Knowl Inf Syst
DOI 10.1007/s10115-016-0981-5

REGULAR PAPER

# A distributed approach for accelerating sparse matrix arithmetic operations for high-dimensional feature selection

Antonela Tommasel[1] · Daniela Godoy[1] · Alejandro Zunino[1] · Cristian Mateos[1]

Received: 13 December 2014 / Revised: 30 June 2016 / Accepted: 4 August 2016

**Abstract** Matrix computations are both fundamental and ubiquitous in computational science, and as a result, they are frequently used in numerous disciplines of scientific computing and engineering. Due to the high computational complexity of matrix operations, which makes them critical to the performance of a large number of applications, their efficient execution in distributed environments becomes a crucial issue. This work proposes a novel approach for distributing sparse matrix arithmetic operations on computer clusters aiming at speeding-up the processing of high-dimensional matrices. The approach focuses on how to split such operations into independent parallel tasks by considering the intrinsic characteristics that distinguish each type of operation and the particular matrices involved. The approach was applied to the most commonly used arithmetic operations between matrices. The performance of the presented approach was evaluated considering a high-dimensional text feature selection approach and two real-world datasets. Experimental evaluation showed that the proposed approach helped to significantly reduce the computing times of big-scale matrix operations, when compared to serial and multi-thread implementations as well as several linear algebra software libraries.

**Keywords** Sparse matrix · Matrix arithmetic operation · Feature selection · Distributed computing

## 1 Introduction

Matrix computations are both fundamental and ubiquitous in computational science. Arithmetic operations between matrices are frequently used in numerous disciplines in the context of scientific computing and engineering. For example, they represent the dominant cost in many iterative methods for solving linear systems, eigenvalue computation, and optimisation

✉ Antonela Tommasel
antonela.tommasel@isistan.unicen.edu.ar

[1] ISISTAN, UNICEN-CONICET, Paraje Arroyo Seco, Campus Universitario, Tandil, Buenos Aires, Argentina

Published online: 26 August 2016

problems. This type of operations usually becomes a performance bottleneck due to their high computational complexity and thus is critical to the performance of a large number of applications. In this context, arithmetic operations need to be efficiently performed. Particularly, sparse matrix operations have proved to be of particular importance in computational science. Since only non-zero elements are stored, they require fewer computational resources than their dense equivalent, which helps to cope with the resources required by their high dimensionality. Furthermore, the performance of several matrix operations can be significantly improved when sparse matrices are involved [24,53]. However, sparse matrices pose additional challenges as the performance of their operations tends to be lower than the dense matrix equivalent due to the overhead of accessing the data and the irregularity of memory accesses [20].

One domain that requires the efficient computation of sparse matrix operations is large-scale text analysis in social environments, in which high-dimensional data is frequent in the form of document-term matrices. Social media data grow at an unprecedented rate due to the massive use of social networking sites. For example, in 2014, the posting activity in *Facebook* increased a 94 %. Meanwhile, in *Twitter* the increment was of a 47 %. Text learning is often susceptible to the problem known as the curse of dimensionality, which refers to the increasing computational complexity of problems as the data that need to be accessed grow exponentially regarding the underlying space dimension. Furthermore, as the data dimensionality increases, the volume of the feature space increases rapidly, so that the available data become sparse. The problem is aggravated if the linked nature of social media data is considered, causing new dimensions (such as users friendship links and co-post relationships) to be added to the feature space [42].

Among the text analysis problems requiring the manipulation of large matrices are feature selection [2], feature enrichment [39,45], sentiment analysis [19], and document retrieval [54]. Feature selection aims at enhancing predictive models and speeding-up data mining algorithms by choosing a small subset of relevant features according to a certain evaluation criterion. In other engineering disciplines, matrix operations are common in pattern recognition tasks, signal processing, image processing, gene expression, and video concept detection, among others. Revising a wide range of works, such as [14,15,39,42], it is possible to conclude that the most commonly used arithmetic operations are addition–subtraction and matrix multiplication, followed by transpose and norm. All the listed operations are also among the most resource-consuming ones.

Simultaneously to the extensive amount of social data generated, computation techniques are constantly advancing due to the advent of new technologies. Single-processor architectures have evolved into multi-core architectures, and in recent years, Graphics Processing Units (GPUs) have emerged as co-processors capable of handling large amount of calculations. However, several complexities also arise from the interaction between computer processors and the data involved in computations [12]. Also, the development of new technologies with complex memory hierarchies leads to a continuous demand for new algorithms and software libraries to efficiently cope with these new architectural features [12].

Although computers have fast performing processors, memory accesses continue to be relatively slow. Thus, the development and advances on computer hardware have thoroughly influenced the development of linear algebra algorithms. In this context, the parallel processing of matrix operations in distributed memory architectures arises as an important field of study [20,38,60]. In particular, operations with dense matrices have been the subject of intensive research [8–10,23,38,59], whereas the problem of operating with sparse matrices has comparatively received less attention. Finally, there are several linear algebra software libraries available in various programming languages, such as Fortran, C++ and Java. However, as the experimental evaluation shows, most of them are not suitable for high-performance applications.

This paper proposes a novel approach for dividing the processing of matrix arithmetic operations into simpler and independent tasks. These tasks are then executed in parallel on a computer cluster for enabling the processing of large-scale sparse matrices. A key factor to this approach is the definition of several strategies that focus on how to divide the matrix arithmetic operations into the independent tasks to be processed in parallel. As each type of operation has different information sharing requirements, the strategies rely on the intrinsic characteristics of the operations and their associated matrices to compute a value called "Parallel-Factor" that determines how matrix operations are partitioned to be processed on a computer cluster. The strategies are applied to the most common operations: addition–subtraction, matrix multiplication and a combination of both, known as Laplacian, applied to a high-dimensional social feature selection approach that considers not only features and posts, but also the social context of posts and relationships between users. Two alternative sparse matrix implementations are proposed and evaluated in terms of resource requirements and performance. The extent to which the sparseness of matrices affects the performance of the proposed strategies and the importance of balancing the amount of work to be performed by each parallel task are discussed. Finally, the performance of the strategies is compared to the performance of several linear algebra libraries.

The rest of this paper is organised as follows: Section 2 presents the motivation for optimising sparse matrix arithmetic operations in the context of scientific computing. Section 3 presents and defines the different strategies for dividing the arithmetic operations for executing them in a computer cluster. Section 4 describes the settings of the experimental evaluation, which was carried out using the feature selection approach presented by [42] as a potential application. Section 5 reports the obtained results and discusses the importance of considering the intrinsic characteristics of matrices and the importance of adequately balancing the work to be performed by each parallel task. Section 6 discusses related research. Finally, Sect. 7 states the conclusions.

## 2 Motivation

Large-scale text analysis, as the one needed in social environments, poses the challenge of efficiently performing arithmetic operations over high-dimensional data represented in a document-term matrix. Text learning tasks are characterised by the high dimensionality of their feature space, even when most terms have a low frequency. Indeed, text learning is often susceptible to the problem known as the "curse of dimensionality," which refers to the increasing computational complexity of learning tasks as the data that need to be accessed grow exponentially regarding the underlying space dimension. Furthermore, as data dimensionality increases, the size of the feature space rapidly grows and the available data become sparser. Thus, in most cases the resulting document-term matrices are sparse. Several methods for large-scale text processing found in the literature are based on different arithmetic operations over document-term matrices. Most of these methods can be mathematically defined as optimisation problems whose solutions, due to their computational complexity, are found by means of iterative algorithms. Each iterative step usually requires the application of one or more arithmetic operations between matrices. In this context, due to the particular characteristics of sparse matrices, using software libraries designed for dense matrices might not be adequate. Hence, new alternatives for operating with sparse matrices are needed.

Among the tasks requiring the manipulation of high-dimensional matrices is feature selection (FS) [2], a commonly used technique for dimensionality reduction. FS aims at choosing a small subset of relevant and discriminative features, excluding redundant, and often correlated and noisy features, according to certain evaluation criterion. High-dimensional feature spaces could lead to low efficiency and poor performance of learning algorithms. FS allows enhancing predictions, speeding-up data mining algorithms and improving mining performance. The feature selection problem of social media texts has been extensively studied [14,26,42,43,46]. Gu and Han [14] proposed a supervised FS method based on Laplacian Regularised Least Squares (LapRLS) for networked data. The authors used linear regression to represent text contents and graph regularisation to consider the social link information. The regularisation technique assumes that if two nodes are linked, their categories or classes are likely to be the same. The FS method is designed to select the subset of features that minimises the LapRLS training error using the sub-gradient descent method as defined by the Nesterov's method [34], involving the same operations between matrices as the regularisation technique. Experimental evaluation on several sparse datasets showed that the approach was able to outperform traditional FS techniques.

Li et al. [26] proposed an unsupervised approach named Clustering-Guided Sparse Structural Learning (CGSSL) that combines non-negative spectral analysis and structural learning with sparsity. Non-negative spectral clustering aims at learning more accurate cluster labels and simultaneously guide FS. FS was performed by means of non-negative spectral clustering, whereas clustering was performed by structural analysis, which requires the $\ell_{2,1}$-norm regularisation. Additionally, the cluster labels are also predicted by linear models that exploit the hidden structure of labels, and help to uncover feature correlations and their semantic meanings. Since spectral clustering involves solving a NP-hard discrete optimisation problem, the authors relaxed the discrete condition and applied an iterative algorithm for obtaining the solution. Experimental evaluation was performed not only on sparse short texts but also on digital handwriting, biomedical data, and face image datasets. According to the authors, the results demonstrated the efficiency and effectiveness of their algorithm.

Tang and Liu [42,43] and Tang et al. [46] applied FS to social media data. Tang et al. [46] proposed an approach for unsupervised FS for multi-view data, which are represented by heterogeneous feature spaces. For example, in *Flickr*, each post can comprise three different views: visual content, tags, and text descriptions. The approach aimed at simultaneously selecting features for all views by using spectral analysis to exploit the relations among them. The authors also introduced the concept of pseudo-class labels to exploit the relations among views as well as the information of each view. Then, the relations among views can be formulated as an optimisation problem. As the problem considers constraint vectors of zero norm mixed with integer programming, its solution was found by iteratively performing multiple matrix operations. Tang and Liu [43] considered traditional attributes and social media data instead of multi-view data. As in the previous case, the problem formulation included spectral analysis and solving the minimisation problem derived from the $\ell_{2,1}$-norm. Finally, Tang and Liu [42] proposed an approach for supervised FS hypothesising about social correlation theories such as homophily [32] and social influence [31]. According to the authors, each approach improved both traditional and state-of-the-art techniques.

Other authors [15,18,30,35,62] presented methods for FS applied to image processing, face recognition, speech recognition, video concept detection, 3D motion data analysis, sonar signal classification, and gene expression and mass spectrometry classifications, which relied on arithmetic operations between matrices. Most of the approaches [30,35,62] are based on solving the optimisation problem posed by the $\ell_{2,1}$-norm regularisation by means of iterative algorithms, involving arithmetic operations between sparse matrices.

Besides FS, other data analysis problems requiring efficient sparse matrix processing are the detection of erroneous or corrupted information [29], i.e. data outliers, collaborative filtering [21,25,63], followee recommendation [57], computational fluid dynamics [55], and signal and image processing [51], among others. Additionally, arithmetic operations between sparse matrices are the basis of other text processing techniques for feature enrichment [39,45], sentiment analysis [19], community detection [44], and document retrieval [54], among others. Regarding feature enrichment, Tang et al. [45] proposed to integrate multi-language knowledge and FS through matrix factorisation techniques, aiming at improving clustering performance for short texts in social media. The matrix factorisation techniques mapped terms and short texts to a joint latent factor space. Feature integration and selection is performed by means of the Goldstein condition and an iterative algorithm based on several arithmetic operations between matrices. Experimental evaluation reported clustering performance improvements. Also, Qi et al. [39] proposed an approach to annotate images by mining context and content links in social media, thus discovering the underlying latent semantic space.

In the sentiment analysis field, Hu et al. [19] studied whether social relations in microblogging can help in sentiment analysis tasks. The authors integrated sentiment relations between messages by building a latent connection based on sentiment consistency (two messages are posted by the same user) and emotional contagion (there is a relationship between two users). The approach was mathematically formulated as a non-smooth optimisation problem, and then solved by means of the Goldstein rule and the Nesterov's method. Experimental evaluation based on *Twitter* showed performance improvements regarding traditional methods. Regarding community detection, Tang et al. [44] proposed a joint optimisation framework to integrate multiple data sources to discover communities in social media. The integration of the two data sources was modelled as a joint optimisation problem through sparse matrix factorisation techniques. Experimental evaluation showed significant improvements in community detection.

The manipulation of high-dimensional data representing document-term matrices is a common concern in short text clustering. As it was previously described, Li et al. [26] simultaneously performed FS and clustering by integrating cluster analysis and sparse structural analysis into a joint framework. On the other hand, Xu et al. [54] proposed a clustering method based on the non-negative factorisation of the document-term matrix. The latent semantic structure of the collection was derived from the non-negative matrix factorisation, in which each axis represented the base topic of a particular cluster, and each point of data was a combination of those topics. Experimental evaluation showed that the approach outperformed spectral and singular vector decomposition clustering approaches in terms of accuracy and precision.

In summary, it has been shown that arithmetic operations between sparse matrices are frequently used in diverse scientific computing areas and thus worthy of intensive research due to their high computational complexity. Text represents a specific kind of data in which the feature space is highly dimensional and frequencies are low or zero for most words [1], i.e. texts are sparse. The sparseness is accentuated when considering social data constraint in their length, for example tweets have 140 characters at most. Furthermore, as synonymy and polysemy are more common in social data than in formal text, the size of the feature space is increased and the frequency of terms reduced, which in turn increases the data sparseness.

Table 1 summarises the arithmetic operations between matrices used by the described methods. In this context, the parallel processing of matrix operations in distributed memory architectures arises as a critical problem that should be addressed in order to cope with both the increasing number of performed operations and the performance requirements, by exploiting the advances of new architectures. As the table shows, the most common operations are addition–subtraction and matrix multiplication, followed by transpose and norm. Given the unquestionable need for efficiently processing the mentioned operations between matrices, this work presents and experimentally evaluates an approach for distributing sparse matrix arithmetic operations on computer clusters, specifically addition–subtraction, matrix multiplication operations and a combination of both of them known as Laplacian. These operations not only are among the most computational expensive ones, but also their frequency of usage makes them critical for the performance of the applications, or other composed operations.

