Time Series Econometrics at Scale: A Practical Guide to Parallel Computing in (Py)Spark

Benjamin Bluhm*

August 6, 2018

Abstract

This paper provides a practical programming guide to setting up a minimum working example of a distributed system for parallel time series analysis. The system is built in Apache Spark on top of Amazon's Hadoop-based service Elastic MapReduce (EMR). A simple forecasting exercise with 1,000 time series illustrates the proposed parallelization scheme, which reduces total runtime performance by about 95% relative to a single-core, single-machine setting. The ease of implementing this scheme makes this guide a useful reference for econometricians with a limited background in parallel programming. To facilitate reproducibility of the practical steps in this guide, the PySpark/Python code is available for download on github.[1]

Keywords: Time Series Econometrics, Distributed Computing, Apache Spark

1 Introduction

Time series analysis plays a crucial role in the decision making process of many public institutions and private firms. The availability of more and more data in recent years offers promising opportunities to optimize this data-driven decision process. At the same time, the need to process ever growing amounts of data poses significant computational challenges requiring new analytical tools and approaches. Real-world forecasting systems in industries including manufacturing, retail, finance and energy nowadays have to process large forecasting workloads scaling to millions of time series. The large size of these datasets requires a high degree of parallelism to enable data scientists and researchers to quickly engage in data exploration, model fitting and parameter tuning. In this context, substantial progress has been made over recent years in developing distributed systems that leverage the power of massive clusters of shared machines.

While some papers by large internet companies illustrate how to take advantage of distributed systems for large-scale parallel time series analysis, the literature provides

*Email: `benjaminbluhm@gmail.com`

[1]The github repository to reproduce the steps in this guide is available via: https://github.com/benjaminbluhm/spark_parallel_forecasting

little guidance on the technical implementation details. The objective of this paper is to bridge this gap by providing a practical programming guide to setting up a minimum working example of a distributed system suitable for parallel time series analysis and forecasting. The system builds on Apache Spark on top of Amazon's Hadoop-based service Elastic MapReduce (EMR) which supports memory and CPU-intensive parallel computations. I will illustrate the parallelization scheme, which reduces total runtime performance by about 95% relative to a single-core single-machine setting, by walking through a simple forecasting exercise applied to a dataset of 1,000 time series. The ease of implementing this scheme makes this guide a useful reference for econometricians with a limited background in parallel programming.

The remainder of the paper is organized as follows. The next section will briefly review the related work on distributed computing frameworks for time series analysis. Section 3 will elaborate on the parallel computing architecture and provide some guidance on setting up a Spark cluster on EMR. The description of the dataset and some important aspects of data partitioning and formatting are given in section 4. A simple time series forecasting example is presented in section 5, while the parallelization logic is outlined in section 6. Evidence on scalability in terms of total runtime performance is presented in section 7, before concluding the paper in section 8.

2 Related work

Different parallel systems for time series forecasting have been proposed in the literature. Stokely et al. [2011] introduce a computational infrastructure for large-scale statistical computing at *Google* using the MapReduce paradigm for R. Their technique is able to generate hundreds of thousands of forecasts in a matter of hours, using the *googleparallelism* package. An end-to-end machine learning system for probabilistic demand forecasting at *Amazon* built on Apache Spark is described in Böse et al. [2017]. The platform scales to large datasets containing millions of time series. The authors propose a trivial parallel execution scheme for what they call a *local learning* approach, using Spark's *map()* operator to distribute model fitting and forecasting tasks across the cluster. Note that the simple parallelization logic described in this practical guide follows a very similar approach. A brief review of other parallel machine learning frameworks is given by Bilenko et al. [2016].

Noteworthy, there are two libraries for distributed time series analysis in Apache Spark. The *spark-ts* package provides functionalities for fitting time series models and manipulating large time series datasets.[2] The package contains some frequently used univariate time series models, however, it is not under active development anymore and does not allow for parallel execution of algorithms not covered by the package (for example, multivariate time series models).[3] Another initiative is *Flint*, a library for highly optimized time series operations in Spark, which provides functionalities to

[2]For further details see: `https://github.com/sryza/spark-timeseries`

[3]The set of supported models is found here: `https://github.com/sryza/spark-timeseries/tree/master/python/sparkts/models`

efficiently compute across large panel and high frequency data.[4]. To the best of my knowledge, at the time of writing this guide *Flint* does not provide methods to fully parallelize all stages of the model fitting and forecasting process.

