Parallel iterative hybridized threshold clustering for massive data

by

Yang Yang

B.S., YangZhou University, 2012

M.S., YangZhou University, 2015

M.S., Kansas State University, 2017

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Statistics
College of Arts and Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2020

Approved by:

Major Professor
Michael Higgins

Abstract

Iterative hybridized threshold clustering (IHTC) is a recently developed algorithm designed to decrease runtime and reduce memory usage of commonly used clustering algorithms under massive data settings. The IHTC pre-processes the data with iterative threshold instance selection (ITIS) to scale down the size of data before proceeding with the standard clustering analysis, such as k-means and hierarchical clustering. However, when dealing with massive amounts of data, for example, when the number of data points $n > 10^8$, the computational cost of IHTC may still be prohibitive. Efficient parallel implementation may provide a pathway to further reduce computational cost and memory usage of IHTC. In this study, we partition the data points into batches and IHTC is performed on each batch. The prototypes generated from IHTC are collected for k-means clustering. We implement the parallelization using the R packages "Rdsm" and "parallel", and test our implementation on simulated data on the Beocat high-performance cluster by varying the number of cores and batches. Performance is evaluated though accuracy, runtime in seconds, and memory usage in GB. We find that parallelization improves the runtime of IHTC substantially. For example, for a dataset of size $n = 10^9$, dividing the data into 500 batches and applying parallellization through the parallel package on a node with 8 cores decreases runtime by a factor of 4.36. Additionally, Rdsm parallelism for small scale data ($n \approx 10^8$) may decrease memory usage while preserving clustering accuracy. We conclude that a parallel programming design should create a proper number of threads to provide enough work for all cores to efficiently use the available computing resources.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my parents for always supporting and encouraging me even though they never read English. They give me endless love and all my effort in my life is to make them pround of me and pay back their love and care.

I also want to thank my friends in department for their support and help in both study and life. I will memorize the precious time we discussed problems for class as well as laugh in get-together.

Then I would like to acknowledge my committee member, Dr. Weixing Song, one of the smartest and most respectable professor to me. The STAT730 taught by him is my first course in Statistics. His clear and logic teaching open the gate of Statistics for me. I will never forget his help when I was worried about funding when I came to the department.

I also wanna thank my committee member, Dr. Haiyan Wang. She is talented in academics, excellent as a professor, responsible as a teacher and adorable as a lady. I enjoyed every class from her and I'm touched by her attitude to teaching. I will always be grateful to her office hour when she help me with knowledge beyond course, it will become a precious memory of my study life.

Last but the most important, I would like to thank my major professor, Dr. Michael Higgins, who is the best ever advisor in the past and also the future of my life. His motivation, patience and wisdom impressed me and tell me what an excellent professor is like. He is excellent in research with smart and creative brain but he always has a humble personality. He is also a caring mentor who always encourage me even though I made mistakes. Actually, this is my third master degree, but I can never imagine a wonderful mentor-ship like this until I met Dr. Higgins. I have gained and grown a lot with his help and support, and I will harbor this gratitude all the time.

Finally, I want to thank my boyfriend who always give me support for study and courage for life. Also I wanna give a lot thanks to all nice people I've met and this beautiful world.

Chapter 1

Introduction

1.1 Clustering Algorithms

Clustering is a machine learning method in which objects with comparatively more similar traits are assigned into the same groups, called clusters. Clustering is unsupervised, meaning that inference are drawn from data without labeled responses. It is one of the most widely used methods in statistical data analysis, and employed to model difficult problems in many fields involving machine learning (Nunez-Iglesias et al., 2013), image analysis (Rocha et al., 2009), and bioinformatics (Olman et al., 2008).

Common techniques of clustering mainly include connectivity models, centroid models, distribution models, and density models. Connectivity models (for example, hierarchical clustering (Eisen et al., 1998)), as its name suggests, are based on distance connectivity. Agglomerative hierarchical clustering starts by classifying each data point as a separate cluster. Then, it repeatedly executes the procedure of identifying two most similar clusters and merging them into a new cluster until the entire dataset is aggregated as one cluster. Its variant, known as divisive hierarchical clustering, in contrast, initially treats all data points as a single cluster, and then successively partitions these clusters. A dendrogram can visualize the process of aggregating or dividing. Since the core idea of the hierarchical clustering algorithm is based on data point being more similar to nearby data points than to those farther away, it has the advantage of being easy to interpret. Hierarchical clustering also provides a theoretical foundation of other clustering algorithms and inspires the development of density-based clustering analysis. However, it lacks scalability for handling large datasets with the complexity of $O(n^3)$ (Bar-Joseph et al., 2001).

A well-known method of centroid models is k-means clustering (Likas et al., 2003), in which each cluster is represented by a *prototype* which is the mean of data points in the cluster. A prototype can also be defined with other central measures such as the median. The objective of k-means clustering is to minimize the total intra-cluster variance. The objective function is described as,

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} (x_i^j - c_j)^2 \tag{1.1}$$

Where k is the number of clusters, n is the number of units, x_i^j is unit i in cluster j and c_j is the center of cluster j. This clustering algorithm heuristically solves an optimization problem defined by iteratively assigning data points to the cluster with the nearest prototype to generate a new prototype such that the within-cluster distances are minimized. The algorithm is conceptually close to the k-nearest neighbor classifier, which is a powerful supervised learning technique in machine learning (Ding and He, 2004). It also has a theoretical relationship with the distribution model-based clustering when modified by using expectation-maximization algorithms (Nasser et al., 2006). Even with intriguing theoretical properties, k-means suffers from both problems of improper initial centroids and sensitiveness to sample outliers (Chawla and Gionis, 2013). Additionally, since the number of clusters needs to be specified beforehand, it is restrictive by demanding prior knowledge before clustering. Choosing initial centroids is an especially important task since it directly affects the performance. Another large drawback is that the convergence to the global optimum is not guaranteed, and performance depends on the choice of initial centroids. Many variations of k-means clustering are proposed to improve different aspects of the algorithm. For example, k-means++ method specifies an algorithmic approach to choosing initial cluster centroids, as opposed to randomly choosing them, before standard k-means is performed (Arthur and

Vassilvitskii, 2007). Clustering through k-medoids, or partitioning around medoids (PAM), is more robust towards outliers by giving constraints on the choice of the centroid from existing members in the cluster (Park and Jun, 2009).

A popular distributional model for clustering is the Gaussian mixture model using expectation-maximization algorithm. This algorithm assumes that each data point is generated from one of several Gaussian distributions. Similar data points are more likely to belong to the same distribution. This algorithm shows a great advantage in optimizing multimodal functions with only a few local optima, but it may struggle if data deviates substantially from a Gaussian mixture distribution (Lu and Yao, 2005). Density-based clustering algorithm separates clusters from each other through contiguous regions of low density of objects. Data objects located in low-density regions are typically considered border points or noise (for example, outliers). DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is one of the most well-known clustering algorithms in this category (Kriegel et al., 2011).

However, these clustering algorithms often require prohibitive computational cost when sample sizes n are massive. The iterative hybridized threshold clustering (IHTC) proposed by Luo et al. (2019) is a novel statistical method which hybridizes with other clustering algorithms to scales down data without loss of information. The parallel implementations of IHTC in this report aim to increase the viability of these clustering methods under massive data settings.

1.2 Dissimilarity measures

Dissimilarity measures often are critical for clustering methods. We now describe them briefly. Suppose each unit i has a response y_i and a p-dimensional covariate vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$. The dissimilarity between unit i and unit j, denoted d_{ij} or $d(x_i, x_j)$, often is a function of differences between covariate vectors x_i and x_j. A common choice—and the one used in this study—is the Euclidean distance,

$$d(x_i, x_j) = \sqrt{\sum_{\ell=1}^{p}(x_{i\ell} - x_{j\ell})^2} \tag{1.2}$$

Other commonly used measures of dissimilarity include the Manhattan distance, the Minkowski distance, and the Mahalanobis distance.

A small value of d_{ij} indicates that unit i and unit j have similar covariates and vice versa. Thus, the objective of clustering is to find a set of clusters such that units belonging to the same cluster have small dissimilarities while units from different clusters have large dissimilarities.

1.3 Iterative Hybridized Threshold Clustering

Threshold clustering (TC) is a recently developed clustering method that employs graph theory to form clusters such that each cluster contains at least a pre-specified number of units t^* and so that maximum dissimilarity between any two units within the same cluster—the maximum within-cluster dissimilarity (MWCD)—is small (Higgins et al., 2016). In particular, TC tends to form clusterings comprised of many small clusters.

The steps of TC are as follows:

1. **Construct** $(t^* - 1)$**nearest-neighbor subgraph**:

   The $(t^* - 1)$-nearest neighbor graph connects unit i and unit j by an edge, if the their distance d_{ij} is among the $(t^* - 1)$-th smallest dissimilarity from unit i to all units. A $(t^* - 1)$-nearest-neighbor subgraph, denoted as NG_{t^*-1}, is constructed based on the Euclidean distances in this study.

2. **Choose block seeds**:

   A set of units S is selected as block seeds which satisfy:

   (1) There is no walk of length one or two in NG_{t^*-1} between any two seeds in S.

   (2) Any unit outside the S can be reached by one seed in S in a walk of length at most two in NG_{t^*-1}.

3. **Grow from block seeds**:

   A cluster is formed by growing from each block seed in S, which contains the block seed and any unit adjacent to it in NG_{t^*-1}.

4. **Assign remaining units**:

   For any unassigned unit, there exists a walk of length two from at least one seed in S in NG_{t^*-1}. Assign any remaining unit to the seed with the smallest dissimilarity.

Threshold clustering was initially proposed to perform statistical blocking of massive experiments (Higgins et al., 2016). Unlike representative centroid models (e.g. k-means), specifying the number of clusters beforehand is not required in TC. Instead, the number of units in each cluster is specified in order to construct the nearest neighbors graph. Additionally, TC is a very efficient method, requiring $O(n)$ runtime and memory outside of the construction of a t^*-nearest-neighbor graph. Finally, the performance of TC has theoretical guarantees. Specifically, out of all clusterings containing at least t^* units, TC produces a clustering with MWCD at most 4 times larger than the smallest MWCD possible.

Recently, Luo et al. (2019) proposed using TC as an instance selection (IS) method called iterative threshold instance selection (ITIS). IS is a pre-processing step in which the size of data is reduced before statistical analysis is performed. Ideally, conclusions made on the reduced dataset would be the same had the analysis been performed on the original dataset(Liu and Motoda, 2002). ITIS proceeds as follows. First, for a given t^*, TC is performed on n units to generate n^* clusters where each cluster contains at least t^* units. The centers of these n^* clusters, called prototypes, are computed and used to represent the original data set. Then TC is performed again on the n^* prototypes until the data size is sufficiently reduced.

Iterative hybridized threshold clustering (IHTC) is designed to increase the scalability of clustering algorithms (Luo et al., 2019). In IHTC, data is pre-processed with ITIS before proceeding with clustering algorithms, for example, k-means and hierarchical clustering (Luo et al., 2019). That is, after condensing the data to a set of prototypes with ITIS, another clustering algorithm, like k-means, is applied to the current prototypes. These prototypes still maintain the information of the original data but dramatically reduce the size of data. IHTC shows advantages in decreasing the required runtime and memory usage of the original clustering algorithms had they been performed on the entire dataset with minimal loss in clustering accuracy. In this study, we propose two methods to implement parallelization of IHTC to increase its ability to deal with massive data.

1.4 IHTC Procedure

The IHTC is performed as follows(Luo et al., 2019):

1. **Threshold clustering**

2. **Create prototypes**:

   A center vector is generated as a prototype for each cluster.

3. **Terminate or continue**:

   Replace the original data with the prototypes generated above. If the data is reduced to the desired size, then ITIS is completed. Otherwise, repeat steps 1 to 3.

4. **Apply clustering algorithm**:

   IHTC can be hybridized with different standard clustering algorithms, such as k-means and hierarchical clustering. We take k-means as an example to describe here.

   (a) **Initialize the cluster centers**:

   A set of cluster centers are selected randomly.

   (b) **Assign units to centers**:

   All units are assigned to a nearest center (with the smallest Euclidean distance to the center vector) and temporary clusters are formed by the cluster center and its assigned units.

(c) **Update the cluster centers**:

The cluster centers are updated by computing the center of each temporary cluster.

