Text mining on High Performace Computing Cluster

Anh Tu Phan, Tuan Dat Nguyen

June 2021

The covid-19 outbreak in over the world increasingly become the most important research topic in last year. A huge amount of articles and publications have been published about many different angles of covid-19. It is paramount important for scientists to keep up with new information on the virus, but the most obstacle is a large number of articles that can not easily process by humans. It is necessary to using data mining algorithms to process this data. However, with this amount of data (GB), some normal serial text mining algorithms still need a lot of time to process.

Therefore, in our project, we will use the High Performace Computing (HPC) cluster to execute some text-mining-parallel algorithms in order to extract knowledge from a big data set of articles. We will cluster similar articles into groups. Then, over articles in each group, we will finding the set of words that frequently appear together in the most of articles.

The structure of this report is organized as follows: In Section 1 we will briefly introduce some data mining algorithms. The used dataset and problem definition is presented in Section 2. Our solution which contains five components and three tasks is described in Section 3. Section 4 present the detail implementation of three tasks in Section 3. The performance evaluation and result is described in Section 5 and Section 6 concludes our report.

1 Introduction

Clustering is a common Machine Learning technique that involves the grouping of data points to gain some valuable insights from our data by seeing what groups the data points fall into. There are several algorithms to solve the clustering problem. K-Means is probably the most well-known clustering algorithm. K-Means will repeatedly assign a point to the cluster and update the center point of each cluster by computing the mean value of points in the cluster. Mean shift clustering is a sliding-window-based algorithm that attempts to find dense areas of data points. The mean shift procedure was originally presented in 1975 by Fukunaga and Hostetler [1]. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based clustered algorithm similar to mean-shift proposed by Martin Ester et.al [2] in 1996. Given a set of points in some space, DBSCAN groups together points that are closely packed together (points with many nearby neighbors), marking them as outliers points that lie alone in low-density regions (whose nearest neighbors are too far away). To improve the drawbacks of K-Means which are the use of the mean value for the cluster center and the fails in cases where the clusters are not circular. Expectation-Maximization (EM) Clustering using Gaussian Mixture Models (GMM) assumes that the data points are Gaussian distributed and using an optimization algorithm called Expectation-Maximization to find the parameters of the Gaussian distribution. Another approach to solving the clustering problem is Hierarchical clustering. Hierarchical clustering algorithms fall into 2 categories: divisive (top-down) and agglomerative (bottom-up). In a bottom-up algorithm, initially, each point is a cluster and repeatedly combines the two nearest clusters into one until all clusters have been merged into a single cluster. In contrast, in a top-down algorithm, Initially, all data is in the same cluster, and the largest cluster is split until every object is separate.

Frequent Item Sets is a popular problem in data mining which is finding a set of items that frequently appear together in many baskets. Many algorithms have been proposed by different researchers to enhance the technique in the Frequent Item Sets problem. By using an iterative level-wise search technique to discover (k+1)-itemsets from k-itemsets, Apriori algorithm [3] mines frequent itemsets for generating Boolean association rules. PCY algorithm proposed to take advantage of the idle memory in pass 1 of the Apriori algorithm. Frequent Pattern Growth [4] compress the frequent itemsets into Frequent Pattern Tree to mine frequent itemsets without a costly candidate generation process. Equivalence Class

Transformation (EClaT) algorithm proposed by Zaki [5] and Transaction Mapping (TM) proposed by Song and Rajasekaran [6] use the vertical data format to mine frequent itemsets. TreeProjection proposed by Agarwal et al.

In this project, we will implement a parallel version of the K-Means algorithm to group similarity articles. After that, we will parallelize the Apriori algorithm which will extract a set of words that frequently appear together in each cluster.

