Journal of Data Science 20 (3), 413–436
July 2022
DOI: 10.6339/22-JDS1035
Data Science Reviews

Econometrics at Scale: Spark up Big Data in Economics☆

Benjamin Bluhm[1] and Jannic Alexander Cutura[2,*]

[1] *Senior Data Scientist*

[2] *European Central Bank, Sonnemannstraße 20, 60314 Frankfurt am Main, Germany*

Abstract

This paper provides an overview of how to use "big data" for social science research (with an emphasis on economics and finance). We investigate the performance and ease of use of different Spark applications running on a distributed file system to enable the handling and analysis of data sets which were previously not usable due to their size. More specifically, we explain how to use Spark to (i) explore big data sets which exceed retail grade computers memory size and (ii) run typical statistical/econometric tasks including cross sectional, panel data and time series regression models which are prohibitively expensive to evaluate on stand-alone machines. By bridging the gap between the abstract concept of Spark and ready-to-use examples which can easily be altered to suite the researchers need, we provide economists and social scientists more generally with the theory and practice to handle the ever growing datasets available. The ease of reproducing the examples in this paper makes this guide a useful reference for researchers with a limited background in data handling and distributed computing.

Keywords *Apache Spark; distributed computing; econometrics*

1 Introduction

Research in economics and finance moved towards ever larger datasets and computationally more advanced methods, including statistical methods borrowed from the *machine learning* (ML) literature (Hamermesh, 2013; Einav and Levin, 2014). A growing number of papers discusses the potential and limits of modern data analysis frameworks such as ML algorithms (Varian, 2014; Mullainathan and Spiess, 2017), text as data (Gentzkow et al., 2019; Grimmer and Stewart, 2013) and the progress on inference methods in high-dimensional settings (Athey and Imbens, 2017; Kleinberg et al., 2015). Finally, Fernández-Villaverde and Valencia (2018) provide an introduction to parallel computing for economics. While most of these works describe the underlying algorithms at great length, there is almost no guidance on how to handle the typically very large datasets used for these tools. As data availability is very likely to grow in the foreseeable future, the inability to handle big data sets poses a severe challenge to empirical economic research.

This paper aims to fill this gap by providing accessible guidance on how to use distributed computing solutions for economic research. With datasets in the billions of observations (e.g. Cavallo and Rigobon (2016); Gao et al. (2019)) and peta-bytes of data (e.g. Ng (2017)) and growing, economists need to be able to handle and analyse those in a practical and efficient manner. We provide such a framework by showing how to run your existing data handling

☆The views expressed in this paper are those of the authors alone and do not represent the view of the European Central Bank (ECB).

*Corresponding author. Email: jannic.cutura@ecb.europa.eu.

© 2022 The Author(s). Published by the School of Statistics and the Center for Applied Statistics, Renmin University of China. Open access article under the CC BY license.

Received November 9, 2021; Accepted January 12, 2022

pipeline on a distributed computing solution. More specifically we illustrate the parallelization of key tasks frequently encountered in economic research and policy analysis: (i) computing summary statistics, (ii) estimating micro econometric models, (iii) panel data models and (iv) time series models using the *Apache Spark* framework. The paper specifically targets a non-expert audience and is therefore useful for researchers in economics and social sciences more general who struggle with datasets larger than their in house resources allow them to handle.

The basic idea behind Spark is that instead of *bringing the data to the computation* (i.e. read the data from your hard drive into your computers memory) you should *bring the computation to the data* (i.e. run several computations in parallel on the machines where the different parts of the dataset are stored). This allows to handle datasets which are much larger than your computers memory usually allows to handle. There are several providers for cloud computing solutions providing a rich ecosystem of tools for distributed data storage and processing including Spark. In this paper we use *Amazon Web Services* (AWS), but the logic described seamlessly extends to other platforms.

To illustrate the use of Spark for economists, we demonstrate four typical use cases. First, we handle and pre-process a real-world dataset of US home mortgage applications with nearly 140 million observations using R's *sparklyr* library, which is built on the popular *dplyr* library providing an efficient and intuitive approach for data pre-processing. We then use this dataset to illustrate the estimation of various micro econometric models where we provide standard regression output tables and information on runtime performance conditional on cluster resource constraints. Next, we use Python's *pyspark* API to fit a static fixed effects model via within-group data transformation using a simulated panel dataset with one billion observations. In this respect we show how to compute panel robust standard errors using a simple customized distribution scheme. Finally, we provide an example that entails forecasting a large number of time series in parallel.

Fernández-Villaverde and Valencia (2018) conclude that Python and R are inferior to higher level programming languages like C++ and Julia, when it comes to run-time performance based on their comparison of value function iteration. Our results indicate, that for empirical economic research Python and R by using Spark are well equipped for data handling and analysis of very large datasets. As Python and R are considerably easier to learn than e.g. C++ (and in fact today's working standard in data science), we view our introduction to Spark (based on Python and R) as a useful guide for economists who want to analyse datasets larger than their computer's memory allows. Our results in terms of runtime and ease of handling suggest Spark is a suitable tool for economic research. Using an *Elastic Map Reduce* (EMR) setup we are able to pre-process a 150 GB dataset in just under five minutes, whereas the standalone approach on our local machine crashes. Similarly, estimating micro econometric and panel regression models on our local machine would require computing crossproducts of very large arrays, which is not feasible on retail grade computers. Moreover, for the time series analysis case, the distribution scheme reduces total runtime performance by about 95% relative to a single-machine setting.

The contribution of the paper is two-fold. First, we demonstrate the usage of Spark for economic research and its superiority in fact for many applications involving large datasets. Secondly, the intuitive explanation of the framework along side the uses cases should enable economists without previous experience in parallel computing to work with Spark. This will allow them to (i) easily migrate their *existing* data handling and analysis to gain significant run time performance and (ii) allow them to handle datasets which were previously not manageable. To ease the process, we provide the codes used in this paper and (in supplementary material) carefully explain how to connect to a cloud service to run the codes. Moreover, we show how

you can easily develop, test and debug your Spark programs (written in Python and R) on your local machine (avoiding paying fees for cloud services).

The remainder of the paper is organized as follows. The next section will briefly motivate the use of distributed computing solutions for empirical economic research. Section 3 will elaborate on the distributed computing architecture and provide a few numerical examples to illustrate the key idea of the Spark framework. Section 4 starts with a description on how to handle and preprocess a large real-world dataset followed by subsection 4.2 which uses this dataset to walk through the estimation of micro econometric models in Spark. Subsection 4.3 shows how to implement a fixed effects regression in Spark and how to obtain panel robust standard errors. The last subsection 4.4 shows a distributed set up for estimating time series models. The last section concludes.

2 Why Distributed Computing?

Parallel Computing has gained a lot attention and is today used across various fields in economic research, in particular for solving highly complex quantitative models. In this paper, we argue that *Distributed Computing* is the next step in the evolution of computational methods for economic research. The major advantage of parallel computing is that it can allow to solve quantitative problems, that were previously prohibitive expensive to evaluate. Fernández-Villaverde and Valencia (2018) provide an excellent overview of parallel computing performance of various programming languages and illustrate the runtime gains for solving a standard value function iteration problem. They point out that (depending on your problem) you can speed up the analysis by a factor equal to the number of your computer's CPU cores. A standard retail grade computer at the time of writing typically comes with 4–8 cores, which means you can speed up the performance of your computations by up to 8 times, if you use parallel computing appropriately. As pointed out in Fernández-Villaverde and Valencia (2018), that very same logic holds true for the case of distributed computing with that exception that instead of being able to only use all your computer's CPU cores, you can use hundreds and even thousands of CPU cores of a cloud computing framework. Therefore, while parallel computing on your own machine certainly speeds up the process of many applications, it pales in comparison with the performance of a Spark cluster.

Secondly, while your own computer comes at a (potentially high) fixed cost and is only used for a certain number of hours a day, the access to a cloud computing instance can be turned on and off at the flick of a switch. From a societal point of view, this will save resources: Instead of individuals owning powerful machines they only use so many hours a day, one can buy runtime on centralized high performance cloud computers.

Thirdly, and most importantly for the scope of this paper, distributed computing allows to tackle data handling and analysis which were previously prohibitively expensive to run. For empiricists, this is usually the case when the dataset under consideration is considerably larger than your computers memory, making any analysis on it painfully slow or when the number of models which need to be evaluated becomes so large that even parallelization on a powerful retail-grade computer does not alleviate overall runtime concerns. Moreover, the researchers training with data handling and analysis shapes and limits the kind of research questions she conceives of in the first place. By lowering the threshold to use distributed computing solutions for economic research, we hope to achieve two goals. Firstly, we enable social scientists to approach their existing (big) data handling more efficiently and tackle *existing* questions that were previously

prohibitively expensive to run. Secondly, by demonstrating the ease of use of cloud computing technologies, we hope to inspire *new* questions which leverage the possibilities of ever growing and ever more accessible datasets in the future.

