Journal of Data Science 22 (2), 208–220
April 2024

DOI: 10.6339/24-JDS1132
Computing in Data Science

A Platform for Large Scale Statistical Modelling in R

Jason Cairns[1,*], Simon Urbanek[1], and Paul Murrell[1]

[1]*Department of Statistics, University of Auckland, Auckland, New Zealand*

Abstract

With the growing scale of big datasets, fitting novel statistical models on larger-than-memory datasets becomes correspondingly challenging. This document outlines the development and use of an API for large scale modelling, with a demonstration given by the proof of concept platform *largescaler*, developed specifically for the development of statistical models for big datasets.

Keywords *big data; distributed computing; modelling*

1 Introduction

The rate of growth of datasets continues to outpace attempts to engage meaningfully with them, as individual computer memory limits are increasingly exceeded (Kleppmann, 2017). At the scale of big data, speed also becomes a constraining factor, with concurrency and parallelism being of increasing importance. The aim of a statistician seeking to gain novel insight from such datasets commonly includes the interactive use of a complex statistical model, often implemented from scratch using R. The main repository for packages in R, the *Central R Archive Network* (CRAN), hosts a multitude of packages engaged in this area of high-performance computing, with a dedicated Task View available on CRAN highlighting many of them (Eddelbuettel, 2024). With review of the existing packages, no single system satisfactorily provides the capacity to meet the demand imposed in providing an expressive and extensible means of conveying a distributed statistical algorithm in R, in a manner that is immediately familiar to regular R users. This is precisely the contribution that *largescaler* seeks to offer as a platform, with a fuller review of features in Section 5.

Existing systems that do come close to meeting the demand provide direction regarding how to gain insight from larger-than-memory datasets. Most importantly, the standard solution for handling big data is to operate over a distributed system (Boja et al., 2012). Several systems have seen widespread use within the context of data and machine learning pipelines, such as *Spark* (Zaharia et al., 2016) and *Hadoop Map-Reduce* (Shvachko et al., 2010). For the statistician mostly familiar with R, these systems provide APIs to R where distributed data may be manipulated and pre-made models fitted. However, these APIs are often found lacking when attempted to be used for the creation of complex statistical models that don't come pre-packaged, due to this not being their primary use-case, and R not being their target language.

For instance, *sparklyr* is an interface to Spark from within R (Luraschi et al., 2020). The user connects to spark and accumulates instructions for the manipulation of a *Spark* DataFrame object using *dplyr* commands, then executing the request on the *Spark* cluster. It works fluidly when intentions meet the *Spark* paradigm, but due to it being only an interface to an external

*Corresponding author. Email: jason.cairns@auckland.ac.nz.

Received July 30, 2023; Accepted April 7, 2024

system, it is commonly required to program directly to *Spark* specifications, using Scala, when more complex and custom analyses are required.

The most notable general interface in R that has capabilities for large data is the the *foreach* package (Weston, 2020). The backends are provided by other packages, typically named with some form of "DoX", and can be drivers to external or R-specific systems. Parallelisation is enabled by some backends, with *doParallel* allowing parallel computations (Weston, 2019a), *doSNOW* enabling cluster access through the *SNOW* package (Weston, 2019b), and *doMPI* allowing for direct access to the Message Passing Interface (MPI) library for parallel computing (Weston, 2017). *foreach* remains an exceptional and general interface, but can only be as performant as the backend driver packages. The necessity of persisting large data objects and allowing for complex manipulation of them is not met by any of the reviewed package backends.

The most complete self-contained interface that was developed specifically for statistical modelling in R is given by the *pbdR* collection. This collection allows for distributed computing with R (Schmidt et al., 2020), with the name being the abbreviation of Programming with Big Data in R. The packages include high-performance communication and computation capabilities, including *RPC*, *ZeroMQ*, and *MPI* interfaces. The collection is extensive, offering several packages for each of the main categories of application functionality, communication, computation, development, I/O, and profiling. While incredibly performant and great care has been taken to provide an interface as close to native R as possible, the provided packages still require an MPI form of shared program amongst all cluster nodes, which presents an unfamiliar form of program specification to most R users. In addition, more complex analyses requiring lower-level data manipulation end up increasingly involved with the complexities of MPI, which was authored as a far more general system than its use-case in *pbdR*, thereby possessing a far greater complexity than necessitated. Regardless, the direction provided of distributing large data across compute nodes, as a means of statistical interaction, is a concept that has been proven well by this package collection.

