# Distributed Text Mining in R

Theußl, Stefan; Feinerer, Ingo; Hornik, Kurt

Published: 01/03/2011

*Document Version*
Publisher's PDF, also known as Version of record

# Distributed Text Mining in R

**Stefan Theußl, Ingo Feinerer, Kurt Hornik**

**Research Report Series**
Report 107, March 2011

Institute for Statistics and Mathematics
**http://statmath.wu.ac.at/**

WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

# Distributed Text Mining in R

Stefan Theußl Ingo Feinerer Kurt Hornik

March 16, 2011

**Abstract**

R has recently gained explicit text mining support with the **tm** package enabling statisticians to answer many interesting research questions via statistical analysis or modeling of (text) corpora. However, we typically face two challenges when analyzing large corpora: (1) the amount of data to be processed in a single machine is usually limited by the available main memory (i.e., RAM), and (2) an increase of the amount of data to be analyzed leads to increasing computational workload. Fortunately, adequate parallel programming models like MapReduce and the corresponding open source implementation called Hadoop allow for processing data sets beyond what would fit into memory. In this paper we present the package **tm.plugin.dc** offering a seamless integration between **tm** and Hadoop. We show on the basis of an application in culturomics that we can efficiently handle data sets of significant size.

**Keywords:** text mining, MapReduce, distributed computing, Hadoop

## 1 Introduction

R, an environment for statistical computing and graphics (R Development Core Team, 2011), has gained explicit text mining support via the **tm** package (Feinerer, 2011) originally presented in Feinerer et al. (2008). This infrastructure package provides sophisticated methods for document handling, transformations, filters, and data export (e.g., document-term matrices). With a focus on extensibility based on generic functions and object-oriented inheritance, **tm** makes it possible to apply a multitude of existing methods in the R world to text data structures as well.

In the information age as statisticians we are confronted with an ever increasing amount of data stored electronically (F.Gantz et al., 2008). This is in particular true for the most natural form of storing information, namely text. Given appropriate tools to handle huge text corpora will enable us to answer many interesting research questions via statistical analysis or modeling. E.g., suppose one wants to check daily sentiment in newspaper articles in order to measure interactions between the media and the stock market (Tetlock, 2007), then one might be interested to use data published by news agencies like Reuters which allow to access their news database containing text automatically being annotated with rich semantic metadata. Or, on documents acquired from repositories such as `http://arXiv.org/` or the CiteSeerX project (`http://citeseerx.ist.psu.edu/`) which allow for harvesting metadata and open access to the corresponding content (i.e., download of full text articles), statisticians may apply a variety of bibliometric and scientometric approaches in order to find patterns like the formation or development of author or topic (Blei and Lafferty, 2007; Griffiths and Steyvers, 2004) networks. Furthermore, access to documents written in several centuries, such as the books made available in project Gutenberg (`http://www.gutenberg.org/`), allows to study how linguistic patterns develop over time. In a recent publication in Science Michel et al. (2011) use 15% of the digitized Google books content (4% of all books ever printed) to study the diffusion of regular English verbs and to probe the impact of censorship on a person's cultural influence over time. This led to the advent of a new research field called *Culturomics*, the application of high-throughput data collection and analysis to the study of human culture.

However, the motivation to analyze huge text corpora is the source of two challenges: (1) the amount of data to be processed in a single machine is usually limited by the available main memory (i.e., RAM), and (2) an increase of the amount of data to be analyzed leads to increasing computational workload. Thus, it is highly imperative to find a solution which overcomes the memory limitation (e.g., by splitting the data into several pieces) and to markedly reduce the runtime by distributing the workload across available computing resources (e.g., CPU cores or cloud instances). Typically, we consider distributed memory platforms (DMPs) like clusters of workstations for such applications since they are scalable in terms of CPUs and memory (disk space and RAM) employed. Additionally, many different programming models and libraries like the message passing interface (MPI) are available that facilitate working with these kind of *high performance computing* (HPC) systems. Many of those libraries can directly be employed in R (see Schmidberger et al., 2009, for further references). Still, one open question remains: is there an efficient way to handle large data sets in parallel, i.e., distributing the data, applying functions on the subsetted data in parallel, and gathering results on DMPs?

Fortunately, Google developed a distributed programming model and a corresponding implementation called MapReduce (Dean and Ghemawat, 2004) which allows for efficient parallel processing of data in a functional programming style (Lämmel, 2007). MapReduce—typically used in combination with another important building block: the distributed file system (DFS, Ghemawat et al., 2003)—readily enables and takes care of data distribution as well as load balancing, network performance and fault tolerance.

In this paper we show the huge potential of making a text mining infrastructure like **tm** recognize such a distributed programming model. Typical tasks in text mining like preprocessing can be easily run as parallel distributed tasks without knowing details about the underlying infrastructure. The corresponding extensions to **tm** are encapsulated in a separate plug-in package called **tm.plugin.dc** (Theußl and Feinerer, 2011b) offering a seamless integration building on functionality provided by interfaces to MapReduce environments.

The remainder of this paper is organized as follows. In Section 2 we review the important building blocks in the MapReduce paradigm. The integration of this programming model into **tm** is discussed in Section 3. Additionally, we show how distributed storage can be utilized to facilitate parallel processing of corpora. In Section 4 we present the results of a benchmarking experiment of typical tasks in text mining showing the actual impact on performance in terms of execution time. In an application in culturomics we analyze in Section 5 a corpus of several GB of newspaper articles showing how word usage in a prominent newspaper has changed over 20 years. The hard- and software employed is summarized in Section 6. Finally we conclude this paper in Section 7, pointing out directions for future research.

# 2 The MapReduce paradigm

In this section we describe the programming model and the corresponding implementation (i.e., the underlying software framework) we employed to achieve parallel text mining in a distributed context. MapReduce is a programming model originally proposed by Google for large scale processing of data sets. It consists of two important primitives related to concepts from functional programming, namely a `map` function and a `reduce` function. Basically, the `map` function processes a given set of input data (e.g., collections of text files, log files, web sites, etc.) to generate a set of intermediate data which may/is to be aggregated by the `reduce` function. We can express many tasks in text mining in this model, e.g., preprocessing tasks like stemming of raw text files.

## 2.1 Programming model

Usually, in this model we consider a set of workstations (nodes) connected by a communication network. Given a set of input data we want to employ these nodes for parallel processing of suitable subsets of this data. Therefore, we distribute the data in such a way that parts of the data can be efficiently processed locally on the nodes by individual `map` operations, and aggregated

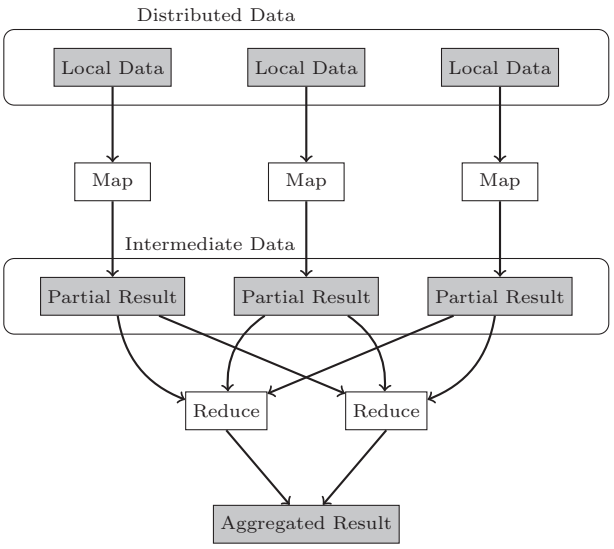


Figure 1: Conceptual Flow

by `reduce` operations. Figure 1 shows this conceptual flow of `map`/`reduce` operations on a given data set.