3 Parallel computing architecture

In this section, I describe the design of the high-level architecture as the basis for a distributed system and I will give some guidance on setting up a Spark cluster using Amazon's EMR service. Note that the choice of Amazon as a service provider is somewhat arbitrary and, thus, the distributed system described in this guide could be implemented on any other suitable cloud-based or on-premise Hadoop platform. Moreover, this system is not limited to the use case of time series analysis and can be applied to any expensive computing workload that can be broken up into subsets of independent tasks. For example, the parallelization scheme in this guide could be adopted to perform tasks such as value function iteration, extreme bounds analysis, forecast combination or complete subset regression. A comprehensive guide on parallelization techniques and use cases in economics is provided by Fernández-Villaverde and Zarruk Valencia [2018].

3.1 EMR architecture

The choice of the parallel computing architecture presented in this section is guided by the following goals:

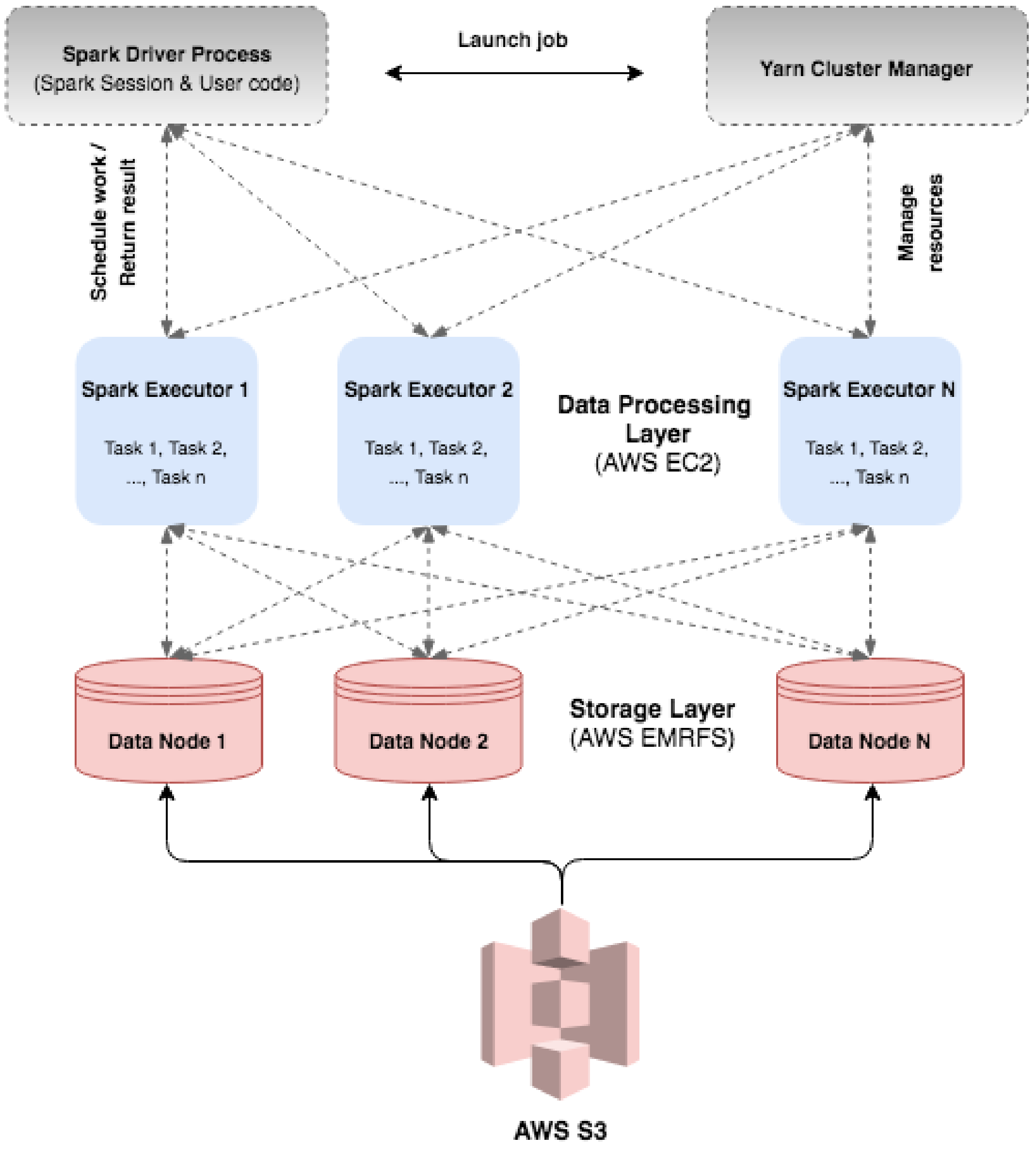
- Facilitate parallel computations on large time series datasets
- Highly scalable to large clusters of machines
- Minimal effort to start a cluster and pre-install user-defined libraries
- Use of Python as a programming language

Regarding the choice of the programming language, Spark applications can be implemented in different languages including Scala, Java, Python and R. For the purpose of this practical guide, I will use Python because it offers a variety of time series libraries and it also has become the default language for many data scientists and researchers. Nonetheless, the steps in this guide can be reproduced in any of the other languages.