In a similar vein is the *bigmemory* collection. Core to this package is the creation of "massive matrices" through a "big.matrix" S4 class with a similar interface to 'matrix' (Kane et al., 2013). These matrices may take up gigabytes of memory, typically larger than RAM, and have simple operations defined that speed up their usage. A variety of extension packages are also available that provide more functionality for big.matrices. The massive capacity of big.matrices is given through their default memory allocation to shared memory, rather than working memory as in most R objects. Importantly, *bigmemory* is single-node only, so can't benefit from the scaling capabilities that a distributed system would provide, nor does it face the same architectural challenges. Of note within the *bigmemory* collection is *biglasso*, which extends *bigmemory* matrices to allow for lasso, ridge and elastic-net model fitting. It can take advantage of multicore machines, with the ability to be run in parallel. The package *biglasso* is described in detail in Zeng and Breheny (2017). Comparability is again limited between *biglasso* and *largescaler*, as *biglasso* is bound to a single machine. In terms of interface, however, the most notable difference is the support by *largescaler* for arbitrary data types with split and combine methods defined for chunking, while *bigmemory* packages depend upon the matrix-based data structure.

Within the motivating context provided, the *largescaler* project has sought to provide an API for working with larger-than-memory data in R, allowing the developer to manipulate distributed data and create arbitrarily complex iterative models with which to fit to the data, over a self-contained user-specified computing cluster.

The following sections offer a structured exploration of *largescaler*. The methodological details of the *largescaler* API construction are given in Section 2. Section 3 describes the API itself. This is followed in Section 4 with example usage of the API to develop a distributed model. Finally, a discussion and summary of features is given in Section 5.

2 Construction of a Large-Data Statistical System API for R

The construction of such *largescaler* has roots in a theoretical reconsideration of precisely what determines the necessary components of a system capable of complex statistical modelling with larger-than-memory data in a distributed fashion. The structure of such considerations have been defined principally by the response to linguistic challenges facing such an API.

In order to perform calculations on larger-than-memory data, we need some means of **representing** the data, in order for it to be tangible and useful. All objects, regular or irregular, are facilitated in their access and manipulation by way of an **object system**, which is an organised manner of interaction with objects. The central constraint is that the data is larger than computer memory. Data must therefore be split into pieces in order to fit into memory. Such a structure, common among big data systems, is known as a shard, or a **chunk**. The chunk thus serves as the lowest level object manipulable by the user, and defines the lower level of the layered object system. The end-user is typically not interested in the chunks making up the object. Therefore higher-level objects are introduced as compositions of chunks, serving as abstract collections. Such higher-level objects are termed **distributed objects**, and serve as the primary data structure interacted with by the high-level user, with a representative example depicted in Figure 1.

With the establishment of an object system, attention may then be turned to the actualities of **computation**. For each chunk, each operation must possess the following properties:

- Operations on chunks must have some means of access to all of the relevant chunk arguments to operate on together;
- The result of the operation must be stored somewhere.

A **chunk function applicator** is required that takes a function and some chunks, sending the operation to a node which applies the function to the specified chunks. The applicator requires some means of access to the respective chunks. An equivalent **distributed object function applicator** is to be defined in a similar manner, though for distributed objects. Likewise, following the conclusion of the operation, some means of access to the result is required - the result is to be kept as a distributed object itself, allowing for further iterative action on it.

Some of the concerns relevent to the applicators include questions of argument sourcing and interaction. With regards to sourcing, the optimal node should be chosen to perform the operation in order to minimise data movement. All chunks not present in the operating node process are needed to be made available in order for the operation to take place. Interaction between chunks, especially in the context of the distributed object function applicator, requires some definition of how these distributed arguments are to be treated in multiple. For instance, whether to recycle chunks, and how they are to be appropriately combined.