2 Problem

We will use a very big and famous data set of covid-19 articles. *CORD-19 is a resource of over 400,000 scholarly articles, including over 150,000 with full text, about COVID-19, SARS-CoV-2, and related coronaviruses.* [7]

The structure of data [8] is described below (we minimize it only keep necessary data):

```
{
    "paper_id": <str>,  #40-character sha1 of the PDF
    "metadata": {
        "title": <str>,
        "abstract": [   #list of paragraphs in the abstract
            {
                "text": <str>,
            },
        ],
        "body_text": [  #list of paragraphs in full body
            {
                "text": <str>,
                "section": "Introduction"
            },
            ...
            {
                "text": <str>,
                "section": "Conclusion"
            }
        ],
}
```

With this data and base on some recent research on this topic, in this project, we will address two main problems:

1. Clustering (group) similarity articles.
   - **Input:** Content of a article. In this project we just use content of title, abstract in the article. Each content of one article will be one file in data foler input.
   - **Output:** K-clusters and each cluster contains articles which have a similar context.
2. Extracting most frequent words on each cluster.
   - **Input:** Articles in one cluster.
   - **Output:** Top K-words usually appear together in the same article over articles in one cluster.

3 Solution

3.1 Components

To address two main problems, our solution will contains five components shown in Figure 1, which are:

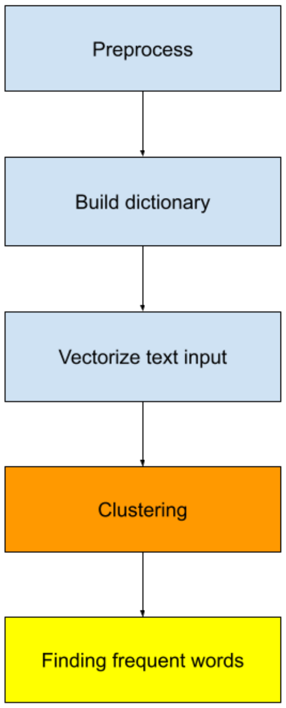

Figure 1: Solution components

1. Preprocess: Combining content of tile and abstract from input file to one document and then applying some preprocess on words of this document such as: removing stop, punctuation, ...
2. Build dictionary: construct an array of all words in the document and map each word to a number which is index of word in array.
3. Vectorize text input: Convert text input to numeric vector.
4. Clustering: Group document which are has similar meaning (the word in document is similar). We will implement parallel K-means clustering.
5. Finding frequent words: Find list of word that are frequent appear together in many document in one cluster. We will implement parallel Apriori algorithm in this step.

3.2 Task Graph

To address five components, we will implement three main tasks. The task graph is shown in Figure 2. In detail:

- Task 1 - Preprocess, Build dictionary, Vectorize text input: The output of this task is the dictionary of the set of articles and the set of vectors, which is the input for the next steps.
- Task 2 - Clustering: Given the vector from the previous task, we use the k-mean algorithm to cluster articles to k clusters.
- Task 3 - Task 3 - Finding frequent words: From the output of the 1st and 2nd task, using the Apriori algorithms, we find a list of words that is frequent appear together, which is the result of the problem.

