Journal of Parallel and Distributed Computing 138 (2020) 211–221

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Designing an efficient parallel spectral clustering algorithm on multi-core processors in Julia

Zenan Huo [a], Gang Mei [a,*], Giampaolo Casolla [b], Fabio Giampaolo [c]

[a] School of Engineering and Technology, China University of Geosciences (Beijing), 100083, Beijing, China
[b] Department of Mathematics and Applications "R. Caccioppoli", University of Naples FEDERICO II, Italy
[c] Consorzio Interuniversitario Nazionale per l'Informatica (CINI), Italy

ARTICLE INFO

Article history:
Received 14 December 2019
Received in revised form 5 January 2020
Accepted 12 January 2020
Available online 20 January 2020

Keywords:
Clustering algorithm
Spectral clustering
Parallel algorithm
Multi-core processors
Julia language

ABSTRACT

Spectral clustering is widely used in data mining, machine learning and other fields. It can identify the arbitrary shape of a sample space and converge to the global optimal solution. Compared with the traditional k-means algorithm, the spectral clustering algorithm has stronger adaptability to data and better clustering results. However, the computation of the algorithm is quite expensive. In this paper, an efficient parallel spectral clustering algorithm on multi-core processors in the Julia language is proposed, and we refer to it as juPSC. The Julia language is a high-performance, open-source programming language. The juPSC is composed of three procedures: (1) calculating the affinity matrix, (2) calculating the eigenvectors, and (3) conducting k-means clustering. Procedures (1) and (3) are computed by the efficient parallel algorithm, and the COO format is used to compress the affinity matrix. Two groups of experiments are conducted to verify the accuracy and efficiency of the juPSC. Experimental results indicate that (1) the juPSC achieves speedups of approximately 14×∼ 18× on a 24-core CPU and that (2) the serial version of the juPSC is faster than the Python version of **scikit-learn**. Moreover, the structure and functions of the juPSC are designed considering modularity, which is convenient for combination and further optimization with other parallel computing platforms.

1. Introduction

In recent years, machine learning has made great progress and become the preferred method for developing practical software, such as computer vision, speech recognition, and natural language processing [37,45,50,55]. Machine learning mainly includes supervised learning and unsupervised learning. Clustering is the main content of unsupervised learning. Among many clustering algorithms, the spectral clustering algorithm has become the popular one [32,53]. Spectral clustering is a technology originating from graph theory [17,29] that uses the edge connecting them to identify the nodes in the graph and allows us to cluster non-graphic data.

Unsupervised clustering analysis algorithm can explore the internal group structure of data, which has been widely used in various data analysis occasions, including computer vision analysis, statistical analysis, image processing, medical information processing, biological science, social science, and psychology [19,44,51]. The basic principle of clustering analysis is to divide the data into different clusters. Members in the same cluster have similar characteristics, and members in different clusters have different characteristics. The main types of clustering algorithms include partitioning methods, hierarchical clustering, fuzzy clustering, density-based clustering, and model-based clustering [38]. The most widely used clustering algorithms are k-means [61], DBSCAN [39], ward hierarchical clustering [47], spectral clustering [53], birch algorithm [66], etc.

It has been proven that the spectral clustering algorithm is more effective than other traditional clustering algorithms in references [46,56], but in the process of spectral clustering computation, the affinity matrix between nodes needs to be constructed, and storage of the affinity matrix requires much memory. It also takes a long time to achieve the first k eigenvectors of the *Laplacian* matrix. Thus, the spectral clustering algorithm is difficult to apply in the large-scale data processing.

For the large-scale spectral clustering problem, we usually adopt approximate technology to solve the dense matrix and its operation. For example, the Nyström expansion method [30] avoids directly calculating the overall affinity matrix while ensuring the accuracy. Several methods are available for achieving the purpose of the sparse matrix [43]. In recent research,

Abbreviations: COO, Coordinate Format; CSC, Compressed Sparse Column Format; CPU, Central Processing Unit; FEM, Finite Element Method; GPU, Graphics Processing Unit; JIT, Just-in-time Compilation; MKL, Intel Math Kernel Library; MPM, Material Point Method

* Corresponding author.
E-mail address: gang.mei@cugb.edu.cn (G. Mei).

https://doi.org/10.1016/j.jpdc.2020.01.003
0743-7315/© 2020 Elsevier Inc. All rights reserved.

Deng et al. [18] proposed a landmark-based spectral clustering algorithm, which scales linearly with the problem size.

In addition to the improvement of the spectral clustering algorithm, many researchers have also focused on the parallel algorithm. Gou et al. [33] constructed a sparse spectral clustering framework based on the parallel computation of MATLAB. Jin et al. [40] combined spectral clustering with MapReduce and, through the evaluation of sparse matrix eigenvalues and the computation of distributed clustering, improved the clustering speed of the spectral clustering algorithm.

The existing spectral clustering algorithms are implemented by static programming languages, such as C/C++ or Fortran. Although there is a certain guarantee of the execution efficiency, high programming skill is required, and code maintenance is difficult, which will lead to more time spent on design and implementation. Advanced dynamic languages, such as Python and MATLAB, have good interactivity, and the code is easier to read. Researchers can concentrate on algorithm design rather than program debugging, but at the cost of computational efficiency.

The Julia language is a new programming language that successfully combines the high performance of static programming languages with the agility of dynamic programming language [16]. The Julia language enables programmers to implement algorithms naturally and intuitively by introducing easy-to-understand syntax. Julia type stability through specialization via multiple dispatch makes it easy to compile programs into efficient code. Julia is widely used in machine learning. There are many excellent packages of clustering algorithms on Julia Observer [1], such as `Clustering.jl` [2], `ScikitLearn.jl` [3], and `QuickShiftClustering.jl` [4]. The package of `Clustering.jl` not only implements a variety of clustering algorithms, but also provides many methods to evaluate the results of clustering algorithms or verify the correctness.

To combine the performance advantages of the Julia language and the characteristics of the parallel algorithm, we have designed and implemented an efficient parallel spectral clustering algorithm on multi-core processors in the Julia language. We refer to it as juPSC. To the best of the authors' knowledge, the juPSC is the first high-performance spectral clustering algorithm designed and implemented in the Julia language. Our contributions in this work can be summarized as follows:

(1) A Julia-based parallel algorithm of the spectral clustering is designed and implemented.

(2) The structure and function of the juPSC are designed considering modularity, and the code is clear and easy to understand, which is convenient for subsequent improvement.

The rest of this paper is organized as follows. Section 2 presents the background introduction to the spectral clustering algorithm and the Julia language. Section 3 introduces the design and implementation details of the parallel algorithm. Section 4 provides several experiments to validate the accuracy and evaluate the efficiency of the juPSC. Section 5 analyzes the performance, advantages, and scalability of the juPSC. Finally, Section 6 concludes this work.

2. Background

In this section, we will present a brief introduction to (1) the spectral clustering algorithm and (2) the Julia language.

2.1. Spectral clustering algorithm

The theoretical basis of the spectral clustering algorithm comes from graph theory, which aims to transform clustering into graph segmentation. Suppose that the data points in the sample data are the nodes V in a graph and that the data pairs in the sample data are set to have a certain similarity, which is expressed by the weight of edge E between the two nodes; thus, an undirected weighted graph $G = (V, E)$ is obtained. The optimal partition criterion based on graph theory is to make the similarity of the nodes in the final partition result be the maximum and the similarity of the nodes belonging to different subgraphs be the minimum.

In the spectral clustering algorithm, we construct an undirected graph based on the similarity between the data and construct the adjacency matrix according to the similarity between the nodes. We turn the problem into the optimal partitioning problem of graph G. The choice of partitioning criteria will directly affect the final clustering result. The common partition rules in graph theory are Minimum cut [54], Normalized cut (N-cut) [27], Ratio cut [34], Average cut [64], and Min–max cut [56]. We construct a new eigenspace using the eigenvectors corresponding to the first k eigenvalues of the *Laplacian* matrix and use traditional clustering algorithms, such as k-means in the new eigenspace. The details of the spectral clustering algorithm using N-cut are as follows.