After laying out the distributed objects and the operations that they may be engaged in, we reach the limits of the construction of the system. The mechanical aspects of the system developed, focus is turned to the more complex and precise notions of the arrangement of these components; let us expand the construction limits now by exploring and describing the component of time as it relates to the system - specifically, **concurrency** within the system. Here,

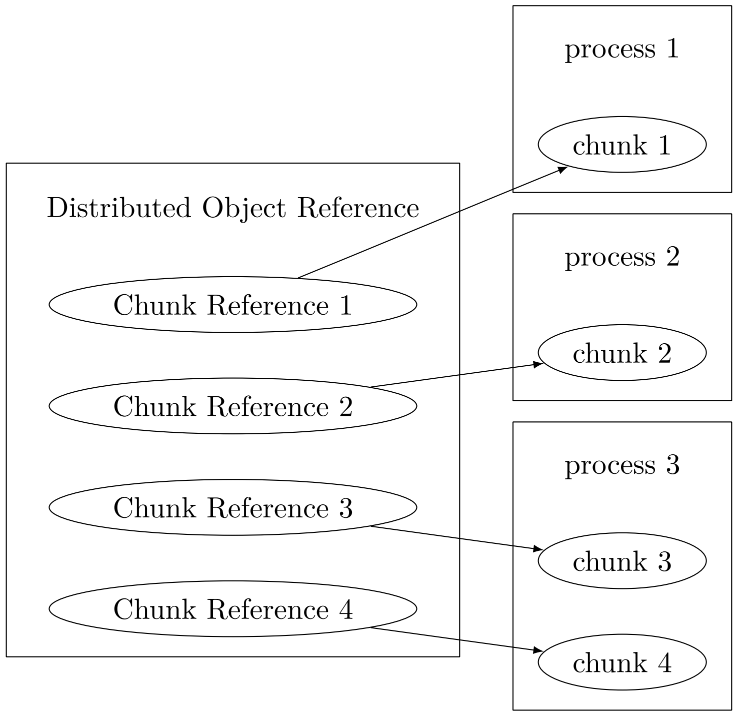

Figure 1: Distributed object, showing chunks and their references across disparate nodes.

we treat concurrency in the manner of Pike (2012), where it is used to refer to the composition of independently executing processes. Considering the system as a whole, and determining the independently executing processes within it and how they may be composed, it is clear that a distributed system has significant room for concurrency, which serves to aid speed and memory usage. Assuming distinct nodes for performing operations on chunks, a remarkable thing would occur on the requesting client: nothing happens. The processing occurs in an entirely separate memory space, with a different processor. Were the client performing operations on regular, non-chunk objects, the processing would bind up the client, and it would block until the operation completed. This recognition opens up a broad mix of possibilities and complications. In terms of possibilities, the potential for chunk operations to be non-blocking to the client means that operations may be **asynchronous**, which would allow for significantly more efficient ordering of events within the system. Long operations on chunks may take place side-by-side with client-side operations on local objects, neither interfering with each other. The converse of such a possibility is the new potential for **race conditions**, where the unordered timing of events may lead to undesirable behaviour.

Through the defining feature of distributed data being incapable of direct reference, we have established that there must be some entity through which to indirectly interact with the distributed data. This entity we have given the appellation of **'Distributed Object Reference'**, and serves as a proxy which is manipulated by the user. This relationship between the reference and the distributed data is known as **"adequacy"** in formal semantics, with the role of the distributed data as the object of indirect interaction being called the **"referent"** (Gordon, 1984). It must be established that the reference itself may be of conceptual interest. The referent chunks themselves, if completely transparent to a reference, possess no information on their relation to each other. This information needs to be captured somewhere accessible to the user, with most distributed algorithms being dependent on the knowledge of how the underlying chunks relate, in dimension or quantity. Generalising the terminology of Quine (1979), the distinction between the **use** and **mention** with respect to distributed objects lies in the contrasting operations that may be intended of a distributed object reference: interaction may be intended for either the referent distributed object (use), or the reference itself (mention). Such semantics of access to the reference and referent serve as the basis for much of metaprogramming, and may be considered a valuable component of a distributed statistical system.