A simple diagram of the Amazon EMR service architecture is illustrated in fig. 1. There are four layers, providing different capabilities and functionalities to the cluster. The storage layer uses the EMR File System (EMRFS) which contains Amazon S3 as a distributed, scalable file system, where input and output data is stored. In the Spark application of this guide, time series data and output from model fitting and forecasting will be stored in Amazon S3. Note that Amazon S3 is a persistent storage device so after terminating a cluster you still have all the data at your disposal and you also have the option to download the data to your local machine.

[4]The github repository can be found at `https://github.com/twosigma/flint`

Figure 1: Amazon EMR Architecture


Spark Driver Process
(Spark Session & User code)
Launch job
Yarn Cluster Manager
Schedule work / Return result
Manage resources
Spark Executor 1
Task 1, Task 2, ..., Task n
Spark Executor 2
Task 1, Task 2, ..., Task n
Data Processing Layer (AWS EC2)
Spark Executor N
Task 1, Task 2, ..., Task n
Data Node 1
Data Node 2
Storage Layer (AWS EMRFS)
Data Node N
AWS S3

The resource management layer uses YARN (Yet Another Resource Negotiator) and is in charge of managing cluster resources and scheduling data-processing jobs. Moreover, EMR's cluster resource manager is responsible for administering YARN components and keeping the cluster in good health.

The data processing framework layer uses Apache Spark, which was first introduced by Zaharia et al. [2010] at UC Berkeley for large-scale machine learning use cases. In the meantime, Spark has turned into an open-source, distributed data processing platform for big data workloads relating to machine learning, stream processing and graph analytics.[5] Spark is based on a master/worker architecture where the Spark driver communicates with the YARN cluster resource manager as a single coordinator which is responsible for managing the Spark workers in which executors run. The Spark driver, also known as the master node, is a Java process that hosts a Spark application and executors are Java processes that run computations and store data defined by your Spark application code (for example, a Python module for time series analysis). A more elaborate description of Spark's architecture can be found, for example, in Chambers and Zaharia [2018]. In the context of the forecasting exercise of this paper, each Spark executor processes a different subset of time series contained in the dataset stored in Amazon S3. As a result the degree of parallelism is crucially determined by the number of executors spawned by the Spark cluster. Note that this approach requires each individual time series to fit into the memory of an individual executor process which is a realistic scenario for many real-world time series datasets.[6]

Finally, the EMR cluster contains a layer for applications and programs that interact with Spark. For the use case presented in this paper, we will deploy various Python libraries to the cluster, facilitating econometric tasks as well as data exchange between the storage layer and Spark.

3.2 Setting up a Spark cluster on EMR

If you do not have an AWS account yet, you need to sign up for one. Once you have set up an account, you have to create an S3 bucket and an Elastic Compute Cloud (EC2) key pair to be able to connect to the cluster nodes via Secure Shell (SSH) protocol. These steps are described in more detail in the official AWS EMR documentation.[7] In what follows, I will only emphasize the steps and configurations in the cluster creation process which are specific to the use case in this guide and not immediate from the official documentation.

As the Python program in the following sections was developed under Spark version 2.2.1, the first step is to select a corresponding EMR release when launching the cluster. Under advanced options, select *emr-5.12.1* in the software configuration panel and make sure to check the box with *Spark 2.2.1*. Next you have to configure the hardware of your cluster including the instance type and the number of instances. While the hardware configuration strongly depends on the resource requirements (and budget considerations) of the specific use, the forecasting example in this paper is based on a general-purpose instance type, providing a balanced ratio of the number of CPUs relative to the amount of RAM.[8]

Once you have selected your preferred hardware configuration, go to the next section

[5]For further details on Spark see: https://spark.apache.org/

[6]High-frequency time series with millions of data points may be a notable exception to the above mentioned scenario. In this case, it may be more appropriate to rely on parallel frameworks like *Flint* or *spark-ts*, explicitly allowing to operate in parallel on individual time series.