Step 1. Defining graph notation

The given data corresponds to the nodes of the graph, and the edges between nodes are weighted so that the undirected weighted graph G:

$$G = (V, E)\ , \quad E = \left\{(i, j)\ , S_{i,j} > 0\right\} \subseteq V \times V \tag{1}$$

where $V = \{1, \ldots, n\}$ is the node-set and E is the edge-set.

Then the clustering problem is transformed into the optimal partition problem of graph G. Graph G can be divided into two disjoint sets A and B (i.e., $A \cup B = V$ and $A \cap B = \emptyset$):

$$cut\ (A, B) = \sum_{u \in A, v \in B} W_{u,v} \tag{2}$$

Step 2. Calculating the affinity matrix

According to the similarity between nodes, the spectral clustering algorithm divides the categories. In the construction of the similarity graph, the accurate relationship between the local neighborhood of nodes can reflect the real clustering structure. We need to build the initial affinity matrix S:

$$S_{i,j} = \left\|x_i - x_j\right\|^2 \tag{3}$$

where $S_{i,j}$ is the distance between x_i and x_j. Then, k-nearest neighbor graphs are used to reconstruct the affinity matrix into the adjacency matrix W:

$$W_{i,j} = W_{j,i} = \begin{cases} 0, \text{ if } x_i \notin kNN\left(x_j\right) \text{ and } x_j \notin kNN\left(x_i\right) \\ e^{\frac{-\left\|x_i - x_j\right\|^2}{2\sigma^2}}, \text{ if } x_i \in kNN\left(x_j\right) \text{ or } x_j \in kNN\left(x_i\right) \end{cases} \tag{4}$$

$$W_{i,j} = W_{j,i} = \begin{cases} 0, \text{ if } x_i \notin kNN\left(x_j\right) \text{ or } x_j \notin kNN\left(x_i\right) \\ e^{\frac{-\left\|x_i - x_j\right\|^2}{2\sigma^2}}, \text{ if } x_i \in kNN\left(x_j\right) \text{ and } x_j \in kNN\left(x_i\right) \end{cases} \tag{5}$$

juPSC uses Eq. (4) to calculate the adjacency matrix.

Step 3. Calculating the *Laplacian* matrix

The degree matrix is calculated as follows:

$$D_{i,j} = \begin{cases} 0, \text{ if } i \neq j \\ \sum_j w_{i,j}, \text{ if } i = j \end{cases} \tag{6}$$

where $W_{i,j}$ are the elements of the adjacency matrix W and $\sum_j w_{i,j}$ is the sum of the weights of the edges of other nodes connected by a node in the graph. Then, calculate *Laplacian* matrix L:

$$L = D - W \tag{7}$$

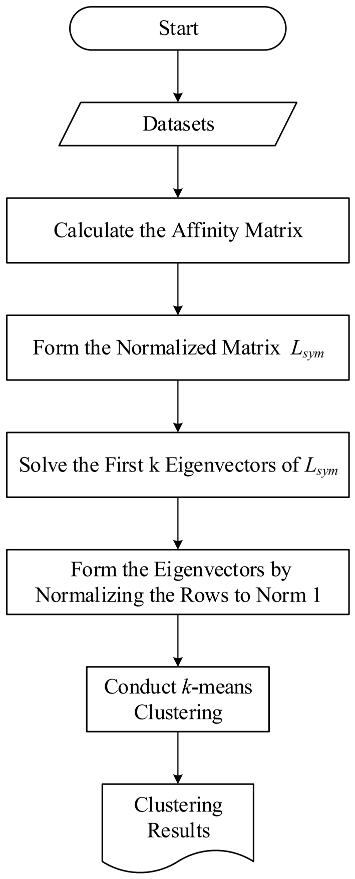

Start
Datasets
Calculate the Affinity Matrix
Form the Normalized Matrix L_{sym}
Solve the First k Eigenvectors of L_{sym}
Form the Eigenvectors by Normalizing the Rows to Norm 1
Conduct *k*-means Clustering
Clustering Results

Fig. 1. Flowchart of the spectral clustering algorithm.

Then, the normalized *Laplacian* matrix is calculated as follows:

$$L_{sym} = D^{-\frac{1}{2}} \times L \times D^{-\frac{1}{2}} \tag{8}$$

Thus, we transform the problem of graph cutting into the first k eigenvectors $u_1, u_2, \ldots, u_k$ for solving L_{sym}.

Step 4. Conducting *k*-means clustering

Let U be the matrix containing the vectors $\{u_1, u_2, \ldots, u_k\}$ as columns. Normalizing each row of U:

$$u_{i,j} = \frac{u_{i,j}}{\sqrt{\sum_k u_{i,k}{}^2}} \tag{9}$$

k-means is used to cluster by row in U. The overall calculation process is illustrated in Fig. 1.

2.2. Julia language

Julia is a high-level, high-performance, dynamic programming language [49]. Julia's standard library supports many built-in mathematical functions, including complex numbers right out of the box. Julia combines three key features of high-performance computing tasks: it is fast, easy to learn and use, and open source. In Julia's competition, C/C++ and Fortran are fast, and the available open-source compilers are excellent, but they are difficult to learn, especially for beginners without programming experience. Python and R are open source languages that are easy to learn and use, but their performance in numerical computation may be disappointing; MATLAB is relatively fast (still slower than Julia) and easy to learn and use, but it is commercial.

Julia has been widely used in machine learning and other scientific computation tasks. The model of machine learning is becoming more and more complex, and when the algorithm is implemented, a large amount of data can lead to performance problems. The Julia language supports functional programming, and researchers can focus on the implementation of algorithms. At the same time, Julia's performance is close to that of C/C++ and other statically compiled languages; see Fig. 2. Julia language has been used in many practical engineering problems. Frondelius et al. [13] proposed a finite element method (FEM) framework in Julia language, which allows the use of simple programming models for distributed processing of large finite element models across computer clusters. Sinaie et al. [57] used Julia language to implement the material point method (MPM). In large strain solid mechanics simulations, using only Julia's built-in characteristics, it performs better than a similar MATLAB based MPM code (with speed up of up to 8).

The Julia language supports parallel computation. In Julia, the parallel computation can be realized in the following ways: (1) coroutines (green threading), (2) multi-threading (experimental interface), (3) multi-core processing, and (4) distributed processing. Coroutines and multi-threading are suitable for small tasks.

The multi-core computation in the Julia language first reallocates the tasks and dynamically allocates the computation tasks to each process. In Julia, we use `SharedArrays` to allocate tasks to shared memory so that different processes can operate on the data at the same time; see Fig. 3.

3. Design and implementation of the parallel spectral clustering in Julia

3.1. Overview

In this paper, we design and implement the parallel spectral clustering algorithm, juPSC, on multi-core processors in Julia. To the best of the authors' knowledge, the juPSC is the first parallel spectral clustering algorithm developed with the Julia language. The juPSC is composed of three procedures: (1) calculating the affinity matrix, (2) calculating eigenvectors, and (3) conducting *k*-means clustering.

(1) **Calculating the affinity matrix**: the data is transformed into a graph, all data points are viewed as nodes in the graph, and the similarity between nodes is quantified as the weight of the corresponding node connection edge. Thus, an undirected weighted graph $G = (V, E)$ based on similarity is obtained. Procedure 1 is handled by the program interface `affinity` provided by juPSC.

(2) **Calculating eigenvectors**: the *Laplacian* matrix is constructed according to the similarity graph, the N-cut method [27] is used to cut the graph, and the first k eigenvectors of the *Laplacian* matrix are obtained. Procedure 2 is handled by the program interface `ARPACKSolver` provided by juPSC.

(3) **Conducting *k*-means clustering**: the *k*-means clustering algorithm is used to cluster k eigenvectors by rows, and the final clustering results are obtained. Procedure 3 is handled by the program interface `cluster` provided by juPSC.

