Heliyon

Big data in multi-block data analysis: An approach to parallelizing Partial Least Squares Mode B algorithm

Alba Martinez-Ruiz [a,*]**, Cristina Montañola-Sales** [b,c]

[a] *Universidad Católica de la Santísima Concepción, Alonso de Ribera 2850, Concepción, Chile*

[b] *IQS-Universitat Ramon Llull (URL), Via Augusta, 390, 08017 Barcelona, Spain*

[c] *Barcelona Supercomputing Center, Centro Nacional de Supercomputación (BSC-CNS), Jordi Girona 29, 08034, Barcelona, Spain*

* Corresponding author.
E-mail address: alba.martinez.ruiz@gmail.com (A. Martinez-Ruiz).

Abstract

Partial Least Squares (PLS) Mode B is a multi-block method and a tightly coupled algorithm for estimating structural equation models (SEMs). Describing key aspects of parallel computing, we approach the parallelization of the PLS Mode B algorithm to operate on large distributed data. We show the scalability and performance of the algorithm at a very fine-grained level thanks to the versatility of `pbdR`, a `R-project` library for parallel computing. We vary several factors under different data distribution schemes in a supercomputing environment. Shorter elapsed times are obtained for the square-blocking factor 16×16 using a grid of processors as square as possible and non-square blocking factors 1000×4 and 10000×4 using an one-column grid of processors. Depending on the configuration, distributing data in a larger number of cores allows reaching speedups of up to 121 over the CPU implementation. Moreover, we show that SEMs can be estimated with big data sets using current state-of-the-art algorithms for multi-block data analysis.

Keywords: Computer science, Computational mathematics

1. Introduction

Early, mathematicians and computer scientists explored methodologies and proposed techniques to process distributed matrices to optimize computing power and profit large computer systems (Golub and Van Loan, 1996). With the software-hardware infrastructures advances, examining large data sets is gradually more feasible. That is the reason why the use of parallel computing technologies has spread by leaps and bounds in many areas (Schmidberger et al., 2009, Pacheco, 2011). This situation makes the analysis of big volumes of data a major challenge of investigating the performance, efficiency, and effectiveness of statistical methods.

From an end-user perspective, the parallelization process of an algorithm is not an easy task. It requires considering many factors, such as data distribution and data processing schema, the understanding of how available computer architectures operate to find the best way to distribute both data and tasks, or determining the appropriate dimension of data blocks for distributing data. As a result, scientific communities and companies are making available computational platforms for parallel statistical analysis, parallel computing and big data endeavors with increasing swiftness. An example of that is the website of "CRAN Task View: High-Performance and Parallel Computing with R" (Eddelbuettel, 2016) which lists a set of R packages and tools to develop parallel R-based applications, the preferred software of the statistical community. The number of applications in the list has at least doubled in the last few years. Most of them provide support to MPI (Message Passing Interface) API which is the standard in parallel computing.

Among the different existing tools (Schmidberger et al., 2009) we would like to highlight snow (Rossini et al., 2007, Tierney et al., 2011), snowfall (Knaus, 2010), parallel (included in R since R 2.14.0) and its extension doParallel (Calaway et al., 2015), Rmpi (Yu, 2009), pbdR and MapReduce. snow and snowfall rely in the typical task parallelism provided by libraries with a Master/Worker approach. They use one function to perform reductions on a whole distributed data set in parallel. Both tools have been used in several applications. For instance, Deb and Srirama (2013) used snow to process bigger gene expression data sets by parallelizing the algorithm of K-Means clustering exploiting the multicore architecture of a desktop computer and Riddick et al. (2011) took advantage of snow package to make more efficient the process of multiple drug responses using Random Forest.

In contrast with this approach, Rmpi exposes MPI routines in R but leaves the parallelization task to the user. In this way, McLeod et al. (2007) used Rmpi to reduce computations by a factor of 30 in running the Durbin-Levinson and Trench algorithms for linear time series analysis and Lê Cao and Chabrier (2008) used Rmpi to faster the classification process of high dimensional data sets. Another example is Varsos et al. (2016), who took advantage of Rmpi to implement single program

and multiple data and develop an interface to perform parallel data analysis for the R-package `vegan`. Parallel R-based packages can also be used for exploring the parameter space of simulations faster. Lawrence and Morgan (2014) used `parallel` package to improve the speed of analysis of genetic variants from a whole genome sequencing experiment, and Luo and Zhang (2015) used `parallel` package provided by R to enhance the detection and extraction of water surface area from individual LiDAR point clouds. Hofert and Mächler (2016) and Górecki and Smaga (2018) used `doParallel` to carry out parallel computations on multiple cores for the simulation of a quantitative risk management problem and multivariate functional data analysis, respectively. `MapReduce` schema in Spark and Hadoop is commonly used in cloud computing but comparisons on clusters of multicore processors show that is not very well-suited for tightly coupled problems (Schmidt et al., 2014), and Single Program Multiple Data approaches provide faster and scalable solutions (Schmidt et al., 2017).

On the other hand, Partial Least Squares (PLS) Mode B is an algorithm for building explicit estimates of standardized variables that describe the relationship between several blocks of variables. PLS has been successfully used to estimate structural equation models and has facilitated the construction and estimation of new models in areas as diverse as marketing, genomics, brain imaging and manufacturing (Esposito-Vinzi et al., 2010, Abdi et al., 2016). In contrast to a loosely coupled algorithm where operations may be easily separated and therefore computed in different processors, PLS Mode B is a tightly coupled algorithm that is composed of a sequence of dense matrix operations that must be executed and iterated in a specific order. From a distributed perspective, the coupled sequence and order of operations make difficult to follow a master-worker approach to perform a parallel implementation of the algorithm. A data parallelism approach, such as Single Program Multiple Data (SPMD), is more suitable in this case.

Recently, there has been some research in relation to the performance of multiblocks algorithms. For instance, to address the big data problem, Fu et al. (2016) proposed a distributed algorithm for Generalized Canonical Correlation Analysis (GCCA) applied to sparse matrices. In this research, each data matrix was stored in different nodes and block components were computed in parallel for each block of variables. In contrast, in our research, we studied how to partition and distribute data matrices in different nodes, and how to tune a set of parameters to achieve the best performance on High Performance Computing architectures. Other works have been published in accelerating CCA algorithms such as Yan et al. (2014) who worked with MKL (Intel Math Kernel Library) and R-project. To our knowledge, no work has been done for multiblock PLS Mode B before.

In this paper, we present a parallel implementation of the multiblock PLS Mode B algorithm. Section 2 shows an outline of the algorithm and presents some

of its features. We also introduce the framework `pbdR`, a set of `R` libraries for High Performance and Distributed Computing. This framework helps implementing the PLS tightly coupled algorithm to operate on distributed data. The versatility offered by `pbdR` to work with High Performance Computing systems with SPMD, its support to Single Process Multiple Data schema (Chen et al., 2012a, 2016), along with their extensive documentation made us choose this library as a good suitable option for parallel PLS implementation. Next, the parallel implementation is presented in Section 3, and we show how the PLS algorithm can be used in a distributed environment to process large or big data sets. Finally, several computational experiments are carried out to study the scalability and performance of the implementation examining several factors such as grid layout and number of observations under different data distribution schemes in a multicore environment in Section 4. Among other results, we found that shorter elapsed times are obtained for the square-blocking factor 16×16 using a grid of processors as square as possible and non-square blocking factors 1000×4 and 10000×4 using an one-column grid of processors. Depending on the configuration, distributing data in a larger number of cores allows reaching speedups of up to 121.

2. Background

2.1. Multiblock PLS Mode B algorithm

PLS Mode B is an iterative algorithm for building a set of standardized variables and estimate the relationships between them (Wold, 1985, Lohmöller, 1989, Tenenhaus et al., 2005, Hanafi, 2007). Let J be the number of standardized variables y_j, J the number of block of variables X_j, J the number of arbitrary initial weights vectors $\tilde{w}_j^{(0)}$ representing the relationships between variables y_j and X_j, $C[c_{jl}]$ a binary matrix with the relationships between variables y_j, $R[r_{jl}]$ a matrix with the correlations between variables y_j, and $\Theta[\theta_{jl}]$ a matrix with the sign of the correlations between variables y_j (centroid weighting scheme), $1 \leq j \leq J$, $1 \leq l \leq J$.