(d) **Repeat and terminate**:

Repeat step (b) and (c) until the cluster centers converge.

The ITIS procedure is illustrated in Figure 1.1 (Luo et al., 2019) and the workflow to implement IHTC in this report is shown in Figure 1.2.

1.5 Parallel Computing

Parallel computing refers to the simultaneous execution of operations in order to solve a computational problem with the use of multiple computing resources, such as Central Processing Unit (CPU), Graphics Processing Unit (GPU), or cluster. Parallel computing dramatically improves the efficiency in runtime and remote memory usage. Parallel computing can also take advantage of remote computing resources and make better use of modern computer hardware. Because of these properties, parallel computing plays a huge role in high-performance computing (Czarnul, 2018) and has broad applications in various areas.

The simulation of large problems is of growing importance in the fields of science and engineering, and greater and greater computing power and memory are needed for precise simulations. Parallel computing has been widely used to model complex problems in a large range of disciplines, like atmosphere (Tolstykh et al., 2017), physics (Amadio et al., 2017), biochemistry (Porter et al., 2019), genetics (Gonzalez-Dominguez and Martin, 2017), geology (Mao et al., 2016), seismology (Corneio-Surez et al., 2018), and other natural sciences. Also, with the rapid development of industrial and commercial data, efficient parallel implementation techniques are the key to meet the required scalability and performance in data analyses.

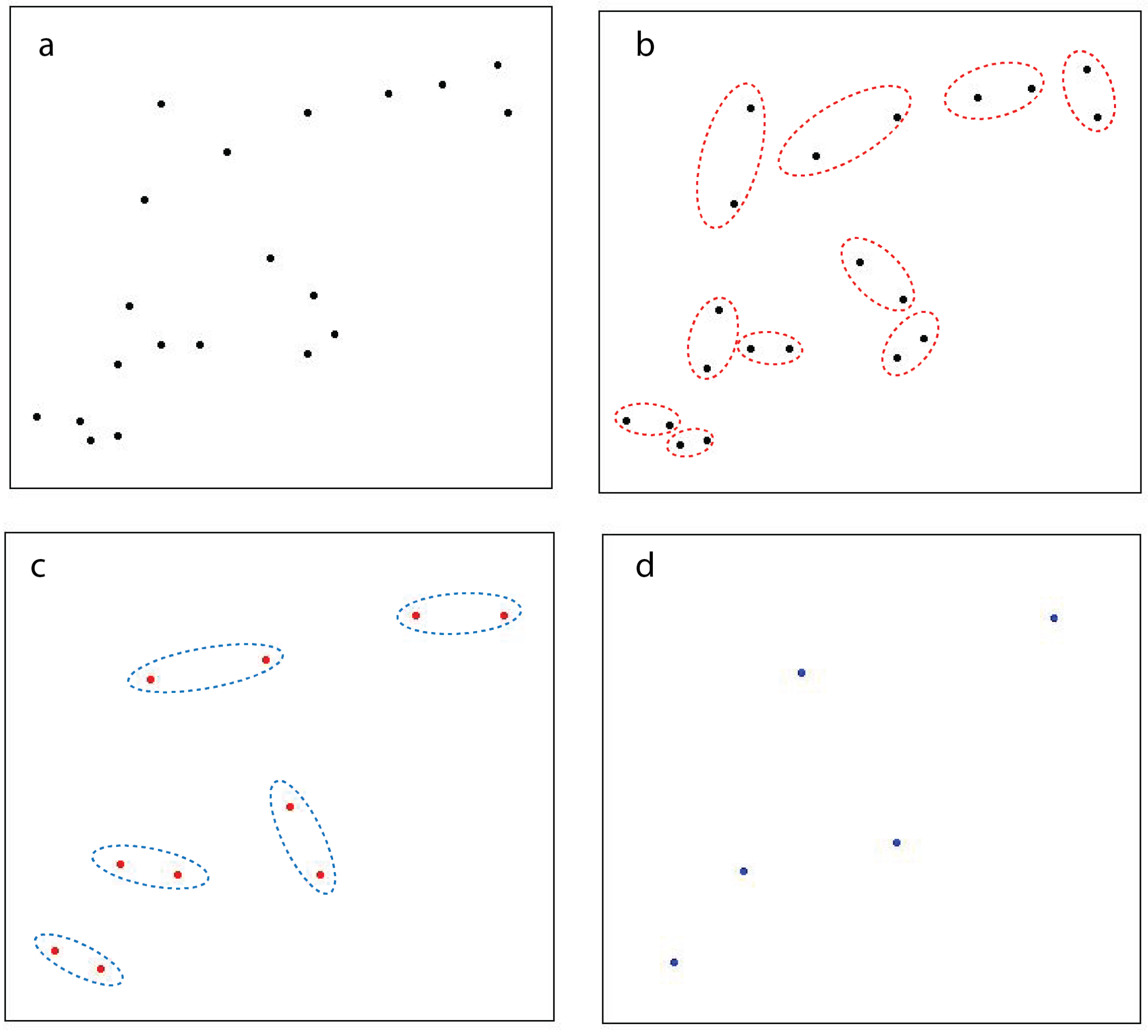

Figure 1.1: Iterative threshold instance selection (ITIS) procedure for a sample of $n = 20$ bivariate data points with 2 iterations of constructing a 2-nearest-neighbors graph.

a: The original data is represented by 20 black points. b: Threshold clustering is performed using 1-nearest-neighbors graph (red dashed circles). c: 10 prototypes are generated and are displayed as red points. Then the threshold clustering (blue dashed circles) is performed again on these prototypes. d: The blue points are final prototypes formed from ITIS.

1.5.1 Evolution in Parallelism

The first discussion of parallelism in numerical calculations dates back to 1958, after the launch of a project in 1956, which aimed to design a supercomputer with 100 times the performance of any available at that time. The technical and commercial success of the first

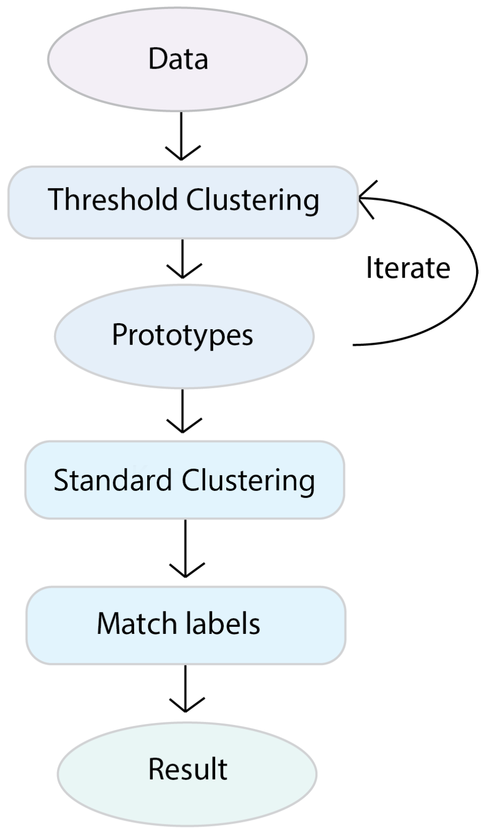

Figure 1.2: Workflow to implement iterative hybridized threshold clustering (IHTC). Apply iterative threshold instance selection (ITIS) on the data to form prototypes. Then k-means clustering is performed and the labels from k-means are matched to the original data points.

supercomputer in 1964 and breakthroughs in shared memory multiprocessors throughout the 60's to 70's helped promote the implementation and application of parallel computing. In the 1980's, massively parallel processors (MPPs)—a single computer with many interconnected networked processors—appeared and dominated the field of computing until the emergence of clusters in the 80's. A cluster is built from a group of loosely coupled computers connected by a network. Since then, cluster computing, with its formidable competitive advantage, became the dominant architecture and eventually displaces MPPs in many fields (Hockney and Jesshope, 2019). Today, about 87% of 500 most powerful commercially available computers are clusters and the remaining are MPPs. From the 1980's to 2000, frequency scaling was the primary computer architecture paradigm to improve computer performance (Malyshkin, 2017). However, increasing the frequency brings the problems of power consumption and consequently heat generation (Mullen et al., 2017).

To deal with the concern of CPU overheating, instead of straining to increase the speed of a single processor, effort was shifted to obtain more computing power by stamping multiple processors on a single chip. Each core in multiple-core computers serves as a computing unit independently, and all cores access shared memory. Instead of increasing clock frequency, enhancing the computing power of multi-core architectures is realized by paralleling the tasks in order to accelerate the runtime. Thus, improving performance achieved through parallel processing is much more efficient than that by frequency scaling. That is, the rapid development of microprocessor performance and network bandwidth, as well as the prevalence of multi-core processors in desktop and laptop systems, make parallel computing the mainstream in computer architecture paradigm.

1.5.2 Types of Parallelism

Parallel computing systems have many different forms, for example, a supercomputer with thousands of processors, a network consisting of computers with hundreds of processors, and even a desktop equipped with multi-core processors (Mashfiq, 2012). Modern parallel architectures can be categorized based on the different processing mechanisms, including manners of memory access and message passing between the processors. The most common classes of parallel computers are scalar computers, parallel vector processors (PVP), symmetrical multi-processing or shared memory multiprocessors (SMP), distributed memory massively parallel processors (MPP), distributed memory shared memory multiprocessors, cluster systems, and grid computing (Peng et al., 2017).

In SMP, shown in Figure 1.3a, the processors operate independently but access a shared memory and connect via a bus. Usually, SMP has small processors and a sufficient amount of memory bandwidth, which are cost-effective. Processors in a MPP (Figure 1.3b) also work independently with its own memory and copy of the operating system and application. Communication between processors is achieved through interconnected networks.

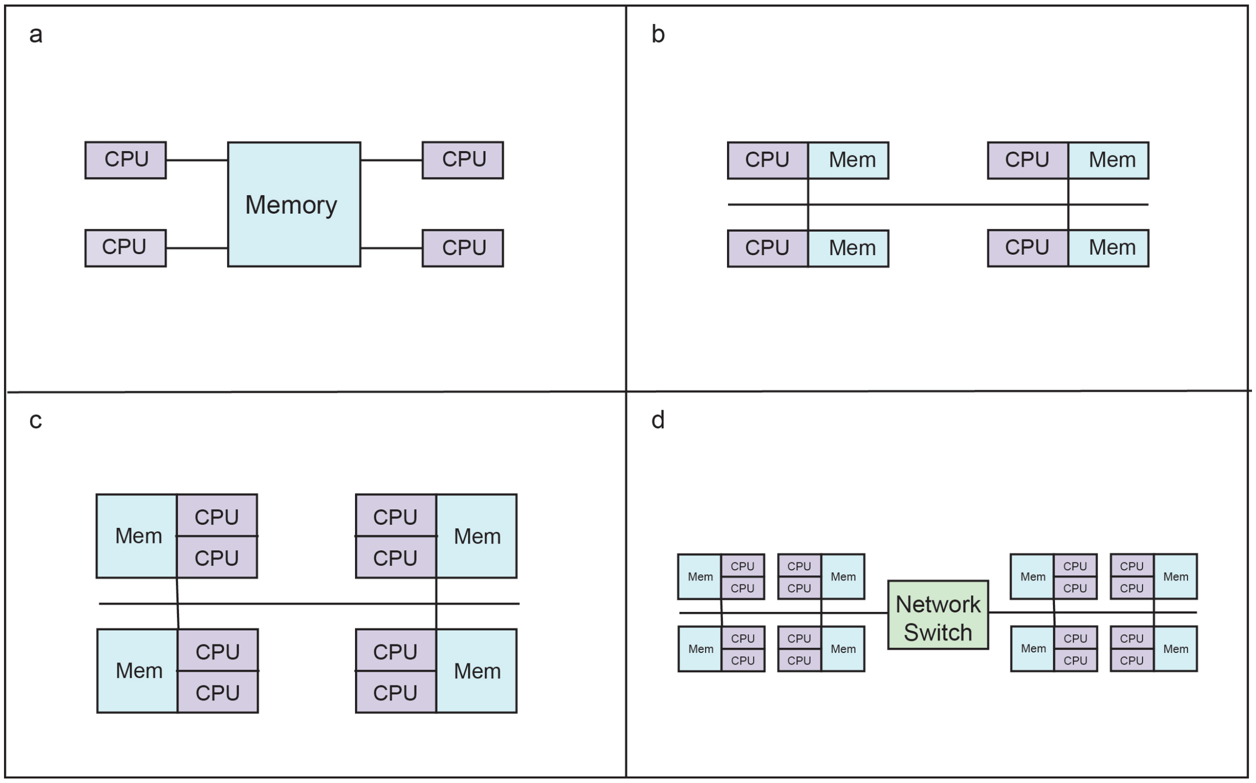

Figure 1.3: Common parallel computing systems.

a: shared memory multiprocessor (SMP); b: distributed memory massively parallel processor (MPP); c: distributed memory shared memory multiprocessor; d: cluster systems.

