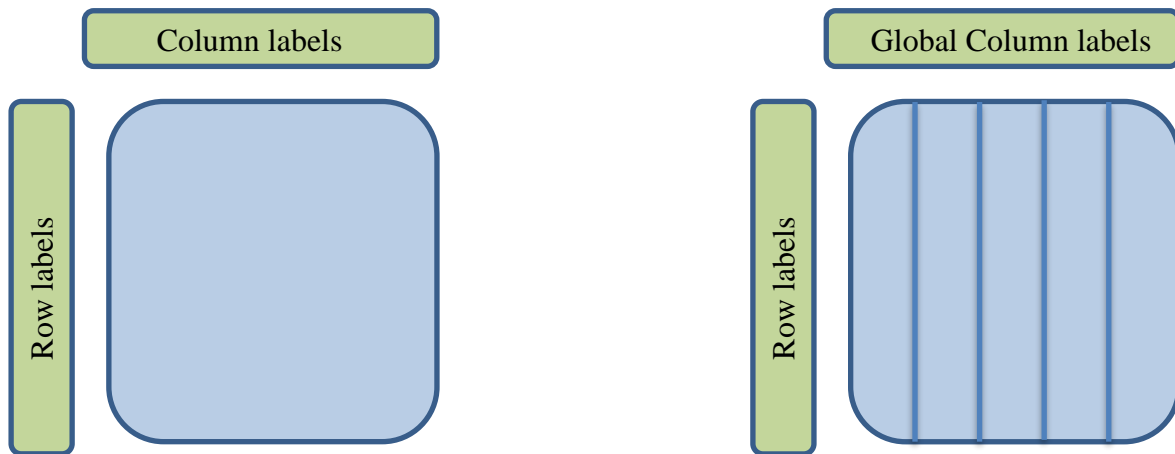


Figure 4-1: Depiction of a Pandas dataframe and a Parallel-Pandas distributed dataframe with ‘columns’ orientation

4.1.1.1 Limitations

Currently, the row-wise distribution is experimental and only supported by a few functions. A row-wise distributed dataframe can only be created via the ‘from_dict’ function with ‘index’

as the ‘orient’ argument. However, the setup is done in such a way that extending this functionality is an easy next step.

4.1.2 Functions Implemented

For the `ParallelDataframe` subclass, in addition to the constructor, the ‘from_dict’ function, the ‘corr’ function, ‘drop’ function, ‘apply’ function and the ‘div’ function have been implemented. For the parallel series subclass, in addition to the partial implementation of the constructor as discussed earlier, ‘value_counts’ function is implemented. Details on what these functions do are below:

- The constructor of each class is responsible for creating an object of that type.
- The ‘from_dict’ function creates a parallel dataframe from a dictionary.
- The ‘corr’ function computes pairwise correlation among the columns of the dataframe, excluding null values. It creates and returns a distributed dataframe when called by a distributed dataframe.
- The ‘drop’ function removes a row or column whose label is specified.
- The ‘apply’ function takes a function and applies it to all the data in the dataframe. It returns a parallel series or a parallel dataframe object.
- The ‘div’ function provides element-wise division. It returns a parallel dataframe which is a result of the arithmetic operation.
- The ‘value_counts’ function counts the unique values in a series excluding null values. It returns a replicated series containing the counts of the unique elements throughout the parallel series.

4.1.3 Other Features Developed

Some other features that have been developed for the `ParallelDataframe` class in form of properties are: `'global_to_local'`, `'globalShape'`, `'globalColumns'`, `'globalIndex'`, and `'loc'`.

For the `ParallelSeries` class, `'global_to_local'`, `'globalIndex'`, and `'collect'` have been developed for helping in testing.

4.2 IMPLEMENTATION DETAILS

Sub-classing of the Pandas data structures has been chosen as the method for extending the Pandas library and developing the Parallel-Pandas library. This enables the Parallel-Pandas data structures to use the underlying base functionality provided by Pandas classes, and functions can be overloaded on an as-needed basis in the incremental development.

As part of the Parallel-Pandas library, a sub-class for the dataframe object has been developed that represents a parallel dataframe. A `ParallelDataframe` can either be distributed, where data is uniformly distributed amongst the cluster or replicated, where the data is repeated across all the nodes in the cluster.

A subclass of the series object has also been developed, to complement the `ParallelDataframe` object. This subclass represents a parallel series which can be either distributed or replicated. However, the constructor of the series is implemented partially, and it expects distributed data for creating a distributed series, this would be the case when a series is created from the distributed dataframe.

This following section will elaborate on the implementation of the Parallel-Pandas library and the functions that have been developed. The function signatures have been kept same as the original Pandas so that it is easier for the users familiar with the Pandas API. Where needed, the additional arguments have been given default values but can be passed to the functions by advanced users who want to optimize their code further.

4.2.1 Constructor of the Parallel Dataframe Subclass

The constructor is responsible for creating a parallel dataframe object. Uniform distribution of data is achieved by sorting the data by size and then distributing it in a round-robin fashion, alternating the order of nodes in an increasing and decreasing order until all the data is distributed. For example, the first process gets the largest piece of data in the first round whereas, the last process gets the largest remaining piece of data in the next round and so on and so forth. This process and code have been adapted from a paper discussing comparison of MPI and Spark [24].

4.2.1.1 Function Signature

The function signature is given in Figure 4-2:

```
def __init__(self, data= None, index = None, columns = None, dtype = None, copy = False,  
            dist = 'distributed', dist_data= True, orient = 'columns', comm = MPI.COMM_WORLD):
```

Figure 4-2: Function signature of the parallel dataframe constructor

The 'dist', 'dist_data', 'orient' and 'comm' arguments have been added and have been given default values to cater for the parallel dataframe. All the other arguments are the same as given to the Pandas functions.

- ‘dist’ represents whether the dataframe is to be ‘distributed’ or ‘replicated’,
- ‘dist_data’ is a boolean and is only applicable when the parallel dataframe is of distributed type, it is true if the data being passed is distributed
- ‘orient’ represents whether the distribution is to be oriented by ‘columns’ or ‘index’ (Pandas name for rows)
- ‘comm’ is the communicator used for MPI communication

4.2.1.2 *Functionality Supported*

The constructor of the parallel dataframe can create the following:

- a distributed or replicated empty dataframe,
- a replicated dataframe from any type of serial input that Pandas is compatible with (dictionary, lists, series, NumPy array, dataframe)
- a column-wise distributed dataframe from a serial dataframe,
- a column-wise distributed dataframe from a serial dictionary
- a distributed dataframe from already distributed data

Creation of a row-wise distributed dataframe from undistributed data and creation of a distributed dataframe from a serial 2D numpy array are not supported yet.

4.2.2 **‘from_dict’ Function**

‘from_dict’ is a class method which constructs dataframe from a dictionary. It can create a distributed dataframe from the dictionary by column or by index depending on the ‘orient’

argument. ‘columns’ orientation means that the keys of the dictionary will be the column labels and ‘index’ means that the keys of the dictionary are the row labels.

In the Parallel-Pandas library’s distributed dataframe the ‘orient’ property translates to a column-distribution and row-distribution. The distribution of the data is done in a similar fashion as explained in the constructor, where data is first sorted by size and then distributed in an alternate round-robin fashion across all the nodes in the cluster.

The data is first distributed by an internal function as described above, once each node knows its local data, the base ‘from_dict’ function from the Pandas library is called upon to create local dataframes for each of the nodes.

4.2.2.1 Function Signature

The function signature is seen in Figure 4-3:

```
@classmethod
def from_dict(cls, data, orient = "columns", columns = None, dtype = None,
              comm = MPI.COMM_WORLD, dist = 'distributed') -> "ParallelDataFrame":
```

Figure 4-3: Function signature of the 'from_dict' method

The signature of this function is same as the Pandas function, the only addition is the argument of ‘comm’ which is needed for MPI communication and the ‘dist’ which specifies if the parallel dataframe is distributed or replicated. The additional arguments have been given default values and the users are not required to specify these additional arguments.

4.2.2.2 *Functionality Supported*

Currently, all the functionality supported by Pandas function of ‘from_dict’ is supported by Parallel_Pandas as well. A parallel dataframe of ‘replicated’ or ‘distributed’ type can be created. For distributed dataframe, when ‘orient’ attribute is ‘columns’, the data will be distributed column-wise, and when the ‘orient’ attribute is ‘index’ the data will be distributed row-wise throughout the nodes of the cluster.

4.2.3 **‘corr’ Function**

The ‘corr’ function finds the correlation between the columns of the dataframe. As a result, a distributed correlation dataframe is returned.

The implementation of the correlation function uses a combination of non-blocking sends and blocking receives. Every processor sends its columns to all the other processors one-by-one, and each processor receives and processes the data in every iteration. Sending of the data is done in form of contiguous NumPy arrays. Processing of the data includes using the underlying Pandas ‘corr’ function, Pandas ‘groupby’ and ‘drop’ functions are used to get rid of the duplicate rows and duplicate entries that exist on other processors. After all the iterations are done, the resulting local matrix is transposed with a Pandas ‘transpose’ operation to get it in a column-wise distributed format. The resulting matrix further undergoes a ‘sort_index’ operation so that the row labels in all the nodes are organized in the same way.

4.2.3.1 Function Signature

As can be seen in Figure 4-4, the function signature of the ‘corr’ function is the same as the corresponding Pandas function.

```
def corr(self, method = 'pearson', min_periods = 1):
```

Figure 4-4: Function signature of the 'corr' method

4.2.3.2 Functionality Supported

- The ‘corr’ function fully supports column-distributed dataframes and replicated dataframes.
- Correlation functionality on the row-distributed dataframes has not been implemented and tested yet.

4.2.4 ‘drop’ Function

The ‘drop’ function removes a column or a row with a given label. It returns a dataframe if ‘inplace’ is False otherwise it returns nothing and modifies the existing dataframe.

4.2.4.1 Function Signature

As can be seen in Figure 4-5, the function signature of the ‘drop’ function is the same as the corresponding Pandas function.

```
def drop(self, labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise'):
```

Figure 4-5: Function signature of the 'drop' function

4.2.4.2 *Functionality Supported*

The 0.21.0 version (and onwards) of Pandas allows users to specify single or list-like labels in the index or columns parameters instead of the combination of axis and labels. For example, specifying axis = 0 and labels can be done by just passing index = labels to obtain the same result. This introduces a new feature where columns and index can both be dropped in a single function call. Before the introduction of this feature, a call to the ‘drop’ would be specific to a single axis (row or column).

‘drop’ function has been tested for the following features that are supported by the Parallel-Pandas library:

- Dropping a column or a row in a replicated dataframe
- Dropping a column or a row in a distributed dataframe whether it is column-distributed or row-distributed
- Modifying the current dataframe and returning a new dataframe are both supported, this is specified by the boolean ‘inplace’ in the function argument
- Supports dropping of a single or multiple columns or rows specified
- Dropping of columns and rows in the same call for a replicated dataframe

Parallel-Pandas does not support dropping of columns and rows in the same call for a distributed dataframe. Currently, such a functionality can be accomplished with two separate function calls.

4.2.5 ‘apply’ Function

The ‘apply’ function is responsible for applying a given function along an axis of the dataframe. It returns a parallel series or a parallel dataframe object.