3.2. Procedure 1: calculating the affinity matrix

We construct the initial affinity matrix according to Eq. (3). Considering that the affinity matrix needs much memory for large-scale data, we use the Coordinate format (COO) [35] to store the affinity matrix. The compressed matrix format supported in Julia is Compressed Sparse Column format (CSC) [59], which reads and writes information in memory by column. Therefore, further converting the sparse matrix to the CSC for subsequent computation will yield performance advantages.

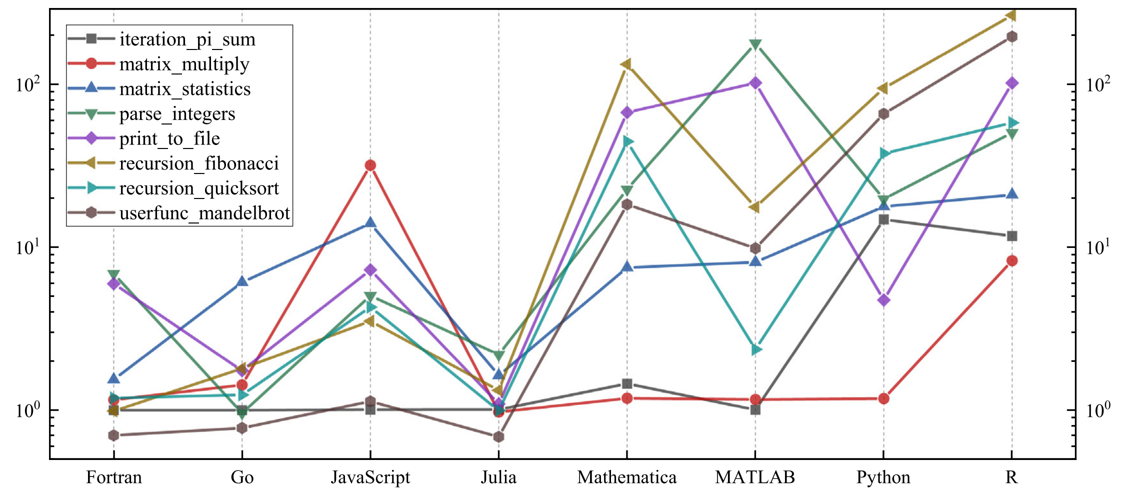


Fig. 2. Julia benchmarks (the benchmark data shown above were computed with Julia v1.0.0, Go 1.9, JavaScript V8 6.2.414.54, MATLAB R2018a, Anaconda Python 3.6.3, and R 3.5.0. C and Fortran are compiled with GCC 7.3.1, taking the best timing from all optimization levels. C performance is 1.0, smaller is better, the benchmark data is download from [5]).

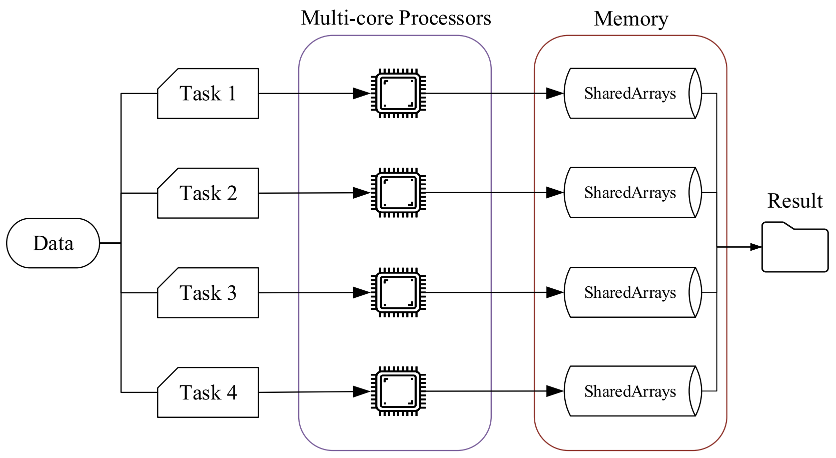


Fig. 3. Illustration of the workflow of multi-core processors in Julia.

The computation of the affinity matrix is implemented by the function `affinity`, and the parallel algorithm in the juPSC is described in Algorithm 1.

```
Algorithm 1 Calculating the Affinity Matrix
Input: The dataset, the number of nearest neighbors in the kNN algorithm
Output: Affinity matrix
1: Calculate the size of the array I, J and V, set it to SharedArrays.
2: Calculate the distance between each node and other nodes, select the nearest $k$ nodes, and calculate the similarity according to Eq. (4).
3: Repeat the above steps until all nodes are calculated.
4: Convert array I, J and V from COO to CSC.
5: return Affinity matrix
```

In our parallel algorithm, we need to determine the size of the arrays `I`, `J` and `V` according to the size of the dataset before the computation. These three arrays store the row index, column index and the value of each element in the affinity matrix. Suppose that, for dataset $X \in R_{(n\times d)}$, n is the number of nodes and d is the dimension of each node. When the number of nearest neighbors in the kNN algorithm is k, $size\ (I) = size\ (J) = size\ (V) = k \times n$, and `I`, `J` and `V` are set as the `SharedArrays`. The first k nearest nodes are used to calculate the adjacency matrix W and stored in the corresponding positions of `I`, `J`, and `V`. In the entire computation process, there is no data dependency between each basic task, and each element in the adjacency matrix W will be stored in the specified location; see Fig. 4(a).

3.3. Procedure 2: calculating eigenvectors

According to step 3 in Section 2, the degree matrix D, *Laplacian* matrix L, and normalized matrix L_{sym} are computed. There is no complex computation in the process of building L_{sym}; only the basic operation of the matrix is involved. Then, we need to solve the first k eigenvectors of L_{sym}.

L_{sym} is a symmetric positive-semidefinite sparse matrix. To implement the solution efficiently, we choose to invoke **ARPACK** [63] instead of the default OpenBLAS [67] in Julia. **ARPACK** is designed to use the implicitly restarted *Lanczos* or *Arnoldi* iterations to solve the eigenvalues and eigenvectors of real symmetric matrices (or general asymmetric matrices). At the same time, **ARPACK** supports the efficient solution of the large sparse matrix; see Fig. 4(b).

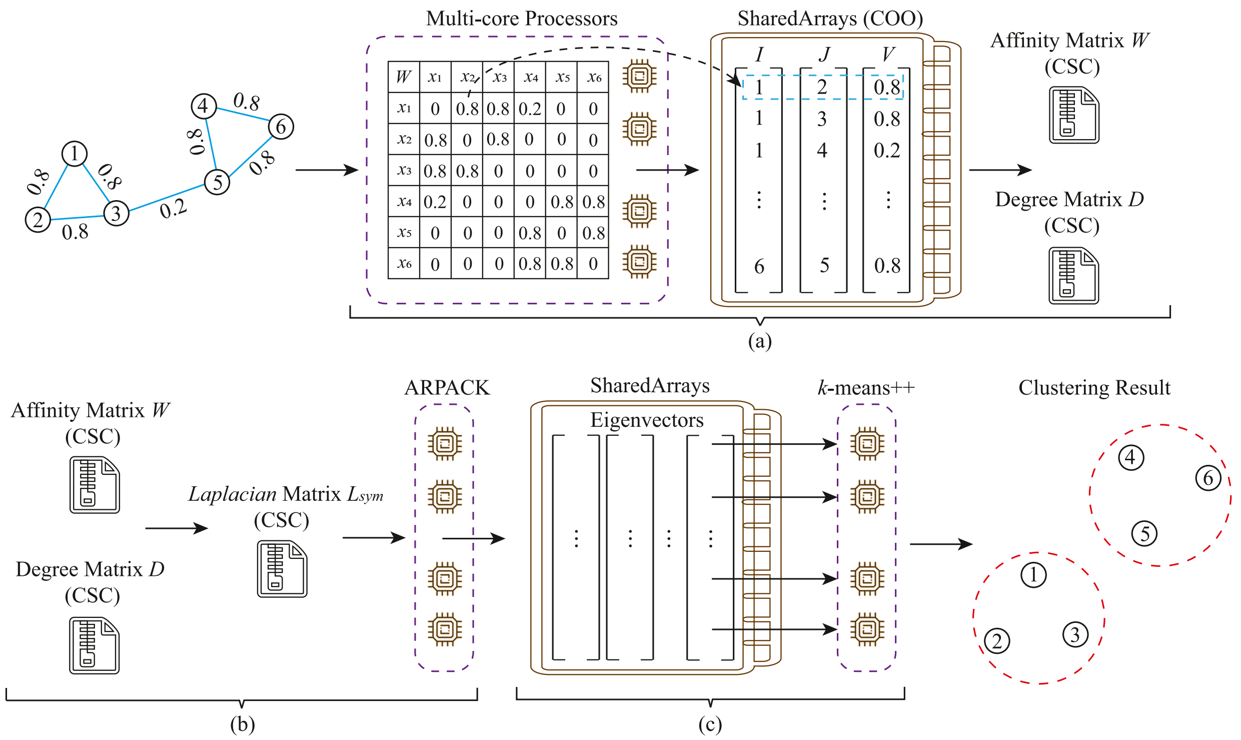


Fig. 4. Computation flow chart of the juPSC ((a) calculating the affinity matrix; (b)calculating eigenvectors; (c)clustering).