Distributed memory shared-memory multiprocessors (Figure 1.3c) is a hybrid system of distributed memory and SMP. Processing elements are connected by a network, within which processors share the same memory. Cluster computing (Figure 1.3d) is composed of a couple of distributed memory computers which are connected by a network (Mashfiq, 2012).

1.5.3 Design of Parallelism

When an algorithm is transformed into a parallel program, there are two important components to consider: concurrency and synchronization. Concurrency refers to the parts of the algorithm which can be carried out independently and are the target for parallel computing. Synchronization is the communication between multiple processors, including synchronization of processes, and synchronization of data. Synchronization of processes refers to the concept that different processors “handshake” to execute a certain sequential action during paralleling. Synchronization of data refers to the coherence of data storage in different processors to maintain data integrity. Therefore, a suitable parallel programming includes enough independent computation and effective synchronization in terms of both time and memory.

Even though the rapid development in hardware brings more support to parallelism, there is an upper bound on the speedup of parallel programming regardless of increased number of processors. Because maximum speedup to parallel an algorithm depends on the parallelizable proportion of computations. The maximum speedup of parallel programming for a designed task is given by the famous Amdahl's law (Amdahl, 1967).

$$Speedup = \frac{T_1}{T_p}, \tag{1.3}$$

where T_1 is the time taken by sequential programming and T_p is the time taken by parallel programming on the same task. Therefore,

$$Speedup = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{m}} = \frac{1}{\alpha + \frac{(1-\alpha)}{m}} \tag{1.4}$$

where, α is the proportion of sequential part of the task before parallelization, and m is the total number of processors for parallel computing.

Figure 1.4 gives a graphical representation of Amdahl's law. Even if we have an extremely efficient program with only a 5% sequential portion, the maximum speedup achieved by parallelization is 20 times. Also, the speedup is not increased much after the number of processor increase to more than 64 when 90% of operations are parallelizable in a program.

In addition to the theoretical upper bound of parallelization, reasonable design of the parallel program also contributes to the attainable speedup. Parallel execution time consists of both the time for the parallel computation by processors and the time for information exchange or synchronization between processors. This includes the time starting from the execution of action on the first processor to the completeness of tasks on all processors.

Firstly, the parallel execution time is affected by the distribution of work to processors.

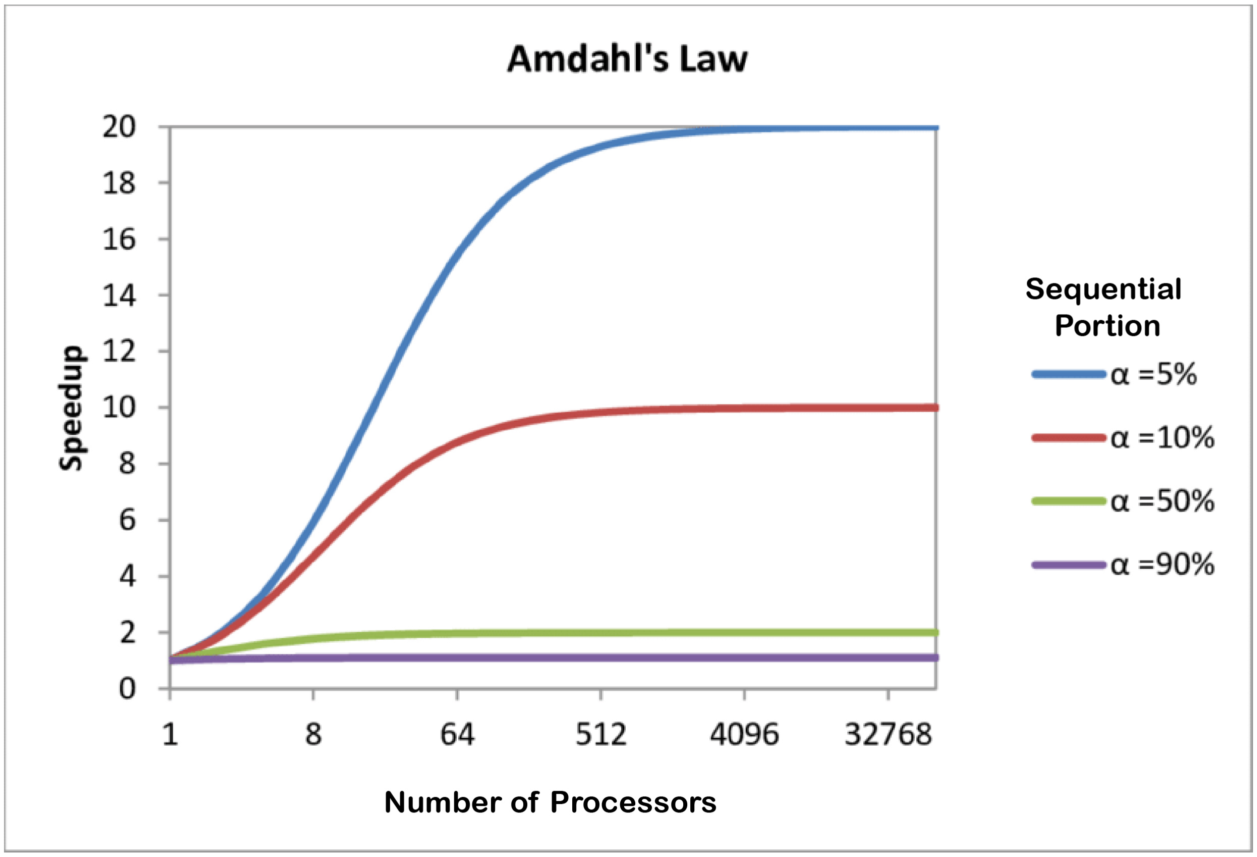

Figure 1.4: A graphical representation of Amdahl's Law. (Mashfiq, 2012)

If the parallelizable computation is too simple and could be carried out in a short time, work distribution may lead to more time than sequential execution, which means designing a parallel program is not worth the effort. Also, for a large number of subtasks, the work assigned to each processor may be quite small, then the time for thread creation, management and termination yields a significant portion of the overall time.

Secondly, the execution time is also influenced by information exchange or synchronization between the subtasks, which are the greatest obstacles to achieve an efficient parallel computing performance. Moreover, longer parallel execution time also results from designs with more idle time. Idle time refers to the time when only one or a small number of processors are active while others are doing nothing but waiting for an operation. For example, this occurs when some processors wait until all other processors have reached a certain point to exchange information, and only then do they continue the execution of the subsequent program code. This results in a sequentialization of execution, and the available parallelism cannot be used. Equally assigning workload to processors, which is called load balancing, can reduce the idle time in some degree. Avoiding frequent information exchanges between processors is also important to design efficient parallelism. In general, a good design and implementation of parallel programming helps obtain greater performance by making effective communication between subtasks.

1.6 Parallel Iterative Hybridized Threshold Clustering

The IHTC shows advantages in the improvement of runtime and memory usage by scaling down the size of original data. However, when dealing with a larger size of data (for example, $n > 10^8$), it is still costly in time and memory. In this study, we propose two paralleled versions of iterative hybridized threshold clustering and use simulated data to evaluate the efficiency in aspects of accuracy, runtime, and memory.

The structure of parallel programming is illustrated in Figure 1.5. First, the simulated data is randomly partitioned into batches without replacement. This partition can be completed in $O(n)$ time(Durstenfeld, 1964)(Knuth, 1997). The prototypes from different batches are collected and used for standard clustering algorithms, which is a step of sequential programming. Then, the labels assigned from the original clustering method are matched to the original data points, which is paralleled also based on batches. Because the labels from the clustering methods are different from those we assigned in simulation, a second step of matching labels is performed sequentially. Finally, the total runtime and memory usage are recorded to evaluate the paralleled algorithms performance. For supervised or simulated data, for which cluster labels are known, the accuracy may also be computed. As shown in Figure 1.5, the green parts are sequential while blue parts are parallel programmed.

This structure of implementation in Figure 1.5 is shared in these two paralleled versions of IHTC in this report. The differences are mostly on data synchronization. The first method to implement parallelism, belonging to distributed memory MPP system (Figure 1.3b), is executed on processors with its own copy of data. The second parallelism implementation is to use shared memory parallel programming from SMP system (Figure 1.3a), where all processors get access to data stored in shared memory. The performance of these two versions differ in the capacity of data magnitude, as well as runtime and memory, but the accuracy is similar and both close to the non-parallel algorithm.

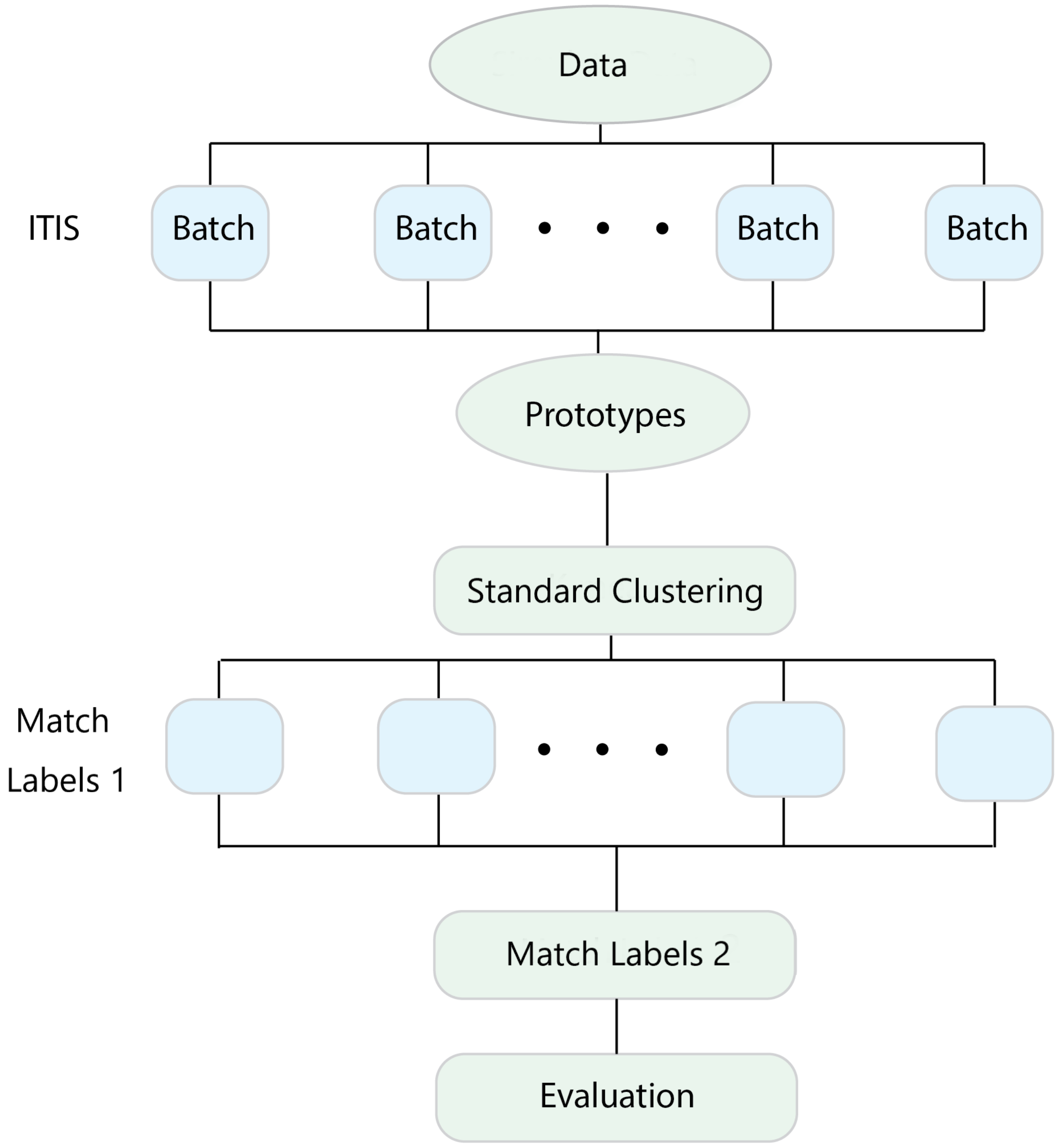

Figure 1.5: An illustration of iterative hybridized threshold clustering (IHTC) parallelism.