## 3 Parallelisation of sparse matrix arithmetic operations

This work proposes a novel approach for distributing the parallel processing of matrix arithmetic operations on computer clusters by dividing them into independent tasks to be processed in parallel. The approach is based on several strategies that are designed to determine how to divide the operations into the independent tasks by means of computing a Parallel Factor (*PF*). The *PF* value is computed for three types of operations: Addition–subtraction, matrix multiplication and Laplacian.

The *PF* calculation is based on the two aspects that distinguish the different arithmetic operations. First, information sharing requirements. For example, in the case of adding the rows of two matrices, only those rows are required for performing the computation, independently of any other row in the matrix. As a result, an addition–subtraction task only needs to know the specific rows to be added from both matrices. On the contrary, in the case of multiplying two matrices, each row in the left matrix is multiplied to each column on the right matrix. Consequently, for computing the resulting cell values involving a specific row, the multiplication task needs to know the row from the left matrix and all of the columns from the right matrix. Second, the relevance of the matrices involved in each operation. For example, in the case of matrix multiplications, it is important to consider the characteristics of the left matrix as its sparseness defines the number of actual multiplications that have to be performed, whereas the Laplacian operation is independent of the sparseness of the matrix involved.

Once it is computed, the *PF* can be used to determine the number of elements assigned to each parallel task to be created and executed on the computer cluster. The amount of work to be performed by each parallel task can be measured by considering the number of individual arithmetic operations that they perform. For example, in the case of adding two matrices $A$ and $B$, the addition of each individual cell, i.e. $A_{ij} + B_{ij}$ where $i$ denotes the row and $j$ denotes the column of cells, defines an individual operation. In order to determine whether balancing the amount of work to be performed by each parallel task has an impact on the overall performance, two kinds of elements were considered: number of rows and number of non-zero values, leading to two different workload distribution strategies. The first workload distribution strategy (*PF*-*R*) distributes work among the parallel tasks regardless of their sparseness level, and hence the amount of work to be performed by each parallel task. In other words, each task is assigned the same fixed number of rows to operate with. The second strategy (*PF*-*NZ*) aims at equitably dividing the number of individual operations to be performed, i.e. the number of non-zero values, and hence the amount of work assigned to

**Table 1** Usage of matrix arithmetic operations in the literature

| Approach | Objective | Addition–subtraction | Matrix multiplication | Scalar multiplication | Laplacian | Transpose | Norm | Trace | Inverse | Others |
|---|---|---|---|---|---|---|---|---|---|---|
| [14] | Feature selection in short texts | X | X | X | X | X | X | X | X | |
| [26] | Feature selection and clustering of short texts | X | X | X | | X | X | X | X | Eigenvector decomposition |
| [46] | Feature selection in short texts | X | X | X | X | X | X | | X | |
| [43] | Feature selection in short texts | X | X | X | X | X | X | | | |
| [42] | Feature selection in short texts | X | X | X | X | X | X | | | |
| [18] | Feature selection in multimedia data | X | X | X | | X | X | | X | |
| [62] | Feature selection in multimedia data | X | X | X | | X | X | | X | |
| [30] | Feature selection in multimedia data | X | X | X | X | X | | | X | |
| [35] | Feature selection in gene expression | X | X | X | | X | X | | X | |
| [15] | Feature selection in multimedia data | X | X | X | | X | | X | X | |
| [45] | Feature enrichment of texts | X | X | X | | X | | | | Dot product |
| [39] | Feature enrichment of images | X | X | X | X | | | | | |
| [19] | Sentiment analysis | X | X | X | X | X | X | | | |
| [44] | Community detection | X | X | X | | X | | | X | |
| [54] | Clustering of short text | X | X | | | X | | | | |
| [29] | Detection of erroneous information | X | X | X | | X | | | X | |

each parallel task. In other words, each task is assigned the same fixed number of non-zero elements to operate with, regardless of the number of rows involved. In all cases, the *PF* value is inversely related to the number of rows or non-zeros per tasks, i.e. the bigger the *PF*, the lower the number of rows or non-zeros per tasks, and thus, the bigger the number of tasks.

All the presented strategies except for the first one rely on the *PF* computation as defined in Eq. (1):

$$\text{Parallel-Factor} = \lfloor \text{\#physical-cores} \times (1 + \gamma + \alpha) \rfloor \tag{1}$$

where $\gamma$ is defined as the ratio of the number of rows and columns, and is shared by all the strategies:

$$\gamma = \begin{cases} 1 - \log\left(\frac{\#rows}{\#columns}\right) & \#rows \leq \#columns \\ 1 - \log\left(\frac{\#columns}{\#rows}\right) & \#rows > \#columns \end{cases} \tag{2}$$

Oppositely, $\alpha$ is defined as a statistic based on the sparseness level of the corresponding matrix, and it varies according to the selected strategy. To prevent assigning zero rows or non-zeros to each task when the *PF* is zero, it is replaced by 1 to always create at least one task. In Eq. (1), #physical-cores refers to the total number of CPU cores that compose the computer cluster.

Based on the concepts and elements described, the rest of this section describes the strategies for parallelising matrix arithmetic operations. All the strategies are formally defined and presented along with an application example. The definition of the matrix used in all of the examples can be found in Fig. 1, which shows the sparseness value of each row and the calculation of $\gamma$ for 1 physical-core. In all cases, the computed *PF* is used to divide the matrix by considering both workload distribution strategies.

## 3.1 Static

This is the simplest strategy as it only depends on a constant value that allows the creation of an arbitrary number of tasks. This constant value, called Granularity-Factor, defines the extend to which the operation is split into tasks, i.e. the granularity of the split operation. As

$$\#Rows = 8 \quad \#Columns = 9 \quad \#\text{Non-Zeros} = 39$$

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 6 & 6 & 6 & 6 & 0 & 6 \\ 7 & 7 & 7 & 0 & 7 & 0 & 7 & 7 & 7 \\ 0 & 6 & 6 & 6 & 6 & 0 & 6 & 6 & 0 \\ 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 \\ 8 & 8 & 0 & 8 & 8 & 8 & 8 & 8 & 8 \end{bmatrix} \quad \begin{array}{ll} \text{Non-Zeros}_1 = 1 & \text{Row-Sparseness}_1 = 0.88889 \\ \text{Non-Zeros}_2 = 1 & \text{Row-Sparseness}_2 = 0.88889 \\ \text{Non-Zeros}_3 = 1 & \text{Row-Sparseness}_3 = 0.88889 \\ \text{Non-Zeros}_4 = 6 & \text{Row-Sparseness}_4 = 0.33333 \\ \text{Non-Zeros}_5 = 7 & \text{Row-Sparseness}_5 = 0.22222 \\ \text{Non-Zeros}_6 = 6 & \text{Row-Sparseness}_6 = 0.33333 \\ \text{Non-Zeros}_7 = 9 & \text{Row-Sparseness}_7 = 0 \\ \text{Non-Zeros}_8 = 8 & \text{Row-Sparseness}_8 = 0.11111 \end{array}$$

$$\text{Parallel-Factor} = PF = \lfloor \text{\#physical-cores} \times (1 + \gamma + \alpha) \rfloor$$

$$\text{\#physical-cores} = 1 \qquad \begin{aligned} \gamma &= 1 - log\left(\frac{\#rows}{\#columns}\right) \\ &= 1 - log(8/9) \\ \gamma &= 1.11778 \end{aligned}$$

**Fig. 1** Example of parallelising strategies—matrix definition

$$\text{Granularity-Factor} = 4 \qquad \begin{aligned} PF &= \lfloor \#\text{physical-cores} \times \text{Granularity-Factor} \rfloor \\ &= \lfloor 1 \times 4 \rfloor \\ PF &= 4 \end{aligned}$$

$\#$Rows-per-Taks$= \lfloor \#rows/PF \rfloor = \lfloor 8/4 \rfloor = 2$

$\#Tasks = 4$

$Task_1 = [row_1, row_2]$

Non-Zeros$_{Task_1} = 2$ $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

$Task_2 = [row_3, row_4]$

Non-Zeros$_{Task_2} = 7$ $\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 6 & 6 & 6 & 6 & 0 & 6 \end{bmatrix}$

$Task_3 = [row_5, row_6]$

Non-Zeros$_{Task_3} = 13$ $\begin{bmatrix} 7 & 7 & 7 & 0 & 7 & 0 & 7 & 7 & 7 \\ 0 & 6 & 6 & 6 & 6 & 0 & 6 & 6 & 0 \end{bmatrix}$

$Task_4 = [row_7, row_8]$

Non-Zeros$_{Task_4} = 17$ $\begin{bmatrix} 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 \\ 8 & 8 & 0 & 8 & 8 & 8 & 8 & 8 & 8 \end{bmatrix}$

$\#$NonZeros-per-Taks$= \lfloor \#\text{non-zero}/PF \rfloor = \lfloor 39/4 \rfloor = 9$

$\#Tasks = 4$

$Task_1 = [row_1, row_2, row_3, row_4]$

Non-Zeros$_{Task_1} = 9$ $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 6 & 6 & 6 & 6 & 0 & 6 \end{bmatrix}$

$Task_2 = [row_5, row_6]$

Non-Zeros$_{Task_2} = 13$ $\begin{bmatrix} 7 & 7 & 7 & 0 & 7 & 0 & 7 & 7 & 7 \\ 0 & 6 & 6 & 6 & 6 & 0 & 6 & 6 & 0 \end{bmatrix}$

$Task_3 = [row_7]$

Non-Zeros$_{Task_3} = 9$ $\begin{bmatrix} 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 \end{bmatrix}$

$Task_4 = [row_8]$

Non-Zeros$_{Task_4} = 8$ $\begin{bmatrix} 8 & 8 & 0 & 8 & 8 & 8 & 8 & 8 & 8 \end{bmatrix}$

**Fig. 2** Example of parallelising strategies—*Static*

Eq. 3 shows, there is a direct correlation between the *PF* and the Granularity-Factor, i.e. the bigger the Granularity-Factor, the bigger the *PF*:

$$\text{Parallel-Factor} = \lfloor \#\text{physical-cores} \times \text{Granularity-Factor} \rfloor \tag{3}$$

Figure 2 shows an example of the *Static* strategy for the matrix *A* and an arbitrary value of 4 for the Granularity-Factor. As it can be seen, although the number of created tasks is the same regardless of the applied workload distribution strategy, the distribution of rows and non-zero elements among tasks differs. When considering the number of non-zero elements assigned to each task for dividing the matrix, the workload assigned to each task is more balanced than when considering the number of rows assigned to each of them. Particularly, the standard deviation of the number of non-zeros when considering the workload strategy distribution based on rows is 5.717, whereas when considering the workload strategy distribution based on non-zeros is 1.920.

### 3.2 Row-Sparseness

This strategy aims at capturing the sparseness of each particular row by considering the mean row sparseness of the matrix. The rationale behind this strategy is to establish an inverse relation between the *PF* and the row sparseness. As a low row sparseness value would indicate highly populated rows, which can imply a higher number of operations to be performed, the number of elements (rows or non-zeros) per tasks should decrease. Equation (4) shows how to compute the $\alpha$ value for this strategy:

$$\alpha = 1 - \overline{\left( \frac{\text{non-zero}_i}{\#columns} \right)} \tag{4}$$

Figure 3 shows an example of the *Row-Sparseness* strategy for the matrix *A*. As for the *Static* strategy, even though the number of created tasks is the same regardless of the applied workload distribution strategy, the distribution of rows and non-zero elements among tasks differs. When considering the number of non-zero elements assigned to each task for dividing the matrix, the workload assigned to each task is more balanced than when considering the number of rows assigned to each of them. Particularly, the standard deviation of the number

$$\begin{aligned}\alpha &= 1-\overline{\left(\frac{\text{non-zero}_i}{\#columns}\right)}\\ &= 1-0.45833\\ \alpha &= 0.54166\end{aligned}\qquad\qquad \begin{aligned}PF &= \lfloor \#\text{physical-cores}\times(1+\gamma+\alpha)\rfloor\\ &= \lfloor 1\times(1+1.11778+0.54166)\rfloor=\lfloor 2.65944\rfloor\\ PF &= 2\end{aligned}$$

$\#$Rows-per-Taks$=\lfloor \#rows/PF\rfloor=\lfloor 8/2\rfloor=4$

$\#Tasks=2$

$Task_1=[row_1,row_2,row_3,row_4]$

Non-Zeros$_{Task_1}=9$ $\begin{bmatrix}1&0&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0&0\\6&0&0&6&6&6&6&0&6\end{bmatrix}$

$Task_2=[row_5,row_6,row_7,row_8]$

Non-Zeros$_{Task_2}=30$ $\begin{bmatrix}7&7&7&0&7&0&7&7&7\\0&6&6&6&6&0&6&6&0\\9&9&9&9&9&9&9&9&9\\8&8&0&8&8&8&8&8&8\end{bmatrix}$

$\#$NonZeros-per-Taks$=\lfloor \#\text{non-zero}/PF\rfloor=\lfloor 39/2\rfloor=19$

$\#Tasks=2$

$Task_1=[row_1,row_2,row_3,row_4,row_5,row_6]$

Non-Zeros$_{Task_1}=22$ $\begin{bmatrix}1&0&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0&0\\6&0&0&6&6&6&6&0&6\\7&7&7&0&7&0&7&7&7\\0&6&6&6&6&0&6&6&0\end{bmatrix}$

$Task_2=[row_7,row_8]$

Non-Zeros$_{Task_2}=17$ $\begin{bmatrix}9&9&9&9&9&9&9&9&9\\8&8&0&8&8&8&8&8&8\end{bmatrix}$

**Fig. 3** Example of parallelising strategies—*Row-Sparseness*

of non-zeros when considering the workload strategy distribution based on rows is 10.5, whereas when considering the workload strategy distribution based on non-zeros is 2.5.