This function is implemented by wrapping the inherited ‘apply’ function provided by Pandas library. The wrapper makes sure to return a parallel series or dataframe as required, the underlying work of applying the functions is executed by the Pandas ‘apply’ function.

4.2.5.1 Function Signature

```
def apply(self, func, axis=0, raw=False, result_type=None, args=(), **kwargs):
```

Figure 4-6: Function signature of the ‘apply’ function

Figure 4-6 shows the function signature of the ‘apply’ function. The function signature of the ‘apply’ function is the same as its Pandas counterpart since the additional arguments needed by the Parallel-Pandas library are included in the parallel dataframe object.

4.2.5.2 Functionality Supported

- For replicated dataframe, all the features of the Pandas ‘apply’ function are supported by the Parallel-Pandas ‘apply’ function.
- For column-distributed dataframe, the ‘apply’ function supports the default functionality when axis = 0, the value of ‘raw’ can be True (row is passed as a ndarray object) or False (row is passed as a Series object).

- Applying a function with `axis=1` for a column-distributed dataframe is not yet supported, consequently having a `'result_type'` other than `'None'`, which is only relevant when `axis = 1`, has not been implemented and tested.
- The `'apply'` function with a row-distributed dataframe has not been tested.

4.2.6 'div' Function

The `'div'` function is responsible for performing division between a dataframe and an entity and provides support for substituting a `fill_value` for missing data. This function returns a dataframe.

This function is implemented by wrapping the inherited `'div'` function, provided by Pandas library, to return a parallel dataframe. The underlying work of `div` is executed by the Pandas `'div'` function.

4.2.6.1 Function Signature

```
def div(self, other, axis='columns', level=None, fill_value=None):
```

Figure 4-7: Function signature of the 'div' function

Figure 4-7 shows the function signature of the `'div'` function. The function signature of the `'div'` function is the same as its Pandas counterpart since the additional arguments needed by the Parallel-Pandas library are included in the parallel dataframe object.

4.2.6.2 *Functionality Supported*

Following describes a list of supported operations and mentions the limitations of this function:

- Division by constants is supported for a parallel dataframe whether it is distributed or replicated
- Division of two replicated dataframes is supported with all the functionality of axis, level and fill_value
- Division of two distributed dataframes or a distributed dataframe with a replicated dataframe is not supported yet.

4.2.7 **Constructor of the Parallel Series Subclass**

The ParallelSeries subclass has been developed to complement the ParallelDataframe and hence only the bare-minimum features have been developed for this class. The constructor of this class expects distributed data when creating a distributed series and can only create replicated series with non-distributed data.

4.2.7.1 *Function Signature*

The function signature can be seen in Figure 4-8:

```
def __init__(self, data=None, index=None, dtype=None, name=None, copy=False, fastpath=False,
             comm=MPI.COMM_WORLD, dist='distributed', dist_data=True):
```

Figure 4-8: Function signature for the constructor of the parallel series

In Figure 4-8, the arguments in the top line are the same as in Pandas, the additional arguments needed for Parallel-Pandas are in the bottom-line. These additional arguments have been given default values so that the novice users do not have to worry about them, but at the same time the ability of changing the default values has been provided for the advanced users. The additional argument of 'comm' specifies the MPI communicator, 'dist' specifies the type of distribution and can have the values of 'distributed' or 'replicated', 'dist_data' is a boolean which specifies whether the incoming data is already distributed or needs to be distributed.

4.2.7.2 *Functionality Supported*

The constructor of the ParallelSeries subclass can create the following:

- a distributed or replicated empty series
- a replicated series from any type of serial input that is compatible with Pandas, namely: a python dictionary, NumPy array, and a scalar value. 'dist = replicated' must be passed while creating this object.
- A distributed series from already distributed data, for example getting a row of a parallel distributed dataframe

Creating a distributed series with non-distributed data is not supported yet.

4.2.8 **'value_counts' Function**

The function of 'value_counts' can be called by a parallel series, which is of replicated or distributed type, the result would be a replicated series with counts of the unique values in the given parallel series.

As a first step, a distributed dictionary containing unique values as keys and counts as values is created. This dictionary is distributed amongst the processors via a hash function depending on the first two characters of the keys. The hashing function provides a total of 716 different hashing keys (26 x 26 number of combinations possible for the first two characters, plus 26 keys for single character words, and 10 keys for words starting with numeric characters). Every processor gets a subset of keys assigned to it by the hashing function and is hence responsible for summing the count of those subset of keys. Each processor prepares tuples, of key and count, to be sent to other processors based on the same hashing function. At the end of this step every processor has a count dictionary with its assigned keys and this dictionary is representative of the unique-counts of the entire original series. This process and code have been adapted from a paper discussing comparison of MPI and Spark [24].

A structure of key and count pair has been created as an MPI datatype and as a NumPy dtype to be able to efficiently transfer the word-count tuples. After the initial step of getting the distributed dictionary is completed, the word-count tuples are collected on every processor by an MPI ‘Gatherall’ operation and a replicated series is created which is then returned by the function of ‘value_counts’.

4.2.8.1 Function Signature

The signature of the function is seen in Figure 4-9:

```
def value_counts(self, normalize = False, sort = True, ascending = False, bins = None, dropna = True):
```

Figure 4-9: Function signature for the 'value_counts' function

As can be seen from the figure above, the signature of the function is the same as its Pandas counterpart. The reason is that, the additional features needed by the Parallel-Pandas version are included in the parallel series object and do not need to be passed as arguments to the function.

4.2.8.2 Functionality Supported

Currently the ‘value_counts’ feature is fully supported; it can find the counts of unique values for a replicated series or a distributed series and these unique values can be decimal numbers, integers or strings.

4.2.9 ‘collect’ Function

The ‘collect’ function for the ParallelSeries class has been developed to help in testing. This function has not been fully optimized for efficiency since it is not expected to be used for purposes other than testing. The ‘collect’ function is responsible for converting a distributed series into a replicated one. The functionality is achieved by performing two ‘all_gather’ operations once for indices and once for values and creating and returning a replicated series with that data.

4.2.10 ‘loc’ Property

‘loc’ is a property of the Pandas dataframe that can access a group of row and columns by labels or a boolean array. Allowed inputs to the Pandas version are a single label, a list or

array of labels ([‘a’, ‘b’, ‘c’]), a slice object with labels (‘a’: ‘f’), a boolean array of the same length as axis is being sliced, or a callable function with one argument.

For the Parallel-Pandas library, the ‘loc’ property has the same functionality and is developed to work with the parallel data structures. The ‘loc’ property is implemented by wrapping the ‘_getitem_axis’ function of the ‘_LocIndexer’ class from the Pandas library. This is accomplished by defining the function in ‘_CustomLocIndexer’ which is defined as a subclass of ‘_LocIndexer’. The wrapper makes sure that a parallel series or parallel dataframe (replicated or distributed as needed) is returned by this property.

4.2.10.1 Functionality Supported

Following is a list of functions supported by the Parallel-Pandas ‘loc’ property, it also points out the limitations.

- Getting of single or multiple rows from a column-distributed dataframe is supported
- Getting of single or multiple rows from a replicated dataframe is also supported
- Getting a cell value by specifying row and column label is supported for replicated dataframes
- The inputs of single label, list or array of labels, slice object with labels, and boolean array have been tested and are supported
- The input of callable function has not been tested and is not fully supported at this time.
- Getting a row or rows in index distributed dataframe is not supported

- Getting a cell value by specifying row and column label for distributed dataframe is not supported yet

4.2.11 Global Properties

Some global attributes have been developed for the `ParallelDataframe` and `ParallelSeries` objects to support the knowledge about the global object's shape, columns, rows etc. The implementation details of these global attributes are discussed below.

4.2.11.1 'global_to_local' and 'globalShape'

The property `'global_to_local'` is applicable to both the `ParallelDataFrame` and the `ParallelSeries` objects and is calculated once, on an as-needed basis. This attribute specifies the mapping of the columns or rows to the nodes depending on the orientation of the distribution in case of a distributed dataframe and specifies the mapping to nodes of the distributed series elements in case of a distributed series.

`'global_to_local'` is a dictionary whose keys are the distributed entity (either rows or columns) and the values are the processor ranks that contain the corresponding entity. A combination of an `'all_gather'` and `'all_reduce'` MPI operations are used to find this mapping. An optimization is implemented that the `'global_to_local'` dictionary is only calculated once when needed and after that it is stored and is available for re-use.

‘globalShape’ function has only been developed for the ParallelDataframe object. The global shape of the dataframe is obtained from the ‘global_to_local’ dictionary and stored for later re-use.

4.2.11.2 *‘globalIndex’ and ‘globalColumns’*

The column labels and the index (Pandas name for row) labels in Pandas are obtained with ‘obj.columns’ and ‘obj.index’ similarly the global column and index labels in the Parallel-Pandas can be obtained by ‘obj.globalColumns’ and ‘obj.globalIndex’ respectively.

‘globalIndex’ is applicable for ParallelDataFrame and ParallelSeries, if the index is distributed amongst the nodes, calling the ‘globalIndex’ returns the keys of the ‘global_to_local’ dictionary.

‘globalColumns’ is applicable for ParallelDataFrame only. If the dataframe is distributed column-wise, calling the ‘globalColumns’ returns the keys of the ‘global_to_local’ dictionary.

5 EVALUATION

The Parallel-Pandas library has been evaluated for functionality through unit-testing and micro-benchmarking; these tests are available in the code-base in ‘tests’ folder. The Parallel-Pandas library has also been evaluated by the use-case of duplicate detection. An application of document duplicate detection has been developed using the Parallel-Pandas library and performance is compared with a serial Pandas implementation, a parallel Pandas implementation and a PySpark implementation.

5.1 PERFORMANCE ANALYSIS METRICS

The performance analysis metrics of speed-up and efficiency with respect to the serial application have been used for evaluation of the Parallel-Pandas library. Speed-up, which is used for assessing performance of parallel applications, was calculated by dividing the execution time of the serial application by the parallel application. Specifically the speed-up formula is given by: $S(p) = \frac{T_{total(serial)}}{T_{total(p)}}$, where ‘p’ is the number of processors and ‘T’ is the total execution time. Efficiency, which is the speed-up normalized by number of processors, was calculated with the formula: $E(p) = \frac{S(p)}{p}$, where p is the number of processors.

To further analyze performance of Parallel-Pandas library, it has also been compared to PySpark, which provides parallelism via a MapReduce framework. This comparison was done via an application of duplicate detection.

5.2 HARDWARE AND SOFTWARE CONFIGURATION

The tests were executed on the Crill cluster at the University of Houston. Sixteen nodes on the Crill cluster were used, each node has four 12-core AMD Opteron 6174 processors and 64 GB of main memory. The cluster has a QDR InfiniBand and a Gigabit Ethernet network interconnect. A single node was used for the unit-testing and for micro-benchmarking.

For the MPI implementation, the code was based on Python 3.4, mpi4py 3.0.0, Open MPI 3.0.1, and Pandas 0.16.2. For the PySpark implementation, the code was based on Python 2.7 and Spark 2.3.4. Two datasets of different sizes were used for micro-benchmarking. Two datasets were used for the duplicate detection application, one was stored in and read from BeeGFS (a parallel file system, optimized for HPC) and the other from the network file system. The Parallel-Pandas library was also tested on a local system with Pandas 0.25.0.