The `map` function usually transforms its input, i.e., the corresponding subset of the data located on each node (local data) into a list of key/value pairs (intermediate data). After the `map` function has been applied we have three choices to handle intermediate data. First, we can write the data to disk. Second, we can apply another `map` function to the data and/or third, we can apply the `reduce` function which takes the resulting list of key/value pairs and typically aggregates this list based on the keys. This aggregation results in a single key/value pair for each key. All of these operations can be parallelized over the nodes, i.e., every node applies the same function on its local set of data. The major advantage of this so-called data parallelism is that if implemented well this approach theoretically scales over any number of nodes. Furthermore, assigning more than one `map` task to each node advances load balancing and eventually increases efficiency.

To apply the MapReduce programming model in a distributed text mining context we rely on the open source implementation Hadoop (The Apache Software Foundation, 2010).

## 2.2 Distributed file system

Implementations of MapReduce are typically coupled to a DFS which assists in data distribution and enables fault tolerance. E.g., Google offers the Google File System (GFS) to store data as well as intermediate and final results in a distributed manner. Such a file system assists in exploiting data locality which makes parallel execution of tasks faster as additional communication overhead is avoided. Only pieces of data local to each node are considered for processing which improves overall I/O bandwidth. Furthermore, data is replicated on multiple nodes so that failures like crashed nodes or network routing problems will not affect computation. This enables automatic recovery of data in case of system failures. Such a fault tolerant environment is well suited for large clusters of commodity machines—the prime platform for many scientific institutions and companies because of its cost-effectiveness (Barroso et al., 2003).

Similar to GFS the Hadoop Distributed File System (HDFS, Borthakur, 2010) provides such a scalable and fault tolerant architecture. Copying data to the HDFS implies that the given input data is split into parts (physical blocks or separate files). These parts are distributed over the system and replicated over a predefined number of machines. Files are organized hierarchically in directories and identified by path names.

### 2.3 Hadoop streaming

In addition to the MapReduce implementation for distributed manipulation of large data sets and the HDFS, the Hadoop framework also includes a utility called *Hadoop Streaming* implementing a simple interface to Hadoop allowing for the usage of MapReduce with any executable or script as the mapper and the reducer. It transforms input data stored on the HDFS and aggregates the results based on the provided scripts. In order to use this tool directly within R we designed a simple interface (the R package **hive**, Theußl and Feinerer, 2011a) which basically creates executable R scripts (Rscript, R Development Core Team, 2011) from provided R functions and automatically runs them on the Hadoop cluster. This approach offers high level access to the Hadoop MapReduce functionality.

## 3 Integration

### 3.1 Classical layout and process flow

The **tm** text mining framework offers functionality for managing text documents, abstracts the process of document manipulation and eases the usage of heterogeneous text formats (Feinerer et al., 2008). The main data structure is a *corpus*, an entity similar to a database holding and managing text documents in a generic way. It can be seen as a container to store a collection of text documents including a set of additional metadata annotations. Conceptually we want a corpus to support a set of intuitive operations, like accessing each document in a direct way, displaying and pretty printing the corpus and each individual document, obtaining information about basic properties (e.g., the number of documents in the corpus), or applying some operation on a range of documents. These requirements are formalized via a set of interfaces which must be implemented by a concrete corpus class:

**Subset** The `[[` operator must be implemented so that individual documents can be extracted from a corpus.

**Display** The `print` and `summary` must convert documents to a format so that R can display them. Additional meta information can be shown via `summary`.

**Length** The `length` function must return the numbers of documents in the corpus.

**Iteration** The `tm_map` function which can be conceptually seen as an `lapply` has to implement functionality to iterate over a range of documents and applying a given function.

The implementation in R via the **tm** package was mainly driven by these conceptual requirements and interface definitions, and is characterized by a virtual corpus class which defines the above set of pre-defined interfaces which must be implemented by actual derived corpus classes to support the full range of desired properties. The main advantage of using a virtual class with well-defined interfaces is that instantiated subclasses work with any function aware of the abstract interface definitions but the underlying implementation and representation of internal data structures is completely abstracted.

So-called *sources* are used to abstract document acquisitions, e.g. files from a hard disk, over the Internet, or by other connection mechanisms. Although we use different sophisticated mechanisms for corpus construction like using database back ends it is conceptually appealing and possible to allocate the storage in a distributed manner since communication is usually not limited by a single bottleneck. We will use this fact for physically distributed allocation in order to reduce communication costs for parallel computation.

E.g., for a non-distributed scenario assume we have our data set stored in a directory on a local hard disk. Then we can simply use a predefined source like `DirSource` which delivers its content. A reader function now specifies how to actually parse each item delivered by the source (like XML or HTML documents). E.g., for some news stories from Reuters (see Section 4.1 for a

detailed description) in XML format stored in the directory `Data/reuters` we can construct the corpus in R via

```
> library("tm")
> corpus <- Corpus(DirSource("Data/reuters"), list(reader = readReut21578XML))
```

For illustration purposes the **tm** package includes a sample data set containing 50 documents of the Reuters corpus on the topic "Acquisitions" (*acq*). We will use it for demonstration (in particular the sixth document) as it provides easy reproducibility. It can be loaded via

```
> data(acq)
> acq[[6]]

A group of affiliated New York
investment firms said they lowered their stake in Cyclops Corp
to 260,500 shares, or 6.4 pct of the total outstanding common
stock, from 370,500 shares, or 9.2 pct.
    In a filing with the Securities and Exchange Commission,
the group, led by Mutual Shares Corp, said it sold 110,000
Cyclops common shares on Feb 17 and 19 for 10.0 mln dlrs.
 Reuter
```

Alongside the data infrastructure for text documents the framework provides tools and algorithms to efficiently work with the documents, like whitespace removal, stemming or stopword deletion. We denote such functions operating on corpora as *transformations*. Another important concept is *filtering* which basically involves applying predicate functions on collections to extract patterns of interest. Since we iterate over all documents in a corpus applying transformations and filters these concepts provide a valuable instance for optimization efforts. Fortunately, both concepts are highly amenable to parallelization by construction: each transformation or filter operation needs to be applied to each document without side effects.