The *Laplacian* matrix is calculated, and the eigenvector is achieved by function `ARPACKSolver`. The procedure is described in Algorithm 2.

Algorithm 2 Calculating Eigenvectors

```
Input: Affinity matrix
Output: Eigenvectors
1: Calculate the degree matrix $D$ and Laplacian matrix $L$.
2: Calculate $L_{sym}$ by Eq. (8).
3: Invoking ARPACK to calculate the first k eigenvectors of the
   compressed matrix $L_{sym}$.
4: return Eigenvectors
```

3.4. Procedure 3: conducting k-means clustering

After we obtain the first k eigenvectors of L_{sym}, we form the matrix U by columns and normalize each row element of U. Finally, we use the k-means++ algorithm [14] to cluster U. The traditional k-means algorithm needs to randomly select k points in the dataset as the clustering center. Therefore, although the implementation is simple, the clustering results will also be affected. Arthur et al. [14] proposed an improved method based on k-means with randomly selected initial points so that the distance between the selected initial points is as great as possible.

In our parallel algorithm, we first set the eigenvector matrix U to the `SharedArrays` and randomly select a row as the first cluster center. The shortest distance $D(x)$ between each node and the current existing cluster center is calculated, and then the probability that the node is selected as the next cluster center is:

$$P(x) = \frac{D(x)^2}{\sum_{x \in U} D(x)^2} \tag{10}$$

We repeat the above steps until k initial clustering centers are selected. When we determine the cluster center, we can use the traditional k-means clustering algorithm to cluster the matrix U. In this process, each node needs to calculate the distance from the cluster center, and these computations are independent. We set the computation of distance as the basic task and assign each node to the nearest cluster center until the nodes belonging to the same cluster center no longer change; see Fig. 4(c). The clustering algorithm is solved by the function `cluster`. The parallel algorithm in the juPSC is described in Algorithm 3.

Algorithm 3 Conducting k-means Clustering

```
Input: Eigenvectors, number of cluster center points
Output: Clustering result
1: Combining eigenvectors into matrix $U$ by columns.
2: Normalize each row of $U$ according to Eq. (9).
3: Using $k$-means++ algorithm to select initial clustering
   centers.
4: The distance between each node and the nearest cluster cen-
   ter is calculated in parallel to determine which cluster center
   it belongs to.
5: Repeat step 4 until the cluster center of each node no longer
   changes (or the iterated number reaches a preset value).
6: return Clustering result
```

In the parallel k-means++ algorithm, we need to calculate the distance between each node and the cluster center to select the closest cluster center. We take the computation of distance as the basic task of each process, which can ensure that the execution time of each task is basically the same, and reduce the synchronization time for waiting for other processes to end. By setting the matrix U as a `SharedArrays`, no node will be obstructed when calculating the distance. As the amount of data increases, the proportion of time used for synchronization and transmission will decrease.

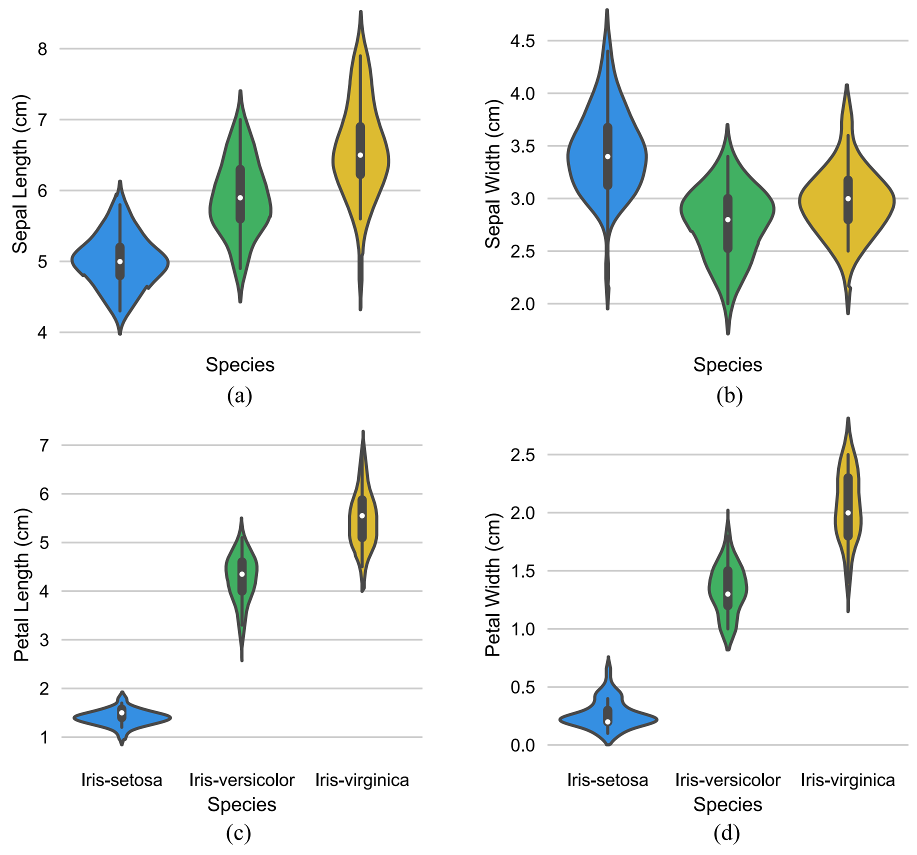


Fig. 5. The violin plot of **Iris**.

Table 1
Specifications of the employed workstation computer.

Specifications	Details
Platform	Windows 10 Professional
CPU	Intel Xeon 5118
CPU Frequency	2.30 GHz
CPU Cores	24
CPU RAM	128 GB
Julia Version	Julia 1.2.0

Table 2
Details of the Iris dataset.

Dataset	Iris
Number of instances	150
Class	Iris Setosa (50) Iris Versicolour (50) Iris Virginica (50)
Attribute Information	Sepal length (cm) Sepal width (cm) Petal length (cm) Petal width (cm)

Table 3
Clustering results.

Class	*k*-means	juPSC
Iris Setosa	50	50
Iris Versicolour	47	47
Iris Virginica	36	37

4. Results

To evaluate the juPSC's accuracy and efficiency, two groups of experiments are conducted on a workstation computer. The specifications of the workstation are listed in Table 1.

4.1. Verification of the accuracy

We use the standard dataset **Iris** to verify the accuracy of the juPSC and compare it with the result of clustering with *k*-means. **Iris** has a total of 150 data instances and 4 attribute features; see Table 2 for details of the dataset.

We input the original **Iris** data directly into the juPSC and *k*-means algorithms for computation. The detailed results are listed in Table 3.

4.2. Analysis of the computational accuracy

We found that the clustering results of *k*-means and the juPSC are almost the same. The accuracy of *k*-means and the juPSC was 88.7% and 89.3%, respectively. They were only different in **Iris Virginia**. We further analyze the **Iris** dataset and find that only **Iris Setosa** is linearly separable with the other two classes. Thus, the accuracy of **Iris Setosa** in the experimental results of the two algorithms is 100%. We explored the distribution of data in the **Iris** dataset and the relationship between each attribute and category, as well as between different attributes; see Figures 5 and 6.