Chapter 2

Simulation

We evaluate the performance of paralleled IHTC using simulated data with two data sizes, 10^8 and 10^9. To be consistent with the previous study, the setting of simulation follows the descriptions in Luo et al. (2019). The data points are sampled independently from a mixture of three bivariate Gaussian distributions with parameters μ_j and Σ_j, $j = 1, 2, 3$. The pdf of mixture distribution is,

$$f(x) = 0.5f(x|\mu_1, \Sigma_1) + 0.3f(x|\mu_2, \Sigma_2) + 0.2f(x|\mu_3, \Sigma_3), \tag{2.1}$$

where $f(x|\mu_j, \Sigma_j)$ is the pdf of Gaussian distributions and the values of parameters are given as,

$$\mu_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mu_2 = \begin{bmatrix} 7 \\ 8 \end{bmatrix}, \mu_3 = \begin{bmatrix} 3 \\ 5 \end{bmatrix}, \tag{2.2}$$

$$\Sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0.5 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}, \Sigma_3 = \begin{bmatrix} 3 & 0 \\ 0 & 4 \end{bmatrix}. \tag{2.3}$$

Based on the three distributions, the data points are grouped into three clusters and labeled with $1, 2,$ or 3 as true labels. IHTC is performed with threshold $t^* = 2$ and the number of iteration $m = 3$ to generate prototypes. Then, k-means is applied with the number of clusters $k = 3$. The Euclidean distance is employed as the dissimilarity measure in this study. Accuracy, run time in seconds, speedup, and memory usage in gigabytes are used to evaluate the performance of two versions of IHTC parallelism, and are compared to those in sequential IHTC.

Simulation and parallel programming are implemented in the `R` language. The `Rdsm` and `parallel` packages are employed for implementing parallelism. The `scclust` package is used for threshold clustering (Sävje et al., 2017), and k-means clustering is performed using the `R` default `kmeans` function. All works are run on the Beocat cluster, a High-Performance Computing (HPC) cluster at Kansas State University. To make the results comparable, we only use the same node, wizard24, with 32 cores and a maximum memory of 1546 GB.

2.1 Implementation and Results

We propose two strategies for parallel processing in this study depending on whether the cores in the cluster share memory. The two versions of parallelism are programmed in `Rdsm` package and `parallel` package.

The code parallelism is done through one of two main approaches, sockets or forking. When setting up a cluster, which refers to launching a collection of cores or threads on the computer, the sockets approach creates a new version of `R` on each core, while the forking approach copies the entire original version of `R` to each new core. Even though forking is faster than sockets, the sockets approach is available for any system while forking only works on POSIX systems (e.g. Linux) but not Windows. The earliest packages for parallel processing in `R` are the `multicore` package (by forking) and `snow` package (by socket). These two powerful packages were merged into the `parallel` package in base `R`, and hence, the `parallel` package can create the clusters via either forking or sockets. In this study, we use the `makeCluster` function with default type, sockets, to make it more comparable to the `Rdsm` library which works only via network sockets.

2.1.1 Parallelism with `Rdsm` Package

Rdsm, named from distributed shared memory for R, creates a programming environment where nodes in a cluster share physical or conceptual memory. The variables shared by different processors must be stored in a special form of matrices using the bigmemory library. The dimension of a shared matrix needs to be specified beforehand, which is a constraint in `Rdsm`. The shared memory reduces memory usage by reducing redundant copies of data. However, there are still some temporal variables in the cache in each processor. In this study, we store the data, intermediate variables, and results in shared variables, which are easy to implement and update. However, due to the capacity of the bigmemory library, the maximum data size in simulation using `Rdsm` parallelism is 10^8.

To perform ITIS, we merge the ITIS functions to an executable paralleling function which is executed by every thread. Another paralleling function is coded for matching prototype labels from k-means to original data points. Then, we export these two paralleling functions and some parameter variables to the working environment of each thread using `clusterExport` function. The clusterEvalQ, which is from snow, is used to launch the threads to run the paralleling function. The `Rdsm` function `getidxs` can partition the workload into chunks according to the number of threads(cores), which is different from parallelism with the `parallel` package. This is achieved via the built-in `Rdsm` variable myinfo, which is a list of the total number of threads (nwrkrs) and the ID number of the thread (id). Then, the work chunk assigned to its core is copied and stored in a temporal variable, but not used directly from shared variables. Employing separate variables to store part of the data may help avoid costly cache misses and keep cache coherency by reducing the number of accesses to shared data.

After the first parallel computation, the prototypes are generated from ITIS. The coordinates of prototypes of threads are exported as a list. The labels to match the prototypes to original data points are stored in shared variables. However, the problem is that, since the speed of thread is different, the work chunks are not necessarily completed as the order of the ID of threads. So we need a variable to keep track of the results, for example, which chunk of the result is from which thread. In order to give a number to each chunk according to the order of completeness, we use a pair of synchronization function, `rdsmlock` and `rdsmunlock`. When a thread executes `rdsmlock`, it will lock the section between `rdsmlock` and `rdsmunlock` so that other threads can not access until it has finished this section and exited. Then the next thread will enter and repeat this process until all threads have executed it. In `Rdsm`, another function called `barr`, which makes the thread wait at this point until all threads have arrived, is also commonly used for synchronization, but is not used in this study.

After that, the k-means is applied on the prototypes, which is sequential programming. Then the second parallel processing is used to match the prototype labels assigned by k-means to the original data points. Because the original data points are associated with their prototypes and labels from k-means on prototypes need to be manually matched to original data points. However, the labels produced from k-means may not be the same as true labels we assign in simulation. So a matching step is taken, which is sequential because the task is too simple to parallel.

2.1.2 Parallelism with `Parallel` Package

The second method of parallelism is through the `parallel` package where each core has its own copy of data and variable. This causes more memory usage when we employ more cores. But it has the advantages of more flexibility in coding and larger capacity of data size. It can even handle data with a size of 10^9, but due to the available memory in the computing cluster, larger scalability of data is difficult to simulate. The parallel processing is very flexible with multiple modes. In this simulation, we use clusterApplyLB, which is a load balancing and dynamic way to launch parallel computing. With the help of snow function `splitIndices`, the data is partitioned into batches. Then each core works on one batch of data and once finished, it continues to take the next available batch. As mentioned in `Rdsm` parallelism, the number of subtasks is constrained to be the number of cores. However, the number of subtasks or batches in the `parallel` package can be any number more than the number of cores. Thus, we set the number of batches to be $50, 100, 200, 500$ or the number of cores to compare the effect of number batches on the performance. As a whole, the entire procedure is almost the same as that in `Rdsm` aside from data structures for input and output of IHTC.

After parallel computing, the intermediate results are exported as a list of lists. As a result, output from the same batch are tied in a list and it is not necessary to track batch unlike `Rdsm`. However, unlisting output by according to different results (such as prototype coordinates, prototype labels, number of prototypes from different processors) becomes more challenging than `Rdsm` implementation.

2.1.3 Results

In this study, we design the parallelism with the combinations of different number of cores and number of batches using `Rdsm` and `parallel` packages. The data size varies from 10^8 to 10^9. `Rdsm` parallelism only works on the data with a size of 10^8 due to the constraint of bigmemory library. The accuracy, run time in seconds and memory usage in GB are given in Table 2.1, 2.2, and 2.3. The speedup is the ratio of the average runtime of non-parallel algorithm and average runtime of parallel design.

To maintain consistency across simulations, we attempt simulations using the same node, Wizard24, on the Beocat research cluster. However, we are not able to get as many replications of experiments because we do not have priority access to this node. Additionally, due to the large size of data in this study, the requirements of number of cores and memory are larger, and execution time is longer compared to other users, which also makes it difficult to get access to computing resource to complete the simulations. Hence, the number of replicates varies among simulations and these results are obtained with substantial effort. Fortunately, the small standard deviations indicate that the results are stable to reproduce.

From the results in Table 2.1 and Figure 2.1, in general, the accuracy tends to decrease with an increase in the number of batches, except simulations in `Rdsm` and two combinations in `parallel`, 8 cores with 8 batches, and 16 cores with 16 batches. These five designs of parallel processing even show higher accuracy than that of the non-parallel algorithm. We can not find a good explanation for accuracy increase at this time, but this makes these designs good candidates for parallelism. Besides, the accuracy depends on the number of batches regardless of the number of cores. Potential interpretation is that sub-grouping data may affect the algorithm accuracy theoretically while the number of cores only represents the productivity of the implementation.

The results in Figure 2.2, Table 2.1 and 2.2 indicate that, for a fixed number of cores, the runtime is shortened as the number of batches increases. When number of batches is fixed, we found that speedup increases and then decreases as the number of cores increases. For both $n = 10^8$ and $n = 10^9$, the maximum speedup occurs using 8 cores with 500 batches. The maximum speedup for $n = 10^8$ is 3.82 and for 10^9 is 4.36. This is reasonable because, with more cores, the execution time becomes longer due to longer synchronization time caused by more frequent communications between threads. By comparing the speedup of data size 10^8 and 10^9, the parallelism performs better when dealing with larger scale data.

`Rdsm` performs better than `parallel` in terms of memory usage. In `parallel` simulations, more cores requires more copies of data, leading to increased memory usage. So parallelism design, for example, the choice of package, number of cores and batches, should be determined by our requirements of saving time or memory as well as balance between speedup and accuracy performance.

2.2 Discussion

On average, parallelized IHTC provides more than 3 times speedup over the original algorithm with a relatively small cost of additional memory. There is still space to improve these implementations of parallelization in terms of time and memory.

The first improvement involves the `R` package `Itis` (Luo et al., 2019) used to implement ITIS, which is a non-public early version of the package designed for massive data. At the first try of parallelism, we failed to install the local `Itis` package successfully in lynx system of Beocat. So we need to code the functions in `Itis` into our functions. The problem is that

10^8	Core	Batch	Accuracy	Run Time (second)	Speedup
Non-parallel	-	-	0.9238708 (0)	1372.44 (84.53)	-
`Rdsm`	4	-	0.9238789 (0)	550.41 (18.92)	2.49
	8	-	0.9239127 (0)	444.37 (12.75)	3.09
	16	-	0.9238757 (0)	418.22 (40.03)	3.28
`Parallel`	4	4	0.9238665 (0)	490.28 (3.07)	2.80
		50	0.9238676 (0)	423.17 (1.99)	3.24
		100	0.9238304 (0)	406.75 (7.69)	3.37
		200	0.9237778 (0)	390.53 (4.55)	3.51
		500	0.9237962 (0)	374.23 (4.18)	3.67
	8	8	0.9238811 (0)	409.44 (5.57)	3.35
		50	0.9238676 (0)	386.47 (4.15)	3.55
		100	0.9238304 (0)	371.14 (3.83)	3.70
		200	0.9237778 (0)	364.18 (1.33)	3.79
		500	0.9237962 (0)	359.52 (0.79)	3.82
	16	16	0.9238865 (0)	421.06 (5.51)	3.26
		50	0.9238676 (0)	406.76 (2.76)	3.37
		100	0.9238304 (0)	396.84 (2.00)	3.46
		200	0.9237778 (0)	396.95 (1.44)	3.46
		500	0.9237962 (0)	387.64 (2.63)	3.54
	32	32	0.9238487 (0)	507.84 (22.47)	2.70
		50	0.9238676 (0)	515.74 (2.69)	2.66
		100	0.9238304 (0)	499.80 (3.45)	2.75
		200	0.9237778 (0)	487.50 (8.47)	2.82
		500	0.9237962 (0)	482.21 (7.04)	2.85

Table 2.1: Average accuracy and run time of the simulation with data size of 10^8. The number of cores and number of batches are given for each simulation. The average accuracy, run time in second and speedup is used to evaluate the performance of parallelism. Dashed lines indicate that the values don't exist. The standard deviation is given in parentheses. The non-parallel result is based on 10 replicates of simulation. The `Rdsm` parallelism is from 8 replicates and `parallel` parallelism is from 4 replicates.