Typical computationally expensive preprocessing steps for raw text documents are: stemming and stopword removal. Stemming denotes the process of deleting word suffixes to retrieve their radicals. It typically reduces the complexity without any severe loss of information (especially for bag-of-words). One of the best known stemming algorithm goes back to Porter (1980) describing an algorithm that removes common morphological and inflectional endings from English words. The **tm** function `stemDocument()` provides an interface to the Porter stemming algorithm, e.g., via

```
> stemmed <- tm_map(acq, stemDocument)
> stemmed[[6]]

A group of affili New York
invest firm said they lower their stake in Cyclop Corp
to 260,500 shares, or 6.4 pct of the total outstand common
stock, from 370,500 shares, or 9.2 pct.
    In a file with the Secur and Exchang Commission,
th group, led by Mutual Share Corp, said it sold 110,000
Cyclop common share on Feb 17 and 19 for 10.0 mln dlrs.
 Reuter
```

we can easily stem all documents from `acq`, i.e., we map (apply) the stemming function on all documents.

Stopwords are words that are so common in a language that their information value is almost zero, i.e., their entropy is very low. Therefore it is a common procedure to remove such stopwords. Similarly to stemming, this functionality is already provided by **tm** via the `removeWords()` function. A single call suffices to remove all English stopwords from a text corpus:

```
> removed <- tm_map(acq, removeWords, stopwords("english"))
> removed[[6]]

A   affiliated New York
investment firms   lowered  stake  Cyclops Corp
 260,500 shares,  6.4 pct   total outstanding common
stock,  370,500 shares,  9.2 pct.
    In  filing   Securities  Exchange Commission,
 , led  Mutual Shares Corp,   sold 110,000
Cyclops common shares  Feb 17  19  10.0 mln dlrs.
 Reuter
```

Removal of whitespace (blanks, tabulators, etc.) and removal of punctuation marks (dot, comma, etc.) can be done via the `stripWhitespace()` and `removePunctuation()` functions.

A very common approach in text mining for actual computation on texts is to build a so-called *document-term matrix* (DTM) holding frequencies of distinct terms, i.e., the *term frequency* (TF) for each document. When using **tm** DTMs are stored using a simple sparse representation implemented in package **slam** by Hornik et al. (2010). It provides several functions like `col_sums()`, `row_means()`, or `rollup()` in order to operate on such matrices memory-efficiently. Thus, this package has to be loaded additionally. DTM construction typically involves preprocessing and counting TFs for each document. A parallel approach for both construction steps seems promising. In a non-distributed environment a DTM can be easily constructed from a corpus via

```
> DocumentTermMatrix(acq, list(stemming = TRUE, removePunctuation = TRUE))

A document-term matrix (50 documents, 1395 terms)

Non-/sparse entries: 3544/66206
Sparsity           : 95%
Maximal term length: 16
Weighting          : term frequency (tf)
```

where the second argument specifies preprocessing steps which should be combined and executed when counting TFs in each document and building the matrix (in this example stemming and punctuation removal).

## 3.2 Integration between Hadoop and tm

The integration between the Hadoop software platform and the **tm** infrastructure is characterized by two main design concepts: *distributed storage* and *parallel computation*. Both concepts are implemented in the package **tm.plugin.dc** in a generic way (Theußl and Feinerer, 2011b). This package is designed in such a way that it provides a corpus implementation of the abstract interfaces as outlined in Section 3.1. Its classes and methods can be transparently used in combination with the existing **tm** infrastructure. The package itself is constructed in such a modular way that several implementations of the concepts distributed storage and parallel computation are possible. We chose to primarily utilize the Hadoop distributed computing environment to provide these capabilities.

### Distributed storage

Typically, a corpus in **tm** is built by constructing an appropriate data structure holding a sequence of single text documents enriched with metadata which further describes textual content and thus offers valuable insights into the document structure. The content for each text document is acquired via source access and copied into main memory. I.e., huge corpora occupy lots of RAM which slows down computations significantly, and even worse there is a practical limit on the

maximum corpus size (by the physical memory size minus overhead by the operating system and other applications). With Hadoop we can circumvent this problem. First, we load the local files delivered by a source instance into the HDFS. More specifically, we construct key/value pairs with the document ID as the key and the corresponding content (a serialized R object) as the value, respectively. Providing the resulting files, i.e., collections of key/value pairs referred to as *chunks* subsequently, in the HDFS can be easily achieved via standard Hadoop functions or via wrapper functions implemented in **hive**. Second, in R we just store unique pointers instead of the actual document. These pointers identify the individual chunks containing the serialized documents in the HDFS. We call such an instance a *distributed corpus*. This technique reduces the main memory consumption drastically (e.g., even for millions of documents the pointers occupy just a few megabytes). Technically, we created the class `'DistributedCorpus'` which inherits from a standard corpus. This allows to utilize this new corpus instance in all use cases of a classical corpus. Since the **tm** infrastructure is designed in a very modular and generic way as described above, we only needed to write methods for a few generic functions (e.g., access or subset operators) encapsulating the different behavior necessary to abstract the underlying distributed storage.

#### Parallel computation

Once we have documents stored in the HDFS, Hadoop is able to perform computations (see `map` and `reduce` steps in Section 2.1) on the data pieces local to each processing node. Such a computation is highly parallel and scales with the amount of available workstations. As indicated in Section 2.2 the distributed storage allows for low communication overhead as each node typically accesses and computes only on the data physically located at its position. Thus, we rewrite **tm** transformations on-the-fly to Hadoop `map` functions, i.e., executable R scripts, which are sent to the Hadoop environment via the Hadoop Streaming utility. In detail, every node is assigned a particular chunk located in the HDFS. Each document in the corresponding chunk gets unserialized and transformations are applied. The (again serialized) results are stored on the HDFS and we only need to update the pointers in our corpus to match the corresponding chunks. Moreover, we store a set of so-called *revisions* for each map call which allows extremely fast switching between various snapshots of the same corpus (like a history with rollback feature known from database systems)—we just need to keep track of the pointers and reroute them to the desired snapshot physically located on the HDFS.

The construction of DTMs additionally needs a `reduce` function. In the `map` steps preprocessing as described above is applied and the remaining terms are counted and stored as intermediate data. In the `reduce` step DTMs are constructed from the individual term vectors and combined on the master node.

### 3.3 Using the distributed corpus

The package **tm.plugin.dc** implements the above concepts and has to be loaded in order to make use of the new class `'DistributedCorpus'`.

```
> library("tm.plugin.dc")
```

Note that `'DistributedCorpus'` allows for many implementations of the main design concepts distributed storage and parallel computation. Typically, a specific parallel computing environment (PE) is associated with a given kind of storage. By default **tm.plugin.dc** uses the temporary directory (e.g., '`/tmp`') on the local file system as "distributed" storage. We refer to this *type* of storage as `local_disk`. Currently, no PE is used with this storage type. However, corpora not fitting into main memory can be easily stored using this technology.

In order to take advantage of the parallel computing paradigm MapReduce for operations on `'DistributedCorpus'` we need to specify to use the HDFS storage type which has Hadoop Streaming associated. In any way, this requires a working Hadoop installation and another R package named **hive** to be loaded (see Appendix A for installing Hadoop and **hive** on a Linux

system). This package offers an interface to file system accessors and high-level access to the PE Hadoop Streaming.

```
> library("hive")
> hive()

HIVE: Hadoop Cluster
- Avail. datanodes: 1
'- Max. number Map tasks per datanode: 2
'- Configured Reducer tasks: 0

> storage <- dc_storage_create(type = "HDFS", base_dir = "/tmp/dc")
```

The function `dc_storage_create()` prepares a given directory `base_dir` on the specified `type` of storage to be used for text mining tasks.