[7]https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-gs-prerequisites.html

[8]A list of available instance types and prices can be found at: https://aws.amazon.com/de/ec2/pricing/on-demand/.

which asks you to specify some general cluster settings. At the bottom of the page there is a bootstrap action panel where custom actions can be specified to install additional software or to customize the configuration of cluster instances. In essence, bootstrap actions are scripts that run on all nodes after the cluster is launched. We will use the bootstrap action to install a few python libraries required for the parallel forecasting exercise. For this purpose, a shell script called *install_python_libraries.sh* with the following content must be uploaded to a folder in the S3 bucket:

Box 1: *install_python_libraries.sh*

```
#!/bin/bash -xe
sudo pip install -U pandas scipy fastparquet s3fs s3io joblib statsmodels
```

Next, add the bootstrap action by selecting *custom action* and, under the *configure and add* button, browse the S3 path to the shell script (no optional arguments needed). After adding the custom bootstrap action, move on to the last step *security settings* where you can simply follow the default instructions. Finally, the cluster can be be created.

In order to deploy the Spark application with our parallel forecasting algorithm to the master node, we have to establish an SSH connection between our local machine and the master node. To enable this connection we need to edit the security rules for the inbound traffic of the master node after the cluster has been successfully created. In particular, you need to add a new inbound security rule to the master group, setting the inbound type to SSH, the port range to 22, and the source to your local machine's IP address.[9]

Once the cluster is up and running and the security rule for SSH inbound traffic has been added, the Spark application can be deployed to the master node. One possible way to deploy an application is to use a professional deployment tool which is included in different integrated development environments (IDE). When working with Python, one good option for a deployment tool is *PyCharm* which is a commonly used IDE among Python developers.[10]

4 Dataset

This section describes the dataset and discusses the choice of data partitioning and file storage format for saving the data in the S3 bucket. Subsequently, I will walk trough a PySpark program that performs the data partitioning and storage tasks.

4.1 Description

The dataset consists of 1,000 simulated time series with each draw of length 1,000. While many real-world time series datasets are considerably larger, the dataset is sufficiently

[9]More details on controlling network traffic on your cluster can be found at: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-security-groups.html

[10]Note that you will need to install the Professional Edition of PyCharm because the community edition does not provide functionality to connect to a remote machine. A brief guide on deploying your PyCharm project to AWS is provided in the following blog: https://minhoryang.github.io/ko/posts/connect-aws-ec2-instance-with-pycharm-professional/

large to demonstrate the performance gains from parallelizing the model fitting and forecasting process. Moreover, the limited size of the dataset facilitates easy reproducibility of the steps in this guide.

The time series are simulated from an *Autoregressive Moving Average Process (ARMA)* process, defined as follows (see, for example, Hamilton [1994]):

$$\left(1 - \sum_{i=1}^{2} \alpha_i L^i\right) X_t = \left(1 + \sum_{i=1}^{2} \theta_i L^i\right) \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(\mu, \sigma^2) \tag{1}$$

where X is a real valued vector ordered by time index t, L is a lag operator, α_i and θ_i define the parameters on the autoregressive (AR) and moving average (MA) component, and ϵ_t is an independent, identically distributed disturbance term sampled from a normal distribution.

Time series draws are generated with the *arima_sim()* method in *R's stats* package (see R Core Team [2016]). Following an example in the official package documentation, the orders of the AR and MA components are restricted to two and the AR and MA coefficients α_1, α_2, θ_1, θ_2 are set to 0.89, -0.49, -0.23, 0.25 respectively. The variance σ^2 of the disturbance term is set to 0.18.[11]

The simulated time series data is written to a csv file with three columns. One column holds the time series data, a second column a unique ID for each series and a third column a sequence of numbers specifying the order of the data for each series.[12] The last column is required because Spark does not preserve the original order of records when distributing the data across the cluster. To be able to fit a time series model after processing the data in Spark we therefore need to add this column in order to recover the original ordering of the data.

4.2 Data partitioning and file storage format

Suppose you have a large time series dataset with thousands of series and you fit models only for small subsets of series on each Spark executor in a parallel fashion as described in section 3. In this setting, it is important to limit the size of the input data processed on each executor. If the input data on a single executor is (too) large this may have different negative implications. First, when the input data doesn't fit into the executor's memory the application will fail and, second, the computational overhead for reading large datasets can slow down the application significantly, potentially eliminating some of the performance gains from parallelization. Thus, it makes sense to break up a large dataset into smaller chunks where each chunk only contains a subset of all items. In a distributed file system, typically these smaller data chunks are stored in a separate directory where the directory name contains an identifier for this chunk. In Spark, the

[11] For further details see: `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/arima.sim.html`

[12] In analogy to a timestamp or date in a real-world time series dataset.

logic of splitting the dataset into smaller pieces is defined by one or multiple partitioning columns, using the *partitionBy()* method.