10^9	Core	Batch	Accuracy	Run Time (second)	Speedup
Non-parallel	-	-	0.9239411 (0)	18383.48 (1875.31)	-
`Parallel`	4	4	0.9239370 (0)	7065.36 (424.09)	2.60
		50	0.9239333 (0)	6235.92 (419.09)	2.95
		100	0.9239426 (0)	5997.54 (457.02)	3.07
		200	0.9239314 (0)	5869.90 (366.82)	3.13
		500	0.9239269 (0)	5520.57 (343.48)	3.33
	8	8	0.9239490 (0)	5148.96 (438.94)	3.57
		50	0.9239333 (0)	4701.94 (526.80)	3.91
		100	0.9239426 (0)	4603.99 (504.99)	3.99
		200	0.9239314 (0)	4484.60 (542.24)	4.10
		500	0.9239269 (0)	4214.93 (561.74)	4.36
	16	50	0.9239333 (-)	4975.82 (-)	3.69
		100	0.9239426 (-)	4697.25 (-)	3.91
		200	0.9239314 (-)	4673.92 (-)	3.93
		500	0.9239269 (-)	4486.62 (-)	4.10

Table 2.2: Average accuracy and run time of the simulation with data size of 10^9. The number of cores and number of batches are given for each simulation. The average accuracy, run time in second and speedup is used to evaluate the performance of parallelism. Dashed lines indicate that the values don't exist. The standard deviation is given in parentheses. The non-parallel result is based on 8 simulations. The `parallel` result with 4 or 8 cores is from 10 simulations and the result with 16 cores is from 1 simulation.

Memory (GB)	Core	Batch	10^8	10^9
Non-parallel	-	-	22.03 (0.40)	201.22 (0.65)
`Rdsm`	4	-	22.00 (0.20)	-
	8	-	36.76 (0.34)	-
	16	-	43.53 (0.86)	-
`Parallel`	4	50	35.80 (0.01)	340.74 (-)
		100	34.93 (0.17)	399.66 (-)
		200	35.45 (0.98)	343.31 (-)
		500	34.30 (0.026)	337.03 (-)
	8	50	51.37 (0.33)	492.13 (-)
		100	50.35 (0.04)	497.18 (-)
		200	51.93 (0.04)	497.18 (-)
		500	50.00 (0.06)	491.01 (-)
	16	50	76.53 (10.85)	796.16 (-)
		100	75.50 (9.96)	790.67 (-)
		200	77.38 (11.58)	800.42 (-)
		500	75.39 (9.87)	797.39 (-)
	32	50	133.28 (9.14)	-
		100	133.30 (9.17)	-
		200	139.02 (9.55)	-
		500	133.22 (9.04)	-

Table 2.3: Memory usage of the simulation with data size of 10^8 and 10^9. The number of cores and number of batches are given for each simulation. The memory usage in GB is used to evaluate the performance of parallelism. Dashed lines indicate that the values don't exist. The standard deviation is given in parentheses. The non-parallel results for size of 10^8 and 10^9 are based on 10 and 8 simulations, respectively. The `Rdsm` parallelism is from 3 replicates. `Parallel` parallelism results for size of 10^8 are from 3 simulations and result for size of 10^9 is from 1 simulation.

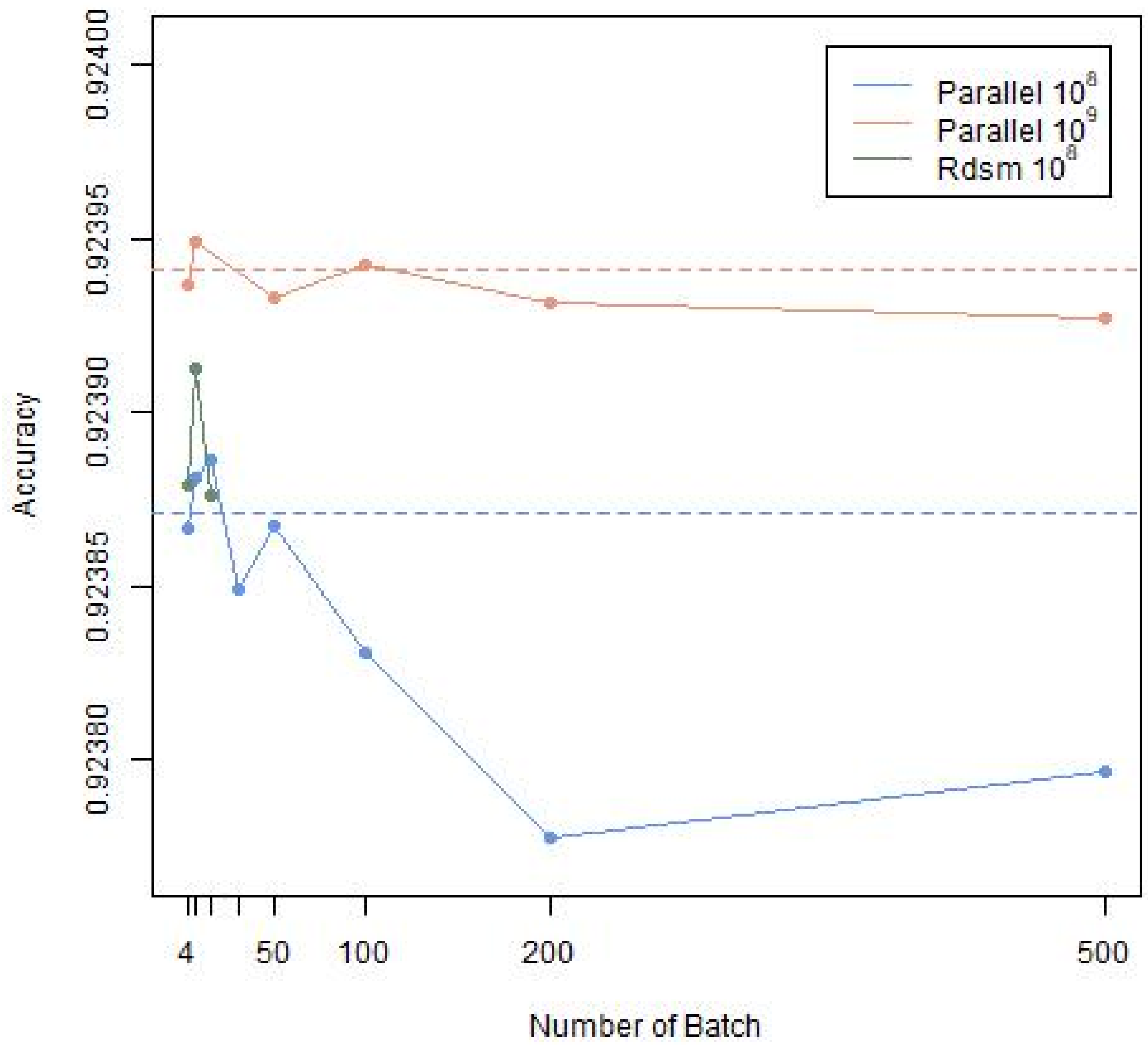

Figure 2.1: Accuracy of parallelism with different number of batches. Red solid line: parallelism using `parallel` package with data size of 10^8. Red dashed line: sequential algorithm with data size of 10^8. Blue solid line: parallelism using `parallel` package with data size of 10^9. Blue dashed line: sequential algorithm with data size of 10^9. Green solid line: parallelism using `Rdsm` package with data size of 10^8.

the `cpp` functions in `Itis` are unable to define beforehand because `cpp` function can't be exported into work environments in each thread. So we have to merge these `cpp` function into paralleling functions, which means `cpp` functions are redefined every time the thread

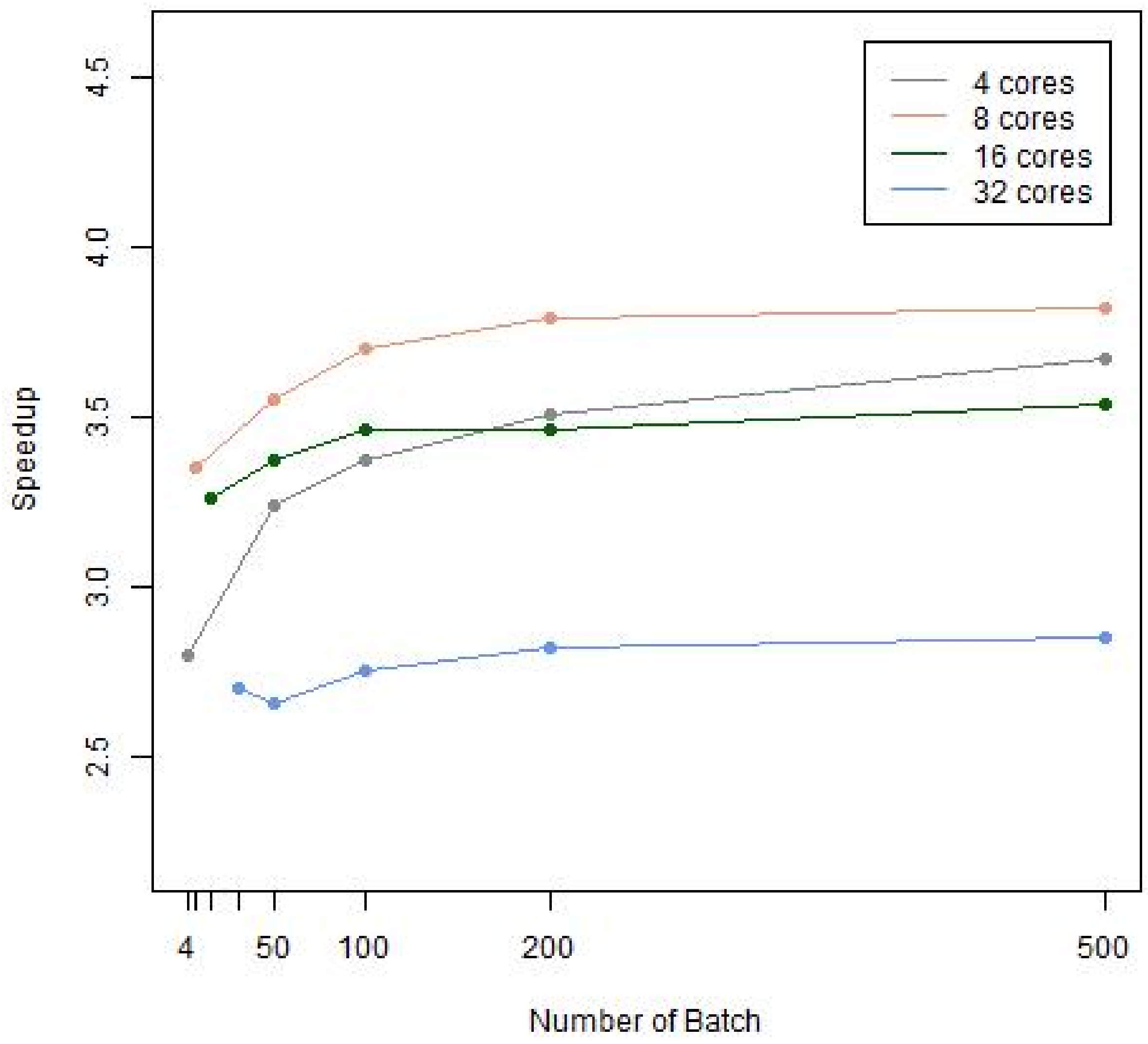

Figure 2.2: Speedup of `parallel` parallelism with different number of batches and cores with data size of 10^8. Grey line: `parallel` parallelism using 4 cores. Red line: `parallel` parallelism using 8 cores. Green line: `parallel` parallelism using 16 cores. Blue line: `parallel` parallelism using 32 cores.

begin a new batch. After simulations are done, we succeed to install the `Itis` package and we could call `Itis` inside paralleling functions which saves a lot of time. A second improvement is to reduce memory usage by removing the original data after data is copied to the thread environment and release intermediate variables once they will not be used.

By these two changes in coding, the memory usage of design with 16 cores and 50 batches for size 10^9 is 428.24 GB, which is 53.79% of the previous result. There is also a decrease in overall runtime despite the time cost required to release memory. The speedup of 16 cores with 50 batches for size 10^9 is 4.09 compared to the previous 3.69 speedup. However, the simulation results in this report are not based on this improved version due to the limited computing resources. Although we are not able to run the simulations using the improved version, the codes are attached in Appendix for future study.