The algorithm repeats 3 steps until convergence: (1) outer estimation of variables y_j, (2) inner estimation of variables y_j, and (3) weight updating. One of the algorithms – the Lohmöller procedure – may be described as follows. To initialize the algorithm, we first calculate the initial weights vectors $\tilde{w}_j^{(0)}$ such that the variance of y_j is equal to one,

$$w_j^{(0)} = \frac{\sqrt{n}\tilde{w}_j^{(0)}}{\|X_j\tilde{w}_j^{(0)}\|}, \quad 1 \leq j \leq J \tag{1}$$

Algorithm 1 Lohmöller iterative algorithm

1: Choose J arbitrary initial vectors $w_j^{(0)}$, $j = 0, 1, 2, ..., J$

2: Choose an initial value for *condition*

3: Initialization

4: $\quad w_j^{(0)} = \dfrac{\sqrt{n}\tilde{w}_j^{(0)}}{\|X_j\tilde{w}_j^{(0)}\|}$

5: $\quad Y_j^{(0)} = X_j w_j^{(0)}$

6: **while** *condition* $> 10^{-16}$ **do**

7: $\quad$ **At iteration** s,

8: $\quad\quad \theta_{jl}^{(s)} = \text{sign}(r(Y_j^{(s)}, Y_l^{(s)}))$

9: $\quad\quad Z_j^{(s)} = \displaystyle\sum_{l=1, l \neq j}^{J} c_{jl}\theta_{jl}^{(s)} Y_l^{(s)}$

10: $\quad\quad \tilde{w}_j^{(s+1)} = (X_j' X_j)^{-1} X_j' Z_j^{(s)}$ (Mode B)

11: $\quad\quad w_j^{(s+1)} = \dfrac{\sqrt{n}\tilde{w}_j^{(s+1)}}{\|X_j\tilde{w}_j^{(s+1)}\|}$

12: $\quad\quad Y_j^{(s+1)} = X_j w_j^{(s+1)}$

13: $\quad\quad condition = (w_j^{s+1} - w_j^s)^2$

14: **end while**

Then, we initialize the value of the outer estimate of y_j as an exact linear combination of its variables X_j, $y_j^{(0)} = X_j w_j^{(0)}$. At this point, we repeat until convergence the next procedure. For iteration s, we calculate an inner estimate z_j of y_j,

$$z_j = \sum_{l=1, l \neq j}^{J} c_{jl} \theta_{jl}^{(s)} y_l^{(s)} \tag{2}$$

where $\theta_{jl}^{(s)} = \text{sign}(r(y_j^{(s)}, y_l^{(s)}))$. After that, we update and normalize the weights vectors $\tilde{w}_j$,

$$\tilde{w}_j^{(s+1)} = (X_j' X_j)^{-1} X_j' z_j^{(s)} \tag{3}$$

$$w_j^{(s+1)} = \frac{\sqrt{n}\tilde{w}_j^{(s+1)}}{\|X_j\tilde{w}_j^{(s+1)}\|} \tag{4}$$

Finally, we update the value of the outer estimate y_j, $y_j^{(s+1)} = X_j w_j^{(s+1)}$.

The outer estimation offers a first estimation of y_j as a linear combination of the measured variables x_{jh}. To consider the relationships between variables y_j, the sign of the correlation between them is computed in the inner estimation (centroid weighting scheme). These signs are used as coefficients to compute the auxiliary variables z_j – counterparts of variables y_j. The variables z_j are a linear combination of the variables y_i with which they are related in the structural model. The last step consists of updating weights. Here, the vector w_j of weights w_{jh} is the vector of the regression coefficients in the multiple regression of z_j on the measured variables x_{jh} (mode B). Lohmöller iterative algorithm for a serial R implementation is shown in Algorithm 1.
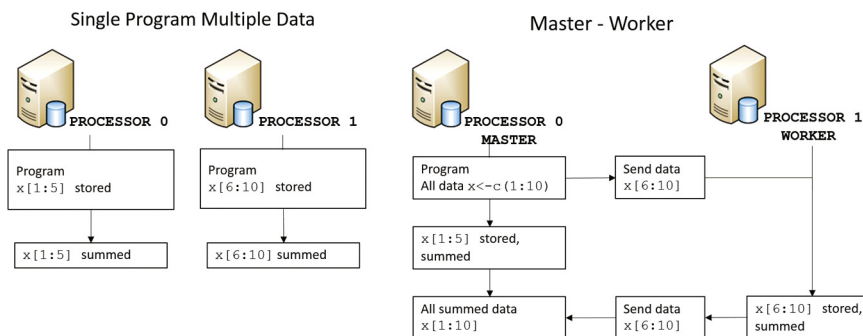

Figure 1. An example of Single Program Multiple Data and Master/Worker approaches with the `R-project` package pbdR. In SPMD, data are distributed among processors (processor rank 0 and 1) and the same program or operations are executed in each portion of data. Here, we could summarize the final result. In the Master/Worker approach, the master prepares the data and distributes the subset to the process by the worker. Both processors execute their respective calculations in their own data sets and finally, the worker sends the results to the master, which summarizes the final result.

2.2. A tightly coupled algorithm and iterations

The PLS Mode B algorithm – and as was described above – consists of a well-defined sequence of dense matrix operations that must be executed in a sequential and specific order. The algorithm is fully written in terms of matrix algebra – making it a good candidate for parallelization – and no operation can be computed if the previous one has not been fully completed. PLS Mode B algorithm is what is called a tightly coupled algorithm where the operations may not be easily separated, and therefore, processed in different processors. For instance, we can not carry out outer and inner estimations in parallel in two processors at the same time because the inner estimation depends on the output values of the outer estimation.

All of this is in contrast to a loosely coupled problem where operations may be easily split, and therefore, processed in parallel in different processors. From a parallelization perspective, tightly coupled sequences make difficult to implement a master-worker framework to perform a parallel implementation. A data parallelism approach, such as SPMD, is more suitable in this case. pbdR puts into practice a SPMD approach (Raim, 2013, Schmidt et al., 2014), thus, it is properly positioned for implementing tightly coupled algorithms and working with dense matrix operations (Figure 1).

Another important characteristic of the PLS Mode B algorithm is that the sequence of dense matrix operations – outer and inner estimation, and weight updating – is repeated until convergence. This has implications in terms of the cost of the algorithm parallelization and it imposes more communications costs among processors, so the distribution of work among them should be optimized as much as possible. All these characteristics, led us to use pbdR for the parallel implementation of the PLS algorithm.
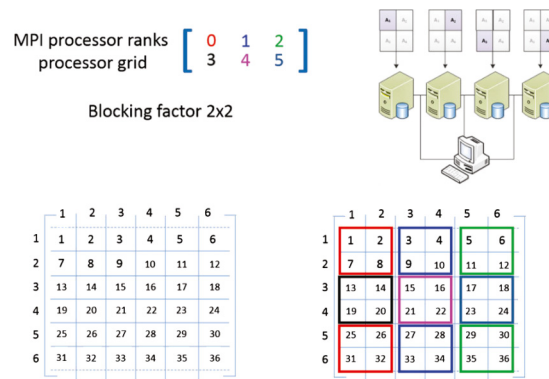
Figure 2. Block-cyclic distribution of a 6 × 6 matrix – global matrix – among 6 processing units. Data blocks are distributed following a 2 × 2 blocking factor, where a square data block of 2 × 2 is assigned to each processor with the order described in the upper left figure. The bottom right figure shows the final distribution of the matrix.

2.3. pbdR programming with big data

pbdR consists of a set of libraries for configuring and establishing an environment to work with parallel computing for big data analysis in R-project in a very similar manner. From a simple perspective, we can easily use R to analyze large data sets (Eddelbuettel, 2016). In terms of performance, pbdR has shown to scale up to 10,000 cores with very good results (Schmidt et al., 2017). There are several distinctive characteristics that make pbdR well-suited for developing parallel applications easily. It offers an almost mid-point between implicit and explicit parallel programming approaches. Users may easily decide whether to set how data are distributed among processors or let the library do it (default values). Thus, users may control some factors such as to examine the application performance while using an environment similar to plain R while operating on distributed data. Moreover, pbdR allows a fine-grained control of the code and thus offers a lot of flexibility when programming an application (Ostrouchov et al., 2013).

pbdR uses block cyclic distribution to distribute data across processors. This is done internally by pbdR but users are able to still set up the blocking factor in the program (Schmidt et al., 2012a, 2014). Data blocks are assigned to a set of processors cyclically. Figure 2 shows an example to illustrate how we can allot a 6 × 6 matrix – global matrix – in a cluster of six processors. The block-cyclic distribution has several advantages. With the allocation of regular data blocks, computational methods can achieve a better performance by balancing the workload among computing units by parallelizing mathematical tasks and thus reducing communication costs (Blackford et al., 1997, Bachmann et al., 2013, Schmidt et al., 2012a, 2014).