5.3 MICROBENCHMARKS

Microbenchmarks have been used in this study to measure performance of few of the parallel functions developed and to understand how they scale with increasing processors. The functions of ‘from_dict’, ‘apply’, ‘value_counts’, and ‘corr’ were chosen for this analysis since they are the most important and commonly used functions amongst the functions that have been developed. Microbenchmark testing was carried out with distributed dataframes and not replicated dataframes. A total of three measurements were taken per function per number of processors and the minimum time was chosen to be analyzed, since that represented the best performance of that function. The serial functions were also timed with

the same size data for comparison purposes. Two data sizes were used to perform this analysis.

For the microbenchmark of ‘apply’ function, two file sizes of 80 KB and 163 KB were used.

The pseudo-code of the micro-benchmark is below:

```
serial_df = read_csv('name_of_file')
dist_df = ParallelDataFrame(serial_df, dist_data = False)
comm.Barrier()
T0 = time.time()
new_dist_df = dist_data.apply(convert to lowercase)
comm.Barrier()
T1 = time.time()
print_to_file(function name and (T1-T0) in seconds)
```

A distributed dataframe was created from the serial dataframe and the ‘apply’ function was called to convert all the string data in the dataframe to lowercase. The timing of this ‘apply’ operation was recorded with changing processors from 1 to 10. The serial function’s minimum execution time observed was 7.74 seconds with 80 KB data and 14.2 seconds with 163 KB data. Figure 5-1 below shows that the execution time reduced with increasing processors. For the 80 KB dataset, execution time reduced from 5.9 seconds for a single processor to 2 seconds with 6 processors, the speed-up seems to taper-off after 6 processors. For the 163 KB data, execution time reduced from 12.9 seconds with a single processor to 4.1 seconds with 8 processors, the speed-up seems to taper-off after 8 processors.

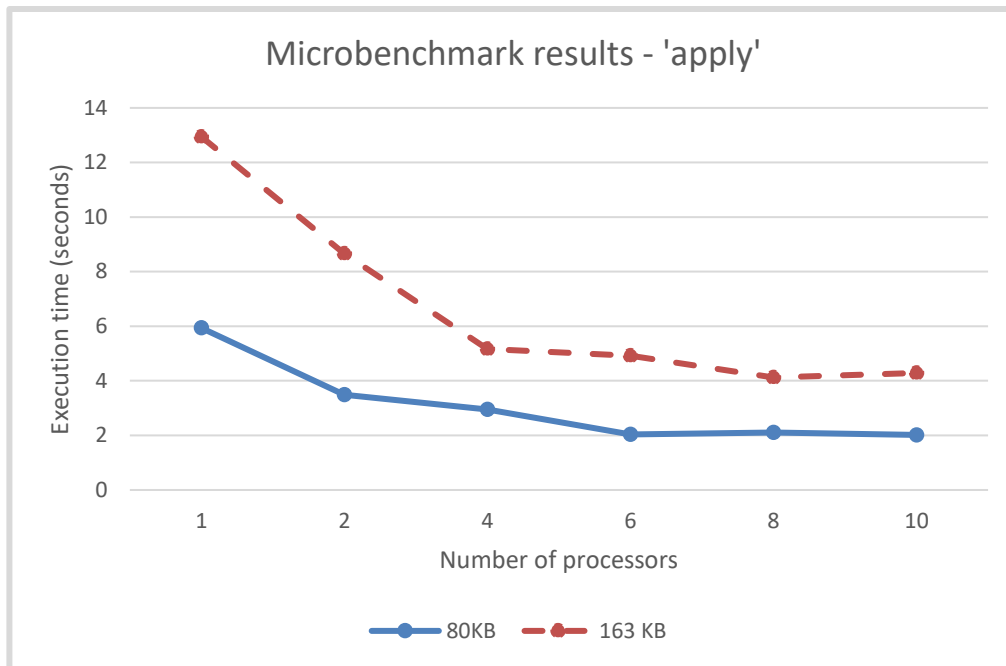


Figure 5-1: Microbenchmark results for 'apply' function

For the microbenchmark of the ‘from_dict’ function, a dictionary created from two file sizes of 80 KB and 163 KB was used. A distributed dataframe was then created from the dictionary with the ‘from_dict’ function. The timing of this ‘from_dict’ operation was recorded with changing processors from 1 to 10. The minimum runtime of the serial function was observed to be 367.7 seconds with the 80 KB data and 743 seconds with the 163 KB data. Figure 5-2 below shows the execution time with changing processors. For the 80 KB dataset, the execution time reduced from 359.3 seconds for a single processor to 157.3 with 8 processors. For the 163 KB dataset, the execution time reduced from 747.3 seconds for a single processor to 318.5 with 8 processors. In both cases, the speed-up seems to taper-off after 8 processors.

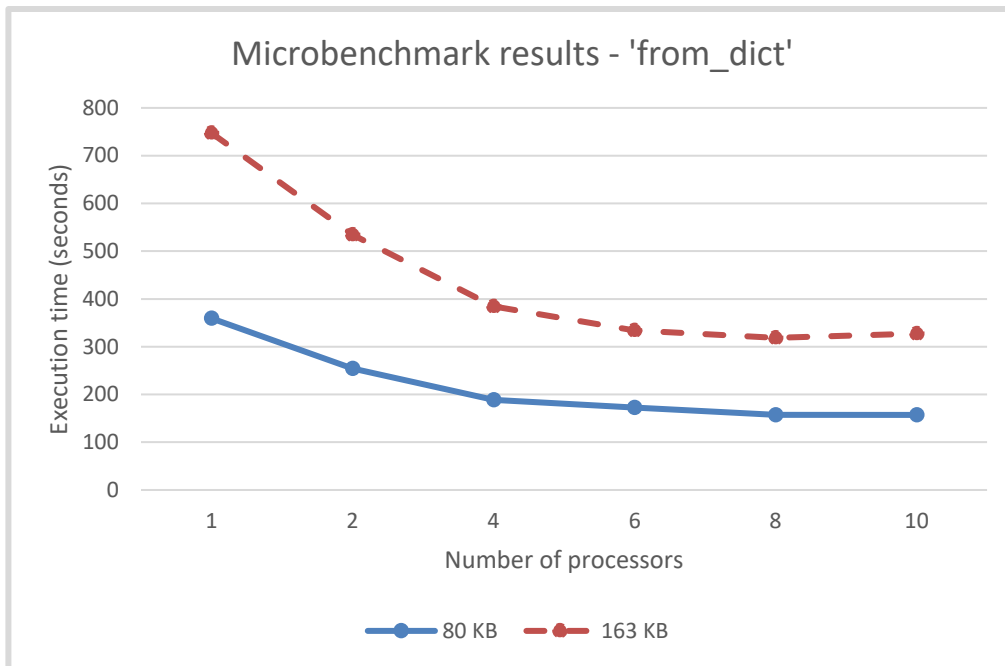


Figure 5-2: Microbenchmark results for 'from_dict' function

For the microbenchmark of the 'value_counts' function, two file sizes of 40 KB and 80 KB were used. The pseudo-code for the microbenchmark is below:

```
serial_df = read_csv('name_of_file')
dist_df = ParallelDataFrame(serial_df, dist_data = False)
dist_series = dist_df.loc['education']
comm.Barrier()
T0 = time.time()
dist_series.value_counts()
comm.Barrier()
T1 = time.time()
print_to_file(function name and (T1-T0) in seconds)
```

A distributed dataframe was created from a serial dataframe, a distributed series was then obtained by using the `ParallelDataframe`'s `'loc'` property. The `'value_counts'` function was then called on the distributed series. The timing of this `'value_counts'` operation was recorded with changing processors from 1 to 10. The function's minimum serial time with the data size of 40 KB was observed to be 0.58 seconds and with 80 KB data it was 1.3 seconds. Figure 5-3 below shows that the execution time reduced with increasing processors. For 40 KB data, the execution time reduced from 0.62 for single processor to about 0.1 with 8 processors. For 80 KB data, the execution time reduced from 1.6 seconds with one processor to 0.3 seconds with 8 processors. For both cases, the speed-up seems to taper-off after 8 processors.

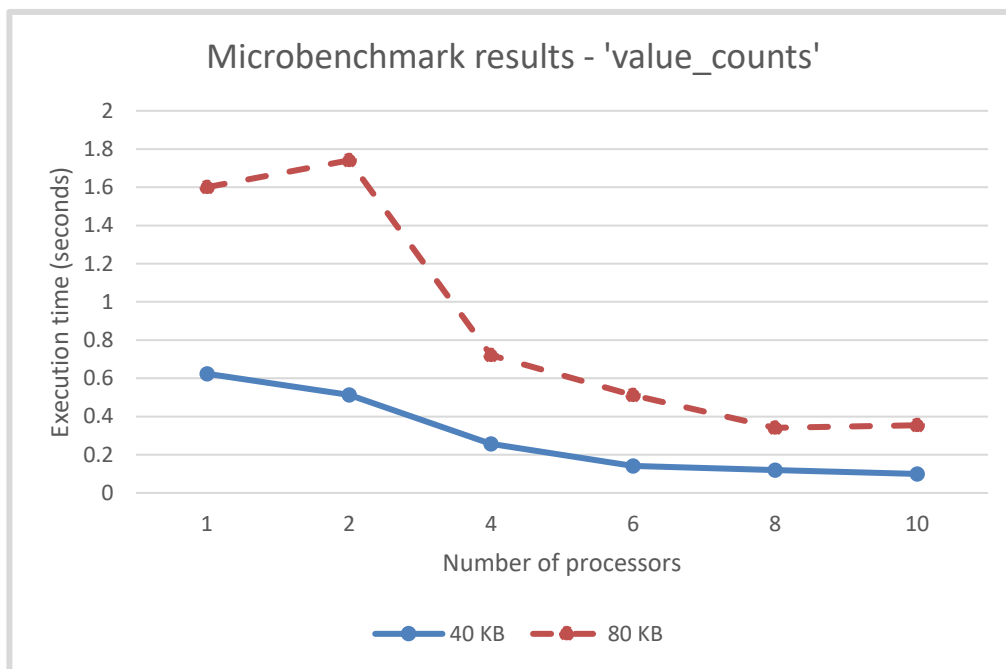


Figure 5-3: Microbenchmark results for 'value_counts' function

For the microbenchmark of the correlation function ('corr'), two files sizes of 82 KB and 164 KB were used. The pseudo-code for the microbenchmark is below:

```
serial_df = read_csv('name_of_file')
dist_df = ParallelDataFrame(serial_df, dist_data = False)
comm.Barrier()
T0 = time.time()
dist_data.corr()
comm.Barrier()
T1 = time.time()
print_to_file(function name and (T1-T0) in seconds)
```

A distributed dataframe was created from a serial dataframe. The 'corr' function was then called on the distributed dataframe, and the execution time of the function was recorded. The minimum runtime of the serial function was observed to be 2.9 seconds with the 82 KB data and 4.6 seconds with the 164 KB data. Figure 5-4 shows that 'corr' function's execution time worsened with increasing processors, which means that there is room for optimization for this function. For the 82 KB data, the execution time increased from 2.3 seconds for single processor to 34.7 seconds for 8 processors. For the 164 KB data, the execution time increased from 6 seconds with a single processor to 68 seconds with 8 processors. This degrading behavior in case of correlation ('corr') function is expected. The data is distributed column-wise and the 'corr' function finds the correlation amongst all the columns of the distributed dataframe to create a distributed correlation matrix. As discussed in the implementation section, in the process of creation of the distributed correlation matrix, every processor shares

its columns with all the other processor in iterations. On top of this communication overhead, the local matrices created need to be transposed and sorted by index so that the resultant local matrices (per processor) have consistent indices. This ‘corr’ function needs to be optimized for better performance.