Chapter 3

Conclusion

3.1 Conclusion

In this study, we proposed two parallelism methods for IHTC to deal with massive data. We implement parallel processing with `Rdsm` and `parallel` packages and test on designs with different number of cores and batches. The increased performance through parallelism is evaluated though accuracy, runtime, and memory usage.

From simulation results, we find that for the IHTC algorithm, the speedup is about 3 times with negligible decrease in accuracy. Parallel processing with 8 cores and 500 batches using `parallel` package achieves the highest speedup, 3.82 for 10^8 and 4.36 for 10^9. `Rdsm` parallelism for small scale data uses memory efficiently. We also conclude that a parallel programming design should create a proper number of threads to provide enough work for all cores to efficiently use the available computing resources.

3.2 Future Work

Although the performance, in terms of accuracy, of parallelized IHTC is comparable to sequential IHTC, we could consider improve the performance by pre-processing data more efficiently. A possible pre-processing is to cheaply sort or group the data points on one covariate and to perform sampling of units according to this sorting, which may yield better performance with, possibly, only a slight increase in runtime. Additionally, we can consider use other `R` paralleling packages, such as `foreach` in snow and `doParallel`, and compare their performances with packages in this study. Finally, implementing parallelization in `C` may provide another path to achieve reduce in runtime. The parallel programming tools in `C`, for example, `OpenMP`, may allow for flexible coding and efficient synchronizations, making it a good candidate for future work, especially since a large portion of the implementation of IHTC uses `C++`.

Bibliography

G Amadio, F Hariri, P Canal, L Duhem, M Gheata, G Cosmo, V Ivantchenko, O Shadura, SP Behera, G Folger, et al. Verification of electromagnetic physics models for parallel computing architectures in the geantv project. In *J. Phys. Conf. Ser.*, volume 898, page 042019, 2017.

Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

Ziv Bar-Joseph, David K Gifford, and Tommi S Jaakkola. Fast optimal leaf ordering for hierarchical clustering. *Bioinformatics*, 17(suppl_1):S22–S29, 2001.

Sanjay Chawla and Aristides Gionis. k-means–: A unified approach to clustering and outlier detection. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 189–197. SIAM, 2013.

Guillermo Corneio-Surez, Leonardo Van der Laat, Esteban Meneses, Javier Pacheco, and Mauricio Mora. Using parallel computing for seismo-volcanic event location based on seismic amplitudes. In *2018 IEEE 38th Central America and Panama Convention (CONCAPAN XXXVIII)*, pages 1–6. IEEE, 2018.

Pawel Czarnul. *Parallel programming for modern high performance computing systems.* Chapman and Hall/CRC, 2018.

Chris Ding and Xiaofeng He. K-nearest-neighbor consistency in data clustering: incorporating local information into global optimization. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 584–589. ACM, 2004.

Richard Durstenfeld. Algorithm 235: random permutation. *Communications of the ACM*, 7 (7):420, 1964.

Michael B Eisen, Paul T Spellman, Patrick O Brown, and David Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences*, 95(25):14863–14868, 1998.

Jorge Gonzalez-Dominguez and Maria J Martin. Mpigenenet: parallel calculation of gene co-expression networks on multicore clusters. *IEEE/ACM transactions on computational biology and bioinformatics*, 15(5):1732–1737, 2017.

Michael J Higgins, Fredrik Sävje, and Jasjeet S Sekhon. Improving massive experiments with threshold blocking. *Proceedings of the National Academy of Sciences*, 113(27):7369–7376, 2016.

Roger W Hockney and Chris R Jesshope. *Parallel Computers 2: architecture, programming and algorithms*. CRC Press, 2019.

Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

Hans-Peter Kriegel, Peer Kröger, Jörg Sander, and Arthur Zimek. Density-based clustering. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(3):231–240, 2011.

Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. The global k-means clustering algorithm. *Pattern recognition*, 36(2):451–461, 2003.

Huan Liu and Hiroshi Motoda. On issues of instance selection. *Data Mining and Knowledge Discovery*, 6(2):115, 2002.

Qiang Lu and Xin Yao. Clustering and learning gaussian distribution for continuous optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(2):195–204, 2005.

Jianmei Luo, ChandraVyas Annakula, Aruna Sai Kannamareddy, Jasjeet S. Sekhon, William Henry Hsu, and Michael Higgins. Hybridized threshold clustering for massive data, 2019.

Victor Malyshkin. Parallel computing technologies 2016. *The Journal of Supercomputing*, 73(2):607–608, 2017.

Xiancheng Mao, Bin Zhang, Hao Deng, Yanhong Zou, and Jin Chen. Three-dimensional morphological analysis method for geologic bodies and its parallel implementation. *Computers & Geosciences*, 96:11–22, 2016.

Khaled Mashfiq. *Nonlinear Earthquake Engineering Simulation using Parallel Computing System*. PhD thesis, 12 2012.

Julia Mullen, Chansup Byun, Vijay Gadepally, Siddharth Samsi, Albert Reuther, and Jeremy Kepner. Learning by doing, high performance computing education in the mooc era. *Journal of Parallel and Distributed Computing*, 105:105–115, 2017.

Sara Nasser, Rawan Alkhaldi, and Gregory Vert. A modified fuzzy k-means clustering using expectation maximization. In *2006 IEEE International Conference on Fuzzy Systems*, pages 231–235. IEEE, 2006.

Juan Nunez-Iglesias, Ryan Kennedy, Toufiq Parag, Jianbo Shi, and Dmitri B Chklovskii. Machine learning of hierarchical clustering to segment 2d and 3d images. *PloS one*, 8(8): e71715, 2013.

Victor Olman, Fenglou Mao, Hongwei Wu, and Ying Xu. Parallel clustering algorithm for large data sets with applications in bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(2):344–352, 2008.

Hae-Sang Park and Chi-Hyuck Jun. A simple and fast algorithm for k-medoids clustering. *Expert systems with applications*, 36(2):3336–3341, 2009.

Zhensheng Peng, Qingge Gong, Yanyu Duan, and Yun Wang. The research of the parallel computing development from the angle of cloud computing. In *Journal of Physics: Conference Series*, volume 910, page 012002. IOP Publishing, 2017.

JR Porter, Maxwell I Zimmerman, and Gregory R Bowman. Enspara: Modeling molecular ensembles with scalable data structures and parallel computing. *The Journal of chemical physics*, 150(4):044108, 2019.

Leonardo Marques Rocha, Fábio AM Cappabianco, and Alexandre Xavier Falcão. Data clustering as an optimum-path forest problem with applications in image analysis. *International Journal of Imaging Systems and Technology*, 19(2):50–68, 2009.

Fredrik Sävje, Michael J Higgins, and Jasjeet S Sekhon. Generalized full matching. *arXiv preprint arXiv:1703.03882*, 2017.

Mikhail Tolstykh, Rostislav Fadeev, Gordey Goyman, and Vladimir Shashkin. Further development of the parallel program complex of sl-av atmosphere model. In *Russian Supercomputing Days*, pages 290–298. Springer, 2017.

Appendix A

R Code

A.1 Improved Parallel

```
library('mvtnorm')
library('parallel')
library('Itis')
library('distances')
library('scclust')
library('pdist')
# n: sample size
# round: number of batches/subgroups
# chunksize: subsample size in each batch/subgroup
# c: number of cores
n=10^9
round=50
chunksize=n/round
c <- 16
# dostep1: a function to do threshold clustering
# ichunk: a list of sequence of index,
#each core get one sequence of index from the list at one time
dostep1 <- function(ichunk) {
  require(distances)
  require(scclust)
  mydat <- as.matrix(x[ichunk,])
  Itis::Itis_fct_nomid(mydat, t, m, d, chunksize)
}

# dostep2: matching the labels of all points to the prototypes
dostep2 <- function(ichunk2) {
  mclu<- unlist(mres[ichunk2])
  ll<- sum(nprotos[1:(ichunk2)])+1
  ul<- sum(nprotos[1:(ichunk2+1)])
  ft <- Itis::fastJoin2(mclu, ktag[ll:ul])
  return(ft)
}

# test: main function
test <- function(cls, n, d, t, m, k,chunksize) {
  #######step1 (parallelized)
  # paralle the threshold clustering step
  ichunk=splitIndices(n, ceiling(n/chunksize))
  # create a list of sequence of index
  clusterExport(cls,c("x","dostep1","n","d","t","m","k",
  "chunksize"))
  # export variables to worker environment
```

```
time0=proc.time()
mres <- clusterApplyLB(cls,ichunk,dostep1)
# execute dostep1 in each cores, mres is a list of b list(b
#is number of batches), each list contains of prototypes
#and labels in this batch
print(proc.time()-time0)
# time of parallelizabel time in step1
protos<- matrix(unlist(lapply(mres, function(u) t(u[1][[1]])
)),ncol=2,byrow=T)# prototype coordinates
nprotos<<- c(0, unlist(lapply(mres, function(u) length(unlist
(u[1]))/2))) # number of prototypes in each batch
########

#######step2 (sequential)
# run the k-means with prototypes
kfit <- kmeans(protos, centers = k, nstart = 10)
ktag<<- kfit$cluster # get kmeans labels for prototypes
mres<<- lapply(mres, function(u) u[-1] )
#remove prototype coordinates to save memory,only keep
#labels of all points according to prototypes,
#because we will pass mres to workers later
#write.table(mres,file='mres.txt', row.names=F)
#write.table(ktag,file='ktag.txt', row.names=F)
#write.table(nprotos,file='nprotos.txt', row.names=F)
########

#######step3 (parallelized)
```

```
# match kmeans labels of prototypes to all points
clusterExport(cls,c("mres","ktag","nprotos"))
# export variables to worker environment
nchunk=n/chunksize # nchunk is just number of batches
ichunk2=splitIndices(nchunk, ceiling(nchunk))
# create a list of sequence of index, each list
#has 1 number(index)
time0=proc.time()
mres2<- clusterApplyLB(cls,ichunk2,dostep2)
# execute dostep2 in each cores
print(proc.time()-time0)
# time of parallelizabel time in step3
mclusterlabel=cbind(unlist(lapply(mres2,function(u) t(u))),y)
#combine labels of kmeans and true labels of all data points
#write.table(mclusterlabel,file='mclusterp1.txt',row.names=F)
########
rm(list=setdiff(ls(),c("mclusterlabel","k")))

#######step4 (sequential)
# match the true lables and labels from k-means
match <- rep(0,k)# k is the number of different true labels,
#k is 3 in our simulation
for (i in 1:k){
  match[i] <- which.max(as.matrix(table(
  c(1:k,mclusterlabel[mclusterlabel[,2]==i,1]))))
  # for points with true label of i,
  #find the most frequent predicted label
  }
  print(match)
  mclusterlabel[,1] <- 0 - mclusterlabel[,1]
  # change the predicted label to negative value, to avoid
  #error when replacing the labels in next loop
  for (i in 1:k){
    mclusterlabel[mclusterlabel[,1]==-match[i],1] <- i
    # replace the predicted label with matching label
  }
  ########

  ######## rest part (sequential)
  #write.table(mclusterlabel,file='clp1.txt',row.names=F)
  ctab=table(mclusterlabel[,1],mclusterlabel[,2],
  dnn=c("Predicted","True"))
  # create a confusion matrix
  #write.table(ctab,file='ctabp1.txt')
  acc=sum(diag(ctab))/n # calculate the accuracy
  ########
  return(acc)
}

# simulation
d=2; t=2; m=3; k=3
x0<-cbind(rbind(rmvnorm(0.5*n,mean=c(1,2),sigma=matrix
(c(1,0,0,0.5),ncol=2,byrow=T)), rmvnorm(0.3*n,mean=c(7,8),
sigma=matrix(c(2,0,0,1),ncol=2,byrow=T)),rmvnorm(0.2*n,mean
=c(3,5),sigma=matrix(c(3,0,0,4),ncol=2,byrow=T))),c(rep
(1,0.5*n),rep(2,0.3*n),rep(3,0.2*n)))
x0<- x0[sample.int(n),]
x<- x0[,1:2]
y<- x0[,3]
rm(x0)# remove x0 to save memory
cls=makeCluster(c)# make a cluster of cores

# call the main function and record time
tim0=proc.time()
acc=test(cls, n, d, t, m, k, chunksize)
(proc.time()-tim0)# total time(data simulation not included)
```