These analyses can provide that the greatest correlation between the class is the petal length and petal width. We only select two attributes of the petal as the dataset and use *k*-means and the juPSC to experiment. The clustering results are listed in Table 4.

After analyzing the data, we obtain more accurate clustering results. In addition, the spectral clustering algorithm has many adjustable parameters. For example, we use the *k*NN algorithm

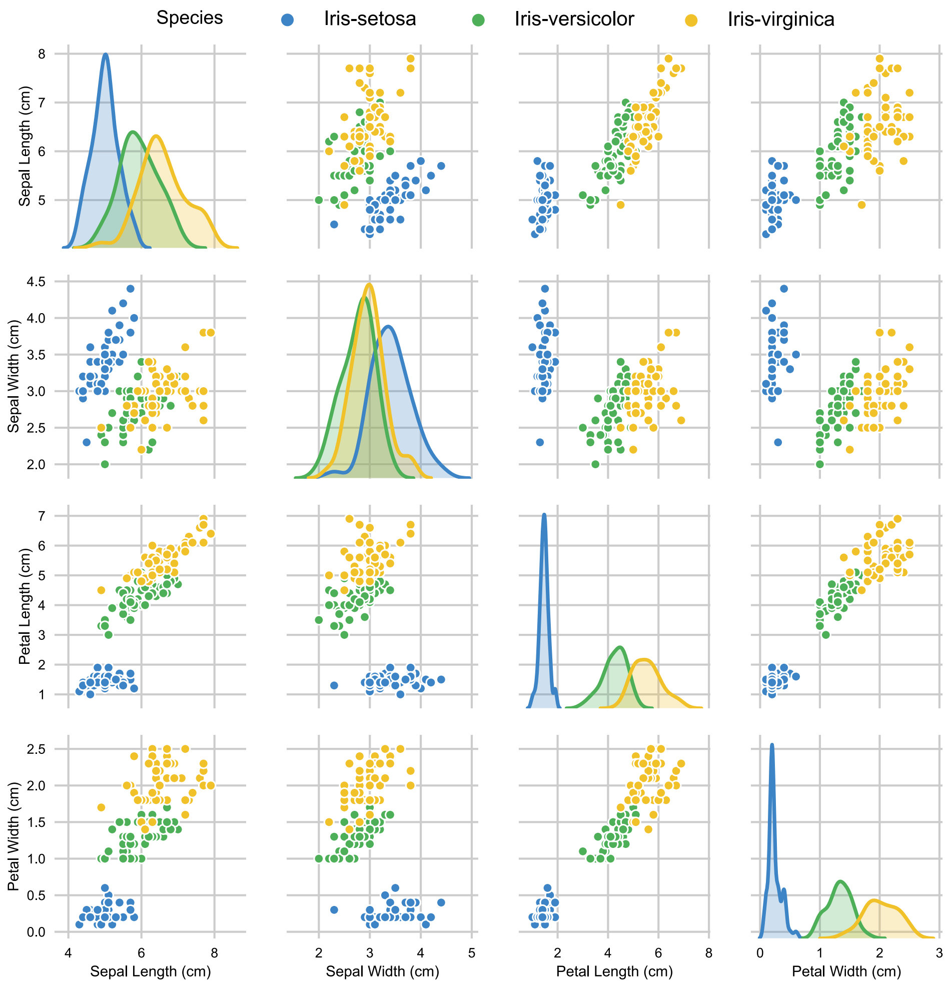


Fig. 6. The relationship between four attributes of **Iris**.

Table 4
Comparison of the clustering results for the Iris dataset.

Methods	Iris Setosa	Iris Versicolour	Iris Virginica	Error
juPSC	50	48	45	4.67%
k-means	50	48	44	5.33%

Table 5
Details of the four generated datasets.

Dataset	Number of instances	Number of attributes	Clusters
1	100,000	6	3
2	200,000	6	3
3	300,000	6	3
4	400,000	6	3

to construct the affinity matrix, in which the value of k needs to be determined before the program runs, the free parameter σ in Eq. (4) needs to be selected, etc. The selection of these parameters is beyond the scope of this paper; see reference [43] for details.

4.3. Evaluation of the efficiency

To analyze the performance of the juPSC intuitively, we focus on its efficiency in large-scale data. We randomly generated four datasets in Julia, and the details of the datasets are listed in Table 5.

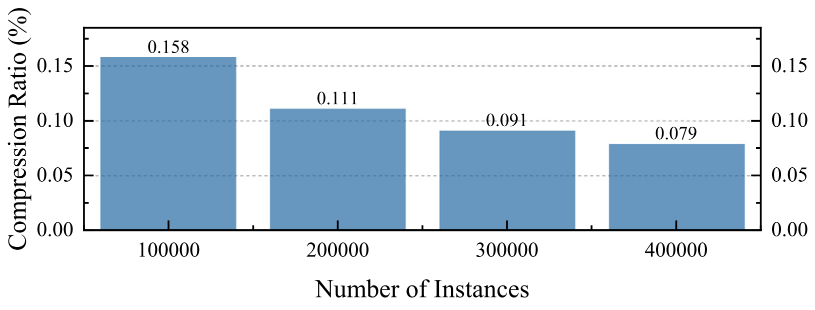
We use juPSC to test the serial computation time and parallel computation time of the above four datasets in the workstation computer. The experimental results of each dataset are listed in Table 6.

Table 6
Running time of the juPSC for the four generated datasets.

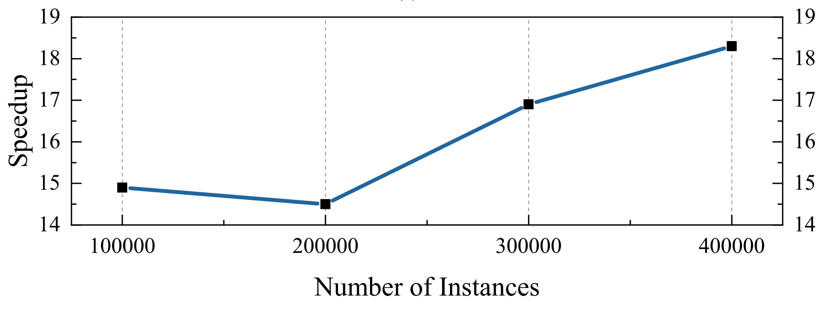
Dataset	Version	Procedure 1	Procedure 2	Procedure 3
1	Parallel	0.19 s	33.12 s	0.44 s
	Serial	4.13 s	28.66 s	5.26 s
2	Parallel	4.91 s	56.74 s	0.67 s
	Serial	72.93 s	55.81 s	7.89 s
3	Parallel	7.53 s	89.52 s	0.86 s
	Serial	131.74 s	85.37 s	9.82 s
4	Parallel	10.16 s	102.19 s	0.94 s
	Serial	192.55 s	118.56 s	10.35 s

4.4. Analysis of the computational efficiency

The juPSC is composed of three procedures: (1) calculating the affinity matrix, (2) calculating eigenvectors, and (3) conducting k-means clustering. Because eigensolvers use **ARPACK** to complete the computation of a large sparse matrix, it will automatically choose the faster method. Therefore, the time consumed in this procedure does not change much regardless of whether it is serial or parallel. The juPSC parallel algorithm is mainly embodied in the first and third procedures. We integrate the time consumed by these two procedures and analyze the speedup and compression ratio of the parallel algorithm; see Figs. 7(a) and 7(b).



(a)



(b)

Fig. 7. The compression ratio and speedup of the juPSC in parallel version.

We have observed that the speedup is less than 15× when the dataset is between 100,000 and 200,000. In the juPSC algorithm, the data needs to be computed in advance to dynamically allocate tasks to the processors. When the number of instances is small, task allocation accounts for a large proportion of the overall computation time, and the effect of parallel acceleration is not enough to offset the time spent on data transmission and allocation. When the number of instances increases gradually, it can be seen that the speedup also increases. In our experiment, when the number of instances is 400,000, the speedup of the juPSC reaches 18× and still shows an upward trend.