The core of pbdR consist of several packages such as pbdMPI, pbdSLAP, pbdBASE, and pbdDMAT. pbdMPI is an interface to MPI, a communication protocol between

processors and the most extended solution so far for clusters and supercomputers (Schmidberger et al., 2009, Eugster et al., 2011, Eddelbuettel, 2016). Thus, `pbdMPI` provides – for instance – functions that allow the data distribution, the movement of these data among processors, and also running the code that should operate on the distributed data. Therefore, to properly establish the communicator – that is the object "to define which collection of processes may communicate with each other" – is of paramount importance. Since `pbdR` is focused on the SPMD programming paradigm (Chen et al., 2012a, Schmidt et al., 2012c, Ostrouchov et al., 2013), users need to initialize the communicator(s) at the beginning of a script with the instruction `init()`. This enables the initialization of the processors (or task IDs) "to specify the source and destination of messages". Naturally, at the end of the script, we have to shut down the communicator(s) with the instruction `finalize()`. The main functions implemented in `pbdMPI` are `reduce` for collecting a set of objects distributed in different processors applying a reduction operation, for example, the sum of the objects; `gather` for collecting a set of objects distributed in different processors, the result is a list of the objects; `comm.set.seed` for setting seeds for random number generation to all processors via `rlecuyer` (Sevcikova and Rossini, 2012); and `*ply` functions such as `pbdApply`, `pbdLapply`, and `pbdSapply`, the counterparts of `apply` and `lapply` in the parallel case. At an early stage, Raim (2013) and Schmidt et al. (2014) showed very useful `pbdMPI` examples and functionalities such as point-to-point and collective communication.

While pbdMPI handles communications among processors, `pbdDMAT` is a library for managing distributed matrix classes, linear algebra, and statistics. `pbdDMAT` presents the class `ddmatrix` that allows the construction of distributed data across multiple processors using a block-cyclic distribution scheme to partition data. One of the main characteristics of the matrix decomposition is that it is not overlapping, meaning that each portion of a matrix is only distributed to one and only one processor. It is worth noting that data partition is performed by row and the blocks of data are handled by column in each processor (Schmidt et al., 2012a, 2014). `pbdDMAT` also provides functions and interfaces to operate on block-cyclic distributed data such as Choleski or QR decomposition, linear algebra functions, principal component analysis, and a fitter for linear models, among others.

Two of the slots of the class `ddmatrix` are `bldim`, and `ICTXT`. `bldim` enables to set up the blocking factor for data distribution, that is the row and column dimensions of the blocks to partition the data matrix. `ICTXT` allows setting up a rectangular grid of processors for distributing data. Three rectangular shapes or contexts are possible. Context 0 sets the processors in a grid as square as possible, context 1 establishes a one-row grid of processors, and context 2 places the processors in a one-column grid. The grid is initialized at the beginning of the R script with the instruction `init.grid()`. By default, the context is 0. Being able to decide the blocking factor

Table 1. Coefficients of the routine `lm.fit()`, distributed matrix solution to the linear least squares problem.

	Non-square blocking factor 10 × 4 `rank 0`		Square blocking factor 4 × 4 `rank 0`
[1,]	-0,1999	[1,]	-0,1999
[2,]	-0,1187	[2,]	-0,1187
[3,]	-0,6524	[3,]	-0,6524
[4,]	0,8255	[4,]	0,8255
[5,]	-0,3261	[5,]	0,2595
[6,]	0,0253	[6,]	0,8592
[7,]	0,0769	[7,]	-1,2341
[8,]	-0,0881	[8,]	-1,0092
[9,]	0,2595		`rank 1`
[10,]	0,8592	[1,]	-0,3261
	`rank 1`	[2,]	0,0253
[1,]	-1,2341	[3,]	0,0769
[2,]	-1,0092	[4,]	-0,0881
[3,]	0,1804	[5,]	0,1804
[4,]	-0,3063	[6,]	-0,3063
[5,]	1,1304	[7,]	1,1304
[6,]	0,0446	[8,]	0,0446

and context offers a lot of flexibility for experimentation and determining the proper set up to perform computations. This has an impact on the achieved scalability and the communications among processors.

Supplementary Material SM_HLY_e01451 shows an example on the use of `pbdLapply` with the Master/Worker and SPMD approaches implemented in `pbdMPI` and a simple example on the use of `pbdDMAT`.

pbdR uses `ScaLAPACK` as the library to perform distributed dense linear algebra operations without adding much computational or memory overhead. `ScaLAPACK` is a widely known library which has been deeply assessed by the scientific community and improved by developers (Blackford et al., 1997). It is well-known that `ScaLAPACK` prefers to work with square blocking factors, that is, equal row and column dimensions to partition a data matrix. Among others – and depending on the designed set-up – our results show that non-square blocking factor can be an alternative for achieving better performance which is particularly conditioned by communication costs among processors. The reference manual of pbdDMAT and the vignettes (Schmidt et al., 2012a) points out that "ScaLAPACK and PBLAS routines *usually* require square blocking" (Schmidt et al., 2012a), but while some routines do not support non-square blocking, others like `lm.fit()` support non-square blocking factors (one must be careful in its use in any case). Table 1 shows the coefficients given by `lm.fit()` for matrices distributed with two blocking factors, a non-square blocking factor of 10 × 4, and a square blocking factor of 4 × 4. The solution corresponds to the solution of the linear least square problem. As can be observed, the same coefficients are obtained in both cases. We execute the script provided here in two ranks with `mpiexec -np 2 Rscript filename.R`.

Despite enabling to set up the size and layout of the grid of processors, and through the blocking factor a certain degree of control over the parallel process, many of the technical details for executing the parallel jobs in `pbdR` are hidden for the end-user (Schmidt et al., 2012a). The packages `pbdSLAP` and `pbdBASE` execute this hidden job. `pbdSLAP` allows to use `ScaLAPACK`'s functions – which includes the scalable linear algebra routines that make possible to perform calculations with distributed data – from within `R` via `pbdMPI` (Chen et al., 2012b, Ostrouchov et al., 2013). `pbdSLAP` is based on `ScaLAPACK` version 2.0.2 which was last updated may 1, 2012. On the other hand, `pbdBASE` presents the necessary wrappers or interfaces and routines for communication with low-level routines written in Fortran and available in `ScaLAPACK` (Schmidt et al., 2012b). All `pbdR` libraries "install and run on a single machine as well as on shared memory and distributed clusters" (Schmidt et al., 2012c, p. 811).

In our experience, one of the main costs of using parallel programming tools such as `pbdR` for statistical analysis are first the initial configuration of hardware (if necessary), second the installation of packages, and third the experimentation and tuning of parameters such as the blocking factor and the context to perform statistical analysis on distributed data.

```
rm(list=ls(all=TRUE))
suppressPackageStartupMessages(library(pbdDMAT, quiet=TRUE))
init.grid()
.pbd_env$BLDIM <- c(4,4) #.pbd_env$BLDIM <- c(10,4)
.pbd_env$ICTXT <- 0
n <- 25
p <- 16
if (comm.rank() == 0){
   comm.set.seed(12345,diff=TRUE)
   X <- matrix(rnorm(n*p, 0, 1), n, p)
   y <- matrix(rnorm(n*1, 0, 1), n, 1)
} else {
   X = NULL
   y = NULL
}
dX <- as.ddmatrix(X)
dy <- as.ddmatrix(y)
myfun <- function(x,y) lm.fit(x,y)$coefficients
result <- myfun(dX,dy)
comm.print("Printing result rank 0...", rank.print=0)
comm.print(submatrix(result), rank.print=0)
comm.print("Printing result rank 1...", rank.print=1)
comm.print(submatrix(result), rank.print=1)
finalize()
```

Algorithm 2 Pseudocode for the parallel implementation of Lohmöller PLS algorithm, where *proc* is the total number of processors, *nr* is the number of rows and *nc* is the number of columns of the submatrices.

Input: X; {large-scale data matrix}

Input: W; {initial weights vector}

Output: dW, dY; {list of distributed weights, list of distributed y_j}

 1: (processor=0)   *Set seed*;

 2:                    *Read* $X = \{X_1, X_2, .., X_J\} \subset \Re^n$;

 3: **for** rank = 0; rank < proc; rank = rank + 1 **do**

 4:            Send(rank, $X[nr * rank + 1 : nr + nr * rank, nc * rank + 1 : nc + nc * rank]$)

 5: **end for**

 6: (All processors) $PLS(X, W, condition)$;

 7: **for** rank = 0; rank <= proc; rank = rank + 1 **do**

 8:            Receive(0, rank, dW_{rank}, dY_{rank})

 9: **end for**

10: *Return* dW, dY;

---

3. Methods

In this section, some considerations about the implementation of the algorithm with pbdR framework are pointed out. The parallelization process considers several steps. We used a virtual machine cluster for testing and tuning the conditions under which the experiments were finally executed, and marenostrum 3 supercomputer for running the experiments.

3.1. Parallel implementation of PLS with pbdR

Using the PLS algorithm presented in Algorithm 1, the new approach operates on $nr * nc$ submatrices of the input matrix X, where nr is the number of rows and nc is the number of columns. The goal is to split a large or big input matrix into different matrices and distribute them accross a number of processors where the PLS method will be applied to different portions of data. The PLS method operated in each submatrix will be collected at the end to compute the final results. The parallel pseudocode to distribute the data is presented in Algorithm 2.