## 3.3 Row-Sparseness Standard Deviation (Row-Sparseness-SD)

Since all data points are used to compute the mean of a distribution, outliers can affect the accuracy of results. This strategy aims at considering the standard deviation of the data distribution in order to add information regarding the existence of outliers. The standard deviation ($\sigma$) measures the dispersion of data from the mean. A low $\sigma$ indicates that the data points are close to the mean, whereas a high $\sigma$ indicates that the data points are spread over a large range of values. As Eq. (5) shows, $\alpha$ considers the difference between the mean and the standard deviation, i.e. the lowest sparseness value in the normal distribution:

$$\alpha = 1-\left(\overline{\left(\frac{\text{non-zero}_i}{\#columns}\right)}-\sigma\left(\frac{\text{non-zero}_i}{\#columns}\right)\right) \tag{5}$$

Figure 4 shows an example of the *Row-Sparseness-SD* strategy for the matrix $A$. In this case, the number of tasks to be created differs according to the applied workload distribution strategy. Four tasks are created when considering the workload distribution based on rows, whereas only 3 tasks are created when considering the workload distribution based on non-zeros. Although when considering the workload distribution strategy based on non-zeros fewer tasks are created, the workload assigned to each task is more balanced than when considering the workload distribution strategy based on rows. In this case, the standard deviation of the number of non-zeros when distributing the workload based on the number of rows is 5.717, whereas when distributing the workload based on the number of non-zeros is 3.559.

## 3.4 Row-Sparseness Mode (Mode)

The mode of a data distribution can be defined as the most frequent value, which in this case can be defined as the most frequent row sparseness value, as shown in Eq. (6):

$$\alpha = 1-\text{most frequent}\left\{\frac{\text{non-zero}_i}{\#columns}\right\} \tag{6}$$

Notice that the mode of a distribution may not accurately represent the data as there may be more than one mode value, or no value at all if no value occurs more than once. Additionally,

$$\begin{aligned}\alpha &= 1-\left(\overline{\left(\frac{\text{non-zero}_i}{\#columns}\right)}-\sigma\left(\frac{\text{non-zero}_i}{\#columns}\right)\right)\\ &= 1-(0.45833-0.34888)\\ \alpha &= 0.89055\end{aligned}$$

$$\begin{aligned}PF &= \lfloor \#\text{physical-cores} \times (1+\gamma+\alpha)\rfloor\\ &= \lfloor 1\times(1+1.11778+0.89055)\rfloor = \lfloor 3.00833\rfloor\\ PF &= 3\end{aligned}$$

$\#$Rows-per-Taks= $\lfloor \#rows/PF\rfloor = \lfloor 8/3\rfloor = 2$

$\#Tasks = 4$

$Task_1 = [row_1, row_2]$

$$\text{Non-Zeros}_{Task_1} = 2 \quad \begin{bmatrix} 1&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\end{bmatrix}$$

$Task_2 = [row_3, row_4]$

$$\text{Non-Zeros}_{Task_2} = 7 \quad \begin{bmatrix} 0&0&1&0&0&0&0&0&0\\ 6&0&0&6&6&6&6&0&6\end{bmatrix}$$

$Task_3 = [row_5, row_6]$

$$\text{Non-Zeros}_{Task_3} = 13 \quad \begin{bmatrix} 7&7&7&0&7&0&7&7&7\\ 0&6&6&6&6&0&6&6&0\end{bmatrix}$$

$Task_4 = [row_7, row_8]$

$$\text{Non-Zeros}_{Task_4} = 17 \quad \begin{bmatrix} 9&9&9&9&9&9&9&9&9\\ 8&8&0&8&8&8&8&8&8\end{bmatrix}$$

$\#$NonZeros-per-Taks= $\lfloor \#\text{non-zero}/PF\rfloor = \lfloor 39/3\rfloor = 13$

$\#Tasks = 3$

$Task_1 = [row_1, row_2, row_3, row_4, row_5]$

$$\text{Non-Zeros}_{Task_1} = 16 \quad \begin{bmatrix} 1&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0\\ 6&0&0&6&6&6&6&0&6\\ 7&7&7&0&7&0&7&7&7\end{bmatrix}$$

$Task_2 = [row_6, row_7]$

$$\text{Non-Zeros}_{Task_2} = 15 \quad \begin{bmatrix} 0&6&6&6&6&0&6&6&0\\ 9&9&9&9&9&9&9&9&9\end{bmatrix}$$

$Task_3 = [row_8]$

$$\text{Non-Zeros}_{Task_3} = 8 \quad \begin{bmatrix} 8&8&0&8&8&8&8&8&8\end{bmatrix}$$

**Fig. 4** Example of parallelising strategies–*Row-Sparseness Standard Deviation*

$$\begin{aligned}\alpha &= 1-\text{most-frequent}\left\{\frac{\text{non-zero}_i}{\#columns}\right\}\\ &= 1-0.88889\\ \alpha &= 0.11111\end{aligned}$$

$$\begin{aligned}PF &= \lfloor \#\text{physical-cores} \times (1+\gamma+\alpha)\rfloor\\ &= \lfloor 1\times(1+1.11778+0.11111)\rfloor = \lfloor 2.22889\rfloor\\ PF &= 2\end{aligned}$$

$\#$Rows-per-Taks= $\lfloor \#rows/PF\rfloor = \lfloor 8/2\rfloor = 4$

$\#Tasks = 2$

$Task_1 = [row_1, row_2, row_3, row_4]$

$$\text{Non-Zeros}_{Task_1} = 9 \quad \begin{bmatrix} 1&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0\\ 6&0&0&6&6&6&6&0&6\end{bmatrix}$$

$Task_2 = [row_5, row_6, row_7, row_8]$

$$\text{Non-Zeros}_{Task_2} = 30 \quad \begin{bmatrix} 7&7&7&0&7&0&7&7&7\\ 0&6&6&6&6&0&6&6&0\\ 9&9&9&9&9&9&9&9&9\\ 8&8&0&8&8&8&8&8&8\end{bmatrix}$$

$\#$NonZeros-per-Taks= $\lfloor \#\text{non-zero}/PF\rfloor = \lfloor 39/2\rfloor = 19$

$\#Tasks = 2$

$Task_1 = [row_1, row_2, row_3, row_4, row_5, row_6]$

$$\text{Non-Zeros}_{Task_1} = 22 \quad \begin{bmatrix} 1&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0\\ 6&0&0&6&6&6&6&0&6\\ 7&7&7&0&7&0&7&7&7\\ 0&6&6&6&6&0&6&6&0\end{bmatrix}$$

$Task_2 = [row_7, row_8]$

$$\text{Non-Zeros}_{Task_2} = 17 \quad \begin{bmatrix} 9&9&9&9&9&9&9&9&9\\ 8&8&0&8&8&8&8&8&8\end{bmatrix}$$

**Fig. 5** Example of parallelising strategies–*Row-Sparseness Mode*

in case the mode exists, it favours the most common sparseness value, which could cause an unbalanced distribution of elements per task in terms of the resulting sparseness, affecting the actual parallelism of the application.

Figure 5 shows an example of the *Mode* strategy for the matrix $A$. In this case, the *PF* has the same value than that of the *Row-Sparseness* strategy. As a result, both strategies result on the same task distributions.

## 4 Experimental settings

This section presents the experimental settings to assess the effectiveness of the proposed approach and is organised as follows. Section 4.1 describes the high-dimensional FS approach that was selected for evaluating the performance of the proposed matrix arithmetic operations parallelisation strategies. Section 4.2 details the implementation of the approach, including the selected programming language, the selected framework for executing the parallel tasks on the computer cluster, and the baseline for comparing and evaluating the performance of the presented strategies. Finally, Sect. 4.3 presents the data collections used in the experimental evaluation of the presented approach.

### 4.1 Feature selection application

The performance of the proposed approach and thus that of the strategies for computing the *PF*, and determining the number of parallel tasks to be created, was evaluated for the feature selection approach presented in [42]. Interestingly, this FS approach considers not only features and posts, but also the social context of the posts and the relationship between users, producing high-dimensional matrices representing texts and social contexts. According to the authors, social media data analysis and social correlation theories such as homophily [32] and social influence [31] suggest the existence of four types of relations: *Co-Post* (posts written by the same user share the same topic), *Co-Following* (if two users follow the same user, their posts are likely to be topically related), *Co-Followed* (if two users are followed by the same user, their posts are likely to be topically related) and *Following* (relations between users are based on the relatedness of the post's topics).

For each type of relation (*Co-Post*, *Co-Following*, *Co-Followed* and *Following*), the authors proposed a hypothesis about how its application affects the feature selection process. The relations were modelled by high-dimensional matrices and arithmetic operations between them. Using those matrices, the authors defined feature selection as an optimisation problem based on the $\ell_{2,1}$-norm to select features across data points with joint sparseness. In this formulation, data instances are assumed to be independent from each other. Each relation adds a new term to the optimisation problem in order to explicitly define its hypothesis. In particular, the *Co-Post* relation adds to the optimisation problem a regularisation term to reinforce the hypothesis that the topics of posts written by the same user are more similar than those of randomly selected posts. The general definition of the optimisation problem involves two matrices, which are specific to each relation, denoted $B$ and $E$. In the case of the *Co-Post* relation, both matrices can be defined as follows:

$$B = XX^T + \beta F L_A F^T \tag{7}$$

$$E = Y^T X^T = (XY)^T \tag{8}$$

where $\beta$ represents the impact of the social relation on the feature selection process, $F$ represents the set of features for a post where $F_{i,j}$ is the frequency of $feature_j$ in $post_i$, $X$ represents a subset of $F$ in which only the labelled post are considered, $Y$ represents the class label matrix for labelled data where $Y_{i,j} = 1$ if $post_i$ is labelled as $class_j$ and zero otherwise, and $L_A$ represents the Laplacian matrix of $A = P^T P$ where $P$ represents the set of posts of a user and then $A_{i,j} = 1$ if $post_i$ and $post_j$ were posted by the same user and zero otherwise.

As it can be difficult to obtain a closed solution of the optimisation problem, the authors proposed an iterative algorithm based on the semi-positive definition of the matrices involved, which converges to the optimal solution. Complete definitions, lemmas, theorems, mathematical proofs, and algorithm can be found in [42]. The iterative algorithm involves performing the same arithmetic operations between matrices as the ones required for computing $B$ and $E$ (addition–subtraction, matrix multiplication, scalar multiplication, transpose), as well as the norm and inverse. As the operations required for computing $B$ and $E$ are also performed by the iterative algorithm, improvements in the operations required for computing $B$ and $E$ would also result in improvements in the iterative algorithm. Additionally, the computation

**Table 2** List of individual operations involved in computing $B$ and $E$

| | |
|---|---|
| $B$ | |
| Matrix-Multiplication I | $A = P^T P$ |
| Addition-Subtraction I | $D_{A_{i,i}} = \sum_j A_{j,i}$ |
| Addition-Subtraction II | $L_A = D_A - A$ |
| Matrix-Multiplication II | $\beta F L_A$ |
| Matrix-Multiplication III | $\beta F L_A F^T$ |
| Matrix-Multiplication IV | $XX^T$ |
| Addition-Subtraction III | $XX^T + \beta F L_A F^T$ |
| $E$ | |
| Matrix-Multiplication V | $(XY)^T$ |

of $B$ and $E$ involves repeatedly performing the most computationally complex operations. In consequence, this work focuses on distributing the parallel processing of the arithmetic operations needed for $B$ and $E$ (addition–subtraction, matrix multiplication and Laplacian) on computer clusters. Table 2 shows the list of the involved individual operations, which were the focus of the experimental evaluation. Note that the Laplacian operation is separated into three individual operations: *Matrix-Multiplication I, Addition-Subtraction I* and *Addition-Subtraction II*.

## 4.2 Implementation details

The Java programming language was chosen for implementing the approach [41], as it was shown that it can achieve a similar performance to optimised languages such as Fortran [33]. Other advantages of using Java include its portability and interoperability with other programming languages. The portability of the application becomes of highly relevance when considering its execution in heterogeneous systems. For example, in the case of an heterogeneous computer cluster including different operating systems or even different architectures, an application coded in Java could be executed in such cluster without the necessity of being re-compiled according to the characteristics of each individual computer. The utilisation of Java as the programming language does not restrict the extend of the utilisation of the application developed, as it can be used in multiple platforms and in combination with applications written in other programming languages, without changes in the original coded application.

Matrices were implemented as sparse memory structures in order to decrease network transfer, disc storage, and RAM memory requirements. The performance of two internal implementations of matrices was evaluated. First, matrices were implemented using the *HashMap* structure provided by Java, which only store objects (non-primitive-type elements) incurring in boxing and un-boxing operations each time an element is retrieved from the structure. Second, matrices were implemented using the *Trove*[1] library of hash structures, which stores primitive types elements to avoid boxing and un-boxing. Although the double type provides more precision in operations than the float type, the latter was chosen as the type of the matrices' elements due to the space required to store a matrix of double elements.

The distribution and execution of tasks on the computer cluster was performed by using the Java Parallel Processing Framework (JPPF)[2] middleware, which allows applications to be

[1] http://trove.starlight-systems.com/.

[2] http://www.jppf.org/.

executed on any number of computers. JPPF is a lightweight and mature middleware for task-oriented distributed computing, which achieves performance similar to classic MPI-based solutions, whilst providing a simpler programming model [40]. JPPF follows a master–slave design in which a master computer distributes the tasks of a job among an arbitrary number of slaves for parallel execution. In order to reduce network transfer times and data duplication, matrices shared by more than a task belonging to the same job are not duplicated in the slave. Further reductions are achieved by using the local class loading option in each slave node.

The baseline for comparing and evaluating the enhancements introduced by using the proposed *PF* to distribute tasks on the cluster was the execution of all the operations in a serial and a multi-thread manner. In particular, for the multi-thread execution pools of 3, 4 and 5 threads were considered (namely *Multi-Thread-3, Multi-Thread-4, Multi-Thread-5*. All the serial and multi-thread executions were performed on a i7-3820 processor running at 3.6 GHz with 32 GB RAM. On the other hand, the computer cluster comprised 5 computers having AMD Phenom II X6 1055T processors with 6 physical-cores each running at 2.8 GHz with 8 GB RAM, and 3 computers having AMD FX-6100 processors with 6 physical-cores each running at 3.6 GHz with 16 GB RAM. All computers were connected by means of a Gigabit network.