3 Distributed Computing Architecture

3.1 General Overview and Cluster Architecture

In this section, we describe the core architecture of a distributed computing system based on Spark. The system is not limited to solving large-scale data handling and econometric tasks as described in this guide, but can be applied to many other expensive computing workloads that can be broken up into subsets of independent tasks. To only mention a few use cases from an economist's perspective, the distributed system in this guide could be adopted to distribute tasks such as for example value function iteration (Aruoba et al., 2003), extreme bounds analysis (Leamer, 1985; Sala-I-Martin, 1997), forecast combination (Clemen, 1989; Timmermann, 2006) or hyperparameter search.

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

- Facilitate distributed computations on datasets which do not fit into a single machine's memory
- Highly scalable to large clusters of machines
- Minimal effort for setting up a cluster and pre-installation of user-defined libraries
- Ease of use to handle and analyse data in a distributed fashion using your existing Python and R data analysis pipelines (Other statistical packages like Stata or eViews unfortunately do not offer any Spark interfaces. Matlab, another popular computing language, recently started to offer Spark/Hadoop support. Java and Scala also offer interfaces, but are less known among social scientists)

A simple diagram of the distributed computing architecture is illustrated Figure 1. There are four layers, providing different capabilities and functionalities to the cluster. The distributed storage layer is based on the Hadoop API and uses Amazon S3 as a distributed, scalable file system, where input and output files from the application are stored on multiple machines, each storing a subset of all files. Hadoop scales to hundreds or even thousands of machines and therefore supports applications that run on very large datasets. A key idea of Hadoop (Ghemawat et al., 2003; Dean and Ghemawat, 2004) is to move the computation to the data (and not vice versa) in order to minimize network congestion which yields large benefits in terms of computational efficiency for huge datasets of gigabytes to terabytes in size.

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

The data processing layer uses 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. The *Resilient Distributed Dataset* (*RDD*) defines the core component of Spark's distributed data processing engine. *RDD*s are collections of lazily evaluated, distributed data objects – also called partitions – which are stored in the data nodes connected to the Spark worker nodes and can be manipulated in a

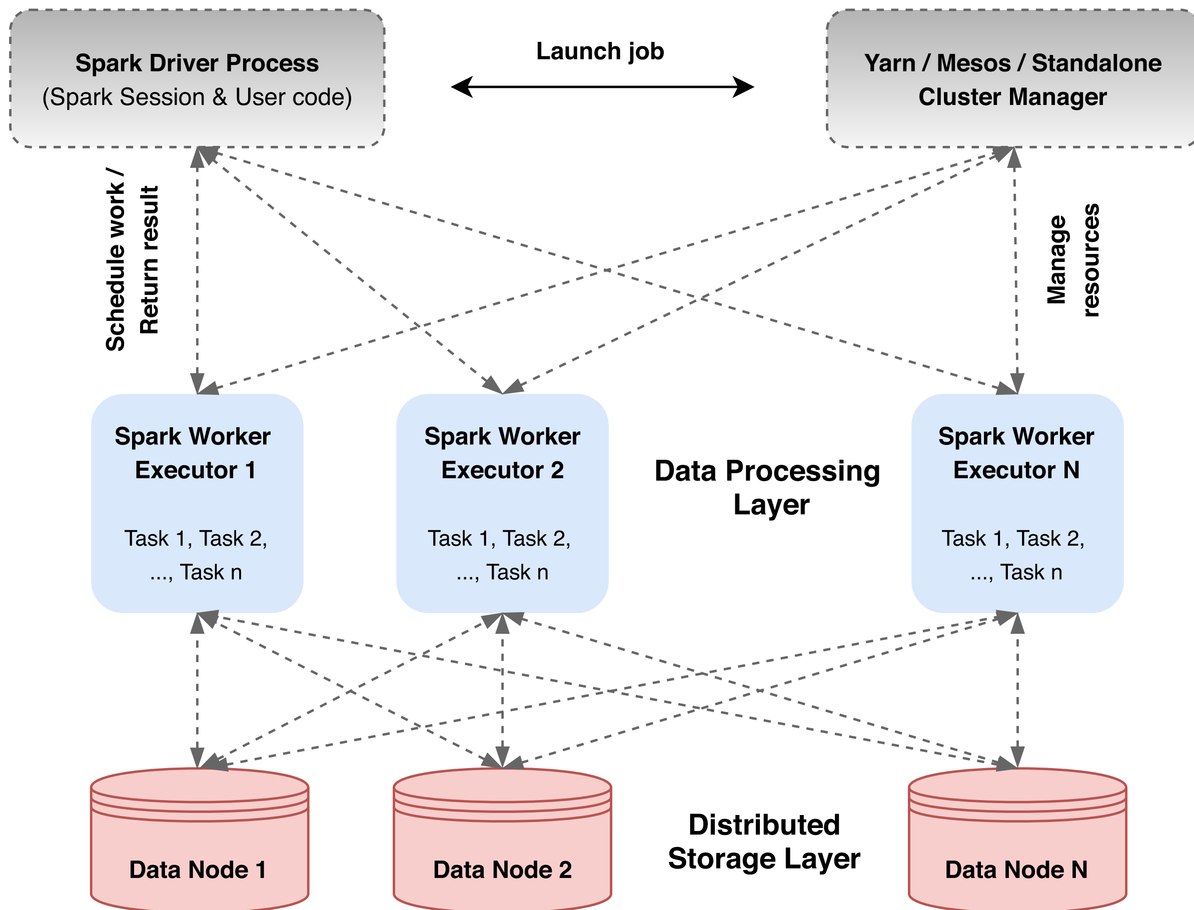

Figure 1: **Spark's distributed computing architecture.** This schema illustrates a distributed computing architecture. When the user submits a Spark application it launches the *Spark Driver* which is the process that takes care of breaking down the user program into individual tasks and coordinating each of these tasks on the *Spark Executors*. The *Spark Driver* submits a resource request to the *Cluster Manager* which launches the *Spark Executors* according to the requested resources. The *Spark Executors* perform the tasks received from the *Spark Driver*. The *Distributed Storage Layer* is based on the Hadoop API and holds the distributed dataset which is partitioned across harddrives of the Spark worker nodes. Distributed datasets can be used on any Hadoop supported storage system including for example Hadoop Distributed File System (HDFS), S3, Cassandra, Hive and HBase. Authors graph based on Karau et al. (2015), Karau and Warren (2017) and Samadi et al. (2018).

parallel fashion on the different executors of the system. Spark is based on a master/worker architecture where the driver communicates with the cluster manager as a single coordinator which is responsible for managing the workers in which executors run. The Spark driver is a process that hosts a Spark application and executors are processes that run computations and store data defined by your application code (for example, a *sparklyr* program for micro econometric analysis). A more elaborate description of the Spark architecture can be found, for example, in Chambers and Zaharia (2018).

3.2 The map-reduce Framework

In this section we illustrate how Spark manipulates data in a parallel fashion using two simple examples. The first example takes as an input an *RDD* with key/value pairs to highlight a map-reduce algorithm that returns the mean value for each key. In the second example we briefly sketch how Spark is applied to approximate the maximum likelihood estimate of a generalized linear model.

Example 1 – A Simple Case for Distributed Computing: Compute the Mean of a Large Data Set

Perhaps the simplest example to demonstrate the map-reduce framework is to compute the average value for each group for a large data set. Consider a data set D = [('A', 5), ('B', 7), ('C', 3), ('D', 4), ('A', 8), ('B', 6), ('C', 2), ('D', 1), ('A', 9), ('B', 3)]. Figure 2 illustrates the map-reduce logic used in Spark. As an input it takes an *RDD* with 10 key/value tuples where the capital letters define the keys. The map function takes as an input the *RDD* with key/value pairs which is distributed across three partitions in this example. The reduce function is called once for each key, takes the input values to compute the average value and returns a key/value tuple. Note that the data is shuffled between the map and reduce stage to ensure that all values of a given key share the same node. After the reduce step is completed, the data is transferred back to the master node, where we can find the output [('A', 7.33), ('B', 5.33), ('C', 2.5), ('D', 2.5)], i.e. the averages for each group A, B, C and D. We can simultaneously count the number of observations per group such that in a second step one can compute a weighted average of the group averages, weighted with the number of observations in each group to obtain the overall