3 User Interface

The *largescaler* platform serves as a functioning proof of concept system, capable of performing complex statistical analyses. The implementation of this system makes use of a layered approach, wherein each layer targets a different category of user. The interface of *largescaler* is the principal new contribution by this project, delivering a novel means of interacting with distributed data through meaningful primitives defined at every level of abstraction. A key offering of the layered approach is the ease by which a user of the package can traverse the levels as needed, with irrelevant information remaining hidden until required. The levels of abstraction correspond to users of the packages, given as the following:

Analyst A user solely interested in using the provided models and statistical functions in order to attain insight into some larger-than-memory data, typically a distributed data frame. All details of distribution are abstracted away.

Researcher A user seeking to develop their own distributed statistical models. Distributed objects are to be considered as singular cohesive objects.

Developer A user seeking greater expressivity in the definition of statistical models. Chunks are considered a relevant concern to be manipulated directly.

Architect A user intending to directly modify the network topology of the distributed system, mainly in order to attain major efficiency gains.

Each of the users are served by the packages composing the *largescaler* framework, in turn serving a logical layer of abstraction. This mapping is given in Table 1.

Owing to considerations of space, this description of the user interface will focus solely on the key offerings of the *largescaler* API, with descriptions of other components and packages having a fuller treatment in Cairns (2024). Use of the API is given by way of example below.

Consider first a simple operation of summation. We have as our object of summation some vector x that is broken up into i non-overlapping chunks, where each chunk x_i is itself a vector consisting of a subset of the elements of x. Each chunk may itself be indexed over its elements by j, with the jth element of chunk x_i denoted as x_{ij}. Summation can be defined for a chunked vector x_i by Equation 1, using associativity to render the sum of the whole as the sum of the sum of the parts.

$$\sum x = \sum_i x_i = \sum_i \sum_j x_{ij}. \tag{1}$$

In an effort to maintain as close proximity as possible between the mathematical description and the provided interface, this may be written in *largescaler* as in the following `R` code:

```
d(sum)(x) |> emerge() |> sum()
```

Here, the `d()` applicator function transforms the base `sum()` function to work over distributed objects, in this example given by `x`. The `sum()` function is therefore sent to each chunk,

Table 1: A mapping of logical layers, users and the respective packages provided by the *largescaler* framework.

Layer	User	Package
Model Usage	Analyst	*largescalemodels*
Model Description	Researcher	*largescaleobjects*
Cluster Interaction	Developer	*chunknet*
Communication	Architect	*orcv*

yielding a new distributed object as the output. This output distributed object can be brought back to the requesting client and combined using the provided `emerge()` function, yielding a regular R numeric vector of sums. This vector of sums may then be summed as normal, providing the final result. The given example is actually a very simple application of map-reduce, and could effectively serve as the `sum()` method for distributed objects

Consider something slightly more complex: the arithmetic mean. Again, a chunked mathematical description is given in Equation 2, where $\dim(x)$ is the dimension of the vector argument x, and $\dim(x_i)$ then being the dimension of the specific chunk x_i

$$\overline{x} = \frac{\sum_i x_i}{\dim(x)} = \frac{\sum_{ij} x_{ij}}{\sum_i \dim(x_i)}. \tag{2}$$

A related means of specification through *largescaler* is possible, given in the following code:

```
sum(x) / sum(d(length)(x))
```

Here we build on the distributed sum introduced above, but the total length of the distributed object is relevent as the denominator. Assuming a `sum()` method for distributed objects as described, and the math group generic defined in a similar fashion, the denominator is defined using the same `d()` function that sends a `length` computation to all of the chunks.