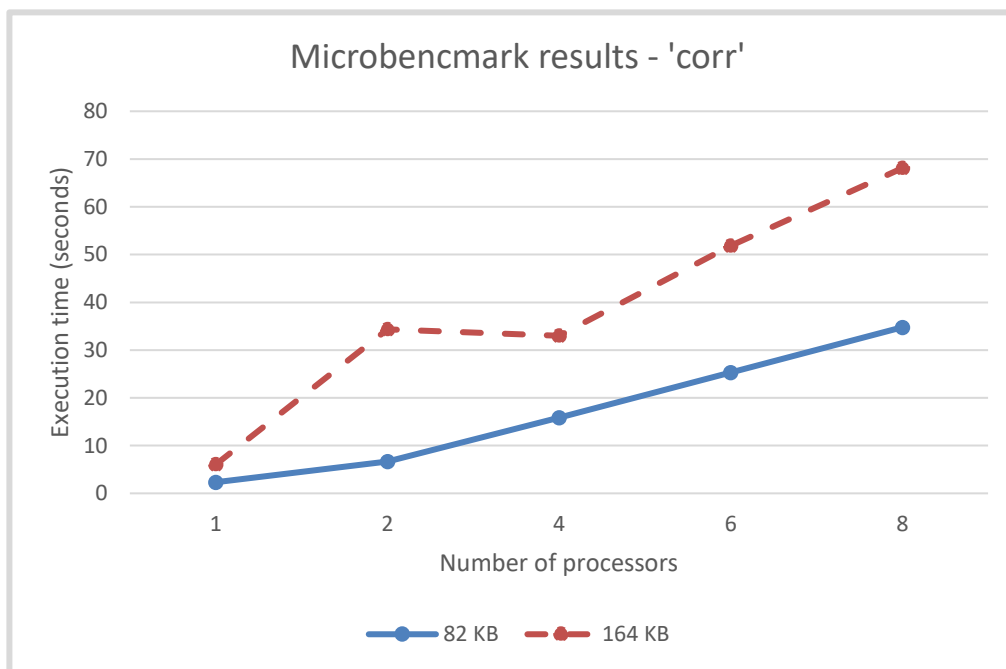


Figure 5-4: Microbenchmark results for 'corr' function

5.4 USE CASE – DUPLICATE DETECTION

The application scenario that was used for testing out the Parallel-Pandas library is duplicate detection. Finding similar items in a dataset is a fundamental data mining problem. It has vast applications whether it is finding similar documents to catch plagiarism, finding mirror web pages (same web pages being hosted in different places) or identifying same articles that have been published by different newspapers (with minor differences such as title etc.). Hence, this problem of similar items becomes very important for search engines to avoid duplication in

the results. On a wider scale, finding similar items also finds applications in the field of e-commerce and media services. It is used in collaborative filtering, specifically suggesting items based on user's similarity to another user [11].

5.4.1 Datasets

Two datasets were used for performance evaluation of the document duplicate detection application. One dataset contained 519 books from the Gutenberg project [21], stored on a network file system in a single directory as one file per book. This dataset was approximately 541 MB in size. A small set of duplicate documents, about 130 books, was manually generated to test out the application.

The other dataset was based on a real-world scenario [10] and the documents were news articles, one article per document. These articles were taken from four different news outlets: New York Times (US), The Wall Street Journal (US), The Guardian (UK) and The Times (UK). The total dataset included 18,698 news articles, with the overall size of 106 MB. Even though the size of the dataset is not as much in this case, but the number of articles creates a scalability problem that is solved with parallel processing.

5.4.2 Algorithm

The algorithm used for finding similar documents has been taken from a paper which describes similarity detection in a MapReduce framework [8]. The crux of the algorithm is given here:

- Pre-process the data by lower case conversion, removal of special symbols, stop word removal, and stemming.
- Find the top 1000 words in the entire corpus.
- Clean the data by removing non-top 1000 words and create an inverted index where each term is associated with a weight per document. Weight is calculated by number of term occurrences divided by total number of words in that document. Hence, for each document d_i and term t there will be a corresponding weight $w_{d_i}^t$ which represents the importance of the word in the document:

$$w_{d_i}^t = \frac{(\text{number of term}_t \text{ occurrences})_{d_i}}{(\text{total terms})_{d_i}}$$

- The inverted index of each file is used to calculate a similarity matrix for pairwise comparison of the files. The similarity of two documents d_i and d_j is created by calculating the sum of the products of all the weights in the documents as below:

$$\text{similarity}(d_i, d_j) = \sum_{(\text{all 1000 terms})} w_{d_i}^t \cdot w_{d_j}^t$$

- The pairs with the maximum similarity score are chosen to finalize the most similar documents in the dataset.

5.4.2.1 Optimization for Parallel Processing

In order to make the above algorithm more efficient by using cluster computing, areas need to be identified for parallel execution.

- Once the dataset is distributed among the nodes in the cluster, pre-processing step can be done in parallel by each node.

- Finding of the top 1000 words in the entire corpus needs internode communication, since every node has a part of the dataset.
- Once each processor knows the top 1000 words, cleaning and finding of the inverted index can be done in parallel by each node on its subset of the data. As a result, the inverted index for the entire dataset will be distributed among all the nodes of the cluster.
- Creation of the similarity matrix needs internode communication, where MPI with combination of NumPy is used to speed up the process. As a result of this step, a distributed similarity matrix is created. The local similarity matrix contains the comparison of its subset of documents with the rest of the dataset.
- Selection of the similar pairs based on a threshold can then be done in parallel by each node.

The steps of the algorithm are depicted in Figure 5-5. The operations in the blue-boxes (or the dashed-outline) are done locally on each processor in parallel, whereas the operations in the green-boxes (or solid outline) need inter-node communication for which MPI is used as the underlying technology.

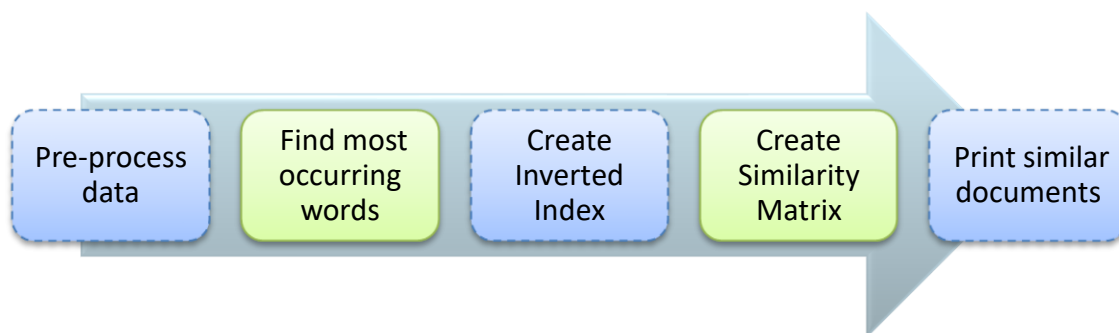


Figure 5-5: Steps of the duplicate detection algorithm, the steps that are contained in boxes with dotted outlines can be done in parallel

5.4.2.2 Optimization by Using Dataframes

The algorithm can be further optimized for the similarity matrix calculation since dataframes are being used. If we visualize how the dataframe for an inverted index looks like, it contains filenames as columns labels, words as row labels, and word weights as the cell contents. Similarity of two documents can be calculated by evaluating the similarity of the columns, and the correlation function from the Pandas (serial dataframe) or the Parallel-Pandas (parallel dataframe) library fits this need exactly. Hence, the similarity matrix calculation step in the algorithm is updated to use the correlation function with the ‘Pearson’ method. Pearson correlation gives the linear correlation between two variable X and Y, gives a value between 1 and -1 where 1 is positively correlated and negative valued numbers correspond to a negative correlation. The Pearson coefficient is defined as:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

Where ρ represents the Pearson correlation, $cov(X,Y)$ stands for the covariance between X and Y and σ stands for standard deviation.

5.4.3 Application I/O (Input/Output)

The inputs to the application are: the location of the dataset, similarity threshold and names of the two output files, one for the results and one for the performance statistics, an optional file for debugging can also be given with the ‘-debug-file’ flag. Similarity threshold needs to be a number between -1 and 1 and defines when the documents are considered similar. After a successful run, the application will create or append results and performance data to the given

files. When formulating the results, every pair deemed similar based on the given threshold is printed to the output file twice since it picks each file as a duplicate of the other.

5.4.4 Different Applications Developed and Compared

Four different types of implementations of the document duplicate detection application will be discussed in the section below. These include a Parallel-Pandas implementation where the focus is on using Parallel-Pandas library, MPI-Pandas implementation where the code has been optimized for performance using Pandas and MPI, Serial-Pandas application which uses Pandas, and PySpark implementation which focuses on using PySpark for implementing duplicate detection. Three of these applications namely the Parallel-Pandas, Serial-Pandas and MPI-Pandas have been developed, whereas the details and results for PySpark have been taken from a paper comparing MPI and Spark [24].

The Serial-Pandas application is used to calculate the speed-up and efficiency of the Parallel-Pandas application. The MPI-Pandas and PySpark applications are used as a further comparison to evaluate the performance of the Parallel-Pandas duplicate detection application.

5.4.4.1 Parallel-Pandas Application

A document duplicate detection application has been developed using the Parallel-Pandas library discussed in section 4, the implementation is based on the algorithm and optimization, discussed in section 5.4. This application has been developed by using the ParallelDataframe functions of ‘from_dict’, ‘apply’, ‘corr’, ‘drop’, ‘div’, the ParallelDataframe property of

‘loc’, the ParallelSeries function of ‘value_counts’ and the ‘concat’ function of the Pandas library.

5.4.4.1.1 Data Distribution

The Parallel-Pandas application has uniform column-wise distribution throughout the application since that is the functionality currently supported by the Parallel-Pandas library. Initially the ‘from_dict’ function is used with the orientation set to ‘columns’ to get a dataframe as shown in the figure below. Figure 5-6 shows a dataframe created with a small test data when the application is run on 5 nodes, as can be seen the data is divided column-wise where every node has a single row labeled ‘text’ and different columns corresponding to different files.

```
1176.txt ... 60390-0.txt
text b'The Project Gutenberg EBook of On Horsemansh... b'\xef\xbb\xbfThe Project Gutenberg EBook of H...
[1 rows x 3 columns]
22364.txt ... noni - Copy.txt
text b'Project Gutenberg\'s The Philosophy of the M... b'\xef\xbb\xbf'
[1 rows x 3 columns]
2momm10u - Copy.utf ... noni.txt
text b'\xff\xfeT\x00h\x00e\x00 \x00f\x00o\x00l\x00l... b'\xef\xbb\xbfJamie'
[1 rows x 3 columns]
2momm10u.utf 4776-Copy.txt
text b'\xff\xfeT\x00h\x00e\x00 \x00f\x00o\x00l\x00l... b'The Project Gutenberg EBook of Political Ide...
1176-Copy.txt 5833-0.txt
text b'The Project Gutenberg EBook of On Horsemansh... b"\xef\xbb\xbfThe Project Gutenberg EBook of H...
[1 rows x 3 columns]
```

Figure 5-6: Initial column-wise data distribution of the dataset in the Parallel-Pandas application

After the data is pre-processed and cleaned, inverted index is created by using a combination of ‘div’ and ‘concat’ functionality in a parallel fashion since communication is not needed for this step. Figure 5-7 shows a snippet of the inverted index matrix on two nodes, it needs to be

emphasized that this is a column-wise distributed dataframe, hence the row-labels are the same on every node and the columns are distributed.