A.2 Rdsm

```
library('mvtnorm')
library('Rdsm')
library('parallel')
library('Rcpp')
library('distances')
library('scclust')
library('pdist')
##############################
##    parallel function     ##
##############################
# The function dostep1 generates the prototypes in shared
#memory parallelization using package 'Rdsm' and 'parallel'.
```

```
# The function dostep2 matches the cluster labels of original
#points to the labels of prototypes by k-means clustering.
# The output includes mclusterlabel and proto in main function.
# mclusterlabel is a n by 4 shared matrix,
#1st column is an index for tasks,
#2nd column is index for myproto in its own chunk of prototypes
#3rd column is cluster label assigned by simulation,
#4th column is the predicted label.
# proto is a f by 3 matrix from mapreduce,
#       f is the number of prototypes,
#       the first 2 columns are prototypes,
#       3rd column is an index for tasks.
dostep1 <- function(n,t,d,m,dat,mclusterlabel,cumm) {
  require(parallel)
  require(Rcpp)
  require(distances)
  require(scclust)
  require(RcppAlgos)
  cppFunction('NumericMatrix␣fastAgg(NumericMatrix␣orgMeans,␣In
long␣catLeng␣=␣cats.length();
int␣numCols␣=␣orgMeans.ncol();
long␣numCats␣=␣max(cats);
IntegerVector␣catSize(numCats+1);
NumericMatrix␣aggMeans(numCats+1,numCols);

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for(int␣j␣=␣0;␣j␣<␣numCols;␣j++){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for(long␣i␣=␣0;␣i␣<␣catLeng;␣i++␣){
```

```
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//Different␣instructions␣if␣j␣=␣0␣and␣if␣j␣nc
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if(j␣==␣0){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣catSize[cats[i]]++;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣aggMeans(cats[i],j)␣=␣(double)(catSize[cats[i
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}else{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣aggMeans(cats[i],j)␣=␣(double)aggMeans(cats[i
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣aggMeans;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}')
  cppFunction('IntegerVector␣fastJoin2(IntegerVector␣mer1,␣Inte
long␣numN␣=␣mer1.length();
IntegerVector␣mer3(numN);

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for(long␣j␣=␣0;␣j␣<␣numN;␣j++){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣mer3(j)=mer2(␣mer1(j)␣);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣mer3;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}')
  myidxs <- getidxs(n)
  mclusterlabel[myidxs,3] <- dat[myidxs,3]
  mydat <- as.matrix(dat[myidxs,1:2])
  n1 <- length(myidxs)

  clusterlabel <- rep(NA, n1)
  for(i in 1:m){
    my_dist <- distances(mydat)
    my_clustering_new <- sc_clustering(my_dist, t)
    aggdata_old <- fastAgg(as.matrix(mydat), my_clustering_new)
    if(i == 1){
      clusterlabel <- as.integer(my_clustering_new)
    }
    else{
      clusterlabel <- fastJoin2(my_clustering_old,
      my_clustering_new)
    }
    mydat <- aggdata_old
    my_clustering_old <- clusterlabel
  }
  mclusterlabel[myidxs,2] <- clusterlabel
  myproto <- mydat
  mynpro <- nrow(myproto)
  rdsmlock('lck')
  cumm[1,1]<- cumm[1,1] + 1
  cumm[1,2]<- cumm[1,2] + mynpro
  cum=cumm[1,1]
  rdsmunlock('lck')
  mclusterlabel[myidxs,1] <- cum
  myproto <- cbind(myproto,rep(cum,mynpro))
  return(myproto)
}

dostep2 <- function(klabel,mclusterlabel) {
  require(parallel)
  require(Rcpp)
  cppFunction('IntegerVector␣fastJoin2(IntegerVector␣mer1,␣Inte
␣␣␣␣␣␣␣␣␣␣␣␣␣␣//mer1␣is␣original␣data␣matrix
␣␣␣␣␣␣␣␣␣␣␣␣␣␣//mer2␣in␣final␣cluster
␣␣␣␣␣␣␣␣␣␣␣␣␣␣//This␣function␣want␣to␣perform␣inner␣join
␣␣␣␣␣␣␣␣␣␣␣␣␣␣//Store␣dimensons␣of␣the␣for␣loop
␣␣␣␣␣␣␣␣␣␣␣␣␣␣long␣numN␣=␣mer1.length();
␣␣␣␣␣␣␣␣␣␣␣␣␣␣IntegerVector␣mer3(numN);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣for(long␣j␣=␣0;␣j␣<␣numN;␣j++){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣mer3(j)=mer2(␣mer1(j)␣);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣mer3;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣}')
  mywok <- myinfo$id
  mklabel <- klabel[klabel[,1]==mywok,2]
  mcluster <- mclusterlabel[mclusterlabel[,1]==mywok,2]
  ft <- fastJoin2(mcluster, mklabel)
  mclusterlabel[mclusterlabel[,1]==mywok,4] <- ft
  return(0)
}

##      main function       ##
# cumm is a 1 by 2 shared matrix,
#      1st number is accumulated number of tasks,
#      2nd number is accumulated number of prototypes.
# klabel is a f by 2 shared matrix,
#         f is the number of prototypes,
#         1st column is an index for tasks,
#         2nd column is labels by kmeans clustering.
# ctab is a tabel for cluster labels, the rows are simulated
#label, the columns are predicted labels.
#       is exported in ctab7.txt.
# acc is the prediction accuracy.
#      is exported in ctab7.txt.
test <- function(datt,cls, n, d, t, m, k) {
  library('parallel')
  library('Rdsm')
  library('Rcpp')
  library('distances')
  library('scclust')
  library('pdist')
  # make shared variables and lock in multiple cores
  mgrinit(cls)  # initial the cores
  makebarr(cls) # shared bar
  mgrmakevar(cls,"dat",n,d+1)#shared variable must be a matrix
  mgrmakevar(cls,"mclusterlabel",n,4)
  mgrmakevar(cls,"cumm",1,2)
  mgrmakelock(cls,"lck") # shared lock
  dat[,]<- datt # initial value of shared varibales
  cumm[,]<- 0
  clusterExport(cls,c("dostep1","dostep2","n","d","t","m","k"))
  # export variables to worker environment
  mproto <- clusterEvalQ(cls,dostep1(n,t,d,m,dat,mclusterlabel,
  cumm)) # execute dostep1 in each cores
  proto <- matrix(unlist(lapply(mproto, function(u) c(t(u)))),
  ncol=d+1, byrow=T) # mapreduce
  print(cumm[,])
  kfit <- kmeans(proto[,1:2], centers = k, nstart = 10)
  # k-means clustering
  mgrmakevar(cls,"klabel",cumm[1,2],2)
  klabel[,1] <- proto[,3]
  klabel[,2] <- kfit$cluster
  clusterEvalQ(cls,dostep2(klabel,mclusterlabel))
  # execute dostep2 in each cores
  #write.table(mclusterlabel[,],file='clusterlabelbef1.txt',
  row.names=F)
  #ctab0=table(mclusterlabel[,4],mclusterlabel[,3],dnn=
  c("Predicted","True"))
  #write.table(ctab0,file='ctab0.txt')
  match <- rep(0,k)
  # match the lables of simulation and k-means
  for (i in 1:k){
    match[i] <- which.max(as.matrix(table(c(1,2,3,
    mclusterlabel[mclusterlabel[,3]==i,4]))))
  }
  print(match)
  mclusterlabel[,4] <- 0 - mclusterlabel[,4]
  for (i in 1:k){
    mclusterlabel[mclusterlabel[,4]==-match[i],4] <- i
  }
```

```
  #write.table(mclusterlabel[,],file='clusterlabelaft1.txt',
  row.names=F)
  ctab=table(mclusterlabel[,4],mclusterlabel[,3],dnn=
  c("Predicted","True"))
  #write.table(ctab,file='ctab7.txt')
  acc=sum(diag(ctab))/n
  print(acc)
  return(acc)
}

##  simulate, test, result ##
# simulation
d=2; t=2; m=3; k=3
n=10^8
c1 <- 4
c2 <- 8
c3 <- 16
c4 <- 32
x<-cbind(rbind(rmvnorm(0.5*n,mean=c(1,2),sigma=matrix
(c(1,0,0,0.5),ncol=2,byrow=T)), rmvnorm(0.3*n,mean=c(7,8),
sigma=matrix(c(2,0,0,1),ncol=2,byrow=T)),rmvnorm(0.2*n,mean
=c(3,5),sigma=matrix(c(3,0,0,4),ncol=2,byrow=T))),c(rep
(1,0.5*n),rep(2,0.3*n),rep(3,0.2*n)))
datt<- x[sample.int(n),]

rm(list=setdiff(ls(),c("datt","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
```

```
cls1=makeCluster(c1)
tim0=proc.time()
acc=test(datt,cls1, n, d, t, m, k)
(tim1=proc.time()-tim0)
stopCluster(cls1)

rm(list=setdiff(ls(),c("datt","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls2=makeCluster(c2)
tim0=proc.time()
acc=test(datt,cls2, n, d, t, m, k)
(tim1=proc.time()-tim0)
stopCluster(cls2)

rm(list=setdiff(ls(),c("datt","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls3=makeCluster(c3)
tim0=proc.time()
acc=test(datt,cls3, n, d, t, m, k)
(tim1=proc.time()-tim0)
stopCluster(cls3)
```