Similar to the standard process flow the data has to be retrieved from the specified source via

```
> dc <- DistributedCorpus( DirSource("Data/reuters"),
+                          list(reader = readReut21578XML), storage )
```

or we can *coerce* a standard `'Corpus'` to `'DistributedCorpus'`:

```
> dc <- as.DistributedCorpus(acq, storage)
> dc

A corpus with 50 text documents
```

Note that one needs to supply the desired storage object as argument to both functions since data is read from the provided source and directly stored as chunks on the given (distributed) storage (see Section 3.2). After that, appropriate methods ensure that `'DistributedCorpus'` can be handled as defined for `'Corpus'`. This allows for a seamless integration of the HDFS or any other storage type defined in **tm.plugin.dc** into **tm** without changing the user interface. E.g., calling `tm_map()` has the same effects as shown in the example above but uses the Hadoop framework for applying the provided `map` functions instead.

```
> dc <- tm_map(dc, stemDocument)
> all(sapply(seq_along(acq), function(x) identical(dc[[x]], stemmed[[x]])))

[1] TRUE
```

Note that the processed documents are still stored on the HDFS since by concept the return value of `tm_map()` must be of the same class as the input value. They can easily be retrieved with corresponding accessor methods. Furthermore, as the amount of usable memory is only restricted by the disk space available in the HDFS when using `'DistributedCorpus'`, we decided to store the original data and the results under different revisions, i.e., different directories on the HDFS. Revisions are highly useful for switching between various snapshots of the same corpus. This allows methods to work on different levels in a preprocessing chain, e.g., before and after stemming. In addition revisions allow backtracking to earlier processing states, a concept similar to rollbacks in database management systems. Revisions can be retrieved or set using the functions `getRevisions()` and `setRevision()`, respectively.

```
> getRevisions(dc)

[[1]]
[1] "20110316181251-2-a"

[[2]]
[1] "20110316181251-1-o"
```

```
> dc <- setRevision(dc, getRevisions(dc)[[1]])
> all(sapply(seq_along(acq), function(x) identical(dc[[x]], acq[[x]])))
```

```
[1] TRUE
```

Another method for `'DistributedCorpus'` ensures that the DTM is constructed in parallel using MapReduce when calling `DocumentTermMatrix()`. Here, term vectors (intermediate data) are generated per document in the `map` step and are aggregated to triplets of the form $(term, ID, tf)$. Finally, after applying `reduce` the resulting matrix is constructed from the aggregated data stored in the HDFS on the master node. Note that the desired number of reducers has to be set via the function `hive_set_nreducer()`.

```
> hive_set_nreducer(2)
> DocumentTermMatrix(dc, list(stemming = TRUE, removePunctuation = TRUE))
```

```
A document-term matrix (50 documents, 1395 terms)

Non-/sparse entries: 3544/66206
Sparsity           : 95%
Maximal term length: 16
Weighting          : term frequency (tf)
```

# 4 Benchmark experiment

In our performance experiment, we studied the runtime behavior of distributed text mining tasks using the integrated MapReduce framework presented in Section 3. Additionally, we compare the results to another parallel programming paradigm—data parallelism via the *Message Passing Interface* (MPI, Message Passing Interface Forum, 1994, 2003)—which is delivered with the **tm** package.

Basically, the MPI approach seems to be promising since it is mainly motivated by the fact that most operations on the documents are independent from the results of other operations. As a consequence there is plenty of room for parallelization, i.e., parallel execution of code fragments on multiple processors with relatively small overhead. This is especially of interest since hardware performance gains during the last years mainly stem from multicore or multiprocessor systems instead of faster (e.g., higher clock frequency) single core processors. However, on the master node the whole data set stays in the main memory while parts of the data are being processed in parallel on the worker nodes. Thus, in contrast to the MapReduce approach the R/MPI implementation is limited by the main memory of the calling machine in terms of data set size. Even if we are able to run several steps in parallel, the whole corpus has to be loaded into RAM initially.

## 4.1 Data

For our benchmark experiment we consider two corpora: the *Reuters-21578* corpus and a collection of *Research Awards Abstracts* from the National Science Foundation (NSF). Both can be represented as standard **tm** corpora needed for running the MPI experiment and thus fit into main memory of a typical workstation.

In subsequent applications we additionally investigate large corpora demanding for a data parallel processing environment like Hadoop circumventing the memory restriction: the *Reuters Corpus Volume 1* (RCV1) and the *New York Times (NYT) Annotated Corpus*. For an overview on the different corpora used in all of our experiments see Table 1. Pre-built data packages (**tm.corpus.<name>**) for the freely redistributable Reuters-21578 and NSF corpora can be downloaded from the data repository of the Institute for Statistics and Mathematics of the WU Wien (`http://datacube.wu.ac.at`). We only provide corpus packages for Reuters-21578 and NSF due to license restrictions. Packages for the RCV1 and NYT corpora can be obtained from the first author if the right to use the data can be verified.

| | # documents | mean # char. per document[1] | corpus size [MB][2] |
|---|---|---|---|
| Reuters-21578 | 21578 | 736.39/834.42 | 87 |
| NSF Abstracts (Part 1) | 51760 | 2895.66 | 236 |
| RCV1 | 806791 | 1426.01 | 3804 |
| NYT Annotated Corpus | 1855658 | 3303.68/3347.97 | 16160 |

Table 1: Number of included documents, average number of characters per document, and uncompressed size on the file system for each corpus.

**Reuters-21578** The Reuters-21578 data set (Lewis, 1997) contains stories collected by the Reuters news agency. The data set is publicly available and has been widely used in text mining research within the last decade. It contains 21578 short to medium length documents in XML format (obtainable e.g., from `http://ronaldo.cs.tcd.ie/esslli07/data/`) covering a broad range of topics, like mergers and acquisitions, finance, or politics. To download the corpus use:

```
> install.packages("tm.corpus.Reuters21578", repos = "http://datacube.wu.ac.at",
+     type = "source")
> library("tm.corpus.Reuters21578")
> data("Reuters21578")
```

**NSF Research Awards Abstracts** This data set consists of 129000 plain text abstracts describing NSF awards for basic research submitted between 1990 and 2003. The data set can be obtained from the *UCI Machine Learning Repository* (`http://archive.ics.uci.edu/ml/`). The corpus is divided into three parts. We used the largest part (*Part 1*) in our experiments.

**Reuters Corpus Volume 1** Lewis et al. (2004) introduced the RCV1 consisting of about 800000 (XML format) documents as a test collection for text categorization research. The documents contained in this corpus were sent over the Reuters newswire (`http://www2.reuters.com/media/newswires/`) during a 1-year period between 1996-08-20 and 1997-08-19. RCV1 covers a wide range of international topics, including business & finance, lifestyle, politics, sports, etc. The stories were manually categorized in three category sets: topic, industry and region.