## 4.3 Datasets

Two data collections were used in the experimental evaluation of the presented approach. The first data collection comprised data extracted from *Digg*[3] that was obtained from [27]. *Digg* is a social news website that allows its users to share and comment content (posts) and to vote the content up or down. In addition, users are encouraged to establish social relationships by adding other users to their friend network and following their activity. The data collection includes information about posts, post authorship, the topic of each post (considered as the class of the post for supervised learning), comment and voting activity on each post, and non-reciprocal social relationships between users. In this dataset, posts were already pre-processed to remove stopwords. For the purpose of the experimental evaluation, the comment and voting activity on each post was discarded. Table 3 summarises the main characteristics of the dataset.

The second data collection comprised data extracted from *BlogCatalog*[4] that was obtained from [52]. *BlogCatalog* is a blog directory in which users can register their blogs under pre-defined categories. In addition, users are also encouraged to establish social relationships by following the activity of other users. The purpose of *BlogCatalog* is to list the blogs available on the Internet and thus provide users with a simple access to find them. The data collection includes a summary of the content of each blog, the categories and tags assigned to each blog, ownership information, and non-reciprocal social relationship between users. The features of each blog were extracted from their summary. Additionally, as each blog may be assigned to more than one category or class, and there was no pre-defined hierarchy between them, classes assigned to each blog were alphabetically sorted and then concatenated to create a unique category. For the purpose of the experimental evaluation, blog summaries were pre-processed to remove all punctuation and stopwords. Then, feature suffixes were automatically removed by means of the Porter Stemmer algorithm [37]. Finally, irrelevant features were also removed by considering their TF-IDF score. In the experimental evaluation, only the

[3] http://www.digg.com/.

[4] http://www.blogcatalog.com/.

**Table 3** *Digg* collection main characteristics

| | |
|---|---:|
| Number of posts | 42,843 |
| Number of features | 8546 |
| Number of classes | 51 |
| Number of following relations | 56,440 |
| Minimum number of followees | 4 |
| Maximum number of followees | 622 |
| Average number of followees | 157 |
| Minimum number of posts per user | 1 |
| Maximum number of posts per user | 1101 |
| Minimum number of features per post | 1 |
| Maximum number of features per post | 10 |
| Average number of features per post | 4 |
| Minimum number of post per class | 38 |
| Maximum number of post per class | 4280 |
| Average number of posts per class | 840 |

**Table 4** *BlogCatalog* collection main characteristics

| | |
|---|---:|
| Number of blogs | 111,648 |
| Number of features | 189,621 |
| Number of classes | 11,701 |
| Number of following relations | 3,348,554 |
| Minimum number of followees | 1 |
| Maximum number of followees | 8368 |
| Average number of followees | 47 |
| Minimum number of blogs per user | 1 |
| Maximum number of blogs per user | 293 |
| Minimum number of features per blog | 1 |
| Maximum number of features per blog | 381 |
| Average number of features per blog | 139 |
| Minimum number of blogs per class | 1 |
| Maximum number of blogs per class | 1047 |
| Average number of blogs per class | 10 |

30 % of the features were considered. Table 4 summarises the main characteristics of the dataset.

## 5 Experimental evaluation

This section reports the results for the four baselines and the four novel strategies presented. In the case of the *Static* strategy, two values for the Granularity-Factor were evaluated, 1 (named *Static-1*) and an arbitrary value of 30 (named *Static-30*). Additionally, two alternatives for each of the strategies based on computing the *PF-NZ* were considered. The non-zero

**Table 5** Matrices' size and sparseness per operation (*Digg*)

| Operation | Size | Sparseness (%) |
|---|---|---|
| *B* | | |
| Matrix-Multiplication I | 42,843 × 42,843 | 99.55 |
| Addition-Subtraction II | 42,843 × 42,843 | 99.55 |
| Matrix-Multiplication II | 8546 × 42,843 | 94.80 |
| $F^T$ | 42,843 × 8546 | 99.96 |
| Matrix-Multiplication III | 8546 × 8546 | 66.94 |
| $X^T$ | 42843 × 8546 | 99.96 |
| Matrix-Multiplication IV | 8546 × 8546 | 99.47 |
| Addition-Subtraction III | 8546 × 8546 | 66.95 |
| *E* | | |
| Matrix-Multiplication V | 8546 × 51 | 66.94 |

elements to assign to each task can be selected either by considering the random order of rows (*PF*-$NZ_r$) or by sorting them in ascending order according to their number of non-zero elements (*PF*-$NZ_s$). Although the number of non-zero elements per tasks is equally computed in both alternatives, sorting rows might change the number and distribution of rows among tasks, which could in turn affect the transfer and communication times involved in performing each task, and then the overall performance of the strategy. For example, if the rows are sorted in ascending order of non-zero elements, the first tasks would include a greater number of rows as they are the most sparse ones, whereas the last tasks would include a lower number of rows as they are the most dense ones. In total, nineteen evaluation cases were considered. All the presented strategies were executed at least five times, so that the mean execution times are reported. The standard deviation of the execution times was lower than a 0.06 % in all cases.

### 5.1 Digg dataset results

Table 5 shows the sparseness level of the result matrices of each of the evaluated operations and several auxiliary matrices in the *Digg* data collection. As it can be observed, the sparseness level of the matrices ranged between 67 and 99 %. The most dense matrix corresponded to the result of a matrix multiplication, which is the most resource-demanding operation.

Figures 6 and 7 compare the results obtained for the matrix representations tested for both *B* and *E* matrices, as defined in Eqs. (7) and (8). The results include the data transfer and storage times, and the computing time of the *PF*, when applicable. As the figures show, the performance of the *Trove*-based implementation was superior to the *HashMap*-based implementation in all cases, improving results from a 7 % when considering the *Serial* execution, up to a 48 % when considering the three different alternatives of the *Static-30* strategy. In addition, the improvements in execution times of the proposed strategies achieved differences up to a 91 % when considering the best performing strategies (*Row-Sparseness*$_{PF\text{-}R}$, *Row-Sparseness*$_{PF\text{-}NZ}$) regarding the *Serial* execution.

Considering the two used matrix representations, the performance difference could be explained in terms of different network transfer and storage times, and thus, it can be stated that the different matrix representations have different storage needs. In this regard, Fig. 8

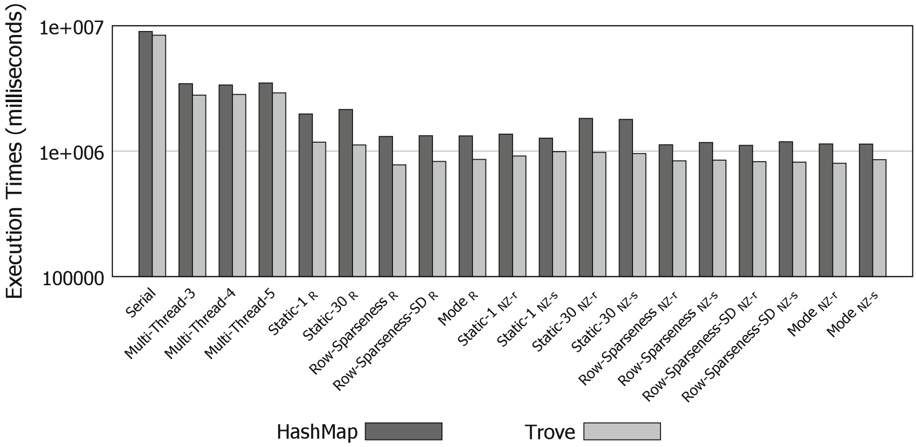

**Fig. 6** *B* matrix overall computing time (*Digg*—logarithmic scale)

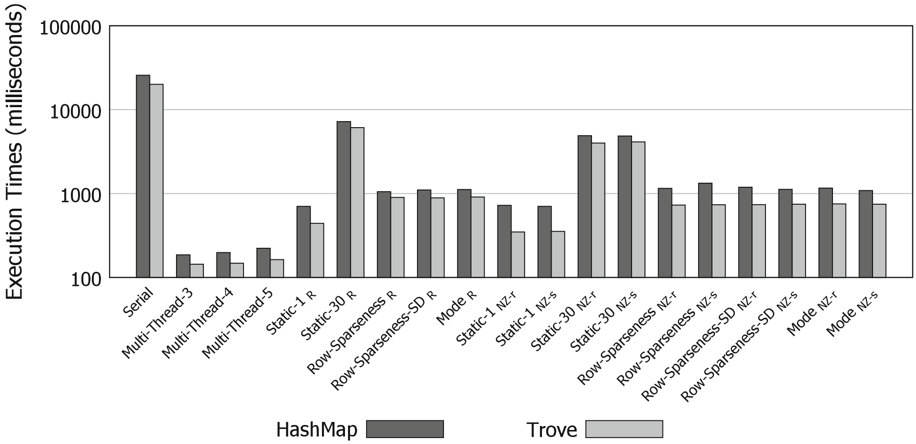

**Fig. 7** *E* matrix overall computing time (*Digg*—logarithmic scale)

shows the size of the matrices involved in the operations presented in Table 5. The *Trove*-based representation required less storage space than the *HashMap*-based representation, with differences ranging from a 11 % for the case of the $F^T$ needed for computing Matrix-Multiplication III up to a 59 % in the case of the *Matrix-Multiplication I*, which is needed for computing the Laplacian. These results could imply that *Trove* uses a more efficient internal structure.

Figure 9 presents the execution times of each operation for the *Serial, Multi-Thread* and *PF-R* and *PF-NZ*-based strategies using the *Trove*-based implementation. The strategies based on distributed executions achieved the best results for most of the operations. The only exception was the *Addition-Subtraction III* operation, in which the Serial execution outperformed the *Multi-Thread* execution and all but one distributed executions. The matrix corresponding to that operation was the smallest matrix and one of the least sparse ones (66 %), and consequently, the one that required more accesses to elements to complete the operation. Additionally, more data needed to be transferred as rows were also less sparse, negatively

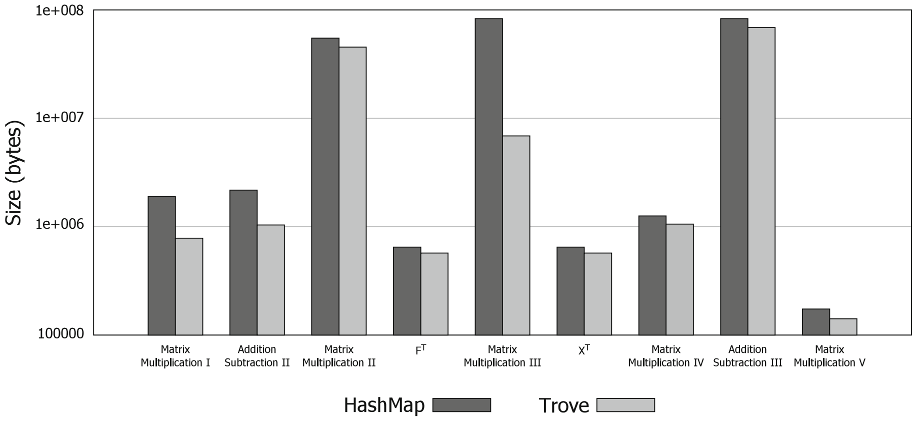

**Fig. 8** Matrix size comparison (*Digg*—logarithmic scale)

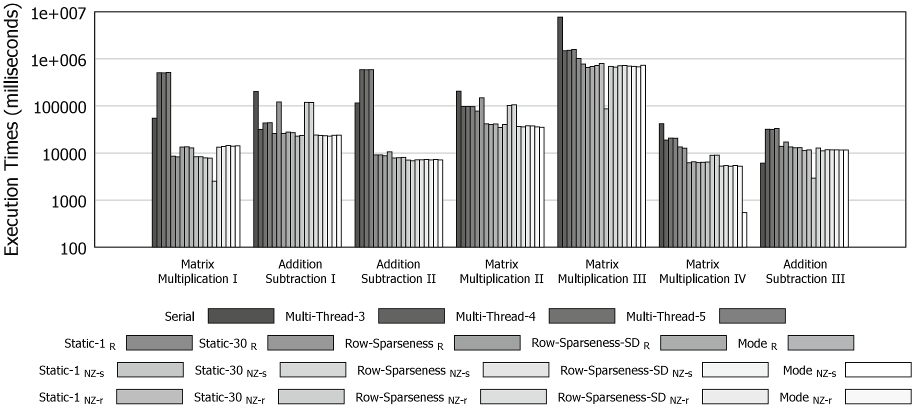

**Fig. 9** Computing time of all individual operations (*Digg*—logarithmic scale)

affecting the transfer and communication times of the distributed executions. Conversely, when considering the *Addition–Subtraction II* operation, the best results were achieved by the distributed executions. An analysis of the result matrix showed that the sparseness level was close to 99 %. These results confirmed the impact that the sparseness level has on the performance of operations as it affects network transfer and storage requirements. In summary, the distributed executions outperformed the *serial* execution by a 74 % in the worst-case (*Matrix-Multiplication I*) and a 94 % in the best-case (*Addition–Subtraction II*), and outperformed the best performing *Multi-Thread* execution by a 54 % in the most time-consuming operation (*Matrix-Multiplication III*) and a 98.79 % in the best-case (*Addition–Subtraction II*), when considering the best overall strategy for computing $B$ (*Row-Sparseness*$_{PF\text{-}R}$). When the computations are performed over small matrices such as the operations involved in $E$, *Multi-Thread* executions tended to perform better than the computer cluster ones. This could be also explained in terms of network transfer times, which negatively impacted on performance, as most part of the time was spent on transferring data instead of performing the actual computations. Although the difference between the execution times represented a 503 %, the absolute difference was 0.754 seconds, which represents less than a 1 % of the overall execution time. In consequence, the *PF* strategies can also be used with small matrices without affecting the overall execution time of the approach.