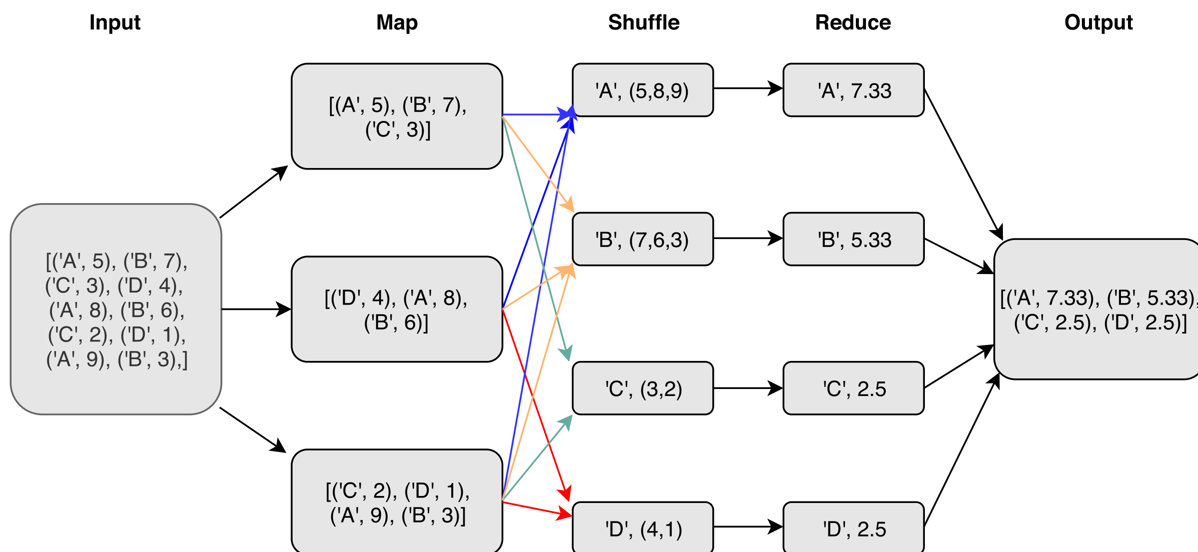

Figure 2: **Distributed computation of the mean.** This illustrates a simple map-reduce logic for computing the average of a list of numbers. The data is first (randomly) *mapped* across workers. In a second step it is *shuffled* such that all values with a given key are allocated to the same worker. In the *reduce* step, the average is computed and finally returned back to the master node. On the master node, one can then compute the weighted average of the groups' averages.

average. While the computational overhead of mapping the data across nodes does not justify the efficiency gains for such a small data set, it becomes increasingly powerful when the size of the input data grows, and in particular if the input data exceeds memory.

Example 2 – A Not so Simple Case for Distributed Computing: Compute Median of a Large Data Set

The previous subsection outlined how to compute the average value for each group of a data set, which is one of the most common examples to illustrate the map-reduce framework. Does the framework easily extend to all frequently used characteristics of data? Consider a data set $D = [3, 4, 2, 6, 7, 1, 2, 4, 5, 6, 4, 4, 5, 6, 4]$. If you sorted that data set by size it would look like $D_{sorted} = [1, 2, 2, 3, 4, 4, 4, 4, 4, 5, 5, 6, 6, 6, 7]$ and its median would be equal to 4. If D is too large to be fit into memory, (how) can we use Spark to compute the median value? A simple application of the map-reduce framework is not possible for this case, since there is no way to allocate the data across workers in a helpful way. There is however a large literature on dealing with these kinds of computational problems. For practical purposes Greenwald et al. (2001) propose an algorithm implemented in Spark that strikes a good balance between accuracy and runtime to compute percentiles on large data sets. The main take away however remains: When computation cannot be broken down across nodes, the trivial map-reduce framework fails to deliver a simple solution and more elaborate algorithms are required.

Example 3 – Back to Econometrics: Distributed Ordinary Least Squares

Linear regression is arguably one of the most popular statistical model used in economics. Statistical software packages like Stata, eViews or SPSS as well as fully fledged programming languages like Python or R use a range of algorithms to solve for the vector of β coefficients. Popular algorithms include the *(stochastic) gradient descent* (SGD) and the quasi-Newton *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) method. Spark uses these methods as well. In this example, we want to demonstrate how to compute a ordinary least squares regression on a distributed system. Note that the underlying algorithm used in Spark might be different and depend on the specific model estimated. For illustrative purposes consider a standard optimization problem on a separable objective function:

$$\min_{w} \sum_{i=0}^{N} F_i(w)$$

where $w \in \mathbb{R}^k$ is a vector of weights we try to find in order to minimize the loss function $F(w)$. For example, in the simple case of OLS with $i = 1, \ldots, N$ observations of (x_i, y_i) where $x_i = [x_{i,1}, \ldots, x_{i,k}]$ is a k dimensional vector, the loss function is well known and given by $F_i(w) = (w^T x_i - y_i)^2$. Recall that in *gradient descent* (GD) we solve the above problem by some (random) initial guess of w and then update it according to:

$$w_{j+1} \leftarrow \sum_{i=1}^{N} w_j - \lambda \nabla F_i(w_j)$$

where λ is the learning rate, $\nabla F_i(.)$ is the gradient and j is the iteration step. For OLS the gradient can be calculated symbolically as $\nabla F_i(w) = 2w^T w x_i - 2wy_i$. Now suppose that the dataset $(X = [x_1, \ldots, x_N], y = [y_1, \ldots, y_N])$ is too large to fit in a single computer. How can we apply a map-reduce framework to compute gradient descent? As the loss is expressed as the

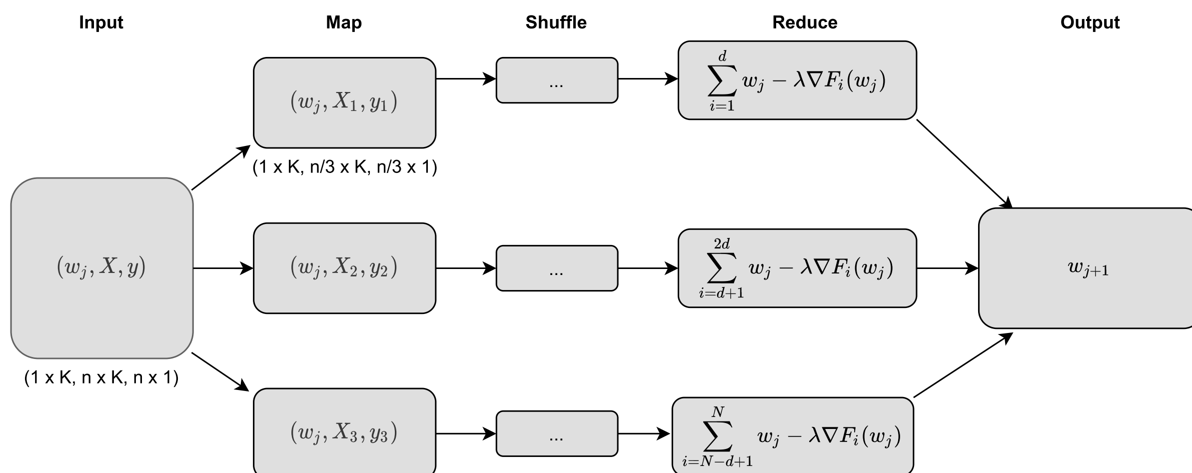

Figure 3: **Distributed linear regression algorithm.** In this example, we illustrate distributed OLS. This example uses three Spark executors $w = 1, 2, 3$. The input data (X, y) has n observations of k variables and a dependent variable y. In the *map* step, random sub-samples of the data (m observations each) are distributed across the executors. In each iteration j, each executor computes a partial sum of the gradient-descent. In the reduce step the partial sums are added up to obtain the new w_{j+1}.

sum of individual losses from each data point, the algorithm easily lends itself to be distributed.

$$\sum_{i=1}^{N} w_j - \lambda \nabla F_i(w_j) = \underbrace{\sum_{i=1}^{d} w_j - \lambda \nabla F_i(w_j)}_{\text{worker 1}} + \underbrace{\sum_{i=d+1}^{2d} w_j - \lambda \nabla F_i(w_j)}_{\text{worker 2}} + \cdots + \underbrace{\sum_{i=N-d+1}^{N} w_j - \lambda \nabla F_i(w_j)}_{\text{worker d}}$$

Since the computation is performed row-by-row, the map step does not induce a shuffle irrespective of how the data is partitioned originally, which is convenient from a run-time perspective. Similarly, to compute the final sum to obtain $w_j + 1$ all partial sums are communicated in a all-to-one fashion. Overall, any computationally expensive all-to-all communication can be avoided. A graphical representation of the procedure (with three workers) is provided in Figure 3.

4 Distributed Econometrics

In this section we discuss how to use well-known econometric techniques in a distributed setting. The first subsection shows how to obtain summary statistics of a big data set. Subsection 4.2 and 4.3 show how to run micro-econometric and panel-regression models in spark on data which would be too expensive to evaluate in a non-distributed fashion. Finally subsection 4.4 demonstrates how to train time series models at scale. In all subsections, we provide information about computing time and cluster resources which may serve as a reference for other researchers confronted with similar big data applications. A word of caution: Some spark algorithms are stochastic (for example stochastic gradient descent) which means that even running it against the very same data twice will yield (slightly) different results. The estimate will oscillate around the true value. In practice the difference between the true (deterministic) result and the stochastic one will be small in most cases, yet it is good to be aware of this limitation.