The first processor ($rank = 0$) prepares the execution by setting the seed that will be used in the algorithm and reads the input dataset (X matrix). After that, X is divided equally and sent to each processor, including the first one. The submatrix will have dimensions $[nr * rank + 1 : nr + nr * rank, nc * rank + 1 : nc + nc * rank]$ where $rank = 0..proc$ for $proc >= 1$. For example, if nr was set to 4 and nc to 5, the first processor will store a submatrix of X, e.g. $X[1 : 4, 1 : 5]$, the second processor will store another submatrix of X, e.g. $X[1 : 4, 6 : 10]$, and so on. The value of nr and nc is determined by the user and it can take into account the memory

Algorithm 3 Pseudocode for the parallel implementation of PLS algorithm introduced in 2 with `pbdR` framework, where N is the total number of processors/ranks

---

Input: X; {large-scale data matrix}

Input: W; {initial weights vector}

Output: $dWls, dYls$; {list of distributed weights, list of distributed y_j}

```
 1: if comm.rank() = 0 then
 2:          Set seed
 3:          X ← matrix();
 4: end if
 5: Set environment variables .pbd_env$BLDIM, .pbd_env$ICTXT
 6: Load dX ← as.ddmatrix(X) in N nodes
 7: Load dW ← as.ddmatrix(W) in N nodes
 8: dXls ← dX;
 9: dWls ← dW;
10: dYls ← mapply(dXls,dWls)
11: while condition > 10⁻¹⁶ do
12:          dZls ← f(dYls)
13:          dWls ← mapply(dXls, dZls)
14:          Update condition
15: end while
16: Resume dWls, dYls from N nodes;
17: Return dWls, dYls;
```

---

capacity of each processor and the number of processors involved. Once the different processors receive the information, they proceed to apply PLS method defined in Algorithm 1 implemented to operate over the assigned submatrices. Finally, results of the algorithm are sent to processor $rank = 0$ and gathered into a unique matrix – in this case dW, dY – before giving the final result. In our proposed algorithm data partition is performed based on data order, which is the way to ensure that the method is giving the appropriate results since the computation of weights presented in Algorithm 1 has a locality constraint.

To parallelize the PLS algorithm, we transformed our optimized serial R version to express it in terms of operations on submatrices, implementing it by using the utilities from `pbdR` and `R-project`. We used the class `ddmatrix` of the `pbdDMAT` package. Thus, we worked with distributed matrices and the code was applied to different portions of data, thanks to basic matrix operations already implemented in `ScaLAPACK` and used by `pbdSLAP` and `pbdBASE` to perform parallel computations (Chen et al., 2012a,b, Schmidt et al., 2012b). The pseudocode of the implementation is presented in Algorithm 3.

There are two possible ways to create a data set for experimentation. First, we can generate a distributed matrix with the following instructions,

```
if (comm.rank() == 0){
   comm.set.seed(12345,diff=TRUE)
   X <- matrix(rnorm(n*p, 0, 1), n, p)
} else X = NULL


dX <- as.ddmatrix(X)
```

where n is the number of observations and p the number of variables. In this case, we are generating random normal data with zero mean and unit variance with the seed 12345 in the `rank = 0` (processor 0), and then distributing it to other processors with the instruction `dX <- as.ddmatrix(X)`. Thus, independently of the setup, the same data set is always distributed. The second way to generate data is as follows:

```
comm.set.seed(12345,diff=TRUE)
dX <- as.ddmatrix(matrix(rnorm(n*p,0,1),n,p))
```

In this other case, data it is automatically generated in each rank (or processor) resulting in different data sets for every setup. To be able to verify the proper implementation of the algorithms we chose the first option. Moreover, we decided to store small vectors and parameters in all ranks to get a more homogeneous parallelization, such as the initial weights vector, the number of observations, the number of variables, the binary matrix with the relationships between variables, the mode for each block of variables, and the number of variables per block of variables.

To manage distributed data within the parallel PLS function, we organize the data into lists, thus, for instance, the first step of the PLS algorithm – the initialization – may be implemented as follows,

```
dWnls <- mapply(f,dXls,dWls,SIMPLIFY=FALSE)
dYls <- mapply("%*%",dXls,dWnls,SIMPLIFY=FALSE)
```

The function f allows us to compute the values of the weights vectors such that the variance of y_j is equal to one (line 4 and 11 in Algorithm 1). This can be obtained by computing the Frobenius norm ("F") of the vector x%*%y. Thus, we call `mapply` to apply the function to the first, second, third, ..., element of each argument, in this case `dXls` and `dWls`, the distributed data set and weights vector. The same procedure is applied to calculate the values of the variables y_j, which are also organized into lists (line 5 and 12 in Algorithm 1). `mapply` is provided for the base distribution of `R-project` and it works fine with distributed data. A similar procedure was used to implement the other steps of the PLS Mode B algorithm (inner estimation and weight updating).

We would like to highlight some stages in the process of creating an implementation that operates in distributed environments. (1) **The serial implementation should be first optimized**, by testing and benchmarking key functions looking for performance improvements. (2) **Determining a degree of parallelism is crucial**. There is a need to decide how data will be distributed and thereafter the implementation should be adapted. (3) **Results obtained should be validated** against the serial version (for instance, the instructions for norm computation, the reciprocal of a number, the fit of linear models, etc. for square and non-square blocking factors and for different contexts). This was rigorously and systematically carried out for (a) each step of the first iteration of the algorithm – outer and inner estimation and weight updating – and (b) the final algorithm results. In this way, we were able to verify the results' correctness and to carefully understand how the operations are executed when using a SPMD approach. (4) Finally, **benchmarks should be performed** to test different alternatives of implementation of the computations and code-granularity.

4. Results and discussion

4.1. Computational experiments

We run a set of computational experiments to study the scalability and performance of the parallel implementation of the PLS Mode B iterative algorithm (centroid scheme). Parallel simulations results were compared with the sequential executions results for correctness whenever possible. We installed pbdR and performed all the experiments in marenostrum 3 supercomputer. Marenostrum3 is equipped with 3,056 nodes containing 2 sockets of Intel SandyBridge-EP E5-2670/1600, with 8 cores each, totaling 16 cores per node and 32 GB of main memory (2 GB per core). The interconnection network is based on Infiniband FDR10 technology. In all nodes, we used R version 3.3.0, OpenMPI 1.8.1, rlecuyer 0.3-4, pbdBASE 0.5-0, pbdMPI 0.3-3, pbdSLAP 0.2-1, and pbdDMAT 0.5-0. In total, we run around 750 experiments.

4.2. Computational performance of the sequential implementation

In order to have a baseline for comparison, the first set of computational experiments was designed to obtain executions times and the relationship between time and the number of observations with the serial implementation of the PLS iterative algorithm. The PLS model setup included a component-based model with three exogenous variables and one endogenous variable. Each variable was related to a block of variables with four indicators in a Mode B. Therefore, we set the complexity level of the multiblock model. Data were generated as random normal data with zero

Table 2. Elapsed times (s) of the sequential implementation of the algorithm.

Number of observations $(x10^6)$	Number of entries $(x10^6)$	Elapsed time (s)
1.0	16	200.592
2.5	40	172.166
5.0	80	394.566
7.5	120	358.604
10.0	160	554.598

mean and unit variance and using the seed 12345. The condition for convergence was set in $1e - 16$. The experiments were executed for tall skinny matrices with five different number of observations: 1, 2.5, 5, 7.5, and 10 million.

As a result, we processed matrices with 16, 40, 80, 120, and 160 million entries, respectively. The experiments were executed on a personal computer under the usual conditions in which researchers and practitioners apply the algorithm to estimate a model: a multicore architecture with 2 to 8 processors with a shared memory. For every case, we measured the elapsed time of the iterative algorithm implementation. We worked with the mean of 5 replications.

Table 2 shows the elapsed time in seconds of the serial implementation of the iterative algorithm to tall and skinny matrices. The algorithm is executed in 200.6 seconds for 1 million observations. Beyond that, we can not observe constant increments of the times. However, we note that the execution time increases close to linearity with the number of observations. For each simulated condition, the same PLS vector of weights was obtained in each execution. Even though there are several factors affecting times, it is worth the attention the fact different seeds give a different set of pseudo-random numbers, and therefore, this will involve different elapsed times. For instance, for a seed 123 and 1 million observations, the execution time of the serial implementation is 34.8 seconds, much less than for the seed 12345. However, for a seed 123 and 7.5 million observations, the same implementation is executed in more than five times the time obtained when generating the data with the seed 12345 (1891.7 seconds). As expected, elapsed times of the PLS algorithm serial implementation are quite smaller for small numbers of observations: matrices with 100, 500, and 1,000 rows were processed in 0.25, 0.05 and 0.08 seconds, respectively (seed 12345).

On the other hand, using the instruction `plspm` of the `plspm` R-package (Sanchez et al., 2009) and the instruction `sempls` of the `semPLS` R-package (Monecke and Leisch, 2012) on the same model resulted in an elapsed time of 0.11 and 0.68 seconds respectively for 1,000 observations (seed 12345). Note that we are comparing the implementation of the iterative algorithm with the instructions just as a reference.