As regards the distributed executions, the worst overall results were obtained for the *Static-1*$_{PF\text{-}R}$ strategy, which was closely followed by *Static-1*$_{PF\text{-}NZ_r}$, stating the importance of considering the intrinsic characteristics of the operations to be performed, in conjunction with those of the matrices involved. Oppositely, the best overall results were obtained for both the *Row-Sparseness*$_{PF\text{-}R}$ and *Row-Sparseness*$_{PF\text{-}NZ_r}$ strategies, outperforming the other strategies by a 40 %. With respect to the individual operations, *Row-Sparseness*$_{PF\text{-}R}$ also achieved the best results for most of the operations with differences up to a 5 % in the most time-consuming operation (*Matrix-Multiplication III*), reinforcing the importance of considering the matrix characteristics when computing *PF*, particularly the mean row sparseness. Although all the strategies based on both *PF-R* and *PF-NZ* outperformed the *Serial* and *Multi-Thread* executions, the results obtained by computing the *PF-NZ* were slightly superior than the performance obtained by computing the *PF-R* for all the presented strategies, with differences up to a 4 % regarding the *Static-1* strategy. This could be explained in terms of the intrinsic characteristics of the data collection. As the *Digg* data collection presents a balanced distribution of features among posts, the resulting matrices also present a balanced distribution of non-zero elements among rows. In this context, computing the *PF-R* by considering the number of rows per task or computing the *PF-NZ* by considering the number of non-zeros per task achieved similar row distributions and thus similar number of individual operations to perform per task.

Furthermore, the differences between the results obtained by considering *PF-NZ*$_r$ or *PF-NZ*$_s$-based strategies were lower than a 8.29 %, representing, in average, a 3.86 % of the overall computing time of $B$. The maximum difference was obtained for the worst performing strategy (*Static-1*), whereas the minimum was obtained for the best performing strategy *Row-Sparseness*. Consequently, the difference between the results can be considered statistically insignificant, and thus, any strategy could be selected. Additionally, the results also highlighted the fact that the time spent computing the $\alpha$ value for the *Row-Sparseness, Row-Sparseness-SD*, and *Mode* strategies was insignificant, consuming, in average, less than a 0.25 % of the total time of each operation. Additionally, the time spent sorting the rows for the *PF-NZ*$_s$-based strategies was insignificant representing in average less than a 0.71 % of the total time of each operation. In conclusion, computing the extra information needed for the presented strategies did not affect the overall performance of the approach.

### 5.2 BlogCatalog dataset results

Table 6 shows the sparseness level of the result matrices of each of the evaluated operations and several auxiliary matrices for the *BlogCatalog* data collection. As it can be observed, this data collection has a higher sparseness level than the one of the *Digg* data collection, ranging between 99.62 and 99.99 %. However, due to the different sizes of both datasets, in average the *BlogCatalog* data collection matrices had more non-zero elements than those of the *Digg* data collection. The most dense matrix corresponded to the result of a multiplication, which is the most resource-demanding operation. Figure 10 compares the storage needs for both matrix implementations tested. As the figure shows, the *Trove*-based implementation required less storage space than the *HashMap* based, confirming the results obtained for the *Digg* data collection, as well as the fact that the reduction in storage space needs depended on the matrix sparseness level. In this case, the reductions in the storage needs ranged from a 13.12 % in the case of the *Matrix-Multiplication I* needed for computing the Laplacian to a 18 % in the case of the *Matrix-Multiplication III*, which is the most dense matrix. Considering

**Table 6** Matrices' size and sparseness per operation (*BlogCatalog*)

| Operation | Size | Sparseness (%) |
|---|---|---|
| *B* | | |
| Matrix-Multiplication I | 111,643 × 111,643 | 99.9959 |
| Addition-Subtraction II | 111,643 × 111,643 | 99.9963 |
| Matrix-Multiplication II | 189,585 × 111,643 | 99.9586 |
| $F^T$ | 111,643 × 189,585 | 99.9809 |
| Matrix-Multiplication III | 189,585 × 189,585 | 99.7885 |
| $X^T$ | 111,643 × 189,585 | 99.9809 |
| Matrix-Multiplication IV | 189,585 × 189,585 | 99.8683 |
| Addition-Subtraction III | 189,585 × 189,585 | 99.7434 |
| *E* | | |
| Matrix-Multiplication V | 11,701 × 189,585 | 99.893 |

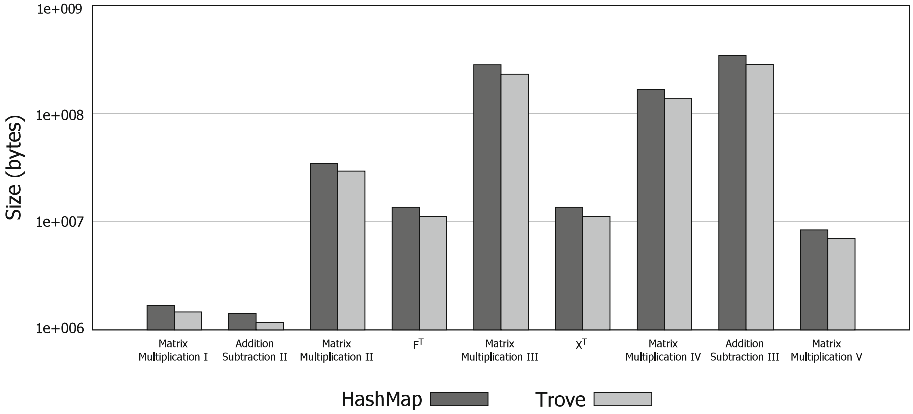

**Fig. 10** Matrix size comparison (*BlogCatalog*—logarithmic scale)

the results obtained for the *Digg* data collection and the differences in storage needs obtained for the *BlogCatalog* data collection, only the *Trove*-based implementation was used in the *BlogCatalog* executions.

Figures 11 and 12 compare the results obtained for both $B$ and $E$ matrices for the *Serial*, *Multi-Thread*, *PF-R* and *PF-NZ*-based strategies using the *Trove*-based implementation. The results include the data transfer and communication times, and the computing time of the *PF*, when applicable. As the figures show, the strategies based on *PF-R* did not improve the overall results of the *Serial* and *Multi-Thread* executions, with the exception of the *Static-30*$_{PF\text{-}R}$ strategy that reduced the $B$ computing time by a 150 % regarding the *Serial* time and a 20 % regarding the best performing *Multi-Thread* execution. When considering the $E$ computing time, the results were similar, as the distributed executions did not improve the *Serial* and *Multi-Thread* executing times. Figure 13 presents the results of each of the individual operations involved in computing $B$. As it is shown, the distributed executions achieved the best computing times for the addition–subtraction operations, whereas for the multiplication operations, the *Multi-Thread* executions achieved the best computing times. Additionally,

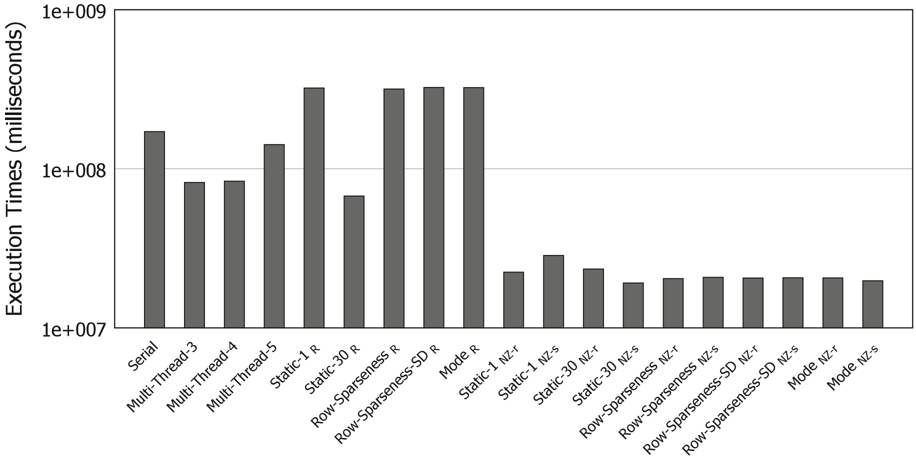

**Fig. 11** *B* matrix overall computing time (*BlogCatalog*—logarithmic scale)

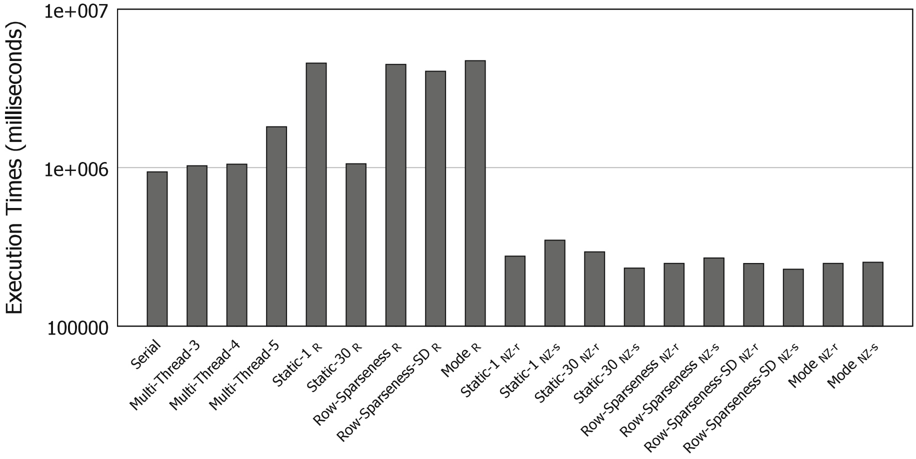

**Fig. 12** *E* matrix overall computing time (*BlogCatalog*–logarithmic scale)

the *PF-R*-based executions increased the computing times up to a 295 % regarding the best *Multi-Thread* strategy. This could be explained in terms of the intrinsic characteristics of the data collection. As the *BlogCatalog* data collection presents an unbalanced distribution of blog features, the resulting matrices also have an unbalanced distribution of non-zero elements among rows. In this context, computing the *PF-R* without considering their individual sparseness level, resulted in tasks with an unbalanced number of individual operations to perform, which in the case of multiplications depended on the number of non-zero elements of the left matrix of each task. In this context, tasks would have different resource needs, resulting in underutilised cluster resources, and hence negatively impacting the real parallelism of tasks. On the contrary, as the addition-subtraction operations do not strictly depend on the sparseness level of the individual matrices, the unbalanced distribution of non-zero elements on tasks did not affect the performance of the distributed executions.

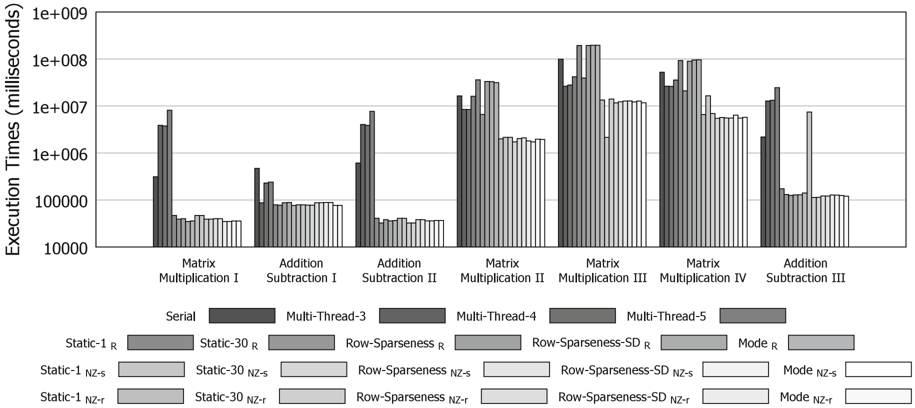

**Fig. 13** Computing time of all individual operations (*BlogCatalog*—logarithmic scale)

The performance of the ten strategies based on computing the *PF-NZ* and executing on the computer cluster was superior to the performance of the *Serial* and *Multi-Thread* executions. The best overall results when computing $B$ were obtained by the *Mode*$_{PF\text{-}NZ_s}$ strategy, which outperformed the best performing *Serial*, *Multi-Thread* and *PF-R*-based results by a 88.47 % (speed-up 8.67), 75.88 % (speed-up 4.14) and 70.67 % (speed-up 3.40), respectively. Those results were followed by the *Row-Sparseness-SD*$_{PF\text{-}NZ_r}$ strategy, with improvements of 88.10 % (speed-up 8.33), 75.12 % (speed-up 3.98) and 69.74 % (speed-up 3.27), respectively. As regards $E$, the performance of the *PF-NZ*-based strategies was superior to the performance of the *Serial* and *Multi-Thread* executions. In this case, the best performing strategy was *Row-Sparseness-SD*$_{PF\text{-}NZ_s}$, which outperformed the *Serial, Multi-Thread*, and *PF-R*-based results by a 75.72 % (speed-up 3.79), 77.75 % (speed-up 4.14) and 78.38 % (speed-up 4.26), respectively. Considering the execution times for all the operations involved in computing $B$, it can be observed that the *PF-NZ*-based strategies were able to effectively reduce the computing time of the most resource-demanding operations, i.e. the multiplications. In most cases, the best results were obtained by the *PF-NZ*-based strategies, outperforming the *Serial* and *Multi-Thread* results by at least a 63 and 37.29 %, respectively.

The differences between the results obtained by considering the *PF-NZ*$_r$ or *PF-NZ*$_s$-based strategies were higher than those obtained for the *Digg* data collection, representing in average a 10.54 % of the overall computing time of $B$. As in the previous case, the biggest difference was obtained for the worst performing strategy (27.32 %), whereas the lowest difference was obtained for the best performing strategy (0.46 %). As sorting the rows before creating the tasks did not lead to significant improvements in the overall computing times when considering the best performing strategy, it can be stated that it is not strictly necessary to sort the rows before creating the tasks. Briefly, all of the *PF-NZ* strategies outperformed all of the *Serial, Multi-Thread* and *PF-R*-based strategies. These results reinforce the need of leveraging the intrinsic characteristics of the involved matrices as well as the distribution of the non-zero elements among rows in order to balance the workload of the tasks to be executed and thus increase the parallelism. Additionally, the results also highlight the fact that the time spent computing the $\alpha$ value for the *Row-Sparseness, Row-Sparseness-SD* and *Mode* strategies was insignificant, representing in average less than a 0.017 % of the total time of each operation. The time spent sorting the rows for the *PF-NZ*$_s$-based strategies was insignificant, representing in average less than a 0.066 % of the total time of each operation.

In conclusion, computing the extra information needed for the presented strategies did not affect the overall performance of the approach.