Moreover, when the similarity matrix stores the information between all nodes, it requires much memory. When there are 400,000 nodes in graph G, nearly 160 GB of memory is needed to store the affinity matrix completely with double precision. In the juPSC, the nearest k nodes of each node are selected according to kNN to construct the affinity matrix. In the experiment of 100,000 nodes, the compression ratio of affinity matrix is 0.158%.

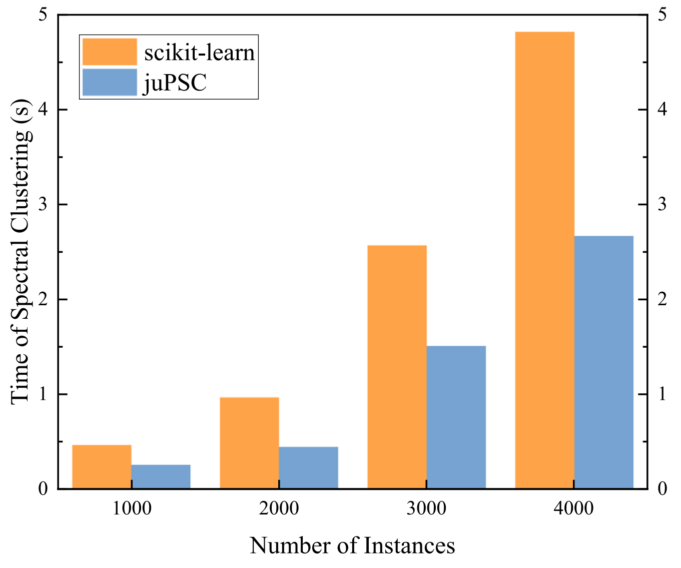


Fig. 8. Comparison of computation time between the juPSC and **scikit-learning**.

5. Discussion

5.1. Comparison with other algorithms

In Python version 3.7.4, we use **scikit-learn** 0.21.3 [48] to compare with the juPSC. In **scikit-learn**, we set `affinity` to `nearest_neighbors` and record the running time under different instances. The comparison between the juPSC and **scikit-learn** is presented in Fig. 8. The juPSC uses the serial version to compare with **scikit-learn**. With the increase of the dataset size, the efficiency advantage of the juPSC becomes more and more obvious.

In addition, we note that there are some parallel algorithms for distributed computing and graphics processing unit (GPU) computing. Chen et al. [23] proposed a parallel spectral clustering algorithm in distributed systems. Although communication and synchronization take a certain amount of time in a distributed system, as the amount of data increases, the effect of parallel algorithms becomes more apparent. A spectral clustering algorithm based on the GPU framework is proposed in the references [31,36], combining CUDA-based third-party libraries such as cuBLAS and cuSparse. The juPSC redesigned the parallel algorithm for the characteristics of the spectral clustering algorithm and applied it to non-graph data. The parallel spectral clustering algorithm was implemented in Julia in a modular way, and Julia's modernized design effectively solves the large-scale spectral clustering problem.

5.2. Performance analysis of juPSC

We analyze the time proportion of each sub-procedure, as shown in Fig. 9. It can be seen that calculating the affinity matrix and eigenvectors account for 90% of the total computation, and calculating the affinity matrix is an important procedure of the

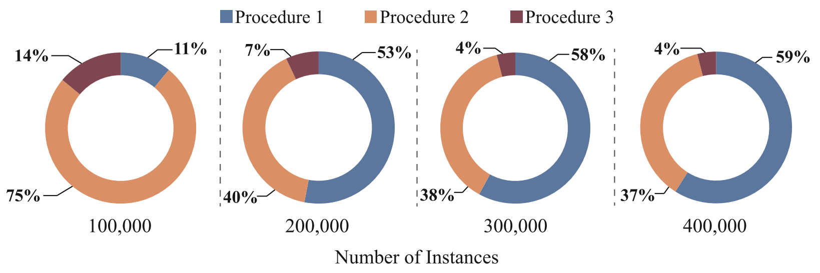


Fig. 9. Execution time percentage of different functions of the juPSC in serial version.

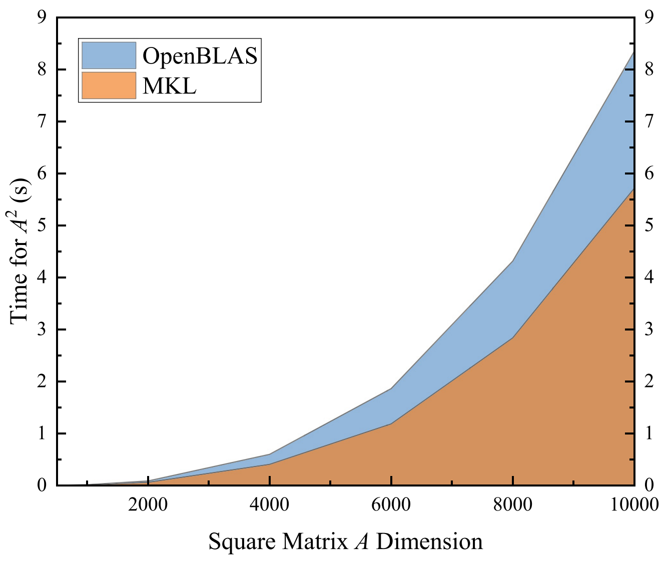


Fig. 10. Comparison of matrices multiplication time between MKL and OpenBLAS.

spectral clustering algorithm and involves many computational steps. We use the parallel algorithm to compute the affinity matrix, and the performance is satisfactory.

The main performance bottleneck of juPSC is to solve the eigenvectors. When the number of edges between nodes in the graph network increases, the non-zero value in the sparse matrix also increases gradually, and the solution speed will further decrease. In addition, the multiplication of $D^{-\frac{1}{2}}$ and three large sparse matrices are computed. When the number of instances increases, these matrix operations will also take some time.

In Julia, the operation of the matrix is completed by OpenBLAS. According to the comparison of reference [6], the computation efficiency of the Intel Math Kernel Library (MKL) [62] will be higher. We use a different matrix size to calculate its square and use MKL and OpenBLAS to experiment in Julia; see Fig. 10. At present, all matrix operation steps of the juPSC are completed by invoking MKL.

5.3. Scalability analysis of juPSC

Large scale data processing is a time-consuming computing process. The traditional single machine system is far from meeting the requirements of big data for computing performance. Machine learning and data mining algorithms are difficult to complete in an acceptable time. With the development of distributed computing and heterogeneous computing, the new high-performance computing system provides a good development opportunity for large-scale scientific computing and engineering simulation.

Among many big data processing technologies, there are mainstream big data processing technologies and system platforms represented by Apache Hadoop [60] and Apache Spark [22,65]. Using large-scale distributed storage and parallel computing technology, it brings effective technical means for big data processing and analysis. In Julia, the functions of distributed computing and cluster management are provided by `Distributed.jl` [7]. `Spark.jl` [8] and `Elly.jl` [9] correspond to Apache Spark and Apache Hadoop computing frameworks respectively. The modular design of juPSC and the modern features of Julia language make the parallel algorithm of spectral clustering easier to expand in the distributed platform.

The CPU–GPU heterogeneous high-performance computing system has been widely used in big data processing and machine learning [21,28]. However, it is a challenge to write efficient device code, which is usually done in the underlying programming language, and the high-level language is rarely supported. In Julia, `CUDAnative.jl` and `CuArrays.jl` [15] provide support for NVIDIA GPUs, while `ArrayFire.jl` [10] and `OpenCL.jl` [11] provide support for other GPU platforms. These excellent third-party packages enable juPSC to directly implement parallel computing on multiple platforms with a small amount of code modification, and high-level program interface encapsulation can also improve the efficiency of program execution.

5.4. Outlook and future work

In the juPSC, each procedure is built in a modular form, which is convenient for algorithm optimization. On a single computer, the juPSC can take full advantage of hardware computation. We can also combine the latest technology to expand the juPSC to other platforms.