4.1 Summarising a Large Data Set

In this subsection, we discuss how to use spark to explore and understand data sets that are too large to fit in memory (referred to as *big data* from here on). With ever more data being both collected and connected, the ability to handle such data amounts is of crucial importance. Even at the time of writing, many existing datasets used in economic research qualify as big data. For example, Edwards et al. (2007); Dick-Nielsen et al. (2012); Jankowitsch et al. (2014) use the TRACE data set on corporate bond trades which as of today consists of more than 230 million observations and 32.9 GB. Many non-public data sets are even larger. For example, the European Central Banks collects daily snapshots of derivative exposures of financial intermediaries in the Euro area, which resulted in the need of a specific IT infrastructure (Boneva et al., 2019). With the trend of using administrative data for research (Einav and Levin, 2014), being able to handle such amounts of data will be crucial both for academic research and policy work (Irving-Fisher-Committee, 2020).

Dataset

For this paper we focus on the well-known HMDA dataset, which contains loan application data from the US, frequently used in economic research (see for example Munnell et al. (1996); Duchin and Sosyura (2014); Gilje et al. (2016)), which is also introduced in Foster et al. (2016). The data can be downloaded in yearly files from the *Federal Financial Institutions Examination Council* (FFIEC) website. The entire data set spans from 2007–2017 and contains 150GB+ of data, which one can reduce to 29GB by replacing character labels with numerical identifiers (for example using the FIPS numeric code “01” instead of spelling out “Alabama”). The dataset contains applications for loan mortgages along several characteristics of borrowers (income, county, etc…) and lenders (bank name, balance sheet info, etc…). We provide a copy of the dataset and the subset we are using on our dropbox. Please note that we do not own or maintain this dataset. Check the FFIEC’s website for the most recent version.

Spark Setup

To analyse the data set, we follow a two-part strategy. We locally develop a Spark application on a randomly selected sub-sample of 200,000 observations. We test and debug our spark program on a retail grade computer and after finding satisfactory performance run the same code in a distributed fashion on AWS. While we restrict ourselves to the main steps of the distributed computing logic here, the complete *sparklyr* code is available including a fully reproducible example on our github repository. To use *sparklyr* for handling and analysing large datasets, we found the following steps useful:

- Upload the data to S3
- Do all heavy computations in *sparklyr* using *dplyr* syntax for dataframe manipulation and `spark_apply()` to use base R functions
- Use `collect()` to get the results back to base R and continue to plot or print tables

Results

With 29.4 GB in size, we were not able to load the entire data set into R or Python on our local machine (which featured 16 GB of memory). Therefore, running the entire analysis in base R was not feasible. Instead we imported a subset of 200,000 randomly selected observations and

developed a spark application which conveniently summarizes the data set. We find a combination of `spark_apply()` and *dplyr* functions very helpful to handle the data set. For example, we need to combine the two-digit state fips code with the three-digit county fips code to create a unique county identifier. To do so we need to pad all single-digit state codes with a leading zero (i.e. changing “1” to “01”). There is no *dplyr* function to do so, so we use `spark_apply` to pass a base R function to create the “state_code_fips” variable:

```
hmda_spark = hmda_spark %>%
  spark_apply(function(e)
    data.frame(sprintf("%02d",as.numeric(e$state_code)), e),
    names = c('state_code_fips', colnames(hmda_spark)))
```

We can use *dplyr* backend to analyse large datasets, which provides very readable code. Consider for example the following six lines of code which generate the raw data used for Figure 4. It first groups the data by year and county and computes the average loan to income ratio for country-year. Subsequently we use `collect()` to read the Spark data back onto driver and further manipulate it:

```
hmda_group = hmda_spark %>%
  group_by(as_of_year,county_fips_code) %>%
  summarise(avg_prc_to_inc = mean(loan_to_inc, na.rm=TRUE)) %>%
  collect() %>%   # read back to memory after heavy lifting is done by Spark
  mutate(log_loan_to_inc  = log(1+loan_to_inc)) %>%
  select(county_fips, log_loan_to_inc, as_of_year)
```

A visual illustration is provided in Figure 4.

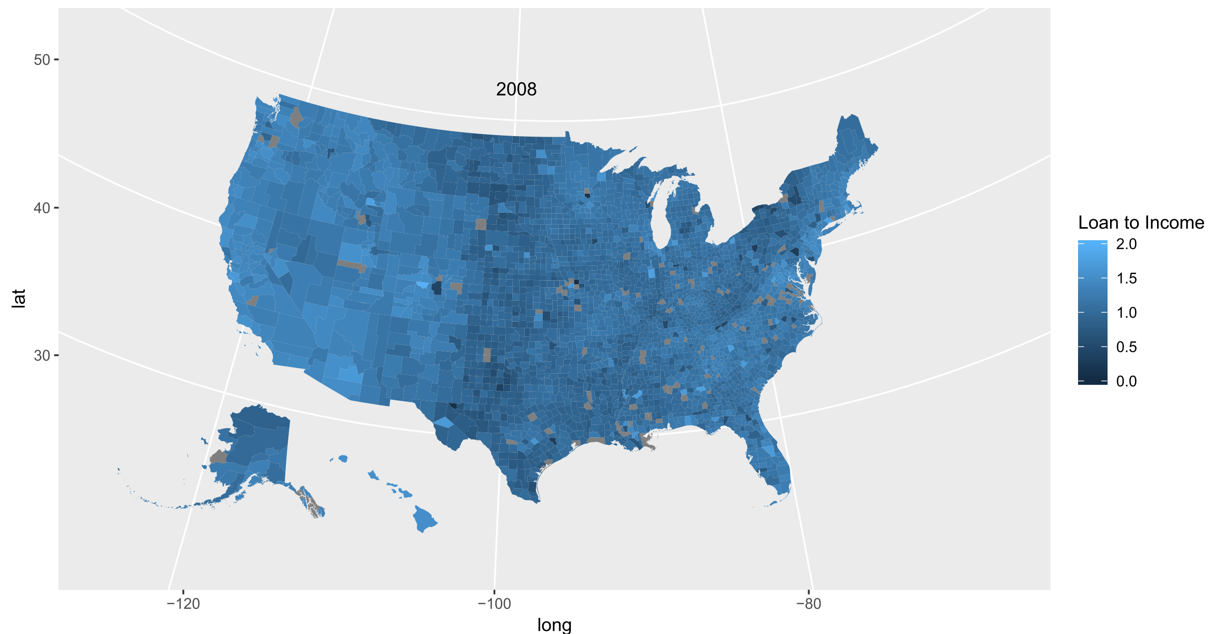

Figure 4: **Heat map US loan-to-income ratios.** This graph plots the log of the loan-to-income of home mortgage application ratio across US counties for 2008.

4.2 Micro Econometrics on Spark

In this section, we explore how to run some popular micro econometric models using Spark. We use the same dataset as in the previous section to estimate linear regression, probit and logit models. We estimate those on a subset of the dataset locally, using base R functions and Spark functions to demonstrate their equivalence. Finally, we run Spark functions on AWS on the entire dataset.

Spark Setup

Using *sparklyr*, we access the Spark's machine learning library *MLlib*, which contains many statistical models used for economic research as well. As outlined in the previous section, we can clean the data using *dplyr* syntax in *sparklyr* to create a spark dataframe against which we can run regressions. *sparklyr* allows us to use base R formulas to be evaluated in their `ml_*` function family. For example, using the 137,819,151 observations strong `hmda_spark` dataframe created in the previous section, we can run a probit regression using the following code:

```
1 glm_model = hmda_spark %>%
2   ml_generalized_linear_regression(application_accepted ~
3    applicant_income_000s , family = "binomial", link = "probit")
```

sparklyr automatically distributes the computation across the cluster, increasing the speed of computation or making it feasible in the first place. We run different versions of the following baseline regression

$$LoanGranted_i = \beta_0 \cdot Income_i + \beta_1 \cdot Male_i + \beta_2 \cdot White_i + \beta_3 \cdot Black_i + 1_{Year(i)} + 1_{LoanPurpose(i)} + \varepsilon_i, \quad (1)$$

where Income is the applicants income in \$1000, Male is a dummy variable which is equal to one if the applicant is male (and zero otherwise), White, Black are dummy variables which are equal to one if the applicant is white or black respectively (with Hispanics being the omitted category), $1_{LoanPurpose(i)}$ and $1_{Year(i)}$ are loan purpose and year fixed effects respectively. LoanGranted is a dummy variable which is equal to one if the loan application was successful. We estimate equation (1) using OLS, probit and logit regressions.