Finally, consider the cumulative sum. It is important in this case to think of chunks as being in series, which is determined by the structure of the distributed object reference. The main difference between a non-distributed and a distributed version of cumulative sum is that for each chunk in the series, computation requires the cumulative sum of the previous chunk as a starting value. Here, we have the cumulative sum S indexed by i, with the next component in the series x. When generalised to a chunked cumulative sum, there is the additional index of the jth chunk, where the total count of observations in the jth chunk is n_j. Using a chunked mathematical description, cumulative sum may be described by Equation 3.

$$\begin{gathered} S_i = S_{i-1} + x_i, \quad S_0 = 0 \\ \iff S_{i,j} = S_{i-1,j} + x_{i,j}, \quad S_{0,j} = S_{n_{j-1},j-1}, \quad S_{0,0} = 0. \end{gathered} \tag{3}$$

This can be expressed in a functional manner using the *reduce* operator, also known as a *fold*, and the *largescaler* framework provides a distributed form of such a function, where the results of one chunk are sent as the initial value to the reduce function as applied to the next chunk and so on in series. An example of a reduce operation is given in Figure 2.

This is put to use for cumulative sum by *largescaler* by the following code:

```
dReduce(cumsum, x) |> emerge()
```

A distributed object is returned that by default just holds the final accumulation, consisting of one single chunk.

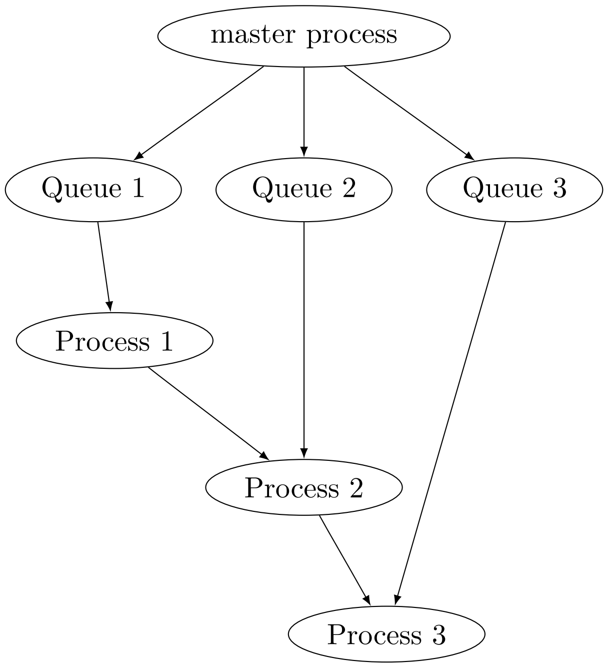

Figure 2: Example distributed reduce pattern from controlling process.

4 A Distributed Model in Largescaler

We now move to the central problem that prompted this work; defining a novel distributed statistical algorithm. Rather than detour with a truly novel algorithm, it is prefereable to engage with something that is familiar, but holds a fairly generic form that novel analyses often share. Let's consider distributed LASSO regression, using the Alternating Direction Method of Multipliers (ADMM), as described by Mateos et al. (2010). We begin by assessing the mathematical form in Subsection 4.1, followed by the "standard R" means of writing a chunked algorithm in contraposition with the *largescaler* manner in Subsection 4.2.

4.1 Distributed LASSO Mathematical Definition

This section seeks to give a brief taste of what a mathematical formulation for a distributed statistical model takes, in order to demonstrate the semantic and syntactic similarity to *largescaler* in Subsection 4.2. The reference for the ADMM LASSO algorithm described in this text is from Boyd et al. (2011), along with a richer description and significant background.

We begin with a description of the input data, as given in Equation 4.

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_N \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix} \qquad (4)$$
$$A_i \in \mathbb{R}^{m_i \times n}, \quad b_i \in \mathbb{R}^{m_i}$$

The starting data includes a column block matrix of explanatory variables, A, consisting of N submatrices. This is equivalent to a distributed object consisting of N chunks. Here, each chunk is of the standard form where rows are individual observations and columns are variables. We also have a block matrix b of the same number of chunks, with each chunk being the column vector of response variables to the corresponding A chunks.