4.3. Computational performance of the parallel implementation

We examined the performance of the parallel PLS algorithm compared to the serial PLS iterative algorithm to find the most suitable computational setup for an effective and efficient algorithm's execution. We varied five factors: number of observations, blocking factor, context, number of cores, and number of nodes. To make results comparable, we generate the same data by setting the seed to 12345 with the instruction `comm.set.seed()`. Data were generated in `comm.rank() == 0` and then distributed to the other cores/nodes as was previously described. The condition for algorithm convergence was set to $1e - 16$. Each experiment was executed in different cores/nodes so that executions were independent and they did not compete with others in the use of resources. In addition, and as for the serial case, we inspected the values obtained for weight vectors. For each simulated condition, the same vector of weights was obtained in each execution.

We performed the first experiment to determine the proper block sizes to distribute data across processors. Moreover, we examined how blocking factors affect the execution time when applying the algorithm to distributed data. It is known that block sizes may be inefficiently large or small (Schmidt et al., 2012b). With this aim, we fixed the size of the data set. We worked with a matrix of 16 variables and 1 million observations. The data matrix was partitioned and distributed using eight different blocking factors: 2×2, 4×4, 8×8, 16×16, 50×4, 100×4, 1000×4 and 10000×4. The first four square blocking factors were also used in Bachmann et al. (2013) to study the performance of parallel implementations of covariance matrices and principal component analysis. In their report, they concluded that "dividing the number of rows and columns evenly are likely more efficient" (Bachmann et al., 2013, p.3). Thus, they chose a matrix where the number of observations is ten times larger than the number of variables to ensure an even distribution of the data and a suitable load balancing. Here, we studied a more general case for experimenting with tall and skinny matrices. Our aim was to see whether to establish the blocking factors according to the column dimension of the data matrices could have an advantage in terms of execution times. Moreover, – and even though square blocking factors are recommended to partition and distribute data and "`ScaLAPACK` and `PBLAS` routines usually require square blocking" (Schmidt et al., 2012a, p.9) – we perform our experiments with the second set of non-square blocking factors – 50×4, 100×4, 1000×4, 10000×4 – where the number of columns of the partition blocks are equal to the number of indicators per variable and the number of rows is up to 2500 the number of columns.

We examined the performance of the algorithm implementation in two different `pbdR` contexts by varying the grid layout by fixing the value of the slot $.ICTXT$ to test the effect of different configurations in the execution time. A context 0 in

Table 3. Elapsed times (et) of the parallel implementation for $.ICTXT = 0$ (et_0, processors in a grid as square as possible) and $.ICTXT = 2$ (et_1, processors placed in one-column grid).

Number of cores	.BLDIM	Elapsed time (s) .ICTXT=0	Elapsed time (s) .ICTXT=2	et_0 - et_1 (s)
2	2x2	1917.927	1914.137	3.790
2	4x4	963.383	962.157	1.226
2	8x8	487.508	486.337	1.171
2	16x16	250.205	248.254	1.951
2	50x4	88.268	86.943	1.325
2	100x4	50.169	48.902	1.267
2	1000x4	16.206	14.727	1.479
2	10000x4	12.668	11.286	1.382
4	2x2	12.339	499.424	-487.085
4	4x4	12.581	252.354	-239.773
4	8x8	12.336	129.219	-116.883
4	16x16	12.124	67.534	-55.410
4	50x4	11.999	25.471	-13.472
4	100x4	11.944	15.589	-3.645
4	1000x4	11.824	6.669	5.155
4	10000x4	11.763	5.805	5.958
8	2x2	8.407	131.334	-122.927
8	4x4	8.474	67.389	-58.915
8	8x8	8.272	35.650	-27.378
8	16x16	8.058	19.844	-11.786
8	50x4	8.022	8.925	-0.903
8	100x4	8.041	6.389	1.652
8	1000x4	7.920	4.109	3.811
8	10000x4	8.359	4.210	4.149
16	2x2	10.208	37.369	-27.161
16	4x4	9.235	20.271	-11.036
16	8x8	8.634	11.759	-3.125
16	16x16	8.213	7.735	0.478
16	50x4	8.472	4.600	3.872
16	100x4	8.387	4.012	4.375
16	1000x4	8.258	3.398	4.860
16	10000x4	8.332	3.630	4.702

which a grid layout is automatically set as square as possible by pbdR, and a context 2 in which the processors are positioned in a one-column grid. For every case, we measured the elapsed time of the application of the iterative algorithm to distributed data in 2, 4, 8, and 16 cores. As in the previous case, the experiments were performed in a multicore environment. Time measurement did not include the time for data generation, the initial data movement for data distribution or the time for collecting the output results. However, the obtained computation time included some data movement within the iterative algorithm.

Table 3 shows the elapsed time in seconds of the parallel implementation operating on distributed data in 2, 4, 8 and 16 cores in a single node, so communication times are minimized. For 2 cores and $.ICTXT = 0$, and taking into account all the blocking factors, the times range from 12.6 s to 1917.9 s (0.2 min to 31.9 min). When $.ICTXT = 2$, the times range from 11.2 s to 1914.1 s (0.1 min to

31.9 min). These results are for a matrix of 16 million entries. These times seem reasonable when they are compared with those obtained by Bachmann et al. (2013), although the highest time obtained for $.ICTXT = 0$ is high. Bachmann et al. (2013) reported the elapsed times of the calculation of the covariance matrix and principal component analysis (PCA) for a matrix of 262 million entries approximately. For the experiments executed in 2 cores, the overall runtime ranged from 3274.6 s to 4131 s (54.5 min to 68.8 min).

For 2 cores, the elapsed times of the parallel implementation are higher than for the serial implementation when distributing data with square blocking factors (see Table 2). However, all the elapsed times obtained when distributing data with non-square blocking factors are smaller than for the serial implementation. Elapsed times decrease in all cases when the number of cores jumps from 2 to 4 as clearly seen in the table. Beyond that – 4, 8, and 16 cores – there is a clear decrease tendency in the times in all cases when increasing the dimension of blocking factors. Moreover, elapsed times remain much smaller than in the serial version in all executions.

On the other hand, when increasing the dimension of blocking factors and the context is 2 ($.ICTXT = 2$ organizing processors in an one-column grid), we can clearly observe a linear decrease in execution times for both square and non-square blocking factors. Nevertheless, the slopes of the curves are higher in the case of square blocking factors. For instance, for 4 cores and square blocking factors, elapsed times range from 499 s to 67.5 s ($.ICTXT = 2$). When increasing the dimension of blocking factors and the context is 0 ($.ICTXT = 0$), we can observe a decrease in the times for both square and non-square blocking factors, but the ranges of variation of times are much smaller than for context 2 ($.ICTXT = 2$). That is the case, for example, for 4 cores and square blocking factors, elapsed times range from 12.5 s to 12.1 s ($.ICTXT = 0$). Since serial experiments were executed on a personal computer, these results are good especially if we consider the architecture of a supercomputer where we might find added latency and communication costs. Therefore, the proposed parallel implementation is justified for a number of cases in a multicore environment.

Furthermore, elapsed times were lower in context 2 than in context 0, 56.25% of executions. For square blocking factors, elapsed times were lower in context 0 than in context 2, 68.7% of executions. For non-square blocking factors, elapsed times in context 2 were below those obtained in context 0, 87.5% of executions. Thus, we conclude square blocking factors work better when processors are arranged in a grid as square as possible ($.ICTXT = 0$), and non-square blocking factors work better when processors are arranged in an one-column grid ($.ICTXT = 2$). These results show that the decision of choosing the blocking factor and context can highly affect the efficiency of the solution and we confirm the results reported by
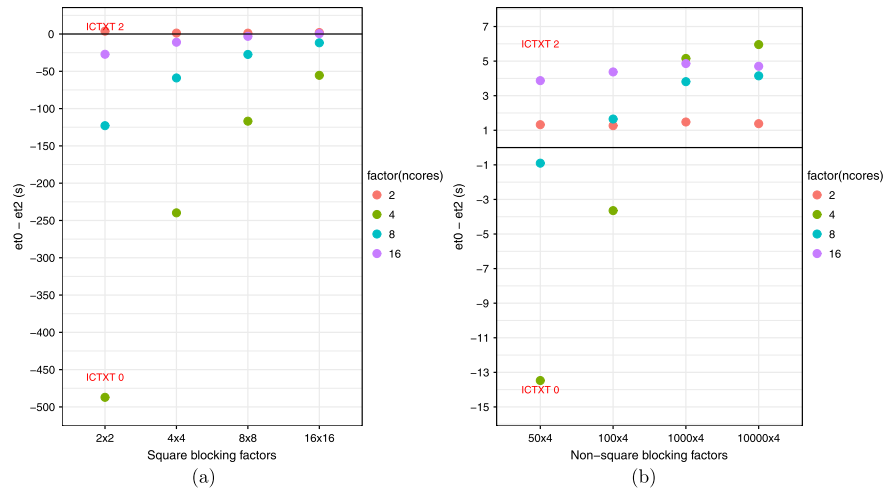
Figure 3. Differences of the elapsed times (s) between context 0 (et_0, $.ICTXT = 0$, processors in a grid as square as possible) and context 2 (et_2, $.ICTXT = 2$, processors placed in one-column grid).