Results

To evaluate the performance of Spark, we run 9 models and report the results in Table 1. We run a linear regression, a probit and a logit model, each one locally on a sub-sample using base R functions and the Spark algorithm as well as the Spark algorithm executed on AWS on the entire dataset. While the empirical estimation confirm some well-known facts (such as white and male privilege in the loan market), the interesting result with regards to this paper's research question is the performance of the *sparklyr* regression commands. For example consider the linear regression estimated in column (1)–(3). Column (1) and (2) estimate a linear regression on the same dataset. Column (1) was estimated using native R's linear regression model, while column (2) used Spark's OLS via MLlib. Both report identical results, as expected. Runtime is considerably larger for the Spark solution. On a small dataset, the computational overhead used for the distribution scheme in Spark outweighs the speed benefits gained by distributed computing. Column (3) provides the same regression on the entire dataset, ran on AWS. Columns (4)–(6) and (7)–(9) repeat this exercise for *probit* and *logit regression* and yield similar conclusions.

Table 1: **Microeconometric regression in Spark.** To estimate a linear regression, specification (1) uses native R's `lm()` on the subsample, specification (2) uses *sparklyr*'s `ml_linear_regression()` on the subsample while specification (3) uses *sparklyr*'s `ml_linear_regression()` on the entire dataset. To estimate a probit regression, specification (4) uses native R's `lm()` on the subsample, specification (5) uses *sparklyr*'s `ml_generalized_linear_regression()` on the subsample while specification (6) uses *sparklyr*'s `ml_generalized_linear_regression()` on the entire dataset. To estimate a logit regression, specification (7) uses native R's `lm()` on the subsample, specification (8) uses *sparklyr*'s `ml_generalized_linear_regression()` on the subsample while specification (9) uses *sparklyr*'s `ml_generalized_linear_regression()` on the entire dataset. Runtime in local spark depends on the machine it is ran on. The results are based on an AWS EC2 instance type *m5.xlarge* (master + 4 nodes). Runtime is measured in minutes. ***, **, *, indicate statistical significance at the 1%, 5%, and 10% respectively.

	OLS			**Probit**			**Logit**		
	Base R (local)	Spark (local)	Spark (AWS)	Base R (local)	Spark (local)	Spark (AWS)	Base R (local)	Spark (local)	Spark (AWS)
Loan granted	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Income (000$)	0.0001***	0.0001***	0.0000061944***	0.0003***	0.0003***	0.0002***	0.0006***	0.0006***	0.0002***
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)
Male	0.0250***	0.0250***	0.0298***	0.0639***	0.0639***	0.070***	0.0982***	0.0982***	0.07006***
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)
Race									
White	0.01383***	0.01383***	0.00829***	0.0354***	0.0354***	0.0264***	0.0613***	0.0613***	0.02645***
	(0.0113)	(0.0113)	(0.0000)	(0.0109)	(0.0109)	(0.0000)	(0.0061)	(0.0061)	(0.0000)
Black	−0.1486***	−0.1486***	−0.1450***	−0.3823***	−0.3823***	−0.3629***	−0.6060***	−0.6060***	−0.3629***
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)
# Observations	147,329	147,329	137,819,151	147,329	147,329	137,819,151	147,329	147,329	137,819,151
Year FE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Loan Purpose FE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Runtime (min)	**0.71**	**0.007**	**9.90**	**0.146**	**1.556**	**33.69**	**0.026**	**0.995**	**19.833**

A concern with big data as input for regression models is that with enough observations, any correlation will be statistically significant at conventional levels (usually 1%). This is true, since the variance of the estimator shrinks towards zero, as the number of observations grows to infinity. However, this does *not* imply that regressions on large datasets will incorrectly find effects, it merely means that the effect size can be arbitrary small and still statistically significant. When discussing these issues with other researchers, we often encountered suggestions like "With big data, the size of the effect is more important". We are sceptical of that interpretation: The size of an effect matters for small and big data, but the larger your dataset, the more likely it is that you are able to pick up even very small effect sizes (Flom, 2013). In fact, you pick up effect sizes which one would simply discard as statistically "insignificant" on small datasets. For a more thorough treatment of large sample theory, the reader is referred to Ferguson (2017).

4.3 Panel Econometrics on Spark

This section illustrates how to estimate a panel data regression in a distributed fashion using Spark. (See Baltagi (2008) for a comprehensive treatment of panel data models). We simulate a big data scenario by generating an artificial panel dataset that is about 90GB in size and contains one billion observations. The key contributions of this section are as follows: First, we illustrate a simple Spark SQL logic to implement within-group data transformation in order to fit a one-way fixed effects estimator. While we restrict the example in this paper to a single fixed-effect, the approach can be easily generalized to a larger number of fixed effects which are encountered in many real-world datasets. Second, we provide empirical results on the validity of the estimated model coefficients and we give an indication of runtime performance and required computing resources. Third, we show how to compute panel-robust standard errors allowing for heteroscedasticity and serial correlation. These standard errors are currently not available in Spark MLlib, however, we show that robust standard errors can be computed in a distributed fashion using our customized distribution scheme.

As for the other subsections, we provide the relevant source code in the format of a Jupyter notebook available on our github repository. Additionally, the data used for this section is provided on our Dropbox.

Dataset

To generate our large artificial dataset we simulate data according to the following linear panel regression model (for a general introduction to panel data and fixed effects see e.g. Wooldridge (2010)):

$$y_{it} = X'_{it}\beta + \epsilon_{it} \quad \forall \quad i = 1, \ldots, N \quad t = 1, \ldots, T \tag{2}$$

where subscript i defines the panel index and refers for example to an individual, subscript t denotes the time period, y_{it} is the dependent variable, X_{it} is the matrix of regressors, β denotes the vector of parameters and ϵ_{it} the error term. Furthermore, we assume that the data contains an unobserved individual-specific fixed effect that is constant over time. In particular, the error term ϵ_{it} is decomposed into an idiosyncratic component u_{it} and an individual-specific effect α_i that is constant over time:

$$\epsilon_{it} = \alpha_i + u_{it} \tag{3}$$

In addition, the individual-specific fixed effect is assumed to enter one of the regressors:

$$X_{1,it} = \alpha_i + \gamma_{1,it} \tag{4}$$

For simplicity all random variables above are drawn from a standard normal distribution. We simulate a dataset with $T = 10$, $N = 100{,}000{,}000$ and $K = 8$ (the number of regressors including the intercept). Furthermore, we impose a coefficient of 0.5 on all regressors without loss of generalization.

Note that the data generating process described by equations (2), (3), (4) introduces correlation between the regressors and the error implying that estimation of β via OLS yields inconsistent results. In the next section, we illustrate how to address this problem in Spark by implementing the well known within-group data transformation scheme which allows for consistent estimation of β via OLS for panels with short T and large N. Hansen (2007) showed that tests based on robust standard errors are consistent even for large T as long as $N \to \infty$. While the within-group estimator is implemented in various standard econometric software packages (see for example *reghdfe* command in Stata (Correia, 2016), *plm* and *lfe* packages in R (Millo, 2017; Gaure, 2019) or *linearmodels* in Python (Sheppard, 2019)), Spark does not yet provide any out-of-the box functionalities for estimating panel data models.

Spark Setup

To apply the within-group data transformation scheme we rewrite (2) by following the definition in Cameron and Trivedi (2005):

$$y_{it} - \bar{y}_i = (X_{it} - \bar{X}_i)'\beta + (\epsilon_{it} - \bar{\epsilon}_i) \tag{5}$$

where $\bar{y}_i = 1/T_i \sum_{t=1}^{T_i} y_{it}$. By subtracting the mean across time for each individual we remove the unobserved fixed-effect such that β can be consistently estimated via OLS. Below, we provide a brief sketch of the Spark SQL logic that implements this simple data transformation scheme for a dataset with one right-hand side variable:

```
1 ## Load panel data into Spark dataframe
2 df = spark.read.parquet('./panel_data')
3
4 ## Create dataframe with mean across time for each individual i
5 df.createOrReplaceTempView("df")
6 df_mean = spark.sql("SELECT i, AVG(y) AS y_bar, AVG(x1) AS x1_bar FROM df
    GROUP BY i")
7
8 ## Apply within-group data transformation scheme via left join
9 df_mean.createOrReplaceTempView("df_mean")
10 df_within_group = spark.sql("
11 SELECT a.i, t, (a.y - b.y_bar) AS y_tilde, (a.x - b.x_bar) AS x_tilde
12 FROM df AS a LEFT JOIN df_mean AS b ON a.i = b.i")
```

Following the transformation defined by (5), we can implement the within estimator using Spark MLlib generalized linear regression class. We should notice though that the standard errors provided by MLlib are the default OLS standard errors which tend to be too low as they do not account for the loss in degrees of freedom arising from demeaning the data. To get a consistent and unbiased estimate for the standard errors we must inflate them by factor $([N(T-1) - K]^{-1}[NT - K])^{1/2}$ (see Cameron and Trivedi (2005) for further details).