The standard form of the LASSO as an optimisation problem is expressed in Equation 5.

$$\begin{aligned} \text{minimise} \quad f(x) + g(z) \quad \text{subject to} \quad x = z\text{where} \quad f(x) &= \frac{1}{2} \|Ax - b\|_2^2 \\ g(z) = \lambda \, |z|_1 \end{aligned} \tag{5}$$

The body of the ADMM loop is given by Equation 7. Of note is the complexity, the presence of iteration, and the interactions between sets of chunks getting reduced and emerged. Here, we have that $\rho > 0$ is a *penalty parameter* and S is a soft thresholding operator, with one formulation given by Boyd et al. (2011) as Equation 6.

$$S_\kappa(a) = \begin{cases} a - \kappa & a > \kappa \\ 0 & |a| \leqslant \kappa \\ a + \kappa & a < -\kappa \end{cases} . \tag{6}$$

$$\begin{aligned} x_i^{k+1} &:= (A_i^T A_i + \rho I)^{-1}(A^T b + \rho(z^k - u_i^k)), \\ z^{k+1} &:= S_{\frac{\lambda}{\rho N}}(\overline{x}^{k+1} + \overline{u}^k), \\ u_i^{k+1} &:= u_i^k + x_i^{k+1} - z^{k+1}. \end{aligned} \tag{7}$$

4.2 Distributed LASSO R Description

This subsection gives both base R syntax for working with a local chunked dataset, as well as the minimal changes that are required when using *largescaler* to transform the expressions to handle distributed data. The core substance of this subsection is to demonstrate the ease with which R is able to meet a mathematical definition, and the successive ease by which *largescaler* is able to turn that into a truly distributed algorithm. As in Subsection 4.1, the working example is given of distributed LASSO. Consider first some chunked data, given as a diff below. In the diff, semantic differences are demarcated through underlining, with shared code given centrally, with no dividing line. On the left of the diff we have how the LASSO as described might be encoded in the absence of the API, assuming that the data fit into memory, and on the right, we make use of *largescaler* with no such constraining assumption.

In the *largescaler* code, distributed data may come from multiple files and multiple hosts holding the chunks, and this is easily provided for. The distribution comes with the necessity to carefully differentiate between the reference of the distributed object, and the distributed object itself, hence the use of the `Ref()` function as provided by *largescaleobjects*. The ability to explicitly distribute local values to particular locations is also demonstrated here. In this illustration of the LASSO with *largescaler*, it is assumed that the input matrix chunk files have been precomputed and written to disk, which *largescaler* provides the facility for. Furthermore, it is assumed that `rho` and `lambda` are scalar numerics. Initialised values for these parameters and the function definition for the soft thresholding operator `S` are passed over in this illustration because they serve to distract from the core demonstration. They are the same in both the distributed and non-distributed versions, and are available in the source code provided in the Supplementary Materials.

```
R                                          | largescaler
---------------------------------------------------------------------------------------
                                           |
A <- read.csv("~/filepath/A")              | A <- read.dmatrix(c("host1:~/filepath/A1",
                                           |                     "host2:~/filepath/A2"))
b <- read.csv("~/filepath/b")              | b <- read.dmatrix(c("host1:~/filepath/b1",
                                           |                     "host2:~/filepath/b2"))
M_N <- dim(A)                              | M_N <- dim(Ref(A))
                                           |
                      m <- ncol(A)
                      S_z <- S(lambda/(rho*M_N\2\))
                      z_curr <- rep(1, m)
                                           |
x_curr <- u_curr <- rep(list(z_curr),      | x_curr <- u_curr <- distribute(z_curr,
                                     N)    |                               where=A)
                                           |
```

The layout of the data is followed by the iterative loop, given in the below diff. Within the iterative loop, we can see that very little is actually needed to be changed in order to distribute this algorithm. We make use of a function that operates on distributed objects which we define ourselves in the successive diff, as well as the emerge to bring the distributed local as we saw before.

```
R                                          | largescaler
---------------------------------------------------------------------------------------
          while (l1_norm(z_curr - z_prev) >tolerance) {
              x_prev <- x_curr; z_prev <- z_curr; u_prev <- u_curr
                                           |
x_curr <- mapply(x.update, x_prev,         | x_curr <- d.x_update(x_prev, A, b,
                      A, b, u_prev,        |                  u_prev, rho, z_prev)
     MoreArgs = list(rho, z_prev))         |
z_curr <- S_z(rowMeans(x_curr) +           | z_curr <- S_z(rowMeans(emerge(x_curr)) +
              rowMeans(u_curr))            |               rowMeans(emerge(u_curr)))
u_curr <- mapply(function(u_prev,          | u_curr <- u_prev + x_curr - z_curr
                  x_curr, z_curr)          |
        u_prev + x_curr - z_curr,          |
                  u_prev, x_curr,          |
         MoreArgs = list(z_curr))          |
                                           |
                                           }
                                        z_curr
```