### 5.3 Comparison with linear algebra software libraries

Table 7 shows a comparison of several general and specialised purpose linear algebra software libraries. The table summarises several characteristics of the libraries such as their accepted data types, matrix implementation, available operations, and ease of use, among others. All the selected libraries are implemented in Java; however, some of them (LAPACK and JBLAS) are ports of libraries that were originally implemented in other languages, such as C++ and Fortran. Interestingly, most of the libraries provide implementations for dense matrices, which may not be suitable for problems that involve sparse matrices. The benchmarks available only test performance for low-dimensional matrices (the number of rows range between 100 and 3000), which are not sufficient to test the scalability of the libraries.

The performance of the mentioned software libraries was compared against the proposed approach for the same arithmetic operations. The experimental evaluation considered the smallest dataset, i.e. the *Digg* dataset. All the executions were performed on the same computer where the *Serial* and *Multi-Thread* strategies were executed. Experimental evaluation showed that several of these libraries had higher resource requirements than the resources available and in turn higher requirements than the approach presented in this work. In fact, the experimental evaluations of JAMA, COLT (dense structure), PCOLT (dense structure), oj!Algorithms, JAMPACK, and JBLAS could not be performed as they required more than the available resources to create the matrices.

Figure 14 presents the $B$ matrix overall execution times for the linear algebra libraries, as well as for the *Serial*, best *Multi-Thread* and the best *PF* strategy presented in this study. For both PCOLT and La4J-CRS, the figure shows the execution time before the library required more than the available resources. As it can be observed in the figure, all the tested software libraries performed poorly when compared to the *PF* strategies presented in this work. For example, PCOLT required more than 18 days to only complete part of the first operation, whereas the best *PF* strategy required 12.95 minutes to compute $B$. The best software library results were obtained for COLT, which required 5.69 days to compute $B$, increasing the best *PF* results more than 630 times. The poor performance of PCOLT on sparse matrix computations can be traced to the unnecessary synchronisation of operations. Even though synchronisation is not needed for performing several operations on sparse matrices, both getters and setters methods are synchronised. As a result, the performance of sparse operations is thoroughly slower than their dense equivalent. Similarly, COLT sparse executions are slower than their dense counterpart. In summary, although the libraries claim to be optimised, simple, user-friendly, and efficient, as the experimental evaluation showed, they are not suitable neither for high-dimensional matrices nor for high-performance applications.

### 5.4 Cost analysis

Predicting resource consumption of a software is crucial in parallel or distributed systems [48]. In this context, the main resources of interest to estimate are bounds for synchronisation costs, and execution and network transfer times. One commonly used model for estimating costs is the Bulk Synchronous Parallel model (BSP) [49]. Unlike simpler models, BSP recognises that synchronisation costs cannot be neglected and that sharing data involves communication times, which depend on the infrastructure of the communication network, and the parallel and distributed processors under analysis. Interestingly, even when the model does not consider

**Table 7** Comparison of linear algebra libraries

| Library-release year | Data type | Matrix type | Available operations | User friendliness | Execution type | Additional characteristics |
|---|---|---|---|---|---|---|
| la4j (Linear algebra for Java) [a]-2014 | Double | Dense and sparse, 1- and 2-dimensional real matrices | • Standard matrix operations<br>• Matrix decompositions<br>• Linear system solvings | Methods have meaningful names | Serial. | Operations are delegated to the matrices. Sparse matrices can be created to be organised by rows or columns. Matrices are immutable and serialisable |
| oj! Algorithms [b]-2013 | Any number or BigDecimal | Dense, real and complex 1- and 2-dimensional matrices | • Standard matrix operations<br>• Matrix decompositions<br>• Financial and statistics tools | Matrices cannot be easily created as they do not have constructors Factory builder methods must be used. Most of the methods have meaningful names. Mutable and immutable matrices are initialised differently | Multi-Thread | Operations are delegated to the matrices. Some types of matrices only accept Double elements. There are several hardcoded hardware profiles. Users must set the hardware settings manually for optimum performance if their hardware is not included on the hardcoded profiles. Matrices are mutable and non serialisable |

**Table 7** continued

| Library-release year | Data type | Matrix type | Available operations | User friendliness | Execution type | Additional characteristics |
|---|---|---|---|---|---|---|
| JAMA (Java Matrix Package) [c]-2012 | Double | Dense real 2-dimensional matrices | • Standard matrix operations<br>• Matrix decompositions | The operations have non-ambiguous and meaningful names | Serial | In some cases, operations are not required to create a result matrix, which can help reduce the storage space consumption. Matrices are mutable and serialisable |
| PCOLT [d]-2010 | Double, float, long and integer | Dense and sparse, 1-, 2- and 3-dimensional real and complex matrices | • All original COLT capabilities<br>• Additional solvers and pre-conditioners | Same as COLT | Multi-Thread execution adapted to the number of available processors | The set of operation available for sparse matrices differs from the set of operation available for dense matrices. Matrices are mutable and non serialisable |

Table 7 continued

| Library-release year | Data type | Matrix type | Available operations | User friendliness | Execution type | Additional characteristics |
|---|---|---|---|---|---|---|
| JBLAS [e]-2010 | Double and float | Dense, real and complex 2-dimensional matrices | • Standard matrix operations<br>• Matrix decompositions<br>• Element-wise logical operations | Most of the methods have meaningful names | Serial | Operations are delegated to the matrices. Matrices are mutable and serialisable |
| COLT [f]-2004 | Double | Dense and sparse 1-, 2- and 3-dimensional real matrices | • Standard matrix operations<br>• Matrix decompositions<br>• Basic and advanced mathematics<br>• Basic and advanced statistics | Most of the standard matrix operations are not directly implemented. The user must defined functions for each of the operations to perform | Serial | Operations are delegated to the matrices. The implementation of multiplication does not vary for sparse and dense matrices. Matrices are mutable and non serialisable |

**Table 7** continued

| Library-release year | Data type | Matrix type | Available operations | User friendliness | Execution type | Additional characteristics |
|---|---|---|---|---|---|---|
| JAMPACK (JAva Matrix PACKage) [g]-1999 | Double. Real and imaginary values must be wrapped in a special type of object | Dense complex 2-dimensional matrices. It also provides diagonal matrices | ● Standard matrix operations adapted for complex matrices<br>● Matrix decompositions | Each operation is implemented in a different class as static methods and throws exceptions. Before creating matrices the user must set their base index. Although the class names are meaningful, their operations do not have meaningful names | Serial | As it only provides representation for complex matrices, each matrix requires the double of the storage space required for a real matrix. The different types of matrices cannot be used interchangeably. Matrices are immutable and non serialisable |

[a] http://la4j.org/
[b] http://ojalgo.org
[c] http://math.nist.gov/javanumerics/jama/
[d] https://sites.google.com/site/piotrwendykier/software/parallelcolt
[e] http://mikiobraun.github.io/jblas/
[f] https://acs.lbl.gov/ACSSoftware/colt/api/
[g] ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html

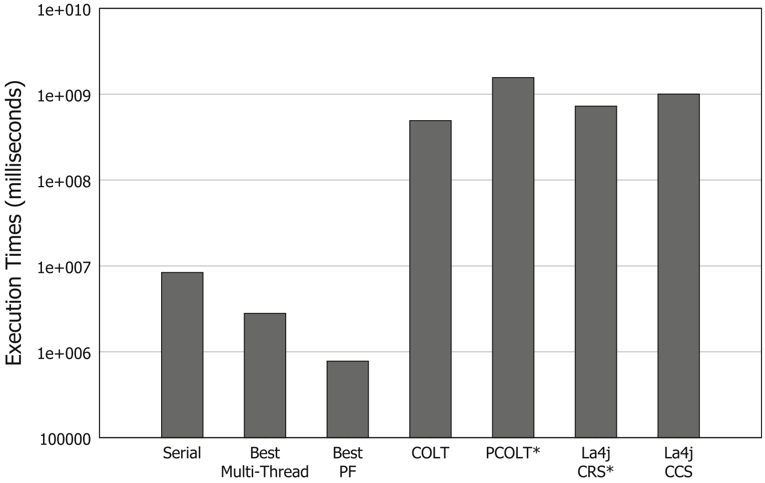

**Fig. 14** *B* matrix overall computing time–linear algebra libraries comparison (logarithmic scale). *Asterisk* (*) indicates incomplete execution

the processor internals, the memory hierarchy of specific communication patterns, it has been proved useful for predicting the true performance of applications [5,48].

BSP models computations as a series of super-steps, representing each of the tasks to be distributed for execution. Then, the cost of each super-step comprises three aspects. First, the independent concurrent computation that occurs in each processor using only local values. Second, a communication step in which processors exchange data. Third, a synchronisation barrier where all processes have to wait until all other processes have finished their communication actions. Equation (9) presents the formal definition of the three aspects of the cost model, where $S$ represents the number of super-steps to perform, $w_s$ is the number of flops in the super-step $s$, $h_s$ is the number of messages exchanges by the super-step $s$, $l$ is the cost of the barrier of synchronisation (including times of synchronisation start-up and checking whether the tasks have finished), and $g$ represents the ability of the communication network for delivering data subjected to the used protocols, the buffer management strategy, and the routing protocols. Both $l$ and $g$ are empirically determined and often expressed as a function of the number of processors $p$.

$$T = \sum_{s=1}^{S} w_s + g * \sum_{s=1}^{S} h_s + S * l \tag{9}$$

Note that as the concurrent computation and communication times are considered separately, the model does not contemplate the possibility of transmitting data and performing computations at the same time. Additionally, as it can be inferred, the cost of computing the super-steps is dominated by the worst-case cost of the local computations. In this regard, the BSP model is only suitable for assessing the cost of applications in which the granularities of the local computations of the super-steps are similar, as in the presented approach. Equation (10) presents an instantiation of the model expressed in Eq. (9) for the approach presented in this work. The equation defines the cost of computing a matrix multiplication between matrices of size $m \times n$ and $n \times c$ using a *HashMap*-based sparse representation.

$$T = \sum_{pf=1}^{PF} \Theta\left(\frac{m}{PF} * n * c\right) + g * \left( n * c * \frac{sparseness}{load\ factor} + \sum_{pf=1}^{PF} \frac{m}{PF} * n * \frac{sparseness}{load\ factor} + \sum_{pf=1}^{PF} \frac{m}{PF} * c * \frac{sparseness}{load\ factor} \right) + PF * l \tag{10}$$

In the equation, $PF$ represents the number of super-steps (i.e. tasks) to be executed. The cost of performing the multiplication associated to each task is $\Theta\left(\frac{m}{PF} * n * c\right)$, where $\frac{m}{PF}$ is the number of rows from the left matrix for which the task performs the computation. Note that the computational cost includes not only the cost of actually performing the operation, but also a time $\alpha$ of accessing to the corresponding elements. Moreover, the cost considers that $\frac{m}{PF} * n * c$ element-to-element multiplications are performed. However, in sparse matrices it can be stated that the number of operations to be performed is equal or lower to the number of non-zero elements in the left matrix of the multiplication; thus, the computational cost can be expressed as $\Theta\,(non\text{-}zero * c)$, assuming that $non\text{-}zero \ll \frac{m}{PF} * n$ for highly sparse matrices.

As expressed, the network transfer times includes three costs. First, the cost of transmitting the right matrix of the multiplication used and shared by every task $\left(n * c * \frac{sparseness}{load\ factor}\right)$. The cost is assessed for a sparse representation of the matrix that considers its $sparseness$ and the $load\ factor$ of the supporting matrix implementation. It is assumed that the network supports broadcasting or multicasting. On the contrary, the cost would be the addition of the cost over the processors: $\sum_{p=1}^{P} n * c * \frac{sparseness}{load\ factor}$. Second, the cost of transferring the rows assigned to each task to the corresponding processor $\left(\sum_{pf=1}^{PF} \frac{m}{PF} * n * \frac{sparseness}{load\ factor}\right)$. Although there might be overhead in transmitting the matrix in parts, instead of complete, such overhead can be disregarded since it represents a small proportion of the total cost of transferring the matrix. Hence, the cost expression can be simplified to $m * n * \frac{sparseness}{load\ factor}$. The same situation arises regarding the cost of transmitting back the task's results $\left(\sum_{pf=1}^{PF} \frac{m}{PF} * c * \frac{sparseness}{load\ factor}\right)$, which can be simplified to $m * c * \frac{sparseness}{load\ factor}$.

Finally, it can be inferred that in order to be convenient to distribute the operations in the computer cluster, the cost given by Eq. (10), should be lower than the cost of performing the operations in a centralised manner $n * m * c$. In other words, $T < \Theta\,(m * n * c)$.

Considering the defined cost model, there are several experimental remarks. First, the cost of transferring all the rows corresponding to each individual task is roughly the same as transferring the matrix as a whole once. Although there is a penalty or overhead for transferring small data packages through a communication network, such penalty can be disregarded due to the fact that represents only a small percentage of the total cost. For example, in a Ethernet Gigabit network (like the one used in the experimental evaluation), the overhead of data transmitted is lower to the 6 % of the total transferring cost. Second, matrices are sparse, i.e. only non-zero elements are stored, which decreases the size of the matrices to share among the processors in the computer cluster. For example, consider the matrix resulting from *Matrix Multiplication I* for the *Digg* dataset, which has a sparseness of 99.55 % (Table 5). In such case, instead of the almost 2 billion elements that a dense representation would store and need to transfer, with a sparse representation, only 8 million elements would need to be transferred. Considering the Gigabit network, that only would require 0.625 seconds to transfer, which represents the 0.38 % of the average time of performing the matrix multiplication. Moreover, frameworks such as JPPF support data compression, which can further decrease the network transfer times. Finally, experimental evaluation showed a case in which the centralised execution was faster than the distributed one, implying that the acceleration obtained in the computation did not compensate for the network transfer times. However, the difference between such executions was lower than a second, from which it can be stated that the overall execution time of the approach is not actually affected. Moreover, the usage of better and faster communication networks (such as an Infiniband) would decrease the network transfer costs, thus increasing the advantages of the presented technique over centralised approaches even for small matrices.

# 6 Related works

Linear algebra has played a central role in computing since the appearance of the first digital computers. Nowadays, arithmetic operations on matrices are commonplace in scientific computing areas, including signal and image processing, document retrieval, computational fluid dynamics, and feature selection. The increasing usage of those operations limits the performance of applications due to their high computational complexity. Also, several complexities arise from the interactions between the computer processors and the data involved in the computations [12]. Although computers have fast performing processors, memory accesses continue to be relatively slow. In consequence, performance is usually limited by the need to share data between processors and memory. The problem is accentuated if all the elements of the involved matrices need to be repeatedly traversed. As a result, computer hardware have strongly influenced the development of linear algebra algorithms.