**NYT Annotated Corpus** The largest data set in our experiment contains over 1.8 million articles published by the New York Times between 1987-01-01 and 2007-06-19 (Sandhaus, 2008). Documents and corresponding metadata are provided in an XML like format: News Industry Text Format (NITF).

## 4.2 Procedure

The benchmark experiments consists of running several preprocessing steps and constructing DTMs which usually constitute the major computation effort. All experiments were repeatedly (three times) run for a selected number of CPUs using the Reuters-21578 corpus and the NSF Research Awards Abstracts (Part 1). We compare the runtime behavior of the Hadoop- with the MPI approach. In a second experiment we show that we can use the Hadoop approach to process data sets of sizes beyond the size of the main memory. In particular we use the same approach as described above but take the RCV1 and NYT annotated corpus as the basis for this experiment.

### Parallelization via MapReduce

With the help of package **tm.plugin.dc** we can easily use the HDFS as "extended" memory to store corpora. Parallelization of transformations (via `tm_map()`) and DTM construction (via

[1] with/without considering empty documents
[2] calculated with the Unix tool `du`

`DocumentTermMatrix()`) is supported by appropriate methods using the Hadoop streaming utility (see Section 3.2).

#### Parallelization via MPI

**tm** supports parallelization of transformations, filter operations, and DTM construction, which constitute the most time-consuming operations in the **tm** infrastructure. Support can be easily (de-)activated via `tm_startCluster()` and `tm_stopCluster()` calls which start up/stop an MPI environment or use an existing instance if already running. Technically the high-level interface **snow** (Rossini et al., 2003; Tierney et al., 2008) is used which in turn delegates the parallel execution to the **Rmpi** package (Yu, 2002, 2010). This allows the code to stay relatively simple because mainly parallel execution of `lapply()` operations needs to be implemented (e.g., via `parLapply()`) but allows both the usage of multi-core systems (via multiple instances on individual cores running on a single physical workstation) and of multiprocessor systems (via accessing multiple physically distributed machines). MPI splits up the input corpus into a set of chunks of documents which get processed by the individual participating nodes. Conducted experiments showed an impressive performance gain, with an almost linear speed-up for each additional processing unit.

### 4.3 Results

Figure 2 shows the runtime improvements in an example where **tm** uses parallelization via MPI (black) and Hadoop (red). As test set we used the complete Reuters-21578 data set (above) and part 1 of the NSF data set (below), and performed stemming (left) and stopword removal (middle) for each document in the corpus. We set the number of processor cores available to a range from one to 32. The figure depicts the averaged runtime (three runs per setting) necessary to complete all operations, showing a clear indication how **tm** profits from multi-core or distributed parallelization of typical preprocessing steps on a realistic data set. Both approaches scale almost linearly with the amount of processing cores. Interestingly, MPI scales superlinearly in a few cases, e.g., for stopword removal on the Reuters-21578 data set. One possible explanation is that `gsub()`, which is internally used to replace stopwords with an empty string, takes significantly more time on longer strings (i.e., its asymptotic runtime behavior is bigger than $O(n)$ where $n$ denotes the input string length).

We can also see that there is an overhead when using Hadoop as it takes notable time to start the framework. This is especially relevant for smaller data sets and for computationally cheap operations. In such cases MPI easily outperforms Hadoop. For large data sets and computationally expensive operations the Hadoop overhead is negligible which makes Hadoop the natural choice for large data sets. In addition a main bottleneck in typical setups are network transfers and we are physically limited by the main memory. These problems are only addressed by the Hadoop high performance computing environment by using a distributed file system which makes it the only choice (compared to MPI) for huge data sets.

Results for constructing DTMs are shown in Figure 2 on the right for Reuters-21578 and NSF, respectively. Interestingly, MPI outperforms Hadoop in this case since communication costs are lower when term vectors are transferred. Furthermore, in contrast to Hadoop where data is almost always kept on disk, MPI holds data in memory. However, for large data set throughput increase significantly using the Hadoop framework as seen in Table 2. Here, throughput is measured as the number of 1000 characters per second ([k char/s]) on the one hand and megabytes per second ([MB/s]) on the other hand. From this table we see some interesting behavior compared to Table 1. Corpora containing documents with more content (in terms of the average number of characters) are processed faster and corpus size (in terms of physical storage) does not necessarily affect runtime. One possible explanation for the former behavior is that fewer I/O operations have to be performed compared to the total corpus size since we iterate over the documents in the corpus and not over equal sized text chunks. Furthermore, this might also influence the latter behavior as only for the largest data set we observed a significant gain in throughput where load balancing of Hadoop is leveraged and becomes effective for the given number of nodes.

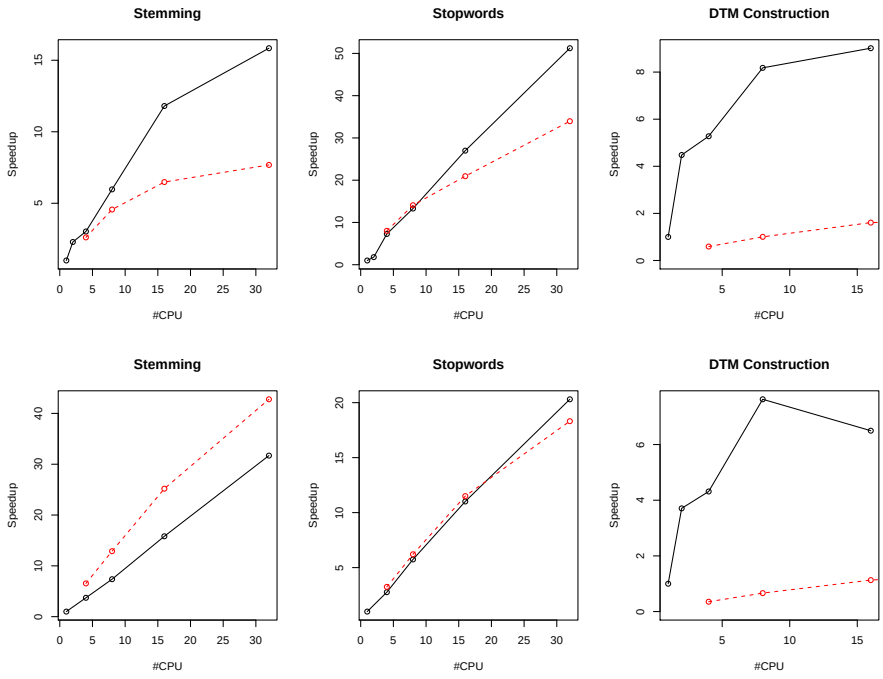


Figure 2: Runtime in seconds for stemming, stopword removal, and DTM construction on the full Reuters-21578 data set (above) and on part 1 of the NSF data set (below) utilizing either Hadoop (red) or MPI (black) with up to 16 processing cores.