	1176.txt	5778-0-Copy.txt	60390-0.txt
xe	0.000000	0.123948	0.024897
wa	0.000549	0.020335	0.024981
hi	0.018882	0.020131	0.014020
thi	0.009222	0.007501	0.010009
said	0.000329	0.018192	0.002042
...
stori	0.000110	0.000383	0.000032
suspicion	0.000110	0.000204	0.000032
administr	0.000000	0.000000	0.000423
proper	0.000110	0.000051	0.000212
connect	0.000110	0.000128	0.000116

[1000 rows x 3 columns]

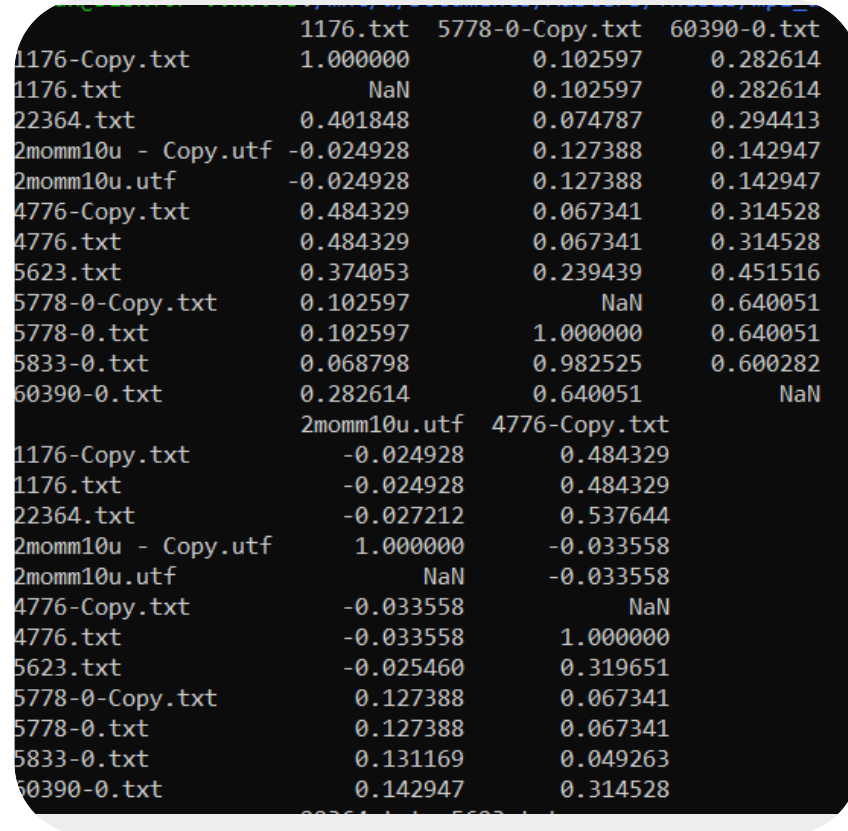
	22364.txt	5623.txt	noni - Copy.txt
xe	0.000000	0.000000	0.0
wa	0.001945	0.021262	0.0
hi	0.014743	0.020016	0.0
thi	0.016939	0.007682	0.0
said	0.000157	0.019019	0.0
...
stori	0.000031	0.000249	0.0
suspicion	0.000188	0.000374	0.0
administr	0.000000	0.000000	0.0
proper	0.000878	0.000042	0.0
connect	0.000878	0.000042	0.0

[1000 rows x 3 columns]

Figure 5-7: A subset of the distributed inverted index in the Parallel-Pandas application

The ‘corr’ function from the Parallel-Pandas library is used to find the similarity matrix from the inverted index. Figure 5-8 shows a snippet of the final similarity matrix on two nodes,

again it is to be noted that it is a column-wise distributed dataframe hence the row-labels are the same and the columns are different since they are distributed.



	1176.txt	5778-0-Copy.txt	60390-0.txt	
1176-Copy.txt	1.000000	0.102597	0.282614	
1176.txt	NaN	0.102597	0.282614	
22364.txt	0.401848	0.074787	0.294413	
2momm10u - Copy.utf	-0.024928	0.127388	0.142947	
2momm10u.utf	-0.024928	0.127388	0.142947	
4776-Copy.txt	0.484329	0.067341	0.314528	
4776.txt	0.484329	0.067341	0.314528	
5623.txt	0.374053	0.239439	0.451516	
5778-0-Copy.txt	0.102597	NaN	0.640051	
5778-0.txt	0.102597	1.000000	0.640051	
5833-0.txt	0.068798	0.982525	0.600282	
60390-0.txt	0.282614	0.640051	NaN	
	2momm10u.utf	4776-Copy.txt		
1176-Copy.txt	-0.024928	0.484329		
1176.txt	-0.024928	0.484329		
22364.txt	-0.027212	0.537644		
2momm10u - Copy.utf	1.000000	-0.033558		
2momm10u.utf	NaN	-0.033558		
4776-Copy.txt	-0.033558	NaN		
4776.txt	-0.033558	1.000000		
5623.txt	-0.025460	0.319651		
5778-0-Copy.txt	0.127388	0.067341		
5778-0.txt	0.127388	0.067341		
5833-0.txt	0.131169	0.049263		
60390-0.txt	0.142947	0.314528		

Figure 5-8: A subset of the distributed similarity matrix in the Parallel-Pandas application

5.4.4.1.2 Use of Parallel-Pandas Dataframe

Use of the Parallel-Pandas dataframe makes it easier to perform various pre-processing and data cleaning activities by using the ‘apply’ function. The ‘loc’ property of a distributed dataframe gives a distributed series upon which the ‘value_counts’ function is used to find the top 1000 words in the entire dataset. ‘drop’ function is used to get rid of the unwanted columns such as counts of the words once the top 1000 words have been found.

5.4.4.2 *MPI-Pandas Application*

A document duplicate detection application has been developed using Pandas and MPI. The algorithm with all the optimizations as discussed in section 5.4 has been implemented using Pandas dataframes and MPI communication using NumPy buffers. As stated earlier, the similarity matrix is calculated by using the correlation function.

5.4.4.2.1 Use of Pandas Dataframe

The data structure of the dataframe is very helpful for the various transformations that are needed by the algorithm. These transformations include pre-processing of the data, cleaning the data to include only the top 1000 words, and transforming the dataframe to have words as row labels, filenames as column labels, and word counts as the data elements in order to prepare for the similarity matrix calculation.

5.4.4.2.2 Optimizing the Task of Finding Top 1000 Words

The task of finding top 1000 words has been optimized for performance. First, similar to the implementation in the Parallel-Pandas library, a distributed dictionary containing words as keys and counts as values is created. This dictionary is distributed amongst the processors via a hash function depending on the first two letters of the words. Every processor gets a subset of words assigned to it by the hashing function and is hence responsible for summing the count of those subset of words. Each processor prepares tuples, of word and count, to be sent to other processors based on the same hashing function. At the end of this step every

processor has a word count dictionary with its assigned words and this dictionary is representative of the word-counts of the entire original series.

After the first step is accomplished, every processor only sends the top 1000 counts to all others via an ‘Allgather’ operation. Based on these counts a cutoff is determined and every processor gets rid of any words that have counts below that cutoff. Only the remaining words (having word count greater than cutoff) are then gathered by an ‘Allgather’ operation and the final top words are then determined.

5.4.4.2.3 Data Distribution

The MPI-Pandas application is optimized for performance and has row-wise distribution of data for the pre-processing and the cleaning step, then it moves to column-wise distribution of data for the inverted index and similarity matrix calculations.

Figure 5-9 below shows the initial dataframe that is created when using a small test dataset with 5 processors. As can be seen the data is distributed row-wise amongst the processors, the indices are the filenames, the dataframe contains a single column called ‘text’ containing the entire text of the file. This format is very useful for the beginning steps of pre-processing, finding top words, and cleaning by removing all non-top words, since we are only concerned with the ‘text’ column. All the pre-processing of stop-word removal etc. can be done by applying the function along the column axis of the dataframe. For getting top words, the column of the dataframe can be easily obtained as a series and then a ‘split’ function can be

applied to get all the words. By the same token, once the top words are found the dataframe can be cleaned by applying a function along the column axis to get rid of the unwanted words.

```

text
filename
2momm10u.utf      b'\xff\xfeT\x00h\x00e\x00 \x0f\x00o\x01\x01...
4776-Copy.txt     b'The Project Gutenberg EBook of Political Ide...
1176.txt          b'The Project Gutenberg EBook of On Horsemansh...
text
filename
60390-0.txt      b'\xef\xbb\xbfThe Project Gutenberg EBook of H...
5623.txt        b'The Project Gutenberg EBook of The Young Exp...
noni.txt        b'\xef\xbb\xbfJamie'
text
filename
22364.txt       b'Project Gutenberg\'s The Philosophy of the M...
5833-0.txt      b"\xef\xbb\xbfThe Project Gutenberg EBook of H...
noni - Copy.txt b'\xef\xbb\xbf'
text
filename
5778-0-Copy.txt b"\xef\xbb\xbfThe Project Gutenberg EBook of T...
5778-0.txt      b"\xef\xbb\xbfThe Project Gutenberg EBook of T...
text
filename
2momm10u - Copy.utf b'\xff\xfeT\x00h\x00e\x00 \x0f\x00o\x01\x01...
4776.txt          b'The Project Gutenberg EBook of Political Ide...
1176-Copy.txt     b'The Project Gutenberg EBook of On Horsemansh...

```

Figure 5-9: Initial dataframe distribution (row-wise) in the MPI-Pandas application

After these initial steps are completed the dataframe is prepared for the inverted index and the similarity matrix calculation. Here, the algorithm benefits from having a column-wise distribution. Every node converts its portion of the dataframe with a combination of ‘str’, ‘split’ and ‘transpose’ operation to prepare the dataframe for inverted index. Figure 5-10 below shows a snapshot of the dataframe of 2 processors in a column-wise distributed format.

```

filename 22364.txt 5833-0.txt noni - Copy.txt
0        project    project    None
1        gutenber    gutenber    None
2         moral      ebook      None
3         feel       help       None
4         nthi       nthi       None
...      ...        ...        ...
26439     None      new        None
26440     None      ebook      None
26441     None      hear       None
26442     None      new        None
26443     None      ebook      None

[26444 rows x 3 columns]
filename 5778-0-Copy.txt 5778-0.txt
0        project    project
1        gutenber    gutenber
2         ebook      ebook
3         tri        tri
4         trust      trust
...      ...        ...
29422     new        None
29423     ebook      None
29424     hear       None
29425     new        None
29426     ebook      None

[29427 rows x 2 columns]

```

Figure 5-10: Column-wise data distribution in the MPI-Pandas application

5.4.4.3 Serial-Pandas Application

A document duplicate detection application has been developed using the standard Pandas functions. All the benefits of the dataframes that have been discussed in the section 5.4.4.2.1 apply to this application as well. The difference from the MPI applications is that the data is not distributed and is present in a standard serial dataframe. The formatting of the dataframe

has been kept similar to the Parallel-Pandas application dataframe where the initial dataframe is created with the default ‘column’ orientation selection in the ‘from_dict’ method.