## 6.1 Advances in dense matrices

Extensive research was carried out to address the sequential implementation of matrix operations. Gustavson et al. [16] proposed to optimise dense matrix representations to be used with dense matrix libraries by maintaining data locality at every level of the memory hierarchy. With the advent of multi-core architectures, research was focused on developing algorithms to transparently scale the parallelism and thus leverage the increasing number of processor cores. The first attempts to leverage multi-core architectures intended to develop compilers that could automatically transform sequential implementations into parallel ones. However, those compilers proved to be efficient only on a restricted set of problems. For example, only applications with loops could be automatically parallelised with compilers. Furthermore, for a loop to be automatically parallelised, it has to comply with certain criteria. First, the number of iterations of the loop must be known in advance. Second, there cannot be jumps into or out of the loops. Third, all loop iterations must be independent, i.e. correct results cannot depend on the order in which each iteration is executed. Fourth, the improvement in the parallelised loop must overcome the overhead of starting and synchronising parallel tasks. Also, users must add specific instructions for the compiler in order to explicitly indicate which loops should be parallelised. In consequence, most of the applications that were not explicitly implemented to be executed in parallel had to be re-implemented to take advantage of multi-core processors, implying considerable work. Furthermore, most general purpose libraries that can be executed in parallel only provide support for multi-thread processes and thus cannot be easily ported to computer clusters [28]. In this context, the parallel processing of matrix operations in distributed memory architectures arises as an important issue to study. Particularly, operating with dense matrices has been the subject of intensive research [6,8–10,12,23,38,59], among others.

Poulson et al. [38] proposed a distributed memory dense matrix library implemented in C++ to perform standard matrix arithmetic operations, several matrix decompositions, and the Herminian matrix. Buttari et al. [8] also presented matrix decomposition algorithms in which each operation was represented as a sequence of small tasks that operate on blocks of data. The algorithms are based on dividing the LAPACK (Linear Algebra PACKage)[5] and BLAS (Basic Linear Algebra Subprograms)[6] implementations into small sequential tasks, and introducing modifications regarding the access of shared data in order to improve performance. Then, the tasks are dynamically scheduled for execution based on the dependencies among them, and on the availability of computational resources. As a result, tasks could be executed out of order, hiding the presence of the original sequential tasks in the factorisation process. Experimental results suggested that fine granularity and asynchronous execution models are desirable characteristics that can help to achieve high performance on multi-core processors, providing considerable benefits over traditional fork-join approaches.

Unlike the previous approaches, Chan et al. [9,10] and Van Zee et al. [59] presented runtime systems to parallelise and schedule sequential code for executing in multi-core processors. The approaches aimed at studying how to hide the parallelism from the software library and, at the same time, achieve high performance. In [9,10], the system performed a dynamic loop unrolling by means of the Tomasulo algorithm [47] of all the tasks that executed on individual sub-matrix blocks. It also implemented different heuristics for scheduling operations, which were independent of the code that enqueued them. The approach detected dependencies between the tasks that read and write blocks to determine the extent to which tasks could be parallelised. However, dependencies can be difficult to detect as different sub-matrices may reference the same block across multiple tasks. According to the authors, their approach improved the performance of any other algorithm implemented with traditional multi-threaded libraries, such as vendor modifications of BLAS. Similarly, Kurzak et al. [23] manually optimised computational micro-kernels for the multi-core Cell Broadband Architecture architecture to exploit instruction-level parallelism. The authors based their work on the loop unrolling technique. Experimental evaluation showed improvements in performance over the BLAS library. However, the authors did not assess whether similar results could be obtained by an automatic optimisation of code. Additionally, the authors claimed that their matrix multiplication implementation was optimal as neither its performance can be further improved nor the code size be decreased.

Related to the organisation of matrices in memory, Elmroth et al. [12] introduced data structures and recursive algorithms to be applied in super-scalar kernels processors with tiered memory structures. The data structure aimed at recursively storing block-partitioned dense matrices across several levels of the memory hierarchy. The algorithms included linear systems and standard matrix equations solvers. The implementation was based on Fortran and used techniques of loop unrolling, and register and cache blocking. Although the algorithms were optimised for a generic super-scalar architecture, experimental evaluation showed the robustness and performance improvements on several platforms, with respect to parallel implementations of BLAS and LAPACK. The authors stated that additional performance improvements could be obtained by optimising the implementations to specific super-scalar architectures. Finally, Bosilca et al. [6] focused on designing fault-tolerant techniques to be applied on matrix multiplications, arguing that standard check-pointing techniques could decrease the overall performance.

[5] http://www.netlib.org/lapack/.

[6] http://www.netlib.org/blas/.

## 6.2 Advances in sparse matrices

In text analysis, as in collaborative filtering and document clustering, among other domains, matrices are sparse. Therefore, only the non-zero elements need to be stored, highlighting the importance of developing memory-efficient representations and algorithms. Notice that the performance of sparse matrix operations tends to be lower than their dense matrix equivalent due to the overhead of accessing the index information of elements in the matrix structure and the irregularity of the memory accesses [20]. Additionally, algorithms that are efficient for dense representations are not suitable for sparse representations as often expend a large fraction of their computational resources performing unnecessary operations that involve zeros [17]. On the other hand, the performance of several matrix operations can be significantly improved only when the involved matrices are sparse [24,53], for example the Cholesky factorisation.

Several approaches have been developed for representing sparse matrices aiming at improving the efficiency of memory usage and the computation of arithmetic operations. In [60], the focus was on reducing the number of accesses for particular matrix operations and defining a more natural mapping between the indices in the physical value matrix and the logical sparse coefficient matrix. Im et al. [20] proposed an approach to customise the matrix representation according to the specific sparseness characteristics of matrices and the target machine by performing register- and cache-level optimisations. For example, the approach could add explicit zeros to improve the memory system behaviour. Results showed that the right choice of optimisations is essential to improve performance as each optimisation technique could benefit only a subset of matrices and negatively affect others. Buluç et al. [7] introduced a storage format for sparse matrices based on dividing a matrix into blocks. As the storage format does not favour rows over columns or vice versa, it allows to efficiently perform parallel operations involving both a matrix and its transpose. Additionally, the authors proposed a parallel algorithm for computing the multiplication of a sparse matrix by a dense vector. The algorithm uses the non-zero element distribution among rows in order to only parallelise the multiplication of dense rows, i.e. rows in which the number of non-zeros is above a pre-defined threshold, whereas multiplication of sparse rows is performed sequentially. Experimental evaluation showed that the storage format performed best on matrices in which the non-zeros were located near the main diagonal, i.e. banded matrices. In this type of matrices, memory accesses tend to be regular, which favours the cache reuse and the automatic pre-fetching. Finally, Kourtis et al. [22] proposed two matrix representations aimed at compressing index and numerical values, respectively. Experimental evaluation showed the drawbacks of both methods. The compression based on index values did not obtain significant speed-ups when compared to traditional sparse row compression techniques. On the other hand, the compression based on numerical values proved only to be useful for matrices with a considerable proportion of repeated values.

In view of the performance decrease in algorithms designed for dense matrices when applied to sparse ones, several authors focused on designing algorithms to perform arithmetic operations specifically on sparse matrices [28,50,58]. Furthermore, MATLAB had to be extended to include sparse structures and operations [13]. Liu et al. [28] proposed an approach for scaling up the non-negative matrix factorisation independently of the internal structure of matrices. The approach was based on partitioning the data, arranging the computations to maximise the data locality and parallelism, and the MapReduce paradigm. Experimental results showed that the approach allowed to process matrices with millions of rows and billions of non-zero elements within tens of hours. Yuster and Zwick [58] combined fast rectangular matrix multiplication algorithms and combinatorics to reduce the number of operations needed for performing sparse matrix multiplications. Moreover, the authors designed improved algorithms for multiplying more than two sparse matrices. However, their experimental results only had theoretical value as no factual experimental evaluation was provided. Zhou et al. [64] proposed an optimised iterative decomposition of the discrete Fourier transform into a set of sparse matrices. The decomposition was based on integrating three orthogonal transforms to reduce the number of both matrix multiplications and total operations. The proposed decomposition was shown to effectively reduce the computational complexity and improve the performance of other state-of-art techniques. Yu et al. [56] proposed a novel parallel matrix factorisation approach to overcome the scalability issues posed by big-scale sparse matrices when using the Alternating Least Squares (ALS) and Stochastic Gradient Descent (SGD) methods. The novel approach was able to decrease the time complexity per iteration of ALS and to achieve a faster convergence than SGD. Finally, Vuduc et al. [50] presented an approach to automatically customise algorithm implementations to specific matrices and super-scalar machines in run-time.

### 6.3 Leveraging GPU technology

In recent years, one of the strategies used to increase the computing power of computers has been the usage of Graphics Processing Units (GPUs) in addition to CPUs. Generally, those applications which can exploit data-level parallelism are able to attain good performance in a GPU environment [24]. Several works [3,4,11,36,65] have presented approaches for optimising matrix multiplications through the usage of GPUs. Both [11,36] focused on the overlapping between computation and data communication in order to minimise the communication and transfer times. Dang and Schmidt [11] presented an approach to efficiently represent sparse matrices and a CUDA (Compute Unified Device Architecture) implementation of matrix multiplications. The matrix representation was based on partitioning the input matrix according to its sparseness patterns. The matrix multiplication implementation considered a bidirectional data transfer. Oyarzun et al. [36] proposed an approach for matrix multiplication in the context of the conjugate gradient method in multi-GPU environments. In particular, the proposed solver was tested for the Poisson equation and reported improvements up to a 200 % versus the CPU-only solver. Zhang et al. [61] developed a parallel algorithm to efficiently compute the projection matrix for solving the non-negative sparse latent semantic space model using CUDA. Additionally, the authors presented a data partitioning scheme to reduce the data transference to the GPU and thus cope with the limited memory capacity of GPUs. Bell and Garland [3,4] compared the performance of several matrix representations on an implementation of the matrix multiplication that was tailored to the data access pattern of a particular GPU architecture to efficiently use the memory bandwidth. Their results remarked the importance of choosing an adequate matrix representation according to its sparseness pattern. Finally, Zuo et al. [65] proposed an approach to accelerate matrix multiplications by performing parallel computations on heterogeneous hardware. The approach aimed at implementing a cross-platform algorithm to take advantage of both multi-core CPUs and GPUs by means of OpenCL.

Although most of the experimental evaluations of GPU-based approaches were performed on matrices with at most 1,000,000 rows and columns, their sparseness levels were higher than a 97 %, thereby hindering the generalisation of results for smaller but less sparse matrices. Notice that smaller matrices with lower sparseness levels could require more resources than bigger matrices with higher sparseness levels. In conclusion, the experimental evidence is not sufficient to convincingly test the scalability of GPU-based approaches and effectively address the needs of document retrieval or feature selection problems.

In summary, GPU devices offer a considerable potential for performance and efficiency in large-scale applications of scientific computing. However, obtaining the desired performance from GPU devices is not a trivial task as it usually requires significant changes in the algorithms and their implementations [36]. In particular, when considering sparse operations, fine-grained parallelism and sufficient regularity on the execution paths and memory access patterns are required to leverage the potential of GPU devices [4]. Furthermore, GPUs have limitations regarding their fixed memory capacity, maximum threads, memory access, and performance of any single device [3]. For example, the non-contiguous access to the GPU memory has a negative effect in the memory bandwidth efficiency and thus in the performance of memory-bound applications, specially when considering big data structures such as high-dimensional matrices [11]. Additionally, the distribution of tasks among GPUs presents some challenges as GPUs are unable to share memory space with CPUs or with each other [36].

## 7 Conclusions

This paper proposed a novel approach for distributing sparse matrix arithmetic operations on computer clusters. The goal of the approach is to speed up the processing of high-dimensional matrices, which would take days, or even weeks, on a single multiprocessor computer.

The approach is based on two aspects that distinguish each arithmetic operation: information sharing requirements and relevance of the matrices involved. As a result, the approach defines several strategies to determine how to partition the matrix operations to be processed by computing the *PF*, which relies on the intrinsic characteristics of the operations and their associated matrices. The *PF* indicates how to split the matrices into parallel tasks to be created by defining the workload (expressed as number of rows or number of non-zeros) to be assigned to each task, which indirectly determines the number of parallel tasks to be created. The strategies were evaluated for the most common arithmetic operations: Addition–subtraction, matrix multiplication, and the Laplacian, which combines the other two operations. The performance of the proposed strategies was evaluated considering the high-dimensional feature selection approach presented in [42] for two real-world datasets. Also, two alternative matrix implementations were analysed in terms of performance and computational resource requirements.

From the performed experimental evaluation, several conclusions can be drawn. First, the performance of the *Trove* implementation of matrices was superior to the *HashMap* one as *Trove* uses a more efficient internal structure requiring less network and storage resources. In terms of computing times, the distributed executions outperformed the *Serial* and *Multi-Thread* executions when high-dimensional matrices were involved. On the contrary, when the operations involved smaller matrices, the *Multi-Thread* executions tended to perform better than the distributed ones. This could account for the network transfer and communication times as most part of the time was spent on transferring data instead of performing the actual computations.

Regarding the proposed strategies, the worst overall results were obtained for the Static strategy as expected. Oppositely, the best overall results were obtained for those strategies that considered the intrinsic information of matrices such as *Row-Sparseness, Row-Sparseness-SD* and *Mode*, which also outperformed the other strategies for most of the individual operations. These results stated the importance of considering both the intrinsic characteristics of the operations to be performed and the characteristics of the matrices involved, in particular the mean *Row-Sparseness*. As regards the difference between using the *PF-R* or *PF-NZ*-based strategies, results showed their dependency on the individual characteristics of the data collections, such as the balancing of features among posts and the unbalance distribution of posts among classes. For unbalanced data collections, it was required to consider the number of individual operations to be performed in each task in order to balance the workload between tasks, whereas for balanced data collections, the results did not allow to determine the superiority of any of the workload balancing strategies. Furthermore, the *PF*-based strategies greatly outperformed the performance of several linear algebra software libraries, regarding not only the execution times, but also the required computational resources. Results also highlighted the fact that the time spent computing the *PF* and sorting the rows was insignificant, not affecting the overall performance of the strategies. In conclusion, the results confirm the feasibility and advantages of the proposed approach for efficiently performing arithmetic operations on high-dimensional matrices within reasonable times.