Yet, researchers often prefer to compute a panel-robust estimate of the variance-covariance matrix which permits serial correlation in the error term ϵ_{it} and heteroskedasticity of arbitrary form. In practice, model errors are often correlated over time for a given individual which violates the assumption of independence in the model errors. This erroneous assumption leads

to a downward bias in conventional standard errors as the benefit of additional time periods is overestimated. Moreover, the failure to control for heteroskedasticity induces additional bias in the standard errors. While at the time of writing this paper, panel-robust standard errors are not available in Spark MLlib or any other Spark package we are aware of, we can define our own simple distribution scheme to compute these standard errors. The distribution scheme can be easily derived from the standard definition of the estimator for the panel-robust asymptotic variance matrix (Arellano, 1987):

$$\hat{V}[\hat{\beta}] = \left[\sum_{i=1}^{N} \tilde{X}_i' \tilde{X}_i\right]^{-1} \sum_{i=1}^{N} \tilde{X}_i' \hat{\tilde{\epsilon}}_i \hat{\tilde{\epsilon}}_i' \tilde{X}_i \left[\sum_{i=1}^{N} \tilde{X}_i' \tilde{X}_i\right]^{-1} \tag{6}$$

where $\tilde{X}_i = X_i - \bar{X}_i$ is a $T \times K$ matrix of the transformed regressors and $\hat{\tilde{\epsilon}}_i = \tilde{y}_i - \tilde{X}_i \hat{\beta}$ is a $T \times 1$ vector of residuals for panel index i from estimating (5) via OLS. Note that in our example $N = 100{,}000{,}000$ making the size of this dataset much too large to compute $\hat{V}[\hat{\beta}]$ on a standard computer. However, we exploit (6) to break the computation into small independent chunks, apply the computation on each of them separately and finally combine the resulting output to construct $\hat{V}[\hat{\beta}]$. Figure 5 provides a graphical representation of the distribution scheme.

A paired RDD with equally sized data partitions is generated where the RDD's key is defined by the panel index and the value holds the data $(\tilde{X}_i, \hat{\tilde{\varepsilon}}_i)$ for that particular index. After mapping the RDD onto the executors a reducer function is called to perform the crossproduct computation of submatrices. The output is then collected back to the master node where $\hat{V}[\hat{\beta}]$

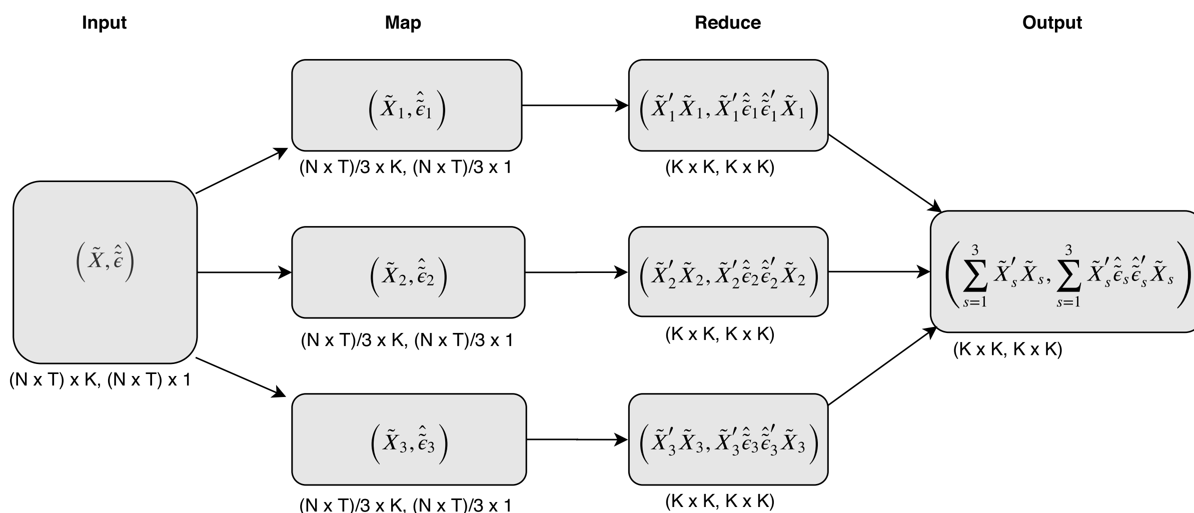

Figure 5: **Distribution scheme for panel robust variance estimate.** In this example, we illustrate a distribution scheme for panel robust variance estimates. This example uses three Spark executors $w = 1, 2, 3$. The input data $(\tilde{X}, \hat{\tilde{\varepsilon}})$ has $N \times T$ observations of K variables and the $N \times T \times 1$ vector of residuals of the regression $\hat{\tilde{\varepsilon}}$. In the *map* step, the data ($N \times T/3$ observations each) is distributed across executors. A hash partitioner is used to ensure that all data for a given panel index is sent to the same executor (for details refer to the code on our github repository). Each executor w computes $(\tilde{X}_w'\tilde{X}_w, \tilde{X}_w'\hat{\tilde{\varepsilon}}_w'\hat{\tilde{\varepsilon}}_w\tilde{X}_w')$ on the sub-sample of data allocated to the executor w. In a final step, the results are returned to the master node where the partial sums are summed up to serve as an input for (6).

can be computed with little computational effort. Given a size of one billion observations and 8 regressors (including the intercept), we assign one million observations to a single partition which is reduced to two small two-dimensional arrays, each of dimension 8×8. As a result, 2,000 of these arrays are collected to the master node which are then used as an input to form (6). Note that in order to obtain correct results when distributing the computation across executors each *RDD* partition must hold all the data for a given panel index which is ensured by the hash partitioner function.

```
1 ## Select relevant columns for computing sandwich VCE
2 df = df.select(["id", "time", "u", "intercept", "x1", "x2", "x3", "x4", "x5",
    "x6", "x7"])
3
4 ## Create hash partitioner assuring that data for each id is in one partition
5 def key_partitioner(id):
6     return hash(id)
7
8 ## Create key-value RDD with 1,000 partitions
9 key_value_rdd = df.rdd.map(lambda x: (x[0], x[1:10])).partitionBy(1000,
    key_partitioner)
10
11 ## Compute array cross-products for sandwich VCE and collect results to master
     node
12 arr_bread_meat = key_value_rdd.mapPartitions(compute_bread_meat).collect()
13
14 ## Construct bread and meat arrays and sandwich VCE
15 bread = np.linalg.inv(sum([item[0] for item in arr_bread_meat]))
16 meat = sum([item[1] for item in arr_bread_meat])
17 vcov = bread.dot(meat).dot(bread)
```

Results

In this section we show results for 3 different model specifications reported in Table 2 as OLS, Fixed Effects and Fixed Effects (Robust VCE). For each specification Table 2 shows estimation and runtime results across three separate estimation runs. The first two columns in each specification provide a comparison of results between Spark in local mode and R's *plm* package using a small subsample of data (100,000 rows). Essentially, this comparison serves to confirm the validity of estimated coefficients and standard errors obtained in Spark taking as a benchmark a popular panel data package from the R community. The estimation and runtime results for the complete dataset can be found in the last column of each specification.

Columns (1)–(3) show coefficient and standard error estimates for the OLS specification corresponding to a linear regression without prior data transformation to account for fixed effects. As expected the coefficient estimates on regressor X_1 show in all three regressions a significant deviation from its true value of 0.5. This bias is corrected for in the fixed effects case when estimation is performed on the transformed dataset as shown in columns (4)–(6) where standard errors have been adjusted for the loss in degrees of freedom. Columns (7)–(9) contain the estimation results with panel robust standard errors using our custom distribution scheme.

A comparison of runtimes across columns shows that the performance advantage of Spark comes into play for large volumes of data. For the cases that consider only a subsample of data the parallelization through Spark does not provide any performance improvement over local model fitting. Columns (3), (6) and (9) show the results when the entire dataset is used. Note that in order to provide a realistic indication of required cluster life time the reported runtime includes

Table 2: **Panel Regression in Spark** To estimate a linear regression, specification (1) uses base R's `lm()` on the subsample, specification (2) uses PySpark's `MLlib LinearRegressionModel()` on the subsample while specification (3) uses PySpark's `MLlib LinearRegressionModel()` on the entire dataset. To estimate a static panel regression, specification (4) uses R's `plm()` package on the subsample, specification (5) uses PySpark's `MLlib LinearRegressionModel()` on the within-group transformed subsample while specification (6) uses PySpark's `MLlib LinearRegressionModel()` on the within-group transformed entire dataset. Standard errors under specification (4)–(6) have been adjusted for the loss in degrees of freedom induced by the within transformation. To estimate a panel regression with robust standard errors, specification (7) uses R's *plm* package on the subsample, specification (8) uses our distributed variance covariance estimator on the subsample while specification (9) uses our distributed variance covariance estimator on the entire dataset. Runtime in local spark depends on the machine it is ran on. The results are based on an AWS EC2 instance type *m4.xlarge* (master + 10 nodes). Runtime is measured in minutes. ***, **, *, indicate statistical significance at the 1%, 5%, and 10% respectively.