Note the significantly simplified and reduced logic in switching to *largescaler*, which bears a far closer resemblance to the mathematical description. The `x_update()` function given above is exemplary of the approach provided by *largescaler*, which allows for the switching of a local to a distributed function through the higher-order `d()` function, as demonstrated below:

R	largescaler
`x_update <- function(x_prev, A, b,`	`d.x_update <- d(function(x_prev, A, b,`

`u_prev, rho, z_prev) {`
`optim(x_prev, function(x_prev)`
`(1/2)*l2_norm(A %*% x_prev - b)^2 +`
`(rho/2)*l2_norm(x_prev - z_prev + u_prev)^2)$par`	

`}`	`})`

And this serves to define a distributed LASSO, using ADMM.

Now, in practical use, we have the following example wrapping the given distributed LASSO description, reproducibly run from the supplementary code files through installing the provided packages and running `make test-lasso` from within the *largescalemodels* directory. Distributed objects `dA` and `db` are distributed matrices provided by the package for testing purposes. The dataset `dA` is a simulated matrix possessing dimensions of 20 × 5e7, and the returned result is a vector of estimated coefficients for the LASSO.

```
> library(largescalemodels)

> print(dA)
server bound to address 192.168.22.224 at port 36125
Distributed Object
Consisting of 6 chunks

> print(db)
Distributed Object
Consisting of 6 chunks

> dpielasso <- dlasso(dA, db, tolerance=1, rho=3, lambda=3)
Iteration:  1
Iteration:  2
Iteration:  3
Iteration:  4
Iteration:  5
Iteration:  6
Iteration:  7
Iteration:  8
Iteration:  9
Iteration:  10

> dpielasso
 [1] 24.94446984  0.19473798  1.26184713  0.26234990  0.32975632  0.09505969
 [7]  0.28556012  0.14810753 82.34335087  0.17584890  0.15710146  0.27761540
[13]  0.26950565  8.38837401  2.14788401  0.17330150  0.61847250  0.23920172
[19]  0.08070681  0.07492101
```

5 Discussion/Conclusions

Data manipulation is a basic necessity, as it is required for modelling, and is provided well by other systems. The *largescaler* platform is capable of a full set of data manipulations, including all that are provided by the *dplyr* package. Model fitting is demonstrated in the proof-of-concept *largescalemodels* package, which includes a variety of models, including linear models and generalised linear models.

A roll-call description of features which *largescaler* offers in unique combination follows:

Distributed Computation The *largescaler* platform holds the capability for distributed computation as a core component of the provided API. This is given by the `do-ccall` and `do-dcall` functions at the chunk and distributed object levels respectively.

Evaluation of User-Specified Code User-specified code is similarly run via the given `do-ccall` and `do-dcall` functions.

Native Support for Iteration The combination of the user-specified code functions with the built-in garbage collection allows for arbitrary looping without memory issues.

Object Persistence at Nodes The chunk and distributed object concepts in *largescaler* depend on exactly this property.

Support for Distributed File Systems Local filesystems over separate hosts are supported. Reading from HDFS or the like is not currently a core component of the API, but could easily be extended to allow for it.

Ease of Setup Initialisation functions are included with the framework; all that is required is that the *largescaler* packages are installed on all the hosts, as well as a reliable ssh connection to them.

Inter-Node Communication The movement from the message-queue-based approach to the final direct approach facilitated this.

Interactive Usage Interactivity has been a capability from the beginning.

Backend Decoupling As a new framework, there is only one backend available, which is provided by *orcv*. The layered approach allows for any other backend that provides the *orcv* interface to be swapped in.

Evaluation of Arbitrary Classes This is a core feature that drives the means of combination of emerged distributed objects.