To sum up, apart from RAM as a limiting factor Hadoop is the technology of choice for (1) corpora containing documents with lots of text and/or (2) corpora which are large in terms of disk space required.

| | runtime [s] | throughput [k char/s] | throughput [MB/s] |
|---|---|---|---|
| Reuters-21578 | 93 | 193.6 | 0.94 |
| NSF Abstracts (Part 1) | 291 | 515.0 | 0.81 |
| RCV1 | 5805 | 198.2 | 0.66 |
| NYT Annotated Corpus | 8330 | 745.8 | 1.94 |

Table 2: Corpus processing statistics. Constructing DTMs with 32 Hadoop nodes on Cluster@WU.

# 5 Application

It is well known (Francis and Kučera, 1982) that if one knows the words with the highest frequency, he or she will quickly know most of the terms in a given English text. With the ability to process large data sets like the NYT Annotated corpus we can answer many interesting questions from linguistics and other disciplines by using appropriate statistical methods. E.g., Michel et al. (2011) analyzed about 4% of all books ever printed in order to investigate cultural trends quantitatively. This even led to the advent of a new research field called *Culturomics*. In this section we add to this discussion how the active vocabulary in newspaper articles of the NYT has changed over time.

The NYT corpus consists of 1855658 documents (docs) published by the New York Times between 1987-01-01 and 2007-06-19. It contains short- to medium-length articles (on average

552 terms per document) from various genres. There are in sum 1023484418 terms in the whole corpus from which we can derive our vocabulary of 547400 words by identifying all unique terms. Typical for English text we observed on average 5.99 bytes per term.

First, we compare the text coverage given a fixed vocabulary size for the NYT corpus with the results of Francis and Kučera (1982) (see also `http://en.wikipedia.org/wiki/Vocabulary`. Table 3 shows that by knowing the 2000 English words with the highest frequency, one would know on average 80% of the terms in English texts or 84% of the terms in NYT articles.

| size | coverage | coverage_NYT |
|---|---|---|
| 1000 | 0.72 | 0.75 |
| 2000 | 0.80 | 0.84 |
| 3000 | 0.84 | 0.88 |
| 4000 | 0.87 | 0.91 |
| 5000 | 0.89 | 0.92 |
| 6000 | 0.90 | 0.93 |
| 15851 | 0.98 | 0.97 |

Table 3: Vocabulary coverage in the NYT Corpus compared to standard English text coverage.

However, text coverage using a given vocabulary is not necessarily stable over time since language use may change (see e.g., Hogg and Denison, 2008). In order to investigate how the active vocabulary changes e.g., if the text becomes simpler in the sense that coverage of texts by the most frequent terms increases, we needed to analyze the term frequencies in this very large corpus, which was conveniently achieved by reading it into a `'DistributedCorpus'` and subsequently deriving the DTM (which contained 547400 unique terms for 1855658 documents, with a sparsity of 99%). Using metadata like *date of publication* contained in the corpus object we can very easily derive a time series of text coverage for a given vocabulary by suitably aggregating the DTM. We know that language texts could simultaneously get "more complicated" in the sense that far tails of the term frequency distribution get heavier, i.e., that increasingly more terms are needed to cover the remaining words. However, we do not pursue the latter issue here as it requires much more extensive Natural Language Processing (NLP) such as named entity recognition and morphological standardization.

Figure 3 shows that text coverage is decreasing almost linearly by roughly 1% over 20 years considering the 1000 and 4000 most often used terms in the whole corpus, respectively. Considering this scenario we might conclude that texts are not getting simpler over time.

However, if we consider only stopwords, i.e., words with a low entropy, the picture changes. Over the whole NYT corpus we found 332 different stopwords. The average text coverage using only stopwords is 0.54. Figure 4 reveals that text coverage driven by stopwords dictionary increases by more than 1% in the first decade until 1997 and remains stable over the second (we do not have a reasonable explanation for this structural break). This suggests that while texts were getting "more specialized", the amount of low entropy content also increased.

# 6 Computational Details

All the runtime experiments and applications presented in this paper were performed on WU's HPC cluster environment (*cluster@WU*). Each node of the cluster consists of an Intel Core 2 Duo CPU 2.4 GHz, 110 GB of local (SATA) hard disk storage reserved for the HDFS, and 4 GB of main memory. All nodes are connected by standard Gigabit Ethernet network in a flat network topology. Parallel jobs have been submitted with the Sun Grid Engine version 6.2 update 3. On cluster@WU the parallel environments Hadoop (MapReduce) and OpenMPI (MPI) are available in versions 0.20.1 and 1.3.3, respectively. All R code has been executed via version 2.12.0 of R.

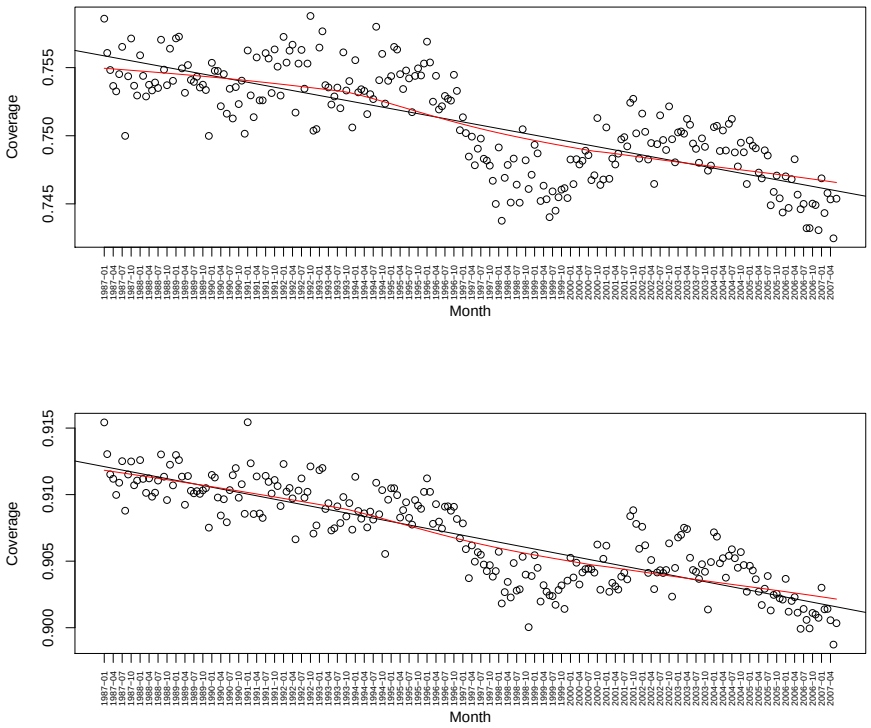


Figure 3: Monthly text coverage using top 1000 (top) and 4000 (bottom) terms in the NYT corpus.