Another aspect of importance is the storage format of the dataset. The example in this guide uses Apache Parquet which is a free and open-source column-storage format of the Apache Hadoop ecosystem.[13] Using Parquet as opposed to traditional storage formats offers some important benefits. As Parquet is known to provide an efficient data compression and encoding scheme, data volume will be substantially lower and data query runtimes will be much faster compared to, for example, csv format. The latter aspect crucially determines the performance gains from parallelization and, thus, also the efficient use of cluster resources.

4.3 Preparing the dataset in Spark

Finally, let's take a look at the PySpark program that creates a Spark session and performs the data preparation tasks, i.e. loading the time series dataset from a csv file into a Spark dataframe and writing it back to S3 in Parquet file format. A Spark session must be created first to be able to load the time series data into Spark, using the *create_spark_session.py* module:[14]

Box 2: *create_spark_session.py*

```python
# Import Python modules
import os
import sys

# Set path
os.environ['SPARK_HOME'] = '/usr/lib/spark'
sys.path.append('/usr/lib/spark/python')
sys.path.append('/usr/lib/spark/python/lib/py4j-0.10.4-src.zip')

# Import PySpark modules
from pyspark.sql import SparkSession

def create_spark_session():

    spark_session = SparkSession.builder.appName('spark_parallel_forecasting')\
        .master('yarn').getOrCreate()

    return spark_session
```

The function call *create_spark_session()* returns a Spark session, encapsulating a Spark context. The Spark context acts as the master of your application and can be used to create distributed entities such as Resilient Distributed Datasets (RDD), a fault-tolerant collection of elements which can be operated on in parallel. The Spark session serves as the entry point to programming in Spark with the Dataset and DataFrame API.

After creating a Spark session, we load a dictionary with some basic configurations, including file path definitions and the AWS S3 endpoint used to establish a connection between the Spark application and the S3 file system:

Box 3: *create_config.py*

[13] For further details on Apache Parquet see: https://parquet.apache.org/documentation/latest/

[14] Note that all Python modules presented in this guide are located in the following directory on the master node: *'/home/hadoop/spark_parallel_forecasting'*.

```
def create_config():

    config = {}

    # Define Path
    config['base_path_hadoop'] = '/home/hadoop/spark_parallel_forecasting/'
    config['base_path_s3'] = 's3://data-folders/spark_parallel_forecasting/'
    config['path_training_data_csv'] = config['base_path_s3'] + 'training_data/csv/rawdata.csv'
    config['path_training_data_parquet'] = config['base_path_s3'] + 'training_data/parquet/'
    config['path_forecasts'] = config['base_path_s3'] + 'forecasts/'
    config['path_models'] = config['base_path_s3'] + 'fitted_models/'

    # Define AWS S3 endpoint for your region
    config['s3_host'] = 's3.eu-central-1.amazonaws.com'

    # Define series and evaluation length
    config['len_series'] = 1000
    config['len_eval'] = 50

    return config
```

Now, the dataset can be prepared by calling the *partition_and_save_dataset()* function. After loading the csv file into a Spark dataframe, the dataset is partitioned by the *ID* column and, thus, a separate Parquet file and directory is created for each time series. Note that other partitioning schemes may be more suitable depending on the use case. For example, if the goal is to jointly model subsets of time series, it would be advisable to allocate these item subsets into one partition:[15]

Box 4: *partition_and_save_dataset.py*

```
def partition_and_save_dataset(spark_session, config):

    df = spark_session.read.option("header", "true")\
        .csv(config['path_training_data_csv'], inferSchema=True)

    df.repartition("ID").write.option("compression", "gzip").mode("overwrite")\
        .partitionBy("ID").parquet(config['path_training_data_parquet'])
```

All main tasks that the program will perform are contained in the *main.py* module:

Box 5: *main.py*

```
# Import Python modules
from create_spark_session import create_spark_session
from create_config import create_config
from partition_and_save_dataset import partition_and_save_dataset


def main():

    # Create Spark session
    spark_session = create_spark_session()

    # Load config dictionary
    config = create_config()

    # Partition and save dataset in Parquet file format to S3
    partition_and_save_dataset(spark_session, config)

if __name__ == '__main__':
    main()
```

[15]As a data compression type, *gzip* is used. Note that I do not investigate the performance impact of the compression type. The other two available Parquet compression types are *snappy* and *uncompressed*.