Schmidt et al. (2012a, p. 7) "there is a strong connection between the process grid and the block-cyclic distribution".

To have a much clearer appreciation of the performance of the parallel implementation when partitioning the data with different blocking factors, Figure 3 shows the differences of the elapsed times in seconds between context 0 and context 2, $et_0 - et_2$; Table 3 also shows this difference. Figure 3(a) presents the results for square blocking factors whereas Figure 3(b) displays the outcome for non-square blocking factors for different number of cores. Negative values of $et_0 - et_2$ indicate the conditions under which the implementation performed better when running the experiments with a context 0 (under the x-axis). Positive values of $et_0 - et_2$ indicate the conditions under which the implementation has better performance when running the experiments with a context 2 (over the x-axis). In general terms for our specific setups, we can clearly see that the parallel implementation of the PLS Mode B algorithm performs better when partitioning the data with non-square blocking factors. For these cases, the differences are smaller than those presented for square-blocking factors, and for a greater number of experiments. For non-square blocking factors, the elapsed times are more similar for both contexts, and although in principle elapsed times are smaller for $.ICTXT = 0$, smaller times are reached for $.ICTXT = 2$. For square blocking factors, the differences are higher and we observe a greater number of experiments achieving a better performance when the processors' grid is arranged as square as possible (context 0).

Our results contrast with the recommendation given by Schmidt et al. (2012c,a) to partition data with square blocking factors. The reason for that is likely due to the fact that the column dimension of the blocking factors was chosen equal to the number of variables related to each variable Y_j, thus facilitating the computation

of the distributed matrix algebra operations considered in the PLS algorithm. This configuration should also facilitate the data operation in each processor, giving that data partition is performed by row and the blocks of data are handled by column in each processor (Schmidt et al., 2012a, 2014).

To summarize, shorter elapsed times are obtained for the following configuration: square-blocking factor 16 × 16 using a grid of processors as square as possible (context 0) and non-square blocking factors 1000 × 4 and 10000 × 4 using an one-column grid of processors. The non-square blocking factors considered here are "too big (relative to the process grid), then the data distribution will be very uneven" (Schmidt et al., 2012a, p. 15), this should reduce communication times among processors and also "the amount of parallelism possible". However, it is good to reduce communication times and data sets considered here are large enough to take advantage of the parallelism.

The second set of experiments was executed in order to measure the execution times when increasing the number of observations. Five number of observations were considered: 1, 2.5, 5, 7.5, and 10 million. First, we studied a general case, experimenting with tall and skinny matrices. Second, the number of observations was 625.000 times larger than the number of variables in the most extreme case. Two blocking factors were selected based on first set of experiments output, 16×16 ($.ICTXT = 0$) and 1000×4 ($.ICTXT = 2$). We run the experiments in 4, 8 and 16 cores in a single node. Performance results presented in this section are based on five replications of the experiments.

Additionally, to examine the performance of the implementation taking into account communication times between nodes, the third set of experiments was designed. We executed the same experiments as described before distributing data in 8 cores in 2, 4, 8, 16 and 32 nodes this time. Therefore, data were distributed among 16, 32, 64, 128, and 256 cores in total. This experiment allows observing if we have improvements in the times by increasing the available memory in each node for each core.

To measure the speedup, we computed the gain obtained in the elapsed time comparing the parallel execution of the PLS Mode B algorithm to the serial one. The speedup of a parallel implementation is defined as $S = T_1/T_p$ where T_1 is the time required for an algorithm running on a computer with one processor and T_P is the time on a computer with P independent processors.

Table 4 and Figures 4(a) and 4(b) show the elapsed times in seconds of the parallel implementation when increasing the number of observations (mean of five replications). Table 4 also shows the elapsed times for each repetition as well as the mean, standard deviations and coefficients of variation. As expected, time increases when the number of observations increases. Besides, all the times are lower for a blocking factor of 1000×4 and $.ICTXT = 2$ than for 16×16 and $.ICTXT =$

Table 4. Elapsed times (s) and speedups of the parallel implementation when increasing the number of observations and the number of cores.

Number of cores	Number of obs. ($x10^6$)	Elapsed time (s) `.BLDIM = c(16,16)`, $.ICTXT = 0$						SD	CV	Speedup
		Rep 1	Rep 2	Rep 3	Rep 4	Rep 5	Mean of 5 reps.			
4	1	12.210	12.271	12.175	12.239	12.249	12.229	0.033	0.003	16.403
4	2.5	462.211	463.222	460.479	461.352	467.360	462.925	2.397	0.005	0.372
4	5	241.662	244.260	243.966	243.336	244.542	243.553	1.027	0.004	1.620
4	7.5	200.057	199.446	200.243	199.662	198.156	199.513	0.734	0.004	1.797
4	10	788.386	794.477	788.940	790.752	792.527	791.016	2.261	0.003	0.701
8	1	8.054	8.060	8.043	8.029	8.051	8.047	0.011	0.001	24.926
8	2.5	279.611	278.184	277.371	279.332	278.824	278.664	0.809	0.003	0.618
8	5	155.320	155.053	154.596	153.694	153.836	154.500	0.645	0.004	2.554
8	7.5	122.799	123.911	123.057	122.796	122.438	123.000	0.496	0.004	2.915
8	10	396.255	399.516	397.846	399.304	398.888	398.362	1.200	0.003	1.392
16	1	8.203	8.184	8.163	8.347	8.237	8.227	0.065	0.008	24.383
16	2.5	277.417	277.024	277.667	277.418	276.749	277.255	0.326	0.001	0.621
16	5	153.750	154.363	154.701	154.001	153.809	154.125	0.359	0.002	2.560
16	7.5	122.429	122.965	123.094	122.481	123.004	122.795	0.281	0.002	2.920
16	10	395.077	396.479	397.655	393.575	393.765	395.310	1.570	0.004	1.403
		Elapsed time (s) `.BLDIM = c(1000,4)`, $.ICTXT = 2$								
4	1	6.680	6.686	6.696	6.698	6.668	6.686	0.011	0.002	30.004
4	2.5	269.963	270.438	268.514	270.244	270.548	269.941	0.741	0.003	0.638
4	5	194.236	193.869	194.489	194.082	194.114	194.158	0.203	0.001	2.032
4	7.5	189.791	190.128	190.053	190.252	190.096	190.064	0.152	0.001	1.887
4	10	725.405	726.650	726.077	725.153	724.833	725.624	0.656	0.001	0.764
8	1	4.102	4.135	4.121	4.109	4.122	4.118	0.011	0.003	48.713
8	2.5	140.400	140.306	140.517	140.430	140.285	140.388	0.085	0.001	1.226
8	5	85.963	85.648	85.639	85.754	85.657	85.732	0.123	0.001	4.602
8	7.5	76.075	76.147	76.097	75.990	76.022	76.066	0.055	0.001	4.714
8	10	286.642	286.047	285.567	286.071	286.425	286.150	0.367	0.001	1.938
16	1	3.403	3.394	3.391	3.397	3.396	3.396	0.004	0.001	59.064
16	2.5	101.459	101.608	101.437	101.823	101.550	101.575	0.138	0.001	1.695
16	5	56.832	56.800	57.014	56.802	56.821	56.854	0.081	0.001	6.940
16	7.5	46.548	46.617	46.525	46.469	46.484	46.529	0.052	0.001	7.707
16	10	157.549	157.238	157.296	157.325	157.271	157.336	0.110	0.001	3.525

0. As can be clearly seen, for matrices with a higher number of observations – 7.5 and 10 million observations – the computations require more time when they are executed with fewer resources as in the case of 4 cores.