	OLS			**Fixed Effects**			**Fixed Effects (Robust VCE)**		
	base R (local) (1)	Spark (local) (2)	Spark (AWS) (3)	R `plm()` (local) (4)	Spark (local) (5)	Spark (AWS) (6)	R `plm()` (local) (7)	Spark (local) (8)	Spark (AWS) (9)
X_1	0.9975***	0.9975***	0.9999***	0.4997***	0.4997***	0.5000***	0.4997***	0.4997***	0.5000***
	(0.0027)	(0.0027)	(0.0000)	(0.003339)	(0.003339)	(0.000)	(0.003342)	(0.003342)	(0.000)
X_2	0.4967***	0.4967***	0.5000***	0.4987***	0.4987***	0.5000***	0.4987***	0.4987***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003337)	(0.003337)	(0.000)	(0.003361)	(0.003361)	(0.000)
X_3	0.4895***	0.4895***	0.5000***	0.4916***	0.4916***	0.5000***	0.4916***	0.4916***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003338)	(0.003338)	(0.000)	(0.003306)	(0.003306)	(0.000)
X_4	0.5028***	0.5028***	0.5000***	0.5039***	0.5039***	0.5000***	0.5039***	0.5039***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003339)	(0.003339)	(0.000)	(0.003350)	(0.003350)	(0.000)
X_5	0.4988***	0.4988***	0.5000***	0.5017***	0.5017***	0.5000***	0.5017***	0.5017***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003334)	(0.003334)	(0.000)	(0.003316)	(0.003316)	(0.000)
X_6	0.4994***	0.4994***	0.5000***	0.5002***	0.5002***	0.5000***	0.5002***	0.5002***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003352)	(0.003352)	(0.000)	(0.003354)	(0.003354)	(0.000)
X_7	0.5101***	0.5101***	0.5000***	0.5036***	0.5036***	0.5000***	0.5036***	0.5036***	0.5000***
	(0.0039)	(0.0039)	(0.0000)	(0.003334)	(0.003334)	(0.000)	(0.003316)	(0.003316)	(0.000)
# Observations	100,000	100,000	1,000,000,000	100,000	100,000	1,000,000,000	100,000	100,000	1,000,000,000
Runtime (min)	**0.019**	**0.017**	**8.33**	**0.183**	**0.083**	**36.09**	**0.386**	**0.367**	**83.52**

not only the time for the actual model fitting stage but also for the time it takes to perform the relevant data transformation steps. A comparison of columns (3) and (6) indicates that this data transformation can be computationally expensive as runtime is substantially higher for the fixed effects specification. The highest runtime is reported for column 9 which is not surprising given that on top of data transformation this specification involves both the computation of residuals and the variance covariance matrix in a distributed fashion. Finally, note that the standard errors in column (7) are the same as in column (8) which confirms that our custom distribution scheme yields valid results for the panel robust variance estimator.

4.4 Time Series Econometrics on Spark

In this section we illustrate how to leverage Spark for large-scale time series analysis which plays a crucial role in the decision making process of many public and private institutions. 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. Moreover, research in economics often requires fitting many time series models. With each individual model typically containing only a limited number of data points, the setup is ideal for distributed computing since existing estimation methods can be executed in parallel across the worker nodes of the cluster framework.

An end-to-end machine learning system for probabilistic demand forecasting at *Amazon* built on Spark is described in Böse et al. (2017). The platform scales to large datasets containing millions of time series. The authors propose a simple distribution 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 distribution logic described in this section follows a very similar approach. A brief review of other distributed machine learning frameworks is given by Chun et al. (2016).

Dataset

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

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

$$\left(1 - \sum_{i=1}^{2} \alpha_i L^i\right) X_t = \left(1 + \sum_{i=1}^{2} \theta_i L^i\right) \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(\mu, \sigma^2) \tag{7}$$

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 (2019)). Following the 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.

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 identifier for each series and a third column a sequence of numbers specifying the order of the data for each series. The last column is required

because Spark may not preserve the temporal 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 temporal ordering of the data.

Spark Setup

This section illustrates the key concept of our distributed forecasting system in Spark. In short, the distribution of model fitting and forecasting is broken up into the following subtasks:

- Create custom Python module (we call it `fit_model_and_forecast.py`) with method that
  - reads time series data based on time series identifier contained in *RDD* partition
  - fits time series models, generates forecasts and saves fitted model
- Read dataset with time series data into Spark dataframe on master node
- Create *RDD* with distinct time series identifiers from Spark dataframe and partition *RDD* into collections of distinct identifiers
- Map *RDD* partitions of identifiers onto Spark executors
- On each Spark executor, import custom Python module and call method from module

To illustrate the simplicity of this approach we provide below the Python code that implements this parallelization logic in less than 10 lines of code:

```
1  ## Load time series data into Spark dataframe
2  df = spark.read.parquet('/path/to/time_series_data')
3
4  ## Create RDD with distinct identifiers and repartition dataframe into 100
       chunks
5  time_series_ids = df.select('ID').distinct().repartition(100).rdd
6
7  ## Add Python module to Spark context for parallel execution
8  spark.sparkContext.addPyFile('/path/to/python_module/fit_model_and_forecast.py
       ')
9
10 ## Function to import Python module on Spark executor for parallel forecasting
11 def import_module_on_spark_executor(time_series_ids):
12       from fit_model_and_forecast import fit_model_and_forecast
13       return fit_model_and_forecast(time_series_ids)
14
15 ## Parallel model fitting and forecasting
16 time_series_ids.foreach(lambda x: import_module_on_spark_executor(x))
```

We first load the entire time series dataset into a Spark dataframe and create a partitioned RDD with distinct time series identifiers. In the context of the exercise in this paper, we set the number of *RDD* partitions to 100 using the *repartition()* function, cutting the collection of distinct identifiers into 100 subsets. Given that we have 1,000 time series in our dataset, the average number of identifiers in each partition is 10. Since Spark will run one task for each partition, the number of *RDD* partitions in combination with the number of executors allocated for the application is an important parameter that determines the degree of parallelism.

In order to make our custom Python module available to all Spark executors, we need to add the module to the Spark context by calling the *addPyFile* method that takes as an argument the file path to the module. Next we define a function that we call below to (i) import the Python module on each Spark executor and (ii) to execute the Python module's method that takes as an input an *RDD* element and performs model fitting and forecasting tasks for the time series in question. Finally, we call this function for each *RDD* partition and its elements in a distributed fashion by calling Spark's `foreach` method.

Table 3: **Runtime for different execution schemes.** The results are based on an AWS EC2 instance type *m4.2xlarge*. Runtime is measured in minutes, Memory is measured in GiB, *Virtual CPUs* refers to the number of *virtual processing units* and *# Partitions* defines the number of *RDD* partitions, containing subsets of distinct time series IDs.

Scenario	Parallel	Virtual CPUs	Memory	# Partitions	Runtime
1	no	16	32	–	201.27
2	yes	192	384	100	6.20

Results

Given a total sample size of 1,000 for each time series, we reserve the last 50 observations of the sample for forecast evaluation while the first 950 observations are used to fit an initial *ARMA(2,2)* model which is then used to produce the first forecast. Subsequently, we 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 file system. For sake of simplicity, only one-step ahead forecasts are generated.

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-distributed fashion and, thus, mirrors a single-core single-machine execution scheme. In this setting, models and forecasts are produced by iterating through all time series identifiers using a for loop. This scenario is used as a benchmark case to evaluate the performance gain from the distributed 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 3 shows the runtime results for the two different scenarios.

The total runtime for the non-distributed scheme is about 200 minutes. This compares to roughly 6 minutes execution time for the distributed scheme, reducing runtime by about 95%. Clearly, the runtime of the distributed approach 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 likely lead to 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 distributed scheme can be used to complete large model fitting and forecasting workloads that would be intractable without substantial parallelization.

5 Conclusion

This paper presents a unified framework for handling large datasets for empirical research. It enables economists to handle and analyse ever growing datasets which are computationally difficult to evaluate on retail-grade computers using their existing data handling pipelines. With datasets becoming larger and larger, these computational constraints are more likely to be binding in the future. With data coming in ever higher frequency, dimensions and potential for being linked, being able to handle such data sources will likely result in novel empirical research

designs. By lowering the threshold of employing cloud computing solutions to handle these kinds of data sets, we aim to contribute to this process.