With the advent of the era of big data, the improvement of personal computer has been unable to meet the needs of computation. With the development of distributed computation technology, it is possible to solve large-scale problems. Distributed computation divides tasks into smaller parts and assigns them to multiple computers for processing at the same time. At present, there are many distributed computing applications, such as **Folding@home** and **SETI@home** [41]. The juPSC has excellent computation performance on single computer, and if extended to a distributed system, it will be able to process large-scale data faster. According to references [12,20,58], many machine learning algorithms have been implemented in distributed system. In Julia, distributed computation is supported natively. The juPSC is also written in the form of modules, which is convenient for expansion.

In recent years, GPU has gained wide attention in machine learning and other fields by virtue of its efficient computation power [24–26,52], and its ability to parallel process data has far exceeded the computation power of traditional CPU. The excellent speed and scalability of the Julia language make it the first choice to combine with GPU technology. At present, Julia can use

`CUDAnative.jl` [15] to write GPU functions similar to CPU code. Combined with Julia's dynamic semantics and just-in-time (JIT) compilation [42], Julia can complete tasks efficiently on the GPU.

In the juPSC, we choose many parameters mainly from the analysis and experience of datasets. At present, we have not identified the criteria to determine these parameters. In the future, we hope to integrate data analysis and data processing into a whole through analysis. According to the characteristics of data, we can automatically select more appropriate parameters or indirectly reveal the relationship between parameters and computation results to form a complete scheme.

6. Conclusion

In this paper, we have designed and implemented an efficient parallel spectral clustering algorithm, juPSC, on multi-core processors in Julia. The juPSC is composed of three procedures: (1) calculating the affinity matrix, (2) calculating eigenvectors, and (3) conducting *k*-means clustering. Procedure 1 and procedure 3 have been designed as parallel algorithms. To verify the accuracy of the juPSC and evaluate its computation performance, we have carried out two sets of experiments. The experimental results indicate that: (1) the parallel version of the juPSC executed on a 24-core CPU is approximately 18× faster than the corresponding serial version; and (2) the serial version of the juPSC is faster than the spectral clustering implemented by Python. In addition, the structure and functions of the juPSC are designed considering modularity, which is convenient for combination and further optimization with other parallel computing platforms.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2020.01.003.

CRediT authorship contribution statement

Zenan Huo: Conceptualization, Data curation, Methodology, Software, Writing - original draft. **Gang Mei:** Supervision, Conceptualization, Methodology, Writing - review & editing. **Giampaolo Casolla:** Visualization, Investigation. **Fabio Giampaolo:** Software, Validation.

Acknowledgments

This research was jointly supported by the National Natural Science Foundation of China (Grant Numbers: 11602235 and 41772326), and the Fundamental Research Funds for China Central Universities, China (Grant Numbers: 2652018091, 2652018107, and 2652018109). The authors would like to thank the editor and the reviewers for their contributions.

References

[1] Julia Observer, URL https://juliaobserver.com/.
[2] Julia Package: Clustering.jl, URL https://github.com/JuliaStats/Clustering.jl.
[3] Julia Package: ScikitLearn.jl, URL https://github.com/cstjean/ScikitLearn.jl.
[4] Julia Package: QuickShiftClustering.jl, URL https://github.com/rened/QuickShiftClustering.jl.
[5] Micro benchmark comparison of Julia against other languages, URL https://julialang.org/benchmarks/.
[6] Intel MKL linear algebra backend for Julia, URL https://github.com/JuliaComputing/MKL.jl.
[7] Julia Package: Distributed.jl, URL https://github.com/JuliaLang/julia/tree/2d5741174ce3e6a394010d2e470e4269ca54607f/stdlib/Distributed.
[8] Julia Package: Spark.jl, URL https://github.com/dfdx/Spark.jl.
[9] Julia Package: Elly.jl, URL https://github.com/JuliaParallel/Elly.jl.
[10] Julia Package: ArrayFire.jl, URL https://github.com/JuliaGPU/ArrayFire.jl.
[11] Julia Package: OpenCL.jl, URL https://github.com/JuliaGPU/OpenCL.jl.
[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M.J.a.p.a. Devin, Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016.
[13] J. Aho, T. Frondelius, ovainola, arilaakk, T. Kelman, V. Stoian, T.G. Badger, M. Rapo, JuliaFEM/JuliaFEM.jl: Julia v1 compatible release.
[14] D. Arthur, S. Vassilvitskii, Siam/Acm k-means plus plus: the advantages of careful seeding, in: Proceedings of the Eighteenth Annual Acm-Siam Symposium on Discrete Algorithms, 2007, pp. 1027–1035.
[15] T. Besard, C. Foket, B.D. Sutter, Effective extensible programming: unleashing Julia on GPUs, IEEE Trans. Parallel Distrib. Syst. 30 (4) (2019) 827–841.
[16] J. Bezanson, A. Edelman, S. Karpinski, V.B. Shah, Julia: a fresh approach to numerical computing, SIAM Rev. 59 (1) (2017) 65–98.
[17] N. Biggs, N.L. Biggs, B. Norman, Algebraic Graph Theory, vol. 67, Cambridge university press, 1993.
[18] D. Cai, X. Chen, Large scale spectral clustering via landmark-based sparse representation, IEEE Trans. Cybern. 45 (8) (2015) 1669–1680.
[19] G. Casolla, S. Cuomo, V.S. Di Cola, F. Picciali, Exploring unsupervised learning techniques for the internet of things, IEEE Trans. Ind. Inf. (2019) 1.
[20] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z.J.a.p.a. Zhang, Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems.
[21] C. Chen, K. Li, A. Ouyang, Z. Tang, K. Li, Gpu-accelerated parallel hierarchical extreme learning machine on flink for big data, IEEE Trans. Syst. Man Cybern. Syst. 47 (10) (2017) 2740–2753.
[22] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, K. Li, A parallel random forest algorithm for big data in a spark cloud computing environment, IEEE Trans. Parallel Distrib. Syst. 28 (4) (2017) 919–933.
[23] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, E.Y. Chang, Parallel spectral clustering in distributed systems, IEEE Trans. Pattern Anal. Mach. Intell. 33 (3) (2011) 568–586.
[24] S. Cuomo, P. De Michele, F. Piccialli, 3D data denoising via nonlocal means filter by using parallel GPU strategies, Comput. Math. Methods Med. (2014).
[25] S. Cuomo, A. Galletti, G. Giunta, A. Starace, Surface reconstruction from scattered point via RBF interpolation on GPU, in: Federated Conference on Computer Science and Information Systems, 2013, pp. 433–440.
[26] P. De Michele, F. Maiorano, L. Marcellino, F. Piccialli, A GPU implementation of OLPCA method in hybrid environment, Int. J. Parallel Program. 46 (3) (2018) 528–542.
[27] C. Ding, X. He, H. Zha, M. Gu, H. Simon, Spectral min-max cut for graph partitioning and data clustering, 2001.
[28] M. Duan, K. Li, X. Liao, K. Li, A parallel multiclassification algorithm for big data using an extreme learning machine, IEEE Trans. Neural Netw. Learn. Syst. 29 (6) (2018) 2337–2351.
[29] P. Erdős, Graph theory and probability, Canad. J. Math. 11 (1959) 34–38.
[30] C. Fowlkes, S. Belongie, F. Chung, J. Malik, Spectral grouping using the nystrom method, IEEE Trans. Pattern Anal. Mach. Intell. 26 (2) (2004) 214–225.
[31] O. Frat, A. Temizel, Parallel spectral graph partitioning on cuda, in: GPU Technology Conference (GTC).
[32] Y. van Gennip, B. Hunter, R. Ahn, P. Elliott, K. Luh, M. Halvorson, S. Reid, M. Valasik, J. Wo, G.E. Tita, A.L. Bertozzi, P.J. Brantingham, Community detection using spectral clustering on sparse geosocial data, SIAM J. Appl. Math. 73 (1) (2013) 67–83.
[33] S. Gou, X. Zhuang, H. Zhu, T. Yu, Parallel sparse spectral clustering for SAR image segmentation, IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. 6 (4) (2013) 1949–1963.
[34] L. Hagen, A.B. Kahng, New spectral methods for ratio cut partitioning and clustering, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 11 (9) (1992) 1074–1085.
[35] A. Jain, Q. Sun, N. Goharian, Comparative analysis of sparse matrix algorithms for information retrieval, in: 6th World Multiconference on Systemics, Cybernetics and Informatics, Vol Vii, Proceedings: Information Systems Development Ii, 2002, pp. 80–85.
[36] Y. Jin, J.F. Jaja, Ieee, A high performance implementation of spectral clustering on CPU-GPU platforms, in: IEEE International Symposium on Parallel and Distributed Processing Workshops, 2016, pp. 825–834.
[37] M.I. Jordan, T.M. Mitchell, Machine learning: trends, perspectives, and prospects, Science 349 (6245) (2015) 255–260.
[38] L. Kaufman, P.J. Rousseeuw, Finding Groups in Data: An Introduction to Cluster Analysis, vol. 344, John Wiley & Sons, 2009.
[39] K. Khan, S.U. Rehman, K. Aziz, S. Fong, S. Sarasvady, DBSCAN: past, present and future, in: The Fifth International Conference on the Applications of Digital Information and Web Technologies, ICADIWT 2014, IEEE, 2014, pp. 232–238.