Times decrease considerably when increasing the number of cores from 4 to 8, but when increasing to 16 cores, communication costs among cores outweigh the savings in time by distributing data to a larger number of processors. This is more evident when distributing data with a blocking factor of 16×16 and context 0. For data distributed across 8 and 16 cores, elapsed time decreases more for non-square blocking factors and context 2 than for square blocking factors and context 0, and smaller times are reached for the first case ($.ICTXT = 2$). Figure 4(b) clearly displays that times are closer to linearity when data are partitioned with a non-square blocking factor. This is the case in all studied cases (number of observations), and as shown, results are precise with coefficients of variation of 0.8% at most.
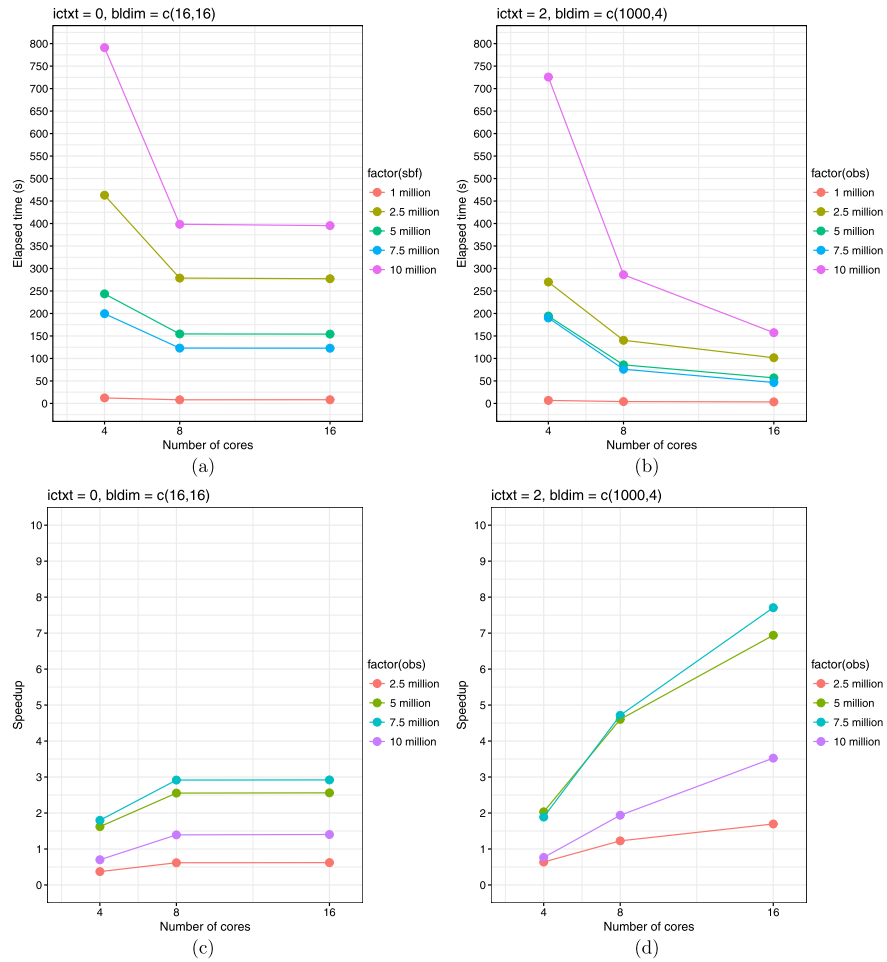
Figure 4. Elapsed times (s) and speedups of the parallel implementation when increasing the number of observations and the number of cores.

Table 4 and Figures 4(c) and 4(d) make visible the resulting speedups. Elapsed times of the parallel implementation were contrasted with those of the serial implementation (Table 2). Results present that the elapsed times are lower than for the serial implementation. The only exceptions are data with 2.5 and 10 million observations distributed in 4 cores ($.ICTXT = 0$ and $.ICTXT = 2$), and 2.5 million observations distributed in 8 and 16 cores ($.ICTXT = 0$). For one million observations, speedups reach values of up to 59. For 2.5, 5, 7.5 and 10 million observations, speedups reach values of up to 1.6, 6.9, 7.7, and 3.5, respectively. The speedups increase for 2.5, 5 and 7.5 million observations but the speedups decrease for 10 million observations. These performance increments are quite good, especially if we contrast them with Schmidt et al. (2012c) who reported speedups of up to 3.58 when testing PCA for a matrix of 100 million entries distributed in 512 cores.

Table 5 and Figure 5 show the elapsed times and the speedups of the parallel implementation when distributing data in 16, 32, 64, 128, and 256 cores in 2, 4,

Table 5. Elapsed times (s) and speedups of the parallel implementation when increasing the number of observations and the number of nodes.

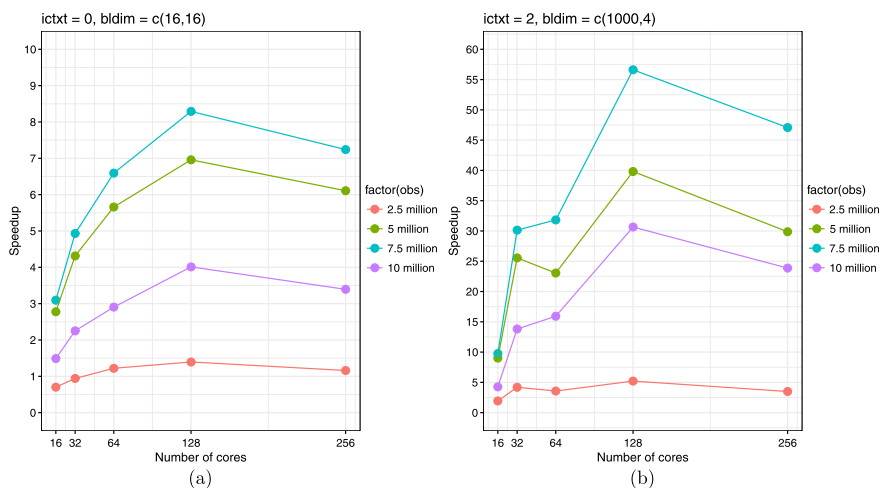| Number of nodes	Number of cores	Number of observations ($\times 10^6$)	Number of entries ($\times 10^6$)	Elapsed time (s) .BLDIM c(16,16)	Elapsed time (s) .BLDIM c(1000,4)	Speedup .BLDIM c(16,16)	Speedup .BLDIM c(1000,4)
2	16	1	16	7.858	3.747	25.527	53.534
2	16	2.5	40	245.505	89.251	0.701	1.929
2	16	5	80	142.072	43.823	2.777	9.004
2	16	7.5	120	115.875	36.787	3.095	9.748
2	16	10	160	372.513	130.205	1.489	4.259
4	32	1	16	6.697	1.946	29.953	103.079
4	32	2.5	40	182.946	41.060	0.941	4.193
4	32	5	80	91.466	15.438	4.314	25.558
4	32	7.5	120	72.712	11.898	4.932	30.140
4	32	10	160	246.372	40.129	2.251	13.820
8	64	1	16	5.089	2.481	39.417	80.851
8	64	2.5	40	140.986	48.217	1.221	3.571
8	64	5	80	69.722	17.118	5.659	23.050
8	64	7.5	120	54.393	11.268	6.593	31.825
8	64	10	160	190.939	34.841	2.905	15.918
16	128	1	16	5.612	1.650	35.743	121.571
16	128	2.5	40	123.460	33.032	1.395	5.212
16	128	5	80	56.705	9.909	6.958	39.819
16	128	7.5	120	43.253	6.334	8.291	56.616
16	128	10	160	138.250	18.094	4.012	30.651
32	256	1	16	6.875	2.854	29.177	70.285
32	256	2.5	40	148.323	49.281	1.161	3.494
32	256	5	80	64.600	13.207	6.108	29.876
32	256	7.5	120	49.528	7.617	7.240	47.079
32	256	10	160	163.386	23.228	3.394	23.876

Figure 5. Speedups of the parallel implementation when increasing the number of observations and the number of nodes.

8, 16, and 32 nodes, respectively, using all available resources at each node. In general terms, the results demonstrate that elapsed times increase when increasing the number of observations. However, as the number of cores increases, the times decrease, although at some point – beyond the 128 cores – communication times among nodes may become important and elapsed times slightly increased again. Furthermore, Figure 5 clearly shows that adding more resources to the experiments or executing the experiments using more resources made it possible to improve the speedups of the parallel implementation when comparing to the results showed in the Figure 4. For these experiments, speedups reach values of up to 121. In general terms, results are better for non-square blocking factors and context 2 ($.ICTXT = 2$).

5. Conclusions

Parallel computing technology is making more and more advances and providing faster solutions for running applications. Technological development in this area is extremely rapid and an increasing number of scientific communities are taking benefit of this technology. It looks like, as computers are a standard today, parallel computing will be tomorrow. From this standpoint, identifying the key aspects of the parallelization process and experimenting in an early stage of the research with different setups allow the user proper decision making. In this sense, the main contributions of this paper are (i) to show the scalability and performance of the Multiblock PLS Mode B algorithm, a tightly coupled algorithm for estimating the relationships among several blocks of variables; scaling an algorithm of this type is a difficult task precisely because of the coupled sequence of matrix operations; (ii) to confirm the applicability and utility of the R-project package pbdR for this implementation; and (iii) to prove that structural equation models can be estimated with big data sets using current state-of-the-art algorithms for multi-block data analysis.

There are several open questions and streams that arise from this research for future work. Areas such as algorithm features, hardware availability, software, linear algebra libraries for processing dense matrix operations, algorithm encoding, among others, could be further addressed. To investigate the use of other linear algebra libraries for distributed data that allow handling non-square blocking factors without restrictions is a pending task. We conclude that non-square blocking factors show the best elapsed times, even though the libraries PBLAS and ScaLAPACK – on which pbdR is based – prefer to work with square-blocking factors and some operations do not support non-square blocking factors. To compare pbdR with other R libraries, or other platforms for big data analysis such as Spark and MapReduce is also a compelling topic for further research as well as to apply our work to real data

sets. Moreover, solving the problem of how to extract and transform big data sets from data sources to a multicore environment should be approached. We plan to tackle also of software installation can future research, especially because the initial configuration is one of the main entry barriers for many users.