**Acknowledgments** This work has been partially funded by CONICET (Argentina) under Grant PIP No. 112-201201-00185.

## References

1. Aggarwal CC, Zhai C (2012) A survey of text classification algorithms. In: Aggarwal CC, Zhai C (eds) Mining text data. Springer, Berlin, pp 163–222
2. Alelyani S, Tang J, Liu H (2013) Feature selection for clustering: a review. In: Data clustering: algorithms and applications, pp 29–60
3. Bell N, Garland M (2008) Efficient sparse matrix–vector multiplication on cuda. NVIDIA technical report NVR-2008-004, NVIDIA Corporation
4. Bell N, Garland M (2009) Implementing sparse matrix–vector multiplication on throughput-oriented processors. In: Proceedings of the conference on high performance computing networking, storage and analysis (SC'09). ACM, New York, NY, USA, pp 18:1–18:11
5. Bisseling RH (2004) Parallel scientific computation: a structured approach using BSP and MPI. Oxford University Press, Oxford
6. Bosilca G, Delmas R, Dongarra J, Langou J (2009) Algorithmic based fault tolerance applied to high performance computing. J Parallel Distrib Comput 69(4):410–416
7. Buluç A, Fineman JT, Frigo M, Gilbert JR, Leiserson CE (2009) Parallel sparse matrix–vector and matrix-transpose–vector multiplication using compressed sparse blocks. In: Proceedings of the 21st symposium on parallelism in algorithms and architectures (SPAA'09). ACM, pp 233–244
8. Buttari A, Langou J, Kurzak J, Dongarra J (2009) A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput 35(1):38–53
9. Chan E, Quintana-Ortí ES, Quintana-Ortí G, Geijn RVD (2007) Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In: Proceedings of the 9th annual ACM symposium on parallel algorithms and architectures (SPAA'07). ACM, pp 116–125
10. Chan E, Zee FGV, Bientinesi P, Quintana-Ortí ES, Quintana-Ortí G, van de Geijn RA (2008) Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In: Chatterjee S, Scott ML (eds) Proceedings of the 13th ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP'08). ACM, pp 123–132
11. Dang HV, Schmidt B (2013) Cuda-enabled sparse matrix–vector multiplication on gpus using atomic operations. Parallel Comput 39(11):737–750
12. Elmroth E, Gustavson F, Jonsson I, Kågström B (2004) Recursive blocked algorithms and hybrid data structures for dense matrix library software. SIAM Rev 46(1):3–45
13. Gilbert JR, Moler C, Schreiber R (1992) Sparse matrices in MATLAB: design and implementation. SIAM J Matrix Anal Appl 13(1):333–356
14. Gu Q, Han J (2011) Towards feature selection in network. In: Macdonald C, Ounis I, Ruthven I (ed) Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM'11). ACM, pp 1175–1184

15. Gu Q, Li Z, Han J (2011) Generalized fisher score for feature selection. In: Proceedings of the 27th conference annual conference on uncertainty in artificial intelligence (UAI-11), pp 266–273. arxiv:1202.3725
16. Gustavson F, Henriksson A, Jonsson I, Kågström B, Ling P (1998) Recursive blocked data formats and BLAS's for dense linear algebra algorithms. In: Proceedings of the 4th international workshop on applied parallel computing. Large scale scientific and industrial problems. Springer, Berlin, pp 195–206
17. Heath L, Ribbens C, Pemmaraju S (2004) Processor-efficient sparse matrix–vector multiplication. Comput Math Appl 48(34):589–608
18. Hou C, Nie F, Yi D, Wu Y (2011) Feature selection via joint embedding learning and sparse regression. In: Walsh T (ed) Proceedings of the 22nd international joint conference on artificial intelligence (IJCAI). AAAI, pp 1324–1329
19. Hu X, Tang L, Tang J, Liu H (2013) Exploiting social relations for sentiment analysis in microblogging. In: Leonardi S, Panconesi A, Ferragina P, Gionis A (ed) Proceedings of the 6th ACM international conference on web search and data mining. ACM, pp 537–546
20. Im EJ, Yelick K, Vuduc R (2004) Sparsity optimization framework for sparse matrix kernels. Int J High Perform Comput Appl 18(1):135–158
21. Kannan R, Ishteva M, Park H (2014) Bounded matrix factorization for recommender system. Knowl Inf Syst 39(3):491–511
22. Kourtis K, Goumas GI, Koziris N (2008) Optimizing sparse matrix–vector multiplication using index and value compression. In: Ramírez A, Bilardi G, Gschwind M (ed) ACM international conference on computing frontiers. ACM, pp 87–96
23. Kurzak J, Alvaro W, Dongarra J (2009) Optimizing matrix multiplication for a short-vector simd architecture—cell processor. Parallel Comput 35(3):138–150
24. Lee A, Yau C, Giles MB, Doucet A, Holmes CC (2010) On the utility of graphics cards to perform massively parallel simulation with advanced monte carlo methods. J Comput Graph Stat 19(4):769–789
25. Li Y, Zhai C, Chen Y (2014) Exploiting rich user information for one-class collaborative filtering. Knowl Inf Syst 38(2):277–301
26. Li Z, Liu J, Yang Y, Zhou X, Lu H (2013) Clustering-guided sparse structural learning for unsupervised feature selection. IEEE Trans Knowl Data Eng 26(9):2138–2150
27. Lin YR, Sun J, Castro P, Konuru R, Sundaram H, Kelliher A (2009) Metafac: community discovery via relational hypergraph factorization. In: Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining (KDD'09), pp 527–536
28. Liu C, Chih Yang H, Fan J, He LW, Wang YM (2010) Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In: Rappa M, Jones P, Freire J, Chakrabarti S (ed) Proceedings of the 19th international conference on world wide web. ACM, pp 681–690
29. Liu H, He J, Rajan D, Camp J (2013) Outlier detection for training-based adaptive protocols. In: IEEE wireless communications and networking conference (WCNC), pp 333–338
30. Ma Z, Nie F, Yang Y, Uijlings JRR, Sebe N, Hauptmann AG (2012) Discriminating joint feature analysis for multimedia data understanding. IEEE Trans Multimed 14(6):1662–1672
31. Marsden PV, Friedkin NE (1993) Network studies of social influence. Sociol Methods Res 22(1):127–151
32. McPherson M, Smith-Lovin L, Cook JM (2001) Birds of a feather: Homophily in social networks. Annu Rev Sociol 27(1):415–444
33. Moreira JE, Midkiff SP, Gupta M, Artigas PV, Wu P, Almasi G (2001) The NINJA project. Commun ACM 44(10):102–109
34. Nesterov Y (2004) Introductory lectures on convex optimization: a basic course (applied optimization), 2nd edn. Springer, Berlin
35. Nie F, Huang H, Cai X, Ding CHQ (2010) Efficient and robust feature selection via joint l2, 1-norms minimization. In: Lafferty JD, Williams CKI, Shawe-Taylor J, Zemel RS, Culotta A (eds) Advances in neural information processing systems. Curran Associates Inc, Red Hook, pp 1813–1821
36. Oyarzun G, Borrell R, Gorobets A, Oliva A (2014) Mpi-cuda sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. Comput Fluids 92:244–252
37. Porter MF (1997) Readings in information retrieval. Morgan Kaufmann Publishers Inc., San Francisco (Chap: An algorithm for suffix stripping)
38. Poulson J, Marker B, van de Geijn RA, Hammond JR, Romero NA (2013) Elemental: a new framework for distributed memory dense matrix computations. ACM Trans Math Softw 39(2):13:1–13:24
39. Qi GJ, Aggarwal CC, Tian Q, Ji H, Huang TS (2012) Exploring context and content links in social media: a latent space method. IEEE Trans Pattern Anal Mach Intell 34(5):850–862
40. Shahrivari S, Sharifi M (2011) Task-oriented programming: a suitable programming model for multicore and distributed systems. In: Proceedings of the 10th international symposium on parallel and distributed computing (ISPDC'11), pp 139–144

41. Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R (2013) Java in the high performance computing arena: research, practice and experience. Sci Comput Program 78(5):425–444
42. Tang J, Liu H (2012a) Feature selection with linked data in social media. In: Proceedings of the 12th SIAM international conference on data mining (SIAM/Omnipress), pp 118–128
43. Tang J, Liu H (2012b) Unsupervised feature selection for linked social media data. In: Yang Q, Agarwal D, Pei J (ed) Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining (KDD'12). ACM, pp 904–912
44. Tang J, Wang X, Liu H (2011) Social media data integration for community detection. In: Postproceedings of MUSE/MSM 2011
45. Tang J, Wang X, Gao H, Hu X, Liu H (2012) Enriching short text representation in microblog for clustering. Front Comput Sci China 6(1):88–101
46. Tang J, Hu X, Gao H, Liu H (2013) Unsupervised feature selection for multi-view data in social media. In: Proceedings of the 13th SIAM international conference on data mining (SIAM), pp 270–278
47. Tomasulo RM (1967) An efficient algorithm for exploiting multiple arithmetic units. IBM J Res Dev 11(1):25–33
48. Trinder PW, Cole MI, Hammond K, Loidl H, Michaelson G (2013) Resource analyses for parallel and distributed coordination. Concurr Comput Pract Exp 25(3):309–348
49. Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111
50. Vuduc R, Demmel JW, Yelick KA (2005) OSKI: a library of automatically tuned sparse matrix kernels. J Phys Conf Ser 16(1):521
51. Wang Q, Li X (2014) Shrink image by feature matrix decomposition. Neurocomputing 140:162–171
52. Wang X, Tang L, Gao H, Liu H (2010) Discovering overlapping groups in social media. In: Proceedings of the 2010 IEEE International Conference on Data Mining (ICDM '10). IEEE Computer Society, Washington, DC, USA, pp 569–578
53. Whiley M, Wilson SP (2004) Parallel algorithms for Markov chain Monte Carlo methods in latent spatial Gaussian models. Stat Comput 14(3):171–179
54. Xu W, Liu X, Gong Y (2003) Document clustering based on non-negative matrix factorization. In: Proceedings of the 26th annual international ACM SIGIR conference on research and development in information retrieval (SIGIR'03). ACM, New York, NY, USA, pp 267–273
55. Yakubovich E, Zenkovich D (2001) Matrix approach to lagrangian fluid dynamics. J Fluid Mech 443:167–196
56. Yu HF, Hsieh CJ, Si S, Dhillon I (2014) Parallel matrix factorization for recommender systems. Knowl Inf Syst 41(3):793–819
57. Yu Y, Qiu RG (2014) Followee recommendation in microblog using matrix factorization model with structural regularization. Sci World J 2014:420841
58. Yuster R, Zwick U (2005) Fast sparse matrix multiplication. ACM Trans Algorithms 1:2–13
59. Van Zee FG, Chan E, van de Geijn RA, Quintana-Ortí ES, Quintana-Ortí G (2009) The libflame library for dense matrix computations. Comput Sci Eng 11(6):56–63
60. Zhang K, Wu B (2012) Parallel sparse matrix multiplication for preconditioning and ssta on a many-core architecture. In: Proceedings of the 7th international conference on networking, architecture, and storage, pp 59–68
61. Zhang Y, Yi D, Wei B, Zhuang Y (2014) A GPU-accelerated non-negative sparse latent semantic analysis algorithm for social tagging data. J Inf Sci 281(0):687–702 (Multimedia modeling)
62. Zhao Z, Wang L, Liu H (2010) Efficient spectral feature selection with minimum redundancy. In: Fox M, Poole D (eds) Association for the advancement of artificial intelligence (AAAI). AAAI Press, Menlo Park
63. Zhou Y, Wilkinson DM, Schreiber R, Pan R (2008) Large-scale parallel collaborative filtering for the netflix prize. In: Fleischer R, Xu J (ed) Proceedings of the 4th international conference on algorithmic aspects in information and management. Lecture notes in computer science, vol 5034. Springer, Berlin, pp 337–348
64. Zhou Y, Cao W, Liu L, Agaian S, Chen CP (2015) Fast Fourier transform using matrix decomposition. J Inf Sci 291:172–183
65. Zuo W, McNeil A, Wetter M, Lee ES (2014) Acceleration of the matrix multiplication of radiance three phase daylighting simulations with parallel computing on heterogeneous hardware of personal computer. J Build Perform Simul 7(2):152–163

**Antonela Tommasel** is a PhD candidate at the Universidad Nacional del Centro de la provincia de Buenos Aires (UNCPBA) and a member of ISISTAN Research Institute, Tandil, Argentina. She is also a teacher assistant at the same university. Her research interests include recommender systems, text mining, and social web. She obtained her system engineer degree (2012) at UNCPBA. Contact her at antonela.tommasel@isistan.unicen.edu.ar.

**Daniela Godoy** is a researcher at CONICET and a member of ISISTAN Research Institute, Tandil, Argentina. She is also a full-time professor in the Department of Computer Science at UNCPBA, Tandil, Argentina. She obtained her Master's degree in Systems Engineering (2001) and her PhD in Computer Science (2005) at the same university. Her research interests include intelligent agents, user profiling and text mining. Contact her at daniela.godoy@isistan.unicen.edu.ar.

**Alejandro Zunino** (http://www.exa.unicen.edu.ar/~azunino) received a PhD degree in Computer Science from The National University of the Center of the Buenos Aires Province (UNCPBA), in 2003, and his M.Sc. in Systems Engineering in 2000. He is a full Adjunct Professor at UNCPBA and member of the ISISTAN and the CONICET. He has published over 120 papers on Distributed Computing, Service-oriented Computing and Mobile Computing.

**Cristian Mateos** has a PhD in computer science from UNCPBA. He is a full-time teaching assistant at UNCPBA and a member of ISISTAN and CONICET. He is interested in parallel and distributed programming, grid middleware, and service-oriented computing. Contact him at www.exa.unicen.edu.ar/~cmateos or cristian.mateos@isistan.unicen.edu.ar.