5 A simple forecasting example

Given a total sample size of 1,000 for each time series, the last 50 observations of the sample are used for forecast evaluation. The first 950 observations are used to fit an initial *ARMA(2,2)* model which is then used to produce the first forecast. I use a recursive estimation scheme, i.e. the size of the estimation sample for model fitting is extended by one observation as one makes forecasts for successive observations. As a result, a total of 50,000 estimations is performed across all time series in the dataset. The forecasts as well as the final model, fitted on the full sample for each time series, are stored in the S3 bucket. For the sake of simplicity, only one-step ahead forecasts are generated. However, the Python module presented in this section can easily be extended to include additional models and forecasts for multiple horizons.

While the forecasts are stored in Parquet file format, the final model is saved as a *pickle* object which is a standard format for model persistence in Python.[16] For model fitting and forecasting, I use Python's *StatsModels* module (see Seabold and Perktold [2010]), providing a class for fitting *ARMA* models via maximum likelihood. The *fastparquet* module, which is a Python interface to the Parquet file format, is used to read and write Parquet files in the S3 bucket from Python. The fitted model objects are saved via the *pickle* module which implements a protocol for serializing and de-serializing Python objects. The set of Python modules installed on the cluster is complemented by the modules *pandas* (see McKinney [2010]), *scipy* (see Jones et al. [2001]), *s3fs*, *s3io*, *joblib*.[17]

A minimum working example of the time series forecasting algorithm is provided in the *fit_model_and_forecast.py* module below:

Box 6: *fit_model_and_forecast.py*

```
# Import python modules
import s3fs
import joblib
import s3io
import boto
import pandas as pd
import numpy as np
from fastparquet import ParquetFile, write
from statsmodels.tsa.arima_model import ARMA

def fit_model_and_forecast(id_list, config):

    # Cast collection of distinct time series IDs into Python list
    id_list = list(id_list)

    # Open connections to S3 File System
    s3 = s3fs.S3FileSystem()
    s3_open1 = s3.open
    s3_open2 = boto.connect_s3(host=config['s3_host'])

    # Loop over time series IDs
    for i, id in enumerate(id_list):

        # Determine S3 file path and load data into pandas dataframe
        file_path = s3.glob(config['path_training_data_parquet'] + 'ID=' + str(id) +
                            '/*.parquet')
        df_data = ParquetFile(file_path,open_with=s3_open1).to_pandas()
```

[16]See for example: `http://scikit-learn.org/stable/modules/model_persistence.html`

[17]The *pandas* module is used to the store forecasts in a dataframe and *scipy* is needed as a dependency for the *StatsModels* module. The other three mentioned modules are required to manage connections and enable file exchange between S3 and Python.

```
        # Sort time series data according to original ordering
        df_data = df_data.sort_values('ORDER')

        # Initialize dataframe to store forecast
        df_forecasts = pd.DataFrame(np.nan, index=range(0, config['len_eval']),
                                    columns=['FORECAST'])

        # Add columns with ID, true data and ordering information
        df_forecasts.insert(0, 'ID', id, allow_duplicates=True)
        df_forecasts.insert(1, 'ORDER', np.arange(1, config['len_eval'] + 1))
        df_forecasts.insert(2, 'DATA', df_data['DATA'][range((config['len_series'] -
                                                          config['len_eval']),
                                                         config['len_series'])].values,
                                                         allow_duplicates=True)

        # Loop over successive estimation windows
        for j, train_end in enumerate(range((config['len_series'] - config['len_eval'] - 1),
                                            (config['len_series'] - 1))):

            # Fit ARMA(2,2) model and forecast one-step ahead
            model = ARMA(df_data['DATA'][range(0, train_end+1)], (2, 2)).fit(disp=False)
            df_forecasts.at[j, 'FORECAST'] = model.predict(train_end+1, train_end+1)

        # Write dataframe with forecast to S3 in Parquet file format
        path = config['path_forecasts'] + 'ID=' + str(id) + '.parquet'
        write(path, df_forecasts, write_index=False, append=False, open_with=s3_open1)

        # Save fitted ARMA model to S3 in pickle file format
        path = config['path_models'] + 'ID=' + str(id) + '.model'
        with s3io.open(path, mode='w', s3_connection=s3_open2) as s3_file:
            joblib.dump(model, s3_file)
```

Note that the function *fit_model_and_forecast()* takes an argument called *id_list*, containing a collection of unique time series IDs. In a non-parallel setting, this collection is simply a Python list with distinct IDs.