5.4.4.4 PySpark Application

The document duplicate detection application in PySpark uses the same initial algorithm discussed in section 5.4.1. To make the comparison fair between the MPI and the PySpark version, the overall processing of the text files such as stop-word removal etc. has been kept same. The timing of the PySpark application excludes the time required for getting the allocation for the job through the scheduler.

The implementation of the PySpark application is achieved by using ‘reduceByKey’ followed by a ‘sort’ operation for the computation of the top 1000 words and the cleaning of the data. Inverted index is created with a ‘join’ operation of the top 1000 words. The similarity matrix calculation is then found with a combination of ‘join’ and ‘reduceByKey’ operations. [24]

5.4.5 Results and Discussion

The following section includes the performance analysis of the Parallel-Pandas library via the application of document duplicate detection.

Comparing the performance of PySpark (MapReduce) and MPI platforms is not a trivial task, since they have different nomenclatures. During this study, to get both the MPI and PySpark applications on equal footing, the hardware resources used by both versions of the application were kept the same. The resources used by MPI is a straight-forward matter, since each MPI

process uses a single core. However, for a Spark job there are multiple ways of assigning resources by varying the number of executors and the number of cores per executor. During this study, the optimal setting of four cores and 18GB memory per executor was used. [24] Hence, 256 nodes in an MPI job is comparable to 64 executors with 4 cores each for a PySpark job.

5.4.5.1 Results for the Books Dataset

The following tables show the execution time in seconds for each of the four implementations of Parallel-Pandas, MPI-Pandas, Serial-Pandas and PySpark for the Gutenberg dataset of 519 books. As explained in section 5.4.1, the dataset was read from network file system.

Execution time of the Parallel-Pandas and MPI-Pandas applications was captured, using 32, 64, 128, and 256 nodes. Execution time of the PySpark application was captured, using 8, 16, 32, and 64 executors with 4 cores per executor. For all the applications, five measurements were taken per datapoint.

The execution times recorded for the Serial-Pandas application can be seen in Table 5-1 below. On average the runtime of a serial application of duplicate detection, with the Gutenberg dataset, using Pandas was seen to be 3.6 hours.

Table 5-1: Execution time in seconds for the Serial-Pandas application using the book dataset

Serial-Pandas							
Processors	Run1	Run2	Run3	Run4	Run5	Avg	Min
1	12724.91	12714.61	12864.77	12927.33	12791.68	12804.66	12714.61

The execution times recorded (in seconds) for the Parallel-Pandas application as well as the speed-up and efficiency statistics can be seen in Table 5-2 below. As can be seen, the performance improved by using more resources. The execution time kept decreasing when the processors were increased, and the speed-up increased with increasing resources.

Table 5-2: Performance statistics for the Parallel-Pandas application using the book dataset

Parallel-Pandas									
Processors	Run1	Run2	Run3	Run4	Run5	Average	Minimum	Speed-up	Efficiency
32	644.31	632.71	661.46	618.52	629.45	637.29	618.52	20.09	0.628
64	409.85	390.28	399.8	385.41	390.91	395.25	385.41	32.40	0.51
128	275.38	293.53	274.79	261.54	292.82	279.61	261.54	45.79	0.358
256	208.66	213.18	202.32	213.6	220.43	211.64	202.32	60.50	0.24

Evaluating the speed-up and taking a closer look at it in Figure 5-11 below, we see that it has a logarithmic trendline. The highest speed-up achieved was 60.5 with 256 processors, the highest efficiency of 0.6 was achieved with 32 processors. The speed-up achieved does not have the ideal linear trendline, this is as expected. As was discussed in section 5.4.1, only sections of the algorithm are embarrassingly parallel where no communication is needed. The most expensive part of the algorithm is the determination of the top 1000 words and the creation of the similarity matrix which cannot be done without communication between nodes which is a relatively costly operation.

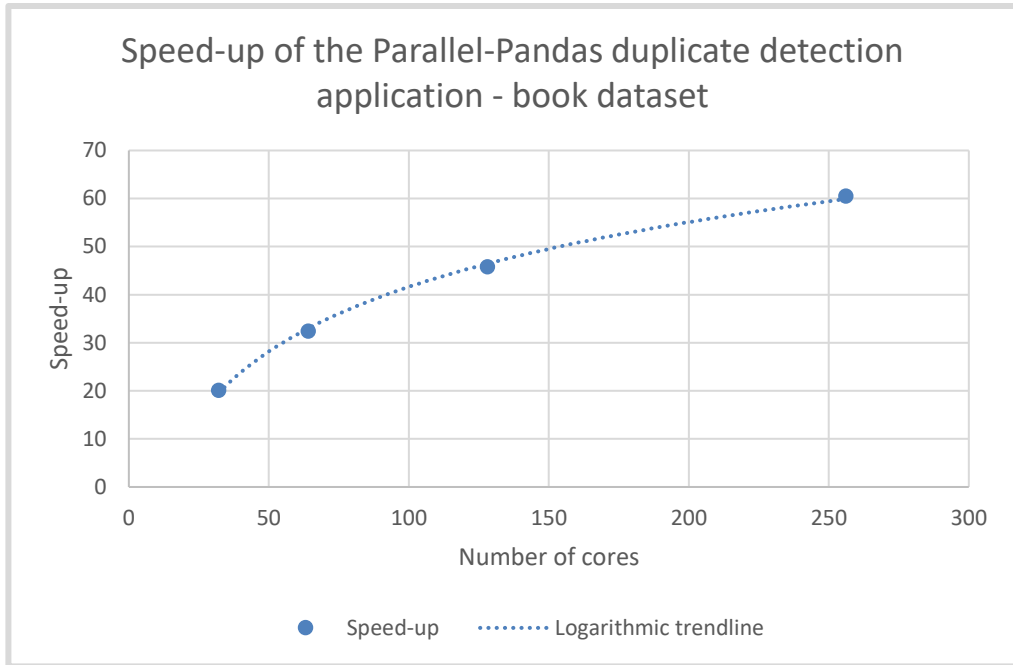


Figure 5-11: Speed-up of the Parallel-Pandas duplicate detection application while using the Gutenberg dataset

The execution times recorded for the MPI-Pandas application can be seen in Table 5-3 below.

As can be seen, the performance improved with increasing resources.

Table 5-3: Execution time in seconds for the MPI-Pandas application using the book dataset

MPI-Pandas							
Processors	Run1	Run2	Run3	Run4	Run5	Average	Minimum
32	353.4065	355.8138	348.8816	349.5711	353.1079	352.1562	348.8816
64	228.693	253.4226	234.0162	205.0649	225.2543	229.2902	205.0649
128	170.3947	174.2834	153.1049	187.781	186.2541	174.3636	153.1049
256	149.1602	133.0076	154.4498	154.3188	153.8913	148.9655	133.0076

The execution times recorded for the PySpark application can be seen in Table 5-4 below. As

can be seen, the performance improved with increasing resources. The performance did not

fluctuate much which is emphasized by the small difference between the average and minimum columns.

Table 5-4: Execution time in seconds for the PySpark application using the book dataset

PySpark							
Executors	Run1	Run2	Run3	Run4	Run5	Average	Minimum
8	1907.681	1922.124	1891.85	1891.915	1891.338	1900.981	1891.338
16	1037.353	1016.552	1028.279	1029.141	1014.788	1025.223	1014.788
32	592.43	582.9129	601.829	583.7176	576.7964	587.5372	576.7964
64	391.9454	389.353	381.3266	386.4814	379.8057	385.7824	379.8057

Comparing the performance of the Parallel-Pandas, PySpark and MPI-Pandas applications while using the Gutenberg dataset, it is seen in Figure 5-12 that the MPI implementations outperform the PySpark implementation for all core counts. Specifically, the Parallel-Pandas application performs about 45% better than PySpark implementation with 256 processors. Worth discussing is the comparison of the Parallel-Pandas and MPI-Pandas applications. The MPI-Pandas application performs about 29% better than Parallel-Pandas application for 256 cores, for two main reasons. One reason is the difference in the data distribution which has been discussed in detail in section 5.4.4 (under the ‘Data Distribution’ sections of the application implementation), MPI-Pandas application changes data distribution as is optimal to that part of the problem, whereas Parallel-Pandas application currently only supports using one-type of data-distribution throughout the application. Another reason is that the step of determining top 1000 words has been optimized by MPI-Pandas application as discussed in section 5.4.4.2.2, whereas the Parallel-Pandas implementation uses the ‘value_counts’ function (discussed in section 4.2.8) which makes the counts and words for all the words

available on all the processors and hence requires more communication, consequently takes more time.

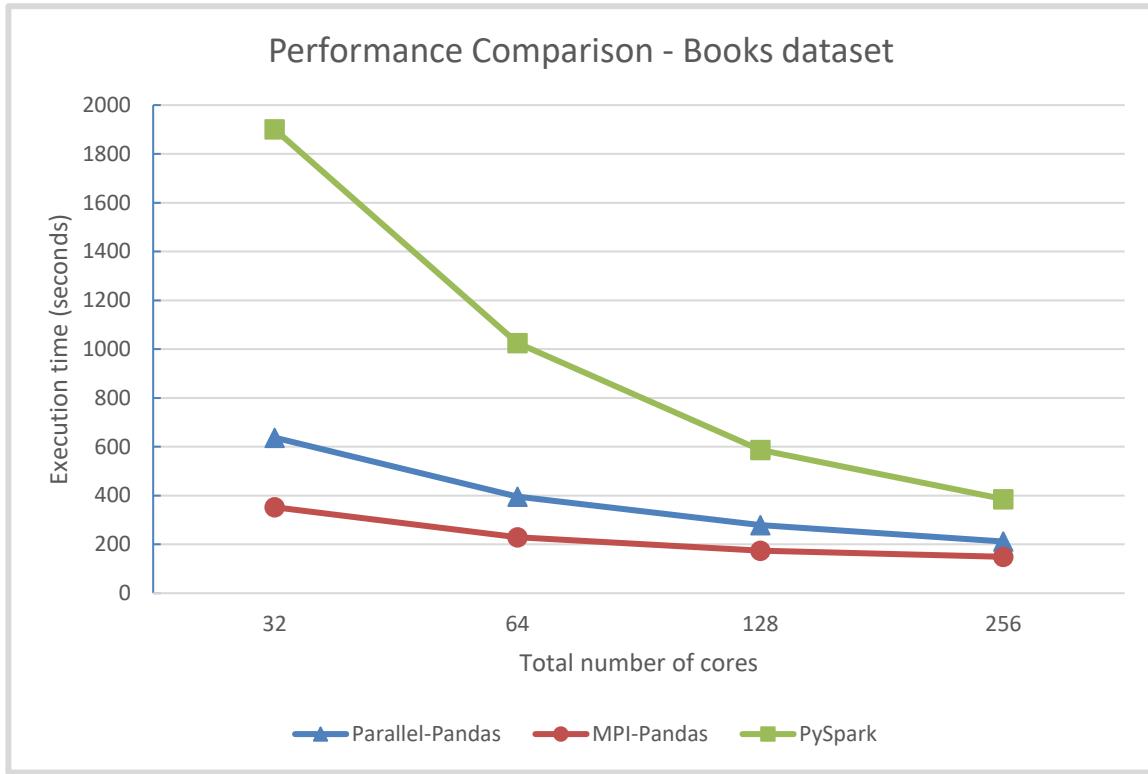


Figure 5-12: Performance comparison of duplicate detection application for the dataset containing 519 Gutenberg books

5.4.5.2 Results for the News Articles Dataset

The following tables show the execution time in seconds for each of the four implementations of Serial-Pandas, Parallel-Pandas, MPI-Pandas and PySpark for the news articles dataset containing 18,698 articles. As explained in section 5.4.1, the dataset was read from BeeGFS.