[40] J.R.C. Kou, R. Liu, Y. Li, Efficient parallel spectral clustering algorithm design for large data sets under cloud computing environment, J. Cloud Comput. Adv. Syst. Appl. 2 (1) (2013) 18.

[41] S.M. Larson, C.D. Snow, M. Shirts, V.S. Pande, Folding@ home and genome@ home: using distributed computing to tackle previously intractable problems in computational biology, 2009.

[42] M. Lubin, I. Dunning, Computing in operations research using Julia, Informs J. Comput. 27 (2) (2015) 238–248.

[43] U. von Luxburg, A tutorial on spectral clustering, Stat. Comput. 17 (4) (2007) 395–416.

[44] A. McCallum, K. Nigam, L.H. Ungar, Efficient clustering of high-dimensional data sets with application to reference matching, in: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Citeseer, 2000, pp. 169–178.

[45] G. Mei, N. Xu, J. Qin, B. Wang, P. Qi, A survey of internet of things (IoT) for geo-hazards prevention: applications, technologies, and challenges, IEEE Internet Things J. (2019) 1–16.

[46] M. Meila, J.B. Shi, Learning segmentation by random walks, in: Advances in Neural Information Processing Systems, in: Advances in Neural Information Processing Systems, vol. 13, 2001, pp. 873–879.

[47] F. Murtagh, P. Legendre, Wards hierarchical agglomerative clustering method: which algorithms implement wards criterion? J. Classif. 31 (3) (2014) 274–295.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

[49] J.M. Perkel, Julia: come for the syntax, stay for the speed, Nature 572 (7767) (2019) 141–142.

[50] F. Piccialli, G. Casolla, S. Cuomo, F. Giampaolo, V. Schiano di Cola, Decision making in IoT environment through unsupervised learning, IEEE Intell. Syst. (2019) 1.

[51] F. Piccialli, S. Cuomo, V.S.d. Cola, G. Casolla, A machine learning approach for IoT cultural data, J. Amb. Intel. Hum. Comput. (2019).

[52] M.M. Rathore, H. Son, A. Ahmad, A. Paul, G. Jeon, Real-time big data stream processing using GPU with spark over hadoop ecosystem, Int. J. Parallel Program. 46 (3) (2018) 630–646.

[53] K. Rohe, S. Chatterjee, B. Yu, Spectral clustering and the high-dimensional stochastic blockmodel, Ann. Statist. 39 (4) (2011) 1878–1915.

[54] S. Sarkar, P. Soundararajan, Supervised learning of large perceptual organization: graph spectral partitioning and learning automata, IEEE Trans. Pattern Anal. Mach. Intell. 22 (5) (2000) 504–525.

[55] J. Schmidhuber, Deep learning in neural networks: an overview, Neural Netw. 61 (2015) 85–117.

[56] J.B. Shi, J. Malik, Normalized cuts and image segmentation, IEEE Trans. Pattern Anal. Mach. Intell. 22 (8) (2000) 888–905.

[57] S. Sinaie, V.P. Nguyen, C.T. Nguyen, S. Bordas, Programming the material point method in julia, Adv. Eng. Softw. 105 (2017) 17–29.

[58] S. Tsironis, M. Sozio, M. Vazirgiannis, L. Poltechnique, Accurate spectral clustering for community detection in MapReduce, in: Advances in Neural Information Processing Systems (NIPS) Workshops, Citeseer.

[59] A. Usman, M. Lujan, L. Freeman, J.R. Gurd, Performance evaluation of storage formats for sparse matrices in fortran, in: Lecture Notes in Computer Science, vol. 4208, 2006, pp. 160–169.

[60] V.K. Vavilapalli, A.C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache hadoop yarn: yet another resource negotiator, in: Proceedings of the 4th Annual Symposium on Cloud Computing, ACM, 2013, p. 5.

[61] K. Wagstaff, C. Cardie, S. Rogers, S. Schrödl, et al., Constrained k-means clustering with background knowledge, in: Icml, vol. 1, 2001, pp. 577–584.

[62] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, Y. Wang, Intel Math Kernel Library, Springer International Publishing, Cham, 2014, pp. 167–188.

[63] T.G. Wright, L.N. Trefethen, Large-scale computation of pseudospectra using ARPACK and eigs, SIAM J. Sci. Comput. 23 (2) (2001) 591–605.

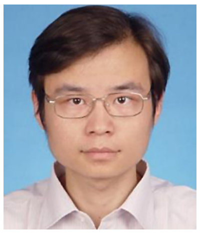
[64] Z. Wu, R. Leahy, An optimal graph-theoretic approach to data clustering - theory and its application to image segmentation, IEEE Trans. Pattern Anal. Mach. Intell. 15 (11) (1993) 1101–1113.

[65] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, et al., Apache spark: a unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65.

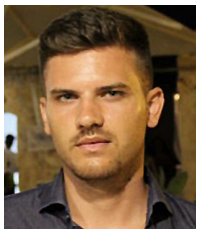
[66] T. Zhang, R. Ramakrishnan, M. Livny, BIRCH: a new data clustering algorithm and its applications, Data Min. Knowl. Discov. 1 (2) (1997) 141–182.

[67] X. Zhang, Q. Wang, W.J.A.A. Saar, OpenBLAS: An optimized BLAS library.

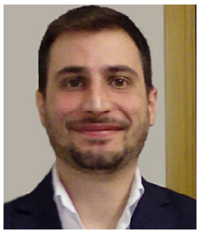
Zenan Huo is currently a master student at China University of Geosciences (Beijing). His research interests are in the areas of Machine Learning, Data Science, and Numerical Simulation, including GPU Computing and FEM Analysis.



Gang Mei received the bachelor's and master's degrees from the China University of Geosciences (Beijing), and the Ph.D. degree from the University of Freiburg, Germany, in 2014. He is currently an Associate Professor in numerical modeling and simulation with the China University of Geosciences (Beijing). He has published more than 50 research articles in journals and academic conferences. His main research interests include the areas of numerical simulation and computational modeling, including computational geometry, FEM analysis, GPU computing, data mining, and complex science and applications. He has been serving as an Associate Editor for the journal IEEE Access.



Giampaolo Casolla is a research fellow in Computer Science and Mathematics at Department of Mathematics and Applications R. Caccioppoli University of Naples Federico II, Naples, Italy. He received the Master Degree in Mathematics at the University of Naples Federico II. His research interests are focused on Machine Learning applications, Graph Mining and Data Visualization techniques.



Fabio Giampaolo is a research fellow in Computer Science at the National Interuniversity Consortium for Computer Science - ITEM-SAVY Laboratory in Naples, Italy. He received the Bachelor Degree in Mathematics at the University of Naples Federico II. His research interests are in Mathematical Modeling, Data Analytics and Python programming.