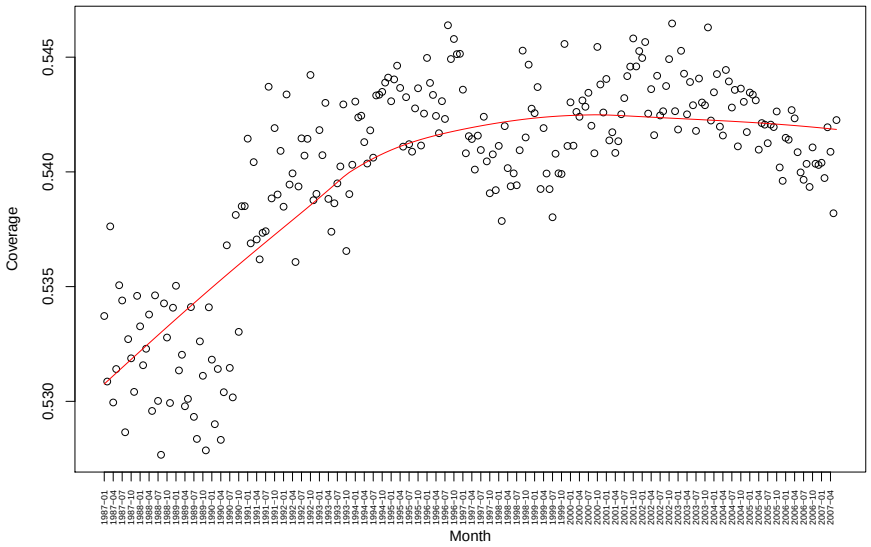


Figure 4: Monthly text coverage using stopwords in the NYT corpus.

# 7 Conclusion and Outlook

This paper has introduced an approach applying the distributed programming paradigm MapReduce to advance performance of suitable text mining tasks in R. We showed that distributed memory systems can be effectively employed within this model to preprocess large data sets by adding layers to existing text mining infrastructure. We also indicated that data parallelism can very easily be achieved using such an integrated framework without altering the handling of current text mining software available in R (i.e., the **tm** package). This is done via the class `'DistributedCorpus'` implemented in package **tm.plugin.dc**. Appropriate methods make use of a distributed storage (HDFS) and a corresponding distributed computing framework (MapReduce). A benchmark experiment showed that applying MapReduce in combination with R on text mining tasks is a very promising approach. The results presented in this paper show a significant performance gain over the sequential code as well as very good scalability when employing the distributed memory model and thus avoiding data transfers. Such an approach allows to process large data sets like the NYT corpus and to use R's rich statistical functionalty for large scale text mining applications. We show this in a culturomics application scenario.

Open issues for further investigations are as follows. Extensions to **tm** in terms of parallel programming models employed are desirable since in particular multi-core systems are now very common, and the number of processors per chip is growing. Thus, it seems natural to provide appropriate **tm** methods which support parallel computation on machines with multiple cores or CPUs, e.g., via the **multicore** package (Urbanek, 2009).

Furthermore, it needs to be investigated which programming models are promising to be employed given a certain text mining application. Criteria for choosing an execution plan are: corpus size, available RAM on main workstation, overhead of available frameworks, and access to distributed computing facilities (number of computing nodes, RAM, network topology, etc.). Ideally, **tm** would implement suitable heuristics for choosing such a execution plan given a set of criteria automatically.

# References


Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.

David M. Blei and John D. Lafferty. A correlated topic model of science. *The Annals of Applied Statistics*, 1(1):17–35, 2007.

Dhruba Borthakur. HDFS architecture. *Document on Hadoop Wiki*, 2010. URL `http://hadoop.apache.org/common/docs/r0.20.2/hdfs_design.html`.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, pages 137–150, 2004. URL `http://labs.google.com/papers/mapreduce.html`.

Ingo Feinerer. **tm***: Text Mining Package*, 2011. URL `http://tm.r-forge.r-project.org/`. R package version 0.5-5.

Ingo Feinerer, Kurt Hornik, and David Meyer. Text mining infrastructure in R. *Journal of Statistical Software*, 25(5):1–54, March 2008. ISSN 1548-7660. URL `http://www.jstatsoft.org/v25/i05`.

John F.Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. *IDC White Paper*, 2008. URL `http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf`.

W. Nelson Francis and Henry Kučera. *Frequency Analysis of English Usage: Lexicon and Grammar*. Houghton Mifflin, 1982.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, New York, NY, USA, October 2003. ACM Press. doi: http://doi.acm.org/10.1145/1165389.945450.

Thomas L. Griffiths and Mark Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences of the United States of America*, 101:5228–5235, 2004. doi: 10.1073/pnas.0307752101.

Richard Hogg and David Denison, editors. *A History of the English Language*. Cambridge University Press, 2008.

Kurt Hornik, David Meyer, and Christian Buchta. *slam: Sparse Lightweight Arrays and Matrices*, 2010. URL `http://CRAN.R-project.org/package=slam`. R package version 0.1-9.

Ralf Lämmel. Google's MapReduce programming model—revisited. *Science of Computer Programming*, 68(3):208–237, 2007.

David Lewis. Reuters-21578 text categorization test collection, 1997. URL `http://www.daviddlewis.com/resources/testcollections/reuters21578/`.

David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. RCV1: A new benchmark collection for text categorization research. *The Journal of Machine Learning Research*, 5:361–397, 2004.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1994.

Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, 2003.

Jean-Baptiste Michel, Yuan Kui Shen, Aviva Presser Aiden, Adrian Veres, Matthew K. Gray, The Google Books Team, Joseph P. Pickett, Dale Hoiberg, Dan Clancy, Peter Norvig, Jon Orwant, Steven Pinker, Martin A. Nowak, and Erez Lieberman Aiden. Quantitative analysis of culture using millions of digitized books. *Science*, 331:176–182, 2011. doi: 10.1126/science.1199644.

Martin Porter. An algorithm for suffix stripping. *Program*, 3:130–137, 1980.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

Anthony Rossini, Luke Tierney, and Na Li. Simple parallel statistical computing in R. *UW Biostatistics Working Paper Series*, (Working Paper 193), 2003. URL `http://www.bepress.com/uwbiostat/paper193`.

Evan Sandhaus. The New York Times annotated corpus, 2008. URL `http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2008T19`.

Markus Schmidberger, Martin Morgan, Dirk Eddelbuettel, Hao Yu, Luke Tierney, and Ulrich Mansmann. State of the art in parallel computing with r. *Journal of Statistical Software*, 31(1): 1–27, 8 2009. ISSN 1548-7660. URL `http://www.jstatsoft.org/v31/i01`.

Paul Tetlock. Giving content to investor sentiment: The role of media in the stock market. *The Journal of Finance*, 62(3):1139âĂŞ1168, 2007.

The Apache Software Foundation. Hadoop, 2010. URL `http://hadoop.apache.org/core/`. Release 0.20.2.

Stefan Theußl and Ingo Feinerer. *hive: Hadoop InteractiVE*, 2011a. URL `http://CRAN.R-project.org/package=hive`. R package version 0.1-10.

Stefan Theußl and Ingo Feinerer. *Text Mining Distributed Corpus Plug-In*, 2011b. URL `http://CRAN.R-project.org/package=tm.plugin.dc`. R package version 0.1-6.

Luke Tierney, Anthony Rossini, Na Li, and Hana Sevcikova. *snow: Simple Network of Workstations*, 2008. R package version 0.3-3.