Execution time of the Parallel-Pandas and MPI-Pandas applications was captured, using 32, 64, 128, and 256 nodes, five measurements were taken per data point. Execution time of the

PySpark application was captured, using 8, 16, 32, and 64 executors with 4 cores per executor. Since the PySpark implementation took a very long time to execute, only a single measurement of the PySpark application was taken.

The execution times recorded for the Serial-Pandas application can be seen in Table 5-5 below. On average the runtime of a serial application of duplicate detection using Pandas was seen to be 1.8 hours.

Table 5-5: Execution time in seconds for the Serial-Pandas application using the news articles dataset

Serial-Pandas							
Processors	Run1	Run2	Run3	Run4	Run5	Average	Minimum
1	6715.743	6675.39	6417.451	6483.01	6363.99	6531.1163	6363.9901

The performance statistics based on the execution times recorded for the Parallel-Pandas application can be seen in Table 5-6 below. The performance improved with increasing resources up till 128 processors. The efficiency achieved is lowest for 256 processors, which means that the problem is not big enough for the resources provided.

Table 5-6: Performance statistics for the Parallel-Pandas application using the news articles dataset

Parallel-Pandas									
Processors	Run1	Run2	Run3	Run4	Run5	Average	Minimum	Speed-up	Efficiency
32	2540.06	2495.3	2456.84	2485.63	2466.85	2488.93	2456.84	2.62	0.082
64	1240.65	1223.2	1225.34	1234.7	1231.73	1231.12	1223.2	5.31	0.083
128	811.71	810.39	803.956	807.76	814.908	809.744	803.956	8.07	0.063
256	933.795	988.91	1008.36	1090.3	1055.73	1015.42	933.795	6.43	0.025

The optimum speed-up of 8 is achieved at 128 processors with this specific problem size. However, 8 seems to be a low number for the speed-up. It seems that for this problem where there are many small articles/files, the speed-up numbers achieved are not as high as with the book dataset where there were fewer bigger files. Figure 5-13 below, visualizes the speed-up for the Parallel-Pandas duplicate application.

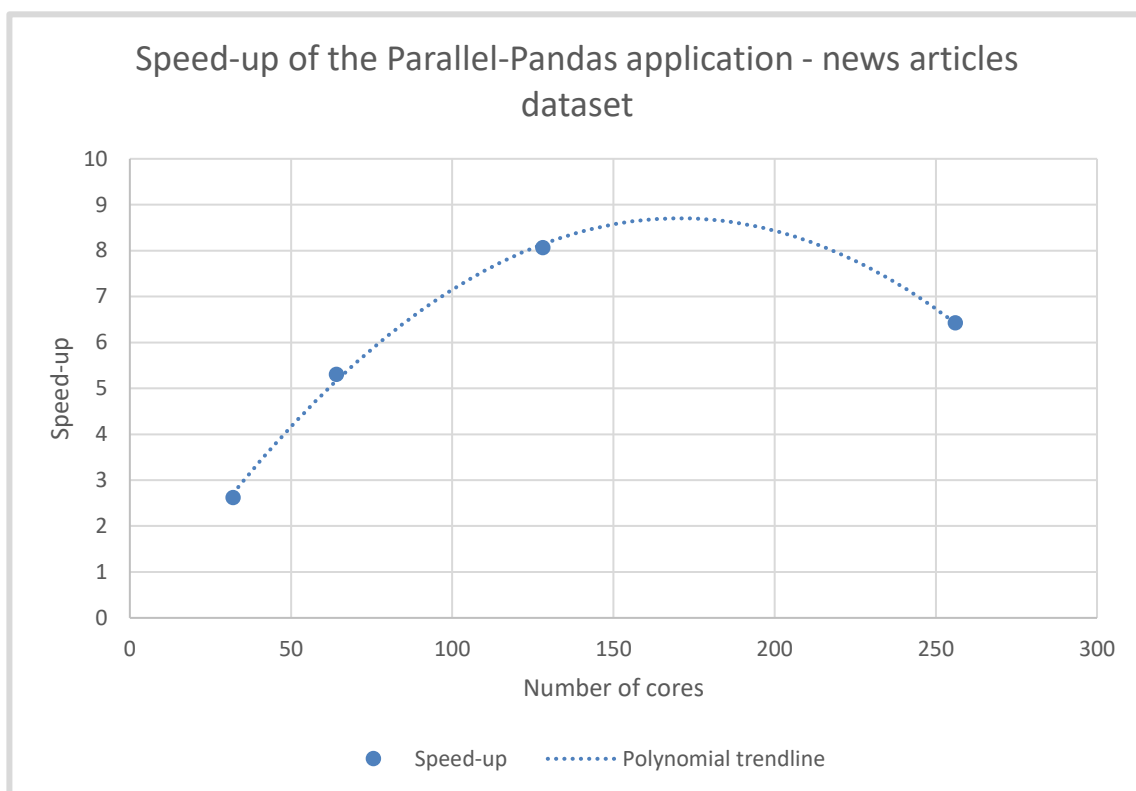


Figure 5-13: Speed-up of the Parallel-Pandas application, while using the news articles dataset

The execution times recorded for the MPI-Pandas application can be seen in Table 5-7 below. The performance improved with increasing resources. It is interesting to note, that the same pattern (as was seen in the Parallel-Pandas) of performance degradation after 128 processors

was not seen to occur for this application. This suggests that there is room for improvement in the Parallel-Pandas library and it can be further optimized.

Table 5-7: Execution time in seconds for the MPI-Pandas application using the news articles dataset

MPI-Pandas							
Processors	Run1	Run2	Run3	Run4	Run5	Average	Minimum
32	2341.401	2365.125	2346.501	2381.007	2471.338	2381.075	2341.401
64	1064.787	1078.738	1083.692	1091.392	1069.326	1077.587	1064.787
128	521.2563	520.9353	520.7503	519.0724	519.3866	520.2802	519.0724
256	485.5719	477.0978	431.3914	347.2313	427.3927	433.737	347.2313

The execution times recorded for the PySpark application can be seen in Table 5-8 below. The performance improved with increasing resources, but the improvement was not as drastic as the MPI applications. It is interesting to note that the Serial-Pandas application (1.8 hours) using a single core out-did the PySpark application, even when PySpark was using 256 cores (12.5 hours). As explained before, in the interest of time only a single measurement was taken for the PySpark application with the news article dataset.

Table 5-8: Execution time in seconds for the PySpark application using the news articles dataset

PySpark	
Executors	Run1
8	53298.28
16	48671.58
32	47387.16
64	45058.07

From a performance comparison perspective of the MPI and the PySpark applications, it is clear that MPI applications of Parallel-Pandas and MPI-Pandas outperformed the PySpark

application. The Pyspark application reduced its time from 14.8 hours with 32 processors to 12.5 hours with 256 processors. The MPI-Pandas application reduced its time from 39.6 minutes with 32 processors to 7.2 minutes with 256 processors and the Parallel-Pandas application reduced its time from 41.5 minutes with 32 processors to 13.5 with 128 processors. Figure 5-14 below, visualizes this comparison. Parallel-Pandas achieves about 98% improved execution time, with 128 processors, as compared to the PySpark application.

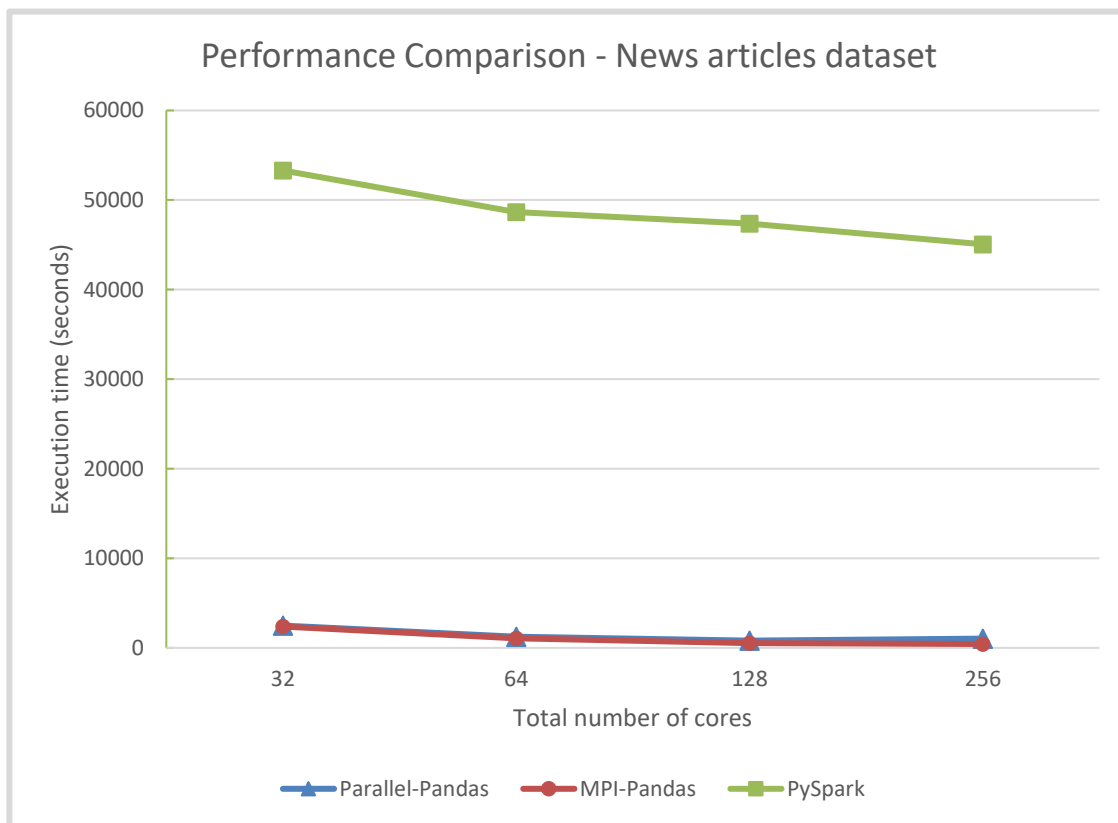


Figure 5-14: Performance comparison of duplicate detection application for the dataset containing 18,698 news articles

There is a difference in the performance of the MPI-Pandas and Parallel-Pandas. Specifically, there is 35% improvement in MPI-Pandas as compared to the Parallel-Pandas with 128

processors. The reasons of this difference have been discussed previously. This difference is visible in Figure 5-15 below. This shows that the Parallel-Pandas library can be optimized further to gain even more speed-up.