The outer loop in the function iterates over the list of IDs, using the ID information to read the time series data via *fastparquet* from the corresponding directory in the S3. The inner loop contains the statements for model fitting and forecasting, iterating recursively over the estimation sample. The last two blocks of code are responsible for storing the fitted model and forecasts respectively.

In order to execute the forecasting algorithm in a non-distributed fashion, we simply import the *fit_model_and_forecast.py* module into another module called *do_non_parallel_forecasting.py* which generates a list of distinct IDs from our time series dataset and calls the *fit_model_and_forecast()* function:

Box 7: *do_non_parallel_forecasting.py*

```
from fit_model_and_forecast import fit_model_and_forecast

def do_non_parallel_forecasting(spark_session, config):

    # Load time series data into Spark dataframe
    df = spark_session.read.parquet(config['path_training_data_parquet'])

    # Create RDD with dictinct IDs
    time_series_ids = df.select("ID").distinct().rdd

    # Create Python list with dictinct IDs
    time_series_ids = [int(i.ID) for i in time_series_ids.collect()]

    # Perform non-parallel forecasting
    fit_model_and_forecast(time_series_ids, config)
```

We can now extend our *main.py* program to include a task for non-distributed forecasting by calling *do_non_parallel_forecasting()*:

Box 8: *main.py*

```
# Import Python modules
from create_spark_session import create_spark_session
from create_config import create_config
from partition_and_save_dataset import partition_and_save_dataset
from do_non_parallel_forecasting import do_non_parallel_forecasting

def main():

    # Create Spark session
    spark_session = create_spark_session()

    # Load config dictionary
    config = create_config()

    # Partition and save dataset in Parquet file format to S3
    partition_and_save_dataset(spark_session, config)

    # Perform non-parallel model fitting and forecasting
    do_non_parallel_forecasting(spark_session, config)

if __name__ == '__main__':
    main()
```

When running this program, the forecasting algorithm will be executed only on the master node of the cluster, i.e. the model fitting and forecasting task is not distributed to the Spark executors. The runtime of this program is taken as a performance benchmark against which the distributed forecasting algorithm can be evaluated.

6 Parallelization

The parallelization of the forecasting algorithm presented in the previous section only requires a few additional programming steps in PySpark. First, we create a new Python module called *do_parallel_forecasting.py*:

Box 9: *do_parallel_forecasting.py*

```
def do_parallel_forecasting(spark_session, config):

    # Load time series data into Spark dataframe
    df = spark_session.read.parquet(config['path_training_data_parquet'])

    # Create RDD with distinct IDs and repartition dataframe into 100 chunks
    time_series_ids = df.select("ID").distinct().repartition(100).rdd

    # Function to import model Python module on Spark executor for parallel forecasting
    def import_module_on_spark_executor(time_series_ids, config):
        from fit_model_and_forecast import fit_model_and_forecast
        return fit_model_and_forecast(time_series_ids, config)

    # Parallel model fitting and forecasting
    time_series_ids.foreach(lambda x: import_module_on_spark_executor(x, config))
```

In the above code, we first load the time series data into a Spark dataframe and create a partitioned RDD with distinct time series IDs. By calling the *repartition()* function, the number of RDD partitions is set to 100, cutting the distributed collection of IDs into 100 chunks. Given that we have 1,000 time series in our dataset, the average number of IDs in each partition is 10.[18] Since Spark will run one task for each partition, the number of RDD partitions is an important parameter that determines the degree of parallelism of your Spark application. In this particular example, the 100 RDD partitions can be

[18]Note that Spark does not automatically distribute the number of elements evenly across partitions. Therefore, it is likely that some partitions contain more and others contain less than 10 elements.

processed in parallel as long as there are sufficient cluster resources available in terms of the number of Spark executors and executor memory.

To apply our *fit_model_and_forecast.py* module to each RDD partition in a distributed fashion, we need to make sure that the module is imported by each Spark executor, before calling the function inside the module. Calling the function *import_module_on_spark_executor()* as part of the *foreach()* operation in the last statement of the *do_parallel_forecasting.py* module ensures that each Spark executor imports the *fit_model_and_forecast.py* module.