Simon Urbanek. *multicore: Parallel processing of R code on machines with multiple cores or CPUs*, 2009. URL `http://www.rforge.net/multicore/`. R package version 0.1-3.

Hao Yu. Rmpi: Parallel statistical computing in R. *R News*, 2(2):10–14, 2002.

Hao Yu. *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface)*, 2010. URL `http://www.stats.uwo.ca/faculty/yu/Rmpi`. R package version 0.5-9.

# A Installing Hadoop

In this section we describe how to set up the Hadoop framework in (pseudo) distributed operation. This description is based on `http://hadoop.apache.org/common/docs/r0.20.2/quickstart.html` and focuses on the Linux operating system as this platform is suggested on the website for production use. For a more detailed installation instruction and for other platforms than Linux we refer to the above website.

## Basic Hadoop environment

Since Java and the command line utility `ssh` are needed for operation both programs have to be installed on all involved machines (on our test platform we used Java version 1.6 update 16 and OpenSSH 5.3). In order to install Hadoop we first downloaded the compressed archive of the framework following the instructions on the release website (`http://hadoop.apache.org/common/releases.html`) and then uncompressed the contents to a directory on a file system which is accessible by all machines running Hadoop (subsequently referred to as `HADOOP_HOME`). E.g., on a single workstation this directory is usually located somewhere on the corresponding local disk. In case of a cluster of workstation, this directory is usually located on a network file system (e.g., NFS). Several components of the Hadoop framework need to know the path to the installation directory, thus it is specified on each machine via the environment variable *`HADOOP_HOME`* (this variable has to be added to the user's environment on each machine if it is not done automatically).

Subsequently, few to several changes have to be made in configuration files located in the `$HADOOP_HOME/conf` directory depending on the desired operating mode. First, the path to a working Java environment has to be specified in '`hadoop-env.sh`'. Second, since we want to operate the framework in *pseudo distributed* (i.e., use the HDFS facilities on a single workstation) or *fully distributed* mode, one needs to setup the configuration files '`core-site.xml`', '`mapred-site.xml`', '`masters`', and '`slaves`'. An example configuration of a pseudo distributed system can be found in Appendix B.

Alternatively, on Debian GNU/Linux systems the Hadoop distribution can be installed from the *unstable* repository very easily using the package manager **aptitude**.

```
aptitude install hadoop-namenoded hadoop-datanoded hadoop-tasktrackerd \
hadoop-jobtrackerd hadoop-secondarynamenoded
```

This completely installs Hadoop and corresponding tools without further interaction needed. Nevertheless, the HDFS has to be configured as described below. Pre-configured packages for other Linux distributions can be obtained e.g., from `http://www.cloudera.com`.

To verify the installation one can execute `hadoop version` on the command line which returns the version number of the installed software package.

### Hadoop distributed file system

The final step before running Hadoop jobs in a pseudo distributed environment involves formatting the HDFS (the default system path on Debian is '`/var/lib/hadoop/cache`'). The command `hadoop namenode -format` serves for this purpose.

The configured Hadoop cluster can be started via two alternative approaches. In the first approach one can use the command `$HADOOP_HOME/bin/start-all.sh` on the command line (non-Debian) or via daemon startup scripts in '`/etc/init.d`' (Debian). This is especially useful if an integration to cluster grid engines is desired in order to automate the startup process. In the second approach the Hadoop cluster can be started directly within R using the `hive_create()` and `hive_start()` functions in **hive**. The resulting `'hive'` object, representing the information about the configured cluster is stored for further use with the help of the function `hive()`. Usually, this is done automatically when the package loads given that the Hadoop framework is referenced to via the `HADOOP_HOME` environment variable or if the executables are in the PATH and configurations are put in '`/etc/hadoop`' (as is with the Debian packages). However, there is a known issue with IPv6 (see `http://wiki.apache.org/hadoop/HadoopIPv6`). Thus, if the Hadoop framework does not start correctly setting the configuration option `net.ipv6.bindv6only` to `0` in '`/etc/sysctl.d/bindv6only.conf`' will help.

```
> library("hive")
> hive()

[1] NA

> hadoop_home <- "~/lib/hadoop-0.20.2"
> hive(hive_create(hadoop_home))
> hive()

HIVE: Hadoop Cluster
- Avail. datanodes: 1
'- Max. number Map tasks per datanode: 2
'- Configured Reducer tasks: 0

> summary(hive())

HIVE: Hadoop Cluster
- Avail. datanodes: 1
'- Max. number Map tasks per datanode: 2
'- Configured Reducer tasks: 0
---
- Hadoop version: 0.20.2
- Hadoop home/conf directory: /home/theussl/lib/hadoop-0.20.2
- Namenode: localhost
- Datanodes:
'- localhost

> hive_is_available()

[1] FALSE

> hive_start()
> hive_is_available()

[1] TRUE
```

When the Hadoop cluster is up and running one can retrieve Hadoop status and job information by visiting specific websites provided by the built-in web front end, two of them are of higher

interest. Assuming that Hadoop is configured to run on `localhost` the websites can be accessed via a standard web browser opening `http://localhost:50030` (JobTracker) and `http://localhost:50070` (NameNode), respectively. From the former one can retrieve information about running, completed or failed jobs. We found the log files showing the error output for the distributed batch jobs very useful as they helped us debugging the R scripts generated by the **hive** package for each mapper and reducer. In the latter website one can inspect the configuration of the HDFS and browse through the file system.

The HDFS can be accessed directly in R with `DFS_*()` functions.

```
> DFS_list("/")

[1] "home" "tmp"

> DFS_dir_create("/tmp/test")
> DFS_write_lines(c("Hello HDFS", "Bye Bye HDFS"), "/tmp/test/hdfs.txt")
> DFS_list("/tmp/test")

[1] "hdfs.txt"

> DFS_read_lines("/tmp/test/hdfs.txt")

[1] "Hello HDFS"   "Bye Bye HDFS"
```

For a complete reference to implemented `DFS_*()` functions see the **hive** help pages.

# B Hadoop configuration

We recommend to set the following parameters in the corresponding configuration files in order to setup pseudo distributed mode on a single machine. In case of a Debian installation only the files '`masters`' and '`slaves`' have to be created in the '`/etc/hadoop/conf`' directory (each containing `localhost`).

'`core-site.xml`' contains:

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/home/${user.name}/tmp/hadoop-${user.name}</value>
  </property>
</configuration>
```

'`mapred-site.xml`' contains:

```
<configuration>
<property>
    <name>mapred.job.tracker</name>
    <value>hdfs://localhost:9001</value>
  </property>
  <property>
```

```
    <name>mapred.tasktracker.map.tasks.maximum</name>
    <value>2</value>
    <description>The maximum number of map tasks that will be run
    simultaneously by a task tracker.
    </description>
  </property>
  <property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>2</value>
    <description>The maximum number of reduce tasks that will be run
    simultaneously by a task tracker.
    </description>
  </property>
</configuration>
```

'`hdfs-site.xml`' contains:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

'`masters`' contains the IP address of the master servers (or `localhost`) to this file.

'`slaves`' contains the IP address of all data nodes (or `localhost`) to this file.