Declarations

Author contribution statement

A. Martinez-Ruiz, C. Montañola-Sales: Conceived and designed the experiments; Performed the experiments; Analyzed and interpreted the data; Contributed reagents, materials, analysis tools or data; Wrote the paper.

Funding statement

Competing interest statement

The authors declare no conflict of interest.

Additional information

Supplementary content related to this article has been published online at https://doi.org/10.1016/j.heliyon.2019.e01451.

Acknowledgements

References

Abdi, H., Esposito-Vinzi, V., Russolillo, G., Saporta, G., Trinchera, L., 2016. The Multiple Facets of Partial Least Squares and Related Methods. Springer Proceedings in Mathematics and Statistics. Springer International Publishing.

Bachmann, M., Dyas, A., Kilmer, S., Sass, J., 2013. Block Cyclic Distribution of Data in pbdR and Its Effects on Computational Efficiency. Technical Report HPCD-2013-11. University of Maryland, Baltimore County.

Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R., 1997. ScaLAPACK: a linear algebra library for message-passing computers. In: SIAM Conference on Parallel Processing, pp. 1–15.

Calaway, R., Weston, S., Tenenbaum, D., 2015. doParallel: Foreach parallel adaptor for the 'parallel' package. http://CRAN.R-project.org/package=doParallel.

Chen, W.-C., Ostrouchov, G., Schmidt, D., Patel, P., Yu, H., 2012a. A quick guide for the pbdMPI package. R Vignette version 0.2-3. http://cran.r-project.org/package=pbdMPI.

Chen, W.-C., Schmidt, D., Ostrouchov, G., Patel, P., 2012b. A quick guide for the pbdSLAP package. R Vignette. http://cran.r-project.org/package=pbdSLAP.

Chen, W.-C., Schmidt, D., Sehrawat, G., Patel, P., Ostrouchov, G., 2016. A quick guide for the pbdPROF package. R Vignette. http://cran.r-project.org/package=pbdPROF.

Deb, B., Srirama, S., 2013. Parallel k-means clustering for gene expression data on snow. Int. J. Comput. Appl. 71 (24).

Eddelbuettel, D., 2016. CRAN task view: high-performance and parallel computing with R. https://cran.r-project.org/web/views/HighPerformanceComputing.html. (Accessed 9August2016).

Esposito-Vinzi, V., Chin, W., Heneler, J., Wang, H., 2010. Handbook of Partial Least Squares: Concepts, Methods and Applications. Springer Handbooks of Computational Statistics. Springer-Verlag, Berlin, Heidelberg.

Eugster, M., Knaus, J., Porzelius, C., Schmidberger, M., Vicedo, E., 2011. Hands-on tutorial for parallel computing with R. Comput. Stat. 26, 219–239.

Fu, X., Huang, K., Papalexakis, E., Song, H., Talukdar, P., Sidiropoulos, N., Faloutsos, C., Mitchell, T., 2016. Efficient and distributed algorithms for large-scale generalized canonical correlation analysis. In: 2016 IEEE 16th International Conference on Data Mining. ICDM, pp. 1–6.

Golub, G., Van Loan, C., 1996. Matrix Computations. Johns Hopkins University Press, Baltimore, US.

Górecki, T., Smaga, Ł., 2018. fdANOVA: an R software package for analysis of variance for univariate and multivariate functional data. Comput. Stat., 1–27.

Hanafi, M., 2007. PLS path modelling: computation of latent variables with the estimation mode B. Comput. Stat. 22, 275–292.

Hofert, M., Mächler, M., 2016. Parallel and other simulations in R made easy: an end-to-end study. J. Stat. Softw. 69 (4).

Knaus, J., 2010. Developing parallel programs using `snowfall`. https://cran.r-project.org/web/packages/snowfall/vignettes/snowfall.pdf.

Lawrence, M., Morgan, M., 2014. Scalable genomics with `R` and `Bioconductor`. Stat. Sci. 29 (2), 214–226.

Lê Cao, K., Chabrier, P., 2008. `Ofw`: an `R` package to select continuous variables for multiclass classification with a stochastic wrapper method. J. Stat. Softw. 28 (9), 1–16.

Lohmöller, J., 1989. Latent Variable Path Modeling With Partial Least Squares. Physica-Verlag, Heidelberg.

Luo, W., Zhang, H., 2015. Visual analysis of large-scale lidar point clouds. In: Big Data (Big Data), 2015 IEEE International Conference on, pp. 2487–2492.

McLeod, A., Yu, H., Krougly, Z., 2007. Algorithms for linear time series analysis: with R package. J. Stat. Softw. 23 (5), 1–26.

Monecke, A., Leisch, F., 2012. sempls: structural equation modeling using partial least squares. J. Stat. Softw. 48 (3), 1–32.

Ostrouchov, G., Schmidt, D., Chen, W.-C., Patel, P., 2013. Combining `R` with scalable libraries to get the best of both for big data. In: Cho, S. (Ed.), Proceedings of IASC Satellite Conference for the 59th ISI WSC & the 8th Conference of IASC-ARS, pp. 85–90.

Pacheco, P., 2011. An Introduction to Parallel Programming. Elsevier, Massachusetts, US.

Raim, A., 2013. Introduction to Distributed Computing With `pbdR` at the UMBC High Performance Computing Facility. Tech. rep. University of Maryland, Baltimore County.

Riddick, G., Song, H., Ahn, S., Walling, J., Borges-Rivera, W., Fine, H., 2011. Predicting in vitro drug sensitivity using random forests. Bioinformatics 27 (2), 220–224.

Rossini, A.J., Tierney, L., Li, N., 2007. Simple parallel statistical computing in R. J. Comput. Graph. Stat. 16 (2), 399–420.

Sanchez, G., Trinchera, L., Russolillo, G., 2009. `plspm`: tools for partial least squares path modeling (PLS-PM). `R` package version 0.4-7. http://cran.r-project.org/package=plspm.

Schmidberger, M., Morgan, M., Eddelbuettel, D., Yu, H., Tierney, L., Mansmann, U., 2009. State-of-the-art in parallel computing with `R`. J. Stat. Softw. 47 (1), 1–51.

Schmidt, D., Chen, W.-C., Ostrouchov, G., Patel, P., 2012a. Guide to the `pbdDMAT` package. `R` Vignette version 2.0 http://cran.r-project.org/package=pbdDMAT.

Schmidt, D., Chen, W.-C., Ostrouchov, G., Patel, P., 2012b. A quick guide for the `pbdBASE` package. `R` Vignette version 2.0 http://cran.r-project.org/package=pbdBASE.

Schmidt, D., Ostrouchov, G., Chen, W.-C., Patel, P., 2012c. Tight coupling of `R` and distributed linear algebra for high-level programming with big data. In: Society, I.C. (Ed.), Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis.

Schmidt, D., Chen, W.-C., Ostrouchov, G., Patel, P., 2014. Speaking serial `R` with a parallel accent. `pbdR` package examples and demonstrations. `R` Vignette version 0.2-0. http://cran.r-project.org/package=pbdDEMO.

Schmidt, D., Chen, W.-C., Matheson, M., Ostrouchov, G., 2017. Programming with big data in `R`: scaling analytics from one to thousands of nodes. Big Data Res. 8, 1–11.

Sevcikova, H., Rossini, T., 2012. `rlecuyer`: `R` interface to RNG with multiple streams. http://cran.r-project.org/package=rlecuyer.

Tenenhaus, M., Esposito Vinzi, V., Chatelin, Y., Lauro, C., 2005. PLS path modeling. Comput. Stat. Data Anal. 48, 159–205.

Tierney, L., Rossini, A., Li, N., Sevcikova, H., 2011. `snow`: simple network of workstations. https://cran.r-project.org/web/packages/snow/.

Varsos, C., Patkos, T., Oulas, A., Pavloudi, C., Gougousis, A., Ijaz, U., Filiopoulou, I., Pattakos, N., Vanden-Berghe, E., Fernández-Guerra, A., Faulwetter, S., Chatzinikolaou, E., Pafilis, E., Bekiari, C., Doerr, M., Arvanitidis, C., 2016. Optimized R functions for analysis of ecological community data using the R virtual laboratory (RvLab). Biodivers. Data J. 4, e8357.

Wold, H., 1985. Partial least squares. In: Kotz, S., Johnson, N. (Eds.), Encyclopedia of Statistical Sciences, vol. 6. Wiley, New York, pp. 581–591.

Yan, J., Zhang, H., Du, L., Wernert, E., Saykin, A., Shen, L., 2014. Accelerating sparse canonical correlation analysis for large brain imaging genetics data. In:

H e l i y o n

XSEDE'14 Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment, pp. 1–7.

Yu, H., 2009. `Rmpi`: interface (wrapper) to `mpi` (message-passing interface) https://cran.r-project.org/web/packages/Rmpi/.