The cloud computing solution we elucidate, is built in Apache Spark and the distribution scheme is suitable for many established econometric methods as well as now popular machine learning models. After providing some background on distributed computing architectures, we demonstrate ease of use and (sizeable) computational gains. We do so by providing codes and configuration instructions for easily reproducible examples featuring a range of applications from micro-, panel- and time-series econometrics. In a first step, we demonstrate how Spark compares to a local execution of base R and Python codes. Intuitively, the computational overhang of mapping data cross the spark cluster is inefficient on small datasets. Yet the empirical results are identical and the Spark code comes at almost no additional complexity. We then take the operation to the cloud: Running the same codes we ran locally on a subset of data, we are able to handle and analyse datasets which would have been difficult to handle on retail grade computers. We provide an overview of popular statistical models which (i) are implemented in Spark as of now, (ii) can be estimated using simple modifications of existing commands and (iii) are difficult to run on Spark.

The presented approach requires minimal installation and configuration effort and it can be implemented with little background in computer science and parallel/distributed computing and without physical access to high performance computers. Additionally, the appendix of this paper contains extremely detailed instructions on how to lunch a computing cluster and provides minimal examples, which can easily be adapted to the readers needs.

Supplementary Material

Supplementary material is available on our github page, containing all codes to replicate the results along links to the data. Additional instructions are also available, detailing how to setup the AWS infrastructure: https://github.com/benjaminbluhm/econometrics_at_scale.

Acknowledgements

The views expressed in this paper are those of the authors alone and do not represent the view of the European Central Bank (ECB). Earlier version of this paper circulated as "Time Series Econometrics at Scale: A Practical Guide to Parallel Computing in (Py)Spark" and "Econometrics at Scale: Spark Up Big Data in Economics". We would like to thank Sanjiv Das, Frauke Kreuter and Satachit Sagade as well as participants from the *GRADE Workshop on Big Data in the Social Sciences*, the *Bank of England Conference on Modelling with Big Data and Machine Learning: Interpretability and Model Uncertainty* (2019), participants of the *Columbia University Data Science Institute Financial and Business Analytics Center Poster Session November 2019* and members of the *International Monetary Funds' BigData@Fund Community of Practice* for helpful comments and suggestions. All remaining errors are our own.

Funding

We gratefully acknowledge a travel grant sponsored by the Bank of England. We gratefully acknowledge research support from the Leibniz Institute for Financial Research SAFE.

References

Arellano M (1987). Practitioners' corner: computing robust standard errors for within-groups estimators. *Oxford Bulletin of Economics and Statistics*, 49(4): 431–434.

Aruoba SB, Fernandez-Villaverde J, Rubio-Ramirez JF (2003). *Comparing Solution Methods for Dynamic Equilibrium Economies*. PIER Working Paper Archive 04-003. Penn Institute for Economic Research, Department of Economics, University of Pennsylvania.

Athey S, Imbens GW (2017). The state of applied econometrics: causality and policy evaluation. *The Journal of Economic Perspectives*, 31(2): 3–32.

Baltagi B (2008). *Econometric Analysis of Panel Data*. John Wiley & Sons.

Boneva L, Böninghausen B, Fache Rousová L, Letizia E, et al. (2019). Derivatives transactions data and their use in central bank analysis. *Economic Bulletin Articles*, 6.

Böse JH, Flunkert V, Gasthaus J, Januschowski T, Lange D, Salinas D, et al. (2017). Probabilistic demand forecasting at scale. *Proceedings of the VLDB Endowment*, 10: 1694–1705.

Cameron AC, Trivedi PK (2005). *Microeconometrics: methods and applications*. Cambridge University Press.

Cavallo A, Rigobon R (2016). The billion prices project: using online prices for measurement and research. *The Journal of Economic Perspectives*, 30(2): 151–78.

Chambers B, Zaharia M (2018). *Spark – The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media, Incorporated.

Chun BG, Cho B, Jeon B, Jeong JS, Kim G, Kim JY, et al. (2016). Dolphin: runtime optimization for distributed machine learning. In: *The ML Systems Workshop at ICML*.

Clemen RT (1989). Combining forecasts: a review and annotated bibliography. *International Journal of Forecasting*, 5(4): 559–583.

Correia S (2016). *Linear Models with High-Dimensional Fixed Effects: An Efficient and Feasible Estimator*. Technical Report. Working Paper.

Dean J, Ghemawat S (2004). Mapreduce: simplified data processing on large clusters. In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 137–150. San Francisco, CA.

Dick-Nielsen J, Feldhütter P, Lando D (2012). Corporate bond liquidity before and after the onset of the subprime crisis. *Journal of Financial Economics*, 103(3): 471–492.

Duchin R, Sosyura D (2014). Safer ratios, riskier portfolios: banks response to government aid. *Journal of Financial Economics*, 113(1): 1–28.

Edwards AK, Harris LE, Piwowar MS (2007). Corporate bond market transaction costs and transparency. *The Journal of Finance*, 62(3): 1421–1451.

Einav L, Levin J (2014). Economics in the age of big data. *Science (New York, N. Y.)*, 346(6210): 1243089.

Ferguson TS (2017). *A Course in Large Sample Theory*. Routledge.

Fernández-Villaverde J, Valencia DZ (2018). *A Practical Guide to Parallelization in Economics*. National Bureau of Economic Research, Cambridge, MA.

Flom P (2013). Hypothesis testing with big data. Cross Validated. (Version: 2013-08-13).

Foster I, Ghani R, Jarmin RS, Kreuter F, Lane J (2016). *Big Data and Social Science: A Practical Guide to Methods and Tools*. Chapman and Hall/CRC.

Gao H, Ru H, Yang X (2019). *What do a Billion Observations Say About Distance and Relationship Lending?* Working Paper. Technical Report.

Gaure S (2019). lfe: Linear Group Fixed Effects. 2.8-7.1 edition.

Gentzkow M, Kelly BT, Taddy M (2019). Text as data. *Journal of Economic Literature*. 57(3):

535–374.

Ghemawat S, Gobioff H, Leung ST (2003). The Google file system. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 20–43, Bolton Landing, NY.

Gilje EP, Loutskina E, Strahan PE (2016). Exporting liquidity: branch banking and financial integration. *The Journal of Finance*, 71(3): 1159–1184.

Greenwald M, Khanna S, et al. (2001). Space-efficient online computation of quantile summaries. *ACM SIGMOD Record*, 30(2): 58–66.

Grimmer J, Stewart BM (2013). Text as data: the promise and pitfalls of automatic content analysis methods for political texts. *Political Analysis*, 21(03): 267–297.

Hamermesh DS (2013). Six decades of top economics publishing: who and how? *Journal of Economic Literature*, 51(1): 162–172.

Hamilton JD (1994). *Time Series Analysis*. Princeton Univ. Press, Princeton, NJ.

Hansen C (2007). Asymptotic properties of a robust variance matrix estimator for panel data when t is large. *Journal of Econometrics*, 141: 597–620.

Irving-Fisher-Committee (2020). Irving Fisher Committee on Central Bank Statistics. 2019 ifc Annual Report. (accessed 15/01/2020).

Jankowitsch R, Nagler F, Subrahmanyam MG (2014). The determinants of recovery rates in the us corporate bond market. *Journal of Financial Economics*, 114(1): 155–177.

Karau H, Konwinski A, Wendell P, Zaharia M (2015). *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, Inc., 1st edition.

Karau H, Warren R (2017). *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly Media, Inc., 1st edition.

Kleinberg J, Ludwig J, Mullainathan S, Obermeyer Z (2015). Prediction policy problems. *The American Economic Review*, 105(5): 491–495.

Leamer EE (1985). Sensitivity analyses would help. *The American Economic Review*, 75(3): 308–313.

Millo G (2017). Robust standard error estimators for panel models: a unifying approach. *Journal of Statistical Software*, 82(3): 1–27.

Mullainathan S, Spiess J (2017). Machine learning: an applied econometric approach. *The Journal of Economic Perspectives*, 31(2): 87–106.

Munnell AH, Tootell GM, Browne LE, McEneaney J (1996). Mortgage lending in Boston: interpreting hmda data. *The American Economic Review*, 86(1): 25–53.

Ng S (2017). *Opportunities and Challenges: Lessons from Analyzing Terabytes of Scanner Data*. Technical Report. National Bureau of Economic Research.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Sala-I-Martin XX (1997). I just ran two million regressions. *The American Economic Review*, 87(2): 178–183.

Samadi Y, Zbakh M, Tadonki C (2018). Performance comparison between hadoop and spark frameworks using hibench benchmarks. *Concurrency and Computation: Practice and Experience*, 30(12): e4367.

Sheppard K (2019). linearmodels: Models for Panel Data, 4.25 edition.

Timmermann A (2006). Forecast combinations. In: *Handbook of Economic Forecasting* (G Elliott, C Granger, A Timmermann, eds.), volume 1 of *Handbook of Economic Forecasting*, Chapter 4. 135–196. Elsevier.

Varian HR (2014). Big data: new tricks for econometrics. *The Journal of Economic Perspectives*,

28(2): 3–28.
Wooldridge JM (2010). *Econometric Analysis of Cross Section and Panel Data*. MIT press.
Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010). Spark: cluster computing with working sets. In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, Boston, MA.