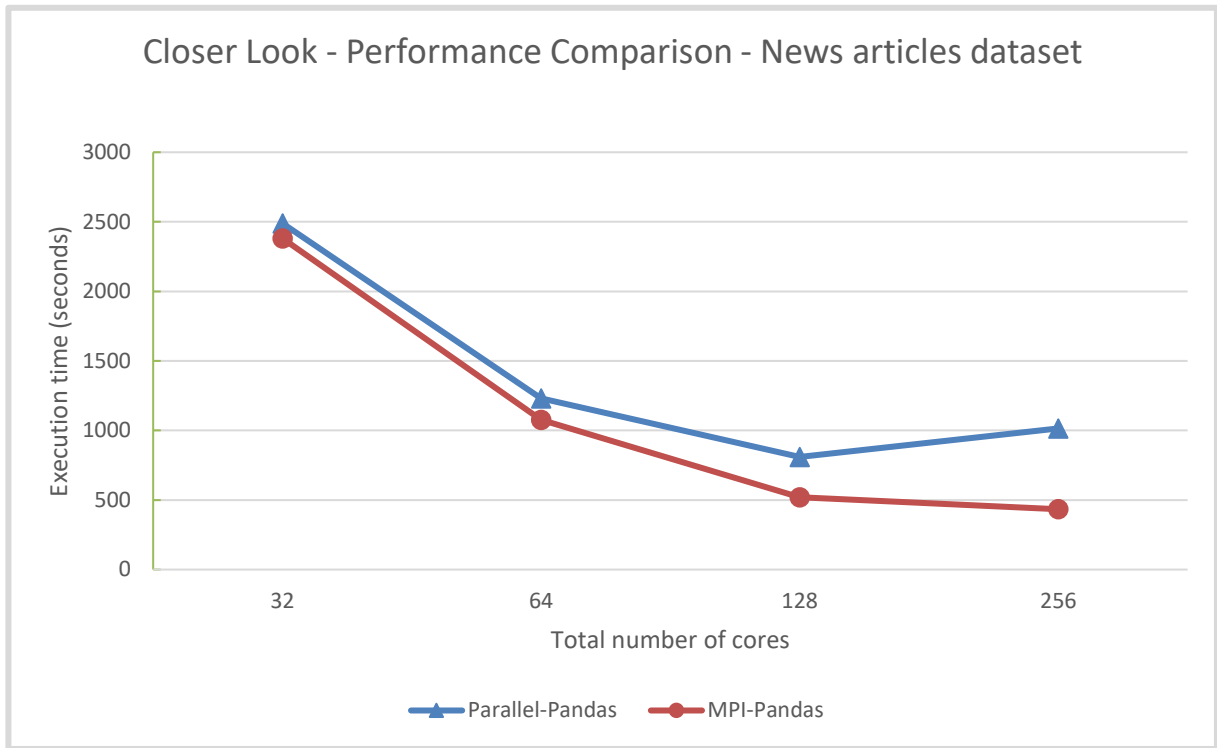


Figure 5-15: Closer look - Performance comparison of duplicate detection application for the dataset containing 18,698 news articles

6 CONCLUSIONS

This thesis introduced the Parallel-Pandas library, which is a library that aims to make existing Pandas applications more efficient by transparently providing parallel implementations of the Pandas functionality. The objective of this library is to make Pandas applications faster, only with a library import, without requiring any code changes. This thesis report detailed the design decisions and the implementation of this library.

The Parallel-Pandas library was evaluated and tested via unit-testing, micro-benchmarking as well as via a real-world document duplicate detection application. This duplicate detection application was evaluated for speed-up and efficiency with respect to the serial version. It was also compared to a PySpark implementation. The duplicate detection application was tested with two types of datasets.

The speed-up results seen for the Gutenberg dataset were much better than the speed-up results seen for the news articles dataset. For the Gutenberg dataset, the highest speedup was 60.5 with 256 processors whereas the highest speed-up for the news articles dataset was 8 with 128 processors.

Parallel-Pandas application always out-performed the PySpark application, 45% better for the Gutenberg dataset with 256 processors and 98% better for the news articles dataset with 128 processors. The optimized MPI-Pandas application always performed a little better than the Parallel-Pandas application, 29% better for the Gutenberg dataset with 256 processors and

35% better for the news articles dataset with 128 processors. The execution time for the PySpark application with 256 cores was worse than the performance of a serial Pandas application with a single core.

Based on these results, the following conclusions can be formulated:

- The Parallel-Pandas library has promising potential, greatly outperforms PySpark for the duplicate detection application.
- The Parallel-Pandas library can be optimized further to gain more speed-up, shown by improved performance of the optimized MPI-Pandas implementation.

7 FUTURE WORK

Currently in the Parallel-Pandas library, from a data distribution standpoint, the data is distributed column-wise or row-wise in the nodes of the cluster. The creation of a row-distributed dataframe is only supported by the ‘from_dict’ function, and a limited number of functions support this distribution. Naturally, a next step would be the full support of the row-distribution by the Parallel-Pandas library. This feature will make the library more versatile, flexible, optimizable and able to handle more problems.

Currently, at the beginning of the application the user must choose a distribution (or is given a column-distribution by default), and then the same distribution is used for the entire application. In order to provide more flexibility and optimization, an important feature to provide would be a conversion from row-distribution to column-distribution and vice-versa.

During this project the focus was to develop the prototype of a Parallel-Pandas library and implement an example application using this library. The focus while developing the Parallel-Pandas library was on transparency and ease-of-use by a Pandas user. The functions have been written for efficient performance however, there might be room for further optimization since that was not the main focus during this study. One of the next steps could be to study the developed functions in-depth and customize and optimize them further.

This project is a first effort of developing a Parallel-Pandas library and provides features and functions as discussed in this thesis. The next step would be to fully implement all the functionality of the Pandas library to be able to support more applications.

8 BIBLIOGRAPHY

- [1] Balaraman, Goutham. “Multi-processing with Pandas”. *Blog*, 2014.
<http://gouthamanbalaraman.com/blog/distributed-processing-pandas.html>
- [2] Bickel, Matti, Adrian Knoth, and Mladen Berekovic. “Evaluation of Interpreted Languages with Open MPI”. *Recent Advances in Message Passing Interface*, EuroMPI 2011.
- [3] Bird, Steven, Ewan Klein, and Edward Loper. “Natural Language Toolkit.” *Natural Language Processing with Python*, 2019. <https://www.nltk.org/book/>
- [4] Dalcin, Lisandro, Rodrigo Paz, and Mario Storti. “MPI for Python”. *Journal of Parallel and Distributed Computing*, 2005.
- [5] Dalcin, Lisandro, Rodgrigo Paz, Mario Storti, and Jorge D’Elia. “MPI for Python: Performance Improvements and MPI-2 Extensions”. *Journal of Parallel and Distributed Computing*, 2008.
- [6] Dask Core Developers. “Dask”. *Dask Documentation*, 2020.
<https://docs.dask.org/en/latest/>
- [7] Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large lusters.” *Communications of the ACM*, 2008.
- [8] Elsayed, Tamer, Jimmy Lin, and Douglas Oard. “Pairwise Document Similarity in Large Collections with MapReduce.” *Association of Computer Linguistics*, 2008.
- [9] Gropp, William, Ewing Lusk, Nathan Doss, and Anthony Skjellum. “A High-performance, Portable Implementation of the MPI Message Passing Interface Standard.” *Parallel Computing*, 789-828, 1996.

- [10] Hellmueller, Lea, Valerie Hase and Peggy Lindner. "Terrorism in the News: Explaining Mediated Visibility of Organized Violence." *69th Annual International Communication Association (ICA) Conference*, 2019.
- [11] Leskovec Jure, Anand Rajaraman, and Jeffrey Ullman. "Chapter 3- Finding Similar Items." *Mining of Massive Datasets*, 2014, 73-77. www.mmds.org/
- [12] Lu, Xiaoyi, Dipti Shankar, Shashank Gugnani, and Dhabaleswar Panda. "High-performance Design of Apache Spark with RDMA and its Benefits on Various Workloads." *IEEE International Conference of Big Data*, 2016.
- [13] Mckinney, Wes. "Pandas: A Foundational Python Library for Data Analysis and Statistics". *Python for High-Performance and Scientific Computing*, Volume 14, 2011.
- [14] Message Passing Interface Forum. "MPI: A Message-Passing Interface Standard Version 3.1." *MPI Documents*, June 2015. <https://www.mpi-forum.org/docs/>
- [15] "Modin." *Modin Documentation*. <https://modin.readthedocs.io/en/latest/index.html>. Accessed: 01 Feb 2020
- [16] Moritz, Phillip, Robert Nishihara, Stephanie Wang, Alexey Tumanov, et al. "Ray: A Distributed Framework for Emerging AI Applications." *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018
- [17] NumPy Developers. "NumPy." *NumPy Documentation*, 2020. <https://numpy.org/>
- [18] NVIDIA. "Cuda Toolkit". High-Performance Computing, 2020. <https://developer.nvidia.com/cuda-toolkit>
- [19] OpenMP Application Review Board. "OpenMP." *OpenMP Application Program Interface, Ver 2.5*, May 2005.

- [20] Petersohn, Devin, William Ma, Doris Lee, Stephen Macke, et al. “Towards Scalable Dataframe Systems.” [arXiv:2001.00888](https://arxiv.org/abs/2001.00888), 2020.
- [21] Project Gutenberg. Accessed: 01, August 2019. www.gutenberg.org
- [22] Python Software Foundation. “Python”. *Python Documentation*, 2020. <https://docs.python.org/3/>
- [23] Rocklin, Mathew. “Dask: Parallel Computation with Blocked Algorithms and Task Scheduling”. *14th Python in Science Conference*, SciPy 2015.
- [24] Saxena, Manvi, Shweta Jha, Saba Khan, John Rodgers, et al. “Comparison of MPI and Spark for Data Science Applications”. *Parallel and Distributed Scientific and Engineering Computing*, PDSEC 2020.
- [25] Smith, Ross. “Performance of MPI Codes Written in Python with NumPy and mpi4py”. *6th Workshop on Python for High-Performance and Scientific Computing*, 2016.
- [26] Strika, Luciano. “How to Run Parallel Data Analysis in Python Using Dask Dataframes.” *Towards Data Science*, 2018. <https://towardsdatascience.com/trying-out-dask-dataframes-in-python-for-fast-data-analysis-in-parallel-aa960c18a915>
- [27] Zaharia, Matei, Mosharaf Chowdhury, Michael Franklin, Scott Shenker, et al. “Spark: Cluster Computing with Working Sets.” *HotCloud*, 2010.
- [28] Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.” *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.