In order to make the *fit_model_and_forecast.py* module available to all Spark executors, we need to add the module to the Spark context by calling the *addPyFile()* function that takes as an argument the file path to the module. To finally execute our forecasting algorithm in a parallel fashion, we extend our *main.py* program as follows:

Box 10: *main.py*

```
# Import Python modules
from create_spark_session import create_spark_session
from create_config import create_config
from partition_and_save_dataset import partition_and_save_dataset
from do_parallel_forecasting import do_parallel_forecasting


def main():

    # Create Spark session
    spark_session = create_spark_session()

    # Load config dictionary
    config = create_config()

    # Partition and save dataset in Parquet file format to S3
    partition_and_save_dataset(spark_session, config)

    # Add Python module to Spark context for distributed model fitting and forecasting
    spark_session.sparkContext.addPyFile(config['base_path_hadoop'] +
                                         'fit_model_and_forecast.py')

    # Perform parallel model fitting and forecasting
    do_parallel_forecasting(spark_session, config)

if __name__ == '__main__':
    main()
```

7 Empirical runtime performance

This section provides experimental results of the runtime performance for the forecasting example described in sections 5 and 6. Table 1 shows the runtime for two different execution schemes. In the first scenario, the forecasting algorithm is executed on the master node in a non-parallel fashion and, thus, mirrors a single-core single-machine execution scheme. This scenario is used as a benchmark case to evaluate the performance gain from the parallel execution scheme.

The cluster hardware has been configured to 13 EC2 instances of type *m4.2xlarge*, comprising a total of 192 virtual CPUs and 384 GiB of RAM for the 12 worker nodes. The number of RDD partitions containing collections of distinct time series IDs is set to 100. Table 1 shows the runtime results for the two different scenarios.

Table 1: Runtime for different execution schemes

Scenario	**Parallel**	**vCPU**	**RAM**	**Partitions**	**Runtime**
1	no	16	32	-	12076
2	yes	192	384	100	372

The results are based on an AWS EC2 instance type *m4.2xlarge*. Runtime is measured in seconds, RAM is measured in GiB, *vCPU* refers to *virtual CPU* and *Partitions* defines the number of RDD partitions, containing subsets of distinct time series IDs.

The total runtime for the non-distributed scheme is about 200 minutes. This compares to roughly 6 minutes execution time for the parallel scheme, reducing runtime by about 95%. Clearly, the runtime of the parallel scheme is strongly affected by the hardware configuration and the number of RDD partitions. An increase in the number of RDD partitions and a more powerful cluster with more CPUs and memory will most likely lead to even higher performance gains. While the impact of different hardware settings on the performance gain is beyond the scope of this paper, the results show that the parallelization scheme can be used to complete large model fitting and forecasting workloads that would be intractable without substantial parallelization.

8 Conclusions

This paper introduced a step-by-step practical guide for setting up a minimum working example of a distributed system for time series analysis and forecasting. The system is built in Apache Spark and the parallelization scheme is suitable (but not limited to) parallel computations on large time series datasets. A simple forecasting exercise illustrates that the parallelization scheme reduces total runtime performance substantially relative to a single-machine setting. The presented approach requires minimal installation and configuration effort and it can be implemented with little background in computer science and parallel programming.

References

Mikhail Bilenko, Tom Finley, Shon Katzenberger, Sebastian Kochman, Dhruv Mahajan, Shravan Narayanamurthy, Julia Wang, Shizhen Wang, and Markus Weimer. Salmon: Towards production-grade, platform-independent distributed ml. In *The ML Systems Workshop at ICML*, 2016.

Joos-Hendrik Böse, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Dustin Lange, David Salinas, Sebastian Schelter, Matthias Seeger, and Yuyang Wang. Probabilistic demand forecasting at scale. 10:1694–1705, 08 2017.

B. Chambers and M. Zaharia. *Spark - The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media, Incorporated, 2018. ISBN 9781491912218.

Jesús Fernández-Villaverde and David Zarruk Valencia. A Practical Guide to Parallelization in Economics. CEPR Discussion Papers 12890, C.E.P.R. Discussion Papers, April 2018.

James Douglas Hamilton. *Time series analysis*. Princeton Univ. Press, Princeton, NJ, 1994. ISBN 0691042896.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001.

Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016.

Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*, 2010.

Murray Stokely, Farzan Rohani, and Eric C Tassone. Large-Scale Parallel Statistical Forecasting Computations in R. *JSM Proceedings, Section on Physical and Engineering Sciences, American Statistical Association*, 2011.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, Berkeley, CA, USA, 2010. USENIX Association.