A.3 Parallel

```
# traditional parallelization
# chunksize=n/batch
# use number of cores: 4, 8, 16, 32
# use number of batches: number of cores, 50, 100, 200, 500
# then chunksize is, n/number of cores,n/50,n/100,n/200,n/500
library('mvtnorm')
library('parallel')
library('Rcpp')
library('distances')
library('scclust')
library('pdist')

##    parallel function    ##
# The function dostep1 generates the prototypes in
#parallelization using package 'parallel'.
# The function dostep1 returns a list of list after
#parallel, the outer list is results from different tasks,
# note: the number of tasks is not the number of cores,
#but the number of batch.
# inside each list (result from each task), there are
#3 elements to form a list,they are,
# mydat: generated prototypes
# clusterlabel: label to match the original data point to
#the prototypes, the label is only valid in each own task.
# npro: the number of prototypes generated from each task
# The function dostep2 matches the cluster labels of original
#points to the labels of prototypes assigned by k-means.
# The output of function dostep2 is the predicted cluster
#for original data point
dostep1 <- function(ichunk) {
  require(parallel)
  require(Rcpp)
  require(distances)
  require(scclust)
  require(RcppAlgos)
  cppFunction('NumericMatrix␣fastAgg(NumericMatrix␣orgMeans,␣In
␣␣␣␣␣␣␣␣␣␣␣␣␣␣long␣catLeng␣=␣cats.length();
␣␣␣␣␣␣␣␣␣␣␣␣␣␣int␣numCols␣=␣orgMeans.ncol();
␣␣␣␣␣␣␣␣␣␣␣␣␣␣long␣numCats␣=␣max(cats);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣IntegerVector␣catSize(numCats+1);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣NumericMatrix␣aggMeans(numCats+1,numCols);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣for(int␣j␣=␣0;␣j␣<␣numCols;␣j++){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for(long␣i␣=␣0;␣i␣<␣catLeng;␣i++␣){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//Different␣instructions␣if␣j␣=␣0␣and␣if␣j␣nc
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if(j␣==␣0){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣catSize[cats[i]]++;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣aggMeans(cats[i],j)␣=␣(double)(catSize[cats[i
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}else{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣aggMeans(cats[i],j)␣=␣(double)aggMeans(cats[i
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣aggMeans;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣}')
  cppFunction('IntegerVector␣fastJoin2(IntegerVector␣mer1,␣Inte
␣␣␣␣␣␣␣␣␣␣␣␣␣␣long␣numN␣=␣mer1.length();
␣␣␣␣␣␣␣␣␣␣␣␣␣␣IntegerVector␣mer3(numN);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣for(long␣j␣=␣0;␣j␣<␣numN;␣j++){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣mer3(j)=mer2(␣mer1(j)␣);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣mer3;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣}')
  mydat <- as.matrix(x[ichunk,])
  n1 <- length(ichunk)
  clusterlabel <- rep(NA, n1)
  for(i in 1:m){
    my_dist <- distances(mydat)
    my_clustering_new <- sc_clustering(my_dist, t)
    aggdata_old <- fastAgg(as.matrix(mydat), my_clustering_new)
    if(i == 1){
      clusterlabel <- as.integer(my_clustering_new)
    }
    else{
      clusterlabel <- fastJoin2(my_clustering_old,
      my_clustering_new)
    }
    mydat <- aggdata_old
    my_clustering_old <- clusterlabel
  }
  npro=nrow(mydat)
  clusterlabel=cbind(clusterlabel,y[ichunk])
  return(list(mydat, clusterlabel, npro))
}
dostep2 <- function(ichunk2) {
  require(parallel)
  require(Rcpp)
  cppFunction('IntegerVector␣fastJoin2(IntegerVector␣mer1,␣Inte
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣long␣numN␣=␣mer1.length();
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣IntegerVector␣mer3(numN);

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for(long␣j␣=␣0;␣j␣<␣numN;␣j++){
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣mer3(j)=mer2(␣mer1(j)␣);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣mer3;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}')
  mclu<- matrix(unlist(mres[ichunk2]),ncol=2,byrow = FALSE)
  ll<- sum(nprotos[1:(ichunk2)])+1
  ul<- sum(nprotos[1:(ichunk2+1)])
  ft <- fastJoin2(mclu[,1], ktag[ll:ul])
  mclu[,1]<- ft
  return(mclu)
}
##      main function      ##
test <- function(cls, n, d, t, m, k,chunksize) {
  library('parallel')
  library('Rcpp')
  library('distances')
  library('scclust')
  library('pdist')
  ichunk=splitIndices(n, ceiling(n/chunksize))
  clusterExport(cls,c("x","y","dostep1","dostep2","n","d","t",
  "m","k"))  # export variables to worker environment
  mres <- clusterApplyLB(cls,ichunk,dostep1)
  # execute dostep1 in each cores
  protos<- matrix(unlist(lapply(mres, function(u) t(u[1][[1]]))
  nprotos<<- c(0, unlist(lapply(mres, function(u) u[3])))
  kfit <- kmeans(protos, centers = k, nstart = 10)
  ktag<<- kfit$cluster
  mres<<- lapply(mres, function(u) u[-c(1,3)] )
  clusterExport(cls,c("mres","ktag","nprotos"))
  nchunk=length(ichunk)
  ichunk2=splitIndices(nchunk, ceiling(nchunk))
  mres2<- clusterApplyLB(cls,ichunk2,dostep2)
  mclusterlabel<- matrix(unlist(lapply(mres2, function(u) t(u)
  )),ncol=2,byrow=T)
  match <- rep(0,k)
  # match the lables of simulation and k-means
  for (i in 1:k){
    match[i] <- which.max(as.matrix(table(c(1:k,
    mclusterlabel[mclusterlabel[,2]==i,1]))))
  }
  print(match)
  mclusterlabel[,1] <- 0 - mclusterlabel[,1]
  for (i in 1:k){
    mclusterlabel[mclusterlabel[,1]==-match[i],1] <- i
  }
  #write.table(mclusterlabel,file='clusterlabelp1.txt',
  row.names=F)
```

```
  ctab=table(mclusterlabel[,1],mclusterlabel[,2],dnn=
  c("Predicted","True"))
  #write.table(ctab,file='ctabp1.txt')
  acc=sum(diag(ctab))/n
  print(acc)
  return(acc)
}
##  simulate, test, result ##
# simulation
d=2; t=2; m=3; k=3
n=10^8
c1 <- 4
c2 <- 8
c3 <- 16
c4 <- 32
x0<-cbind(rbind(rmvnorm(0.5*n,mean=c(1,2),sigma=matrix
(c(1,0,0,0.5),ncol=2,byrow=T)), rmvnorm(0.3*n,mean=c(7,8),
sigma=matrix(c(2,0,0,1),ncol=2,byrow=T)),rmvnorm(0.2*n,mean
=c(3,5),sigma=matrix(c(3,0,0,4),ncol=2,byrow=T))),c(rep
(1,0.5*n),rep(2,0.3*n),rep(3,0.2*n)))
x0<- x0[sample.int(n),]
x<- x0[,1:2]
y<- x0[,3]

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls1=makeCluster(c1)
```

```
tim0=proc.time()
acc10=test(cls1, n, d, t, m, k, n/c1)
(tim11=proc.time()-tim0)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls1=makeCluster(c1)
tim0=proc.time()
acc1=test(cls1, n, d, t, m, k, n/50)
(tim11=proc.time()-tim0)
stopCluster(cls1)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls1=makeCluster(c1)
tim0=proc.time()
acc12=test(cls1, n, d, t, m, k, n/100)
(tim1=proc.time()-tim0)
stopCluster(cls1)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls1=makeCluster(c1)
tim0=proc.time()
acc13=test(cls1, n, d, t, m, k, n/200)
(tim1=proc.time()-tim0)
stopCluster(cls1)
```

```
rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls1=makeCluster(c1)
tim0=proc.time()
acc14=test(cls1, n, d, t, m, k, n/500)
(tim1=proc.time()-tim0)
stopCluster(cls1)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls2=makeCluster(c2)
tim0=proc.time()
acc20=test(cls2, n, d, t, m, k, n/c2)
(tim1=proc.time()-tim0)
stopCluster(cls2)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls2=makeCluster(c2)
tim0=proc.time()
acc21=test(cls2, n, d, t, m, k, n/50)
(tim1=proc.time()-tim0)
stopCluster(cls2)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
```

```
cls2=makeCluster(c2)
tim0=proc.time()
acc22=test(cls2, n, d, t, m, k, n/100)
(tim1=proc.time()-tim0)
stopCluster(cls2)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls2=makeCluster(c2)
tim0=proc.time()
acc23=test(cls2, n, d, t, m, k, n/200)
(tim1=proc.time()-tim0)
stopCluster(cls2)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls2=makeCluster(c2)
tim0=proc.time()
acc23=test(cls2, n, d, t, m, k, n/500)
(tim1=proc.time()-tim0)
stopCluster(cls2)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls3=makeCluster(c3)
tim0=proc.time()
acc30=test(cls3, n, d, t, m, k, n/c3)
```

```
(tim1=proc.time()-tim0)
stopCluster(cls3)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls3=makeCluster(c3)
tim0=proc.time()
acc31=test(cls3, n, d, t, m, k, n/50)
(tim1=proc.time()-tim0)
stopCluster(cls3)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls3=makeCluster(c3)
tim0=proc.time()
acc32=test(cls3, n, d, t, m, k, n/100)
(tim1=proc.time()-tim0)
stopCluster(cls3)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls3=makeCluster(c3)
tim0=proc.time()
acc33=test(cls3, n, d, t, m, k, n/200)
(tim1=proc.time()-tim0)
stopCluster(cls3)
```

```
rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls3=makeCluster(c3)
tim0=proc.time()
acc34=test(cls3, n, d, t, m, k, n/500)
(tim1=proc.time()-tim0)
stopCluster(cls3)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls4=makeCluster(c4)
tim0=proc.time()
acc40=test(cls4, n, d, t, m, k, n/c4)
(tim1=proc.time()-tim0)
stopCluster(cls4)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls4=makeCluster(c4)
tim0=proc.time()
acc41=test(cls4, n, d, t, m, k, n/50)
(tim1=proc.time()-tim0)
stopCluster(cls4)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls4=makeCluster(c4)
```

```
tim0=proc.time()
acc42=test(cls4, n, d, t, m, k, n/100)
(tim1=proc.time()-tim0)
stopCluster(cls4)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls4=makeCluster(c4)
tim0=proc.time()
acc43=test(cls4, n, d, t, m, k, n/200)
(tim1=proc.time()-tim0)
stopCluster(cls4)

rm(list=setdiff(ls(),c("x","y","n","d","t","m","k","c1","c2",
"c3","c4","dostep1","dostep2","test")))
cls4=makeCluster(c4)
tim0=proc.time()
acc44=test(cls4, n, d, t, m, k, n/500)
(tim1=proc.time()-tim0)
stopCluster(cls4)
```

A.4 Non-parallel

```
library('mvtnorm')
library(Rcpp)
library(distances)
library(scclust)
```

```
# Define cpp function #
cppFunction('NumericMatrix␣fastAgg(NumericMatrix␣orgMeans,␣Inte
long␣catLeng␣=␣cats.length();
int␣numCols␣=␣orgMeans.ncol();
long␣numCats␣=␣max(cats);
IntegerVector␣catSize(numCats+1);
NumericMatrix␣aggMeans(numCats+1,numCols);
for(int␣j␣=␣0;␣j␣<␣numCols;␣j++){
for(long␣i␣=␣0;␣i␣<␣catLeng;␣i++␣){
if(j␣==␣0){
catSize[cats[i]]++;
aggMeans(cats[i],j)␣=␣(double)(catSize[cats[i]]-1)/catSize[cats
}
else{
aggMeans(cats[i],j)␣=␣(double)aggMeans(cats[i],j)␣+␣(double)1/c
}}}
return␣aggMeans;
}')
cppFunction('IntegerVector␣fastJoin2(IntegerVector␣mer1,␣Intege
long␣numN␣=␣mer1.length();
IntegerVector␣mer3(numN);
for(long␣j␣=␣0;␣j␣<␣numN;␣j++){
mer3(j)=mer2(␣mer1(j)␣);
}
return␣mer3;
}')
```

```
# Itis function #
Itis_fct_nomid <- function(dat, t, m, d, n){
  clusterlabel <- rep(NA, n)
  for(i in 1:m){
    my_dist <- distances(dat)
    my_clustering_new <- sc_clustering(my_dist, t)
    aggdata_old <- fastAgg(as.matrix(dat), my_clustering_new)
    if(i == 1){
      clusterlabel <- as.integer(my_clustering_new)
    }
    else{
      clusterlabel <- fastJoin2(my_clustering_old,
      my_clustering_new)
    }
    dat <- aggdata_old
    my_clustering_old <- clusterlabel
  }
  return(list(dat, clusterlabel))
}

Km_itis_nomid <- function(proto, clusterlabel, k){
  kfit <- kmeans(proto, centers = k, nstart = 10)
  ft <- fastJoin2(clusterlabel, kfit$cluster)
  return(ft)
  #assign("finalcluster_threshold", ft, envir = globalenv())
}
```

```
##      main function       ##
test1 <- function(datt, n, d, t, m, k) {
  try <- Itis_fct_nomid(datt[,1:2], t, m, d, n)
  proto <- matrix(unlist(try[1]),ncol=2,byrow = FALSE)
  clusterlabel <- unlist(try[2])
  res <- Km_itis_nomid(proto, clusterlabel, k)
  tru <- datt[,3]
  match <- rep(0,k)
  # match the lables of simulation and k-means
  for (i in 1:k){
    match[i] <- which.max(as.matrix(table(c(1:k,res[tru==i]))))
  }
  print(match)
  res <- 0 - res
  for (i in 1:k){
    res[res==-match[i]] <- i
  }
  ctab=table(res,tru,dnn=c("Predicted","True"))
  #write.table(ctab,file='ctabn1_9.txt')
  acc=sum(diag(ctab))/n
  print(acc)
  return(acc)
}

## simulate, test, result ##
# simulation
```

```
d=2; t=2; m=3; k=3
n=10^8
x<-cbind(rbind(rmvnorm(0.5*n,mean=c(1,2),sigma=matrix
(c(1,0,0,0.5),ncol=2,byrow=T)), rmvnorm(0.3*n,mean=c(7,8),
sigma=matrix(c(2,0,0,1),ncol=2,byrow=T)),rmvnorm(0.2*n,mean
=c(3,5),sigma=matrix(c(3,0,0,4),ncol=2,byrow=T))),c(rep
(1,0.5*n),rep(2,0.3*n),rep(3,0.2*n)))
datt<- x[sample.int(n),]

rm(list=setdiff(ls(),c("datt","n","d","t","m","k","test1",
"Itis_fct_nomid","Km_itis_nomid","fastAgg","fastJoin2")))
# record time and accuracy in scn_tim7.txt
tim0=proc.time()
acc1=test1(datt, n, d, t, m, k)
tim1=proc.time()-tim0
print(tim1)
#cat(tim1, acc1, '\n', file='scn1_tim9.txt')
```