Package-specific API Unique API, following the principle of least surprise, with naming and semantics similar to base R.

Methods for Standard Generics Some provided; e.g. `summary`, `ops`, etc.

Methods for dplyr Generics Provided by the *largescaleobjects* package, including true `group-by`.

Current limitations are given largely by the fact that the software itself is a Proof of Concept, used to demonstrate the possibility of combining the described features to provide a unique and expressive API for distributed statistical model definition in R. One such limitation is the necessitation of a unix machine to run it. This is due to the direct use of the *pthreads* library for low-level concurrency primitives, requiring a special port to Windows, which is out of scope for such a platform. Furthermore, there is the possibility that this impediment to access for

Windows users has also reduced the uptake of *largescaler*, thereby constraining the user base and preventing what may have otherwise been more open source contributions to fix bugs, add features, and allow for a more robust platform.

In spite of this the potential for future work remains significant, enabled by the high level of extensibility provided by the system. External systems which serve to monitor performance or take up the role of garbage collection would grant the possibility of greater reliability. Robustness could be gained through self-healing datasets, a potential that has a precedent in a current prototype, which allows for resiliance to node failure in a more efficient manner than that of the current Resilient Distributed Datasets (Zaharia et al., 2012). Further resiliance can be gained within the system through operating the location service as a distributed hash table, leaving no central point of failure in a fully peer-to-peer system.

Supplementary Material

The supplementary material includes a zipped directory of the source packages composing *largescaler*. The packages can also be accessed on GitHub through the following hyperlinks:

- orcv
- chunknet
- largescaleobjects
- largescalemodels

References

Boja C, Pocovnicu A, Batagan L (2012). Distributed parallel architecture for big data. *Informatică Economică*, 16(2): 116. MR2965745

Boyd S, Parikh N, Chu E, Peleato B, Eckstein J, et al. (2011). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1): 1–122. https://doi.org/10.1561/2200000016

Cairns J (2024). A Platform for Large-Scale Statistical Modelling in R, Ph.D. thesis, University of Auckland.

Eddelbuettel D (2024). CRAN task view: High-performance and parallel computing with r.

Gordon MJC (1984). *The Denotational Description of Programming Languages*. 1st edition. Springer, New York, NY.

Kane MJ, Emerson J, Weston S (2013). Scalable strategies for computing with massive data. *Journal of Statistical Software*, 55(14): 1–19. https://doi.org/10.18637/jss.v055.i14

Kleppmann M (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc.

Luraschi J, Kuo K, Ushey K, Allaire J (2020). *Sparklyr: R interface to Apache Spark*. R package version 1.1.0.

Mateos G, Bazerque JA, Giannakis GB (2010). Distributed sparse linear regression. *IEEE Transactions on Signal Processing*, 58(10): 5262–5276. https://doi.org/10.1109/TSP.2010.2055862 MR2722673

Pike R (2012). Concurrency is not parallelism. Heroku.

Quine WV (1979). *Mathematical Logic*. Harvard University Press, London, England. MR0695499

Schmidt D, Chen WC, de la Chapelle SL, Ostrouchov G, Patel P (2020). pbdBASE: pbdR base wrappers for distributed matrices. R package version 0.5-3.

Shvachko K, Kuang H, Radia S, Chansler R (2010). The Hadoop distributed file system. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 1–10. IEEE.

Weston S (2017). doMPI: Foreach parallel adaptor for the Rmpi package. R package version 0.2.2.

Weston S (2019a). doParallel: Foreach parallel adaptor for the 'Parallel' package. R package version 1.0.15.

Weston S (2019b). doSNOW: Foreach parallel adaptor for the 'SNOW' package. R package version 1.0.18.

Weston S (2020). *Foreach: Provides Foreach Looping Construct*. R package version 1.4.8.

Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauly M, et al. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 15–28.

Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, et al. (2016). Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11): 56–65. https://doi.org/10.1145/2934664

Zeng Y, Breheny P (2017). The biglasso package: A memory-and computation-efficient solver for lasso model fitting with big data in R. arXiv preprint: https://arxiv.org/abs/1701.05936.