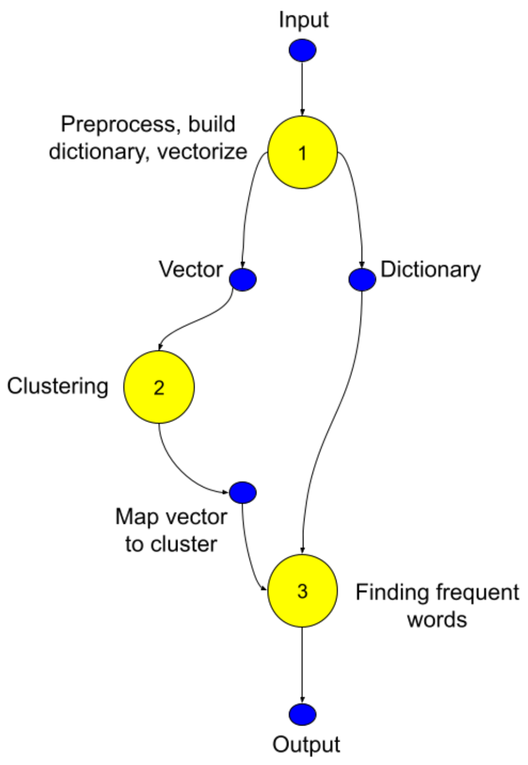

Figure 2: Task Graph

Task 1

- **Input** are raw files which are title, abstract, and body of articles
- **Output** contains two folders:
  - A Folder contains a set of vectors, each vector is a binary file.
  - A Folder contains a set of files, each file contains a part of a whole dictionary. In each file, an integer that represents the index of a word in the dictionary which will be mapped to a word.
- **Parameters**
  * -inputDataFolder: path to the raw files
  * -numFileInput: number of article files want to read from inputDataFolder
  * -vectorOutputFolder: path to vector output folder
  * -dictOutputFolder: path to dictionary output folder
  * -stopWordFilePath: path to stop word files

Task 2

- **Input** is a set of vector, each vector corespond to a binary file. The process will scan the input folder and looking for the indicated file type.
- **Output** of the clustering process is three files:
  - a backup for cluster mean name *k-mean-global-mean.txt*
  - a text file map each example with cluster.
  - a binary file map each example with cluster.
- **Parameters**

    * -k (–cluster) number of cluster.
    * -n number of process.
    * -ne (–size) number of example, make sure this number is multiple of the process number.
    * -mi (–maxiterator) maxium of clustering steps.
    * -d (–dimension) dimension of input vector.
    * -inputFolderPath path to inputer folder - this parameter is required.
    * -inputFileType indicate the file should process - this parameter is required.
    * -outputFileName indicate the output file name (only filename).
    * -nThreads number of thread using by openmp.
    * -nChunk number of thread using by openmp.

Task 3

- **Input**
    - The integer vectors in one cluster
    - Dictionary files in which integer value in vector will be mapped to a word
- **Output**: set of words that frequently appear together in input cluster
- **Parameters**
    * -dictInputFolder: path to dictionary files folder
    * -vectorInputFolder: path to vector files of one cluster
    * -numFileInput: number of vector want to read from vectorInputFolder
    * -sentenceSize: dimension of one input vector
    * -dictSize: number of words in dictionary
    * -dictWordPerFile: nubmer of words in one dictionary file
    * -supportThres: the minimum number of occurence to be considered as frequent
    * -maxNumberBasketSize: the maximum number set of words in one frequent set
    * -clusterId: id of cluster

4 Implementation

4.1 Preprocess, build dictionary and vectorize text input

To run mining algorithms, firstly we need to build dictionary which contains all different words from dataset. The overal steps is described in Algorithm 1.

Algorithm 1: Build dictionary

```
1 for process in COMM_WORLD do
2     read_doc_from_file();
3     pre_process_doc();
4     build_local_dict();
5 reduce_merge_dict();
```

- For each process
    - Each process will just read its own file (not read all dataset) (*read_doc_from_file*()).
    - The stop word (*the, where, when, ...*), punctuation (*. ! - , ...*), the word with len smaller than 3 are removed (*pre_process_doc*()).

  - We will use the set [9] structure to ensure that each word will occur one times in dictionary ($build\_local\_dict()$).

- After each process has its own local dictionary, we will merge all local dictionary to one global dictionary which contains all word in dataset ($reduce\_merge\_dict$). Instead of merging all local dictionary from other process at master process (Point-to-Point Communications), we will use the Collective Communications. Because of in MPI_Reduce and $MPI\_Allreduce$ function, the input and output data have to have the same dimension. Meanwhile, the dimension (number of words) of merged dictionary output of two dictionary input is different. By modifying the implementation of MPI_Reduce from [10], we will implement merge local dict based on tree-structured by just using MPI_Send and MPI_Recv. The Figure 3 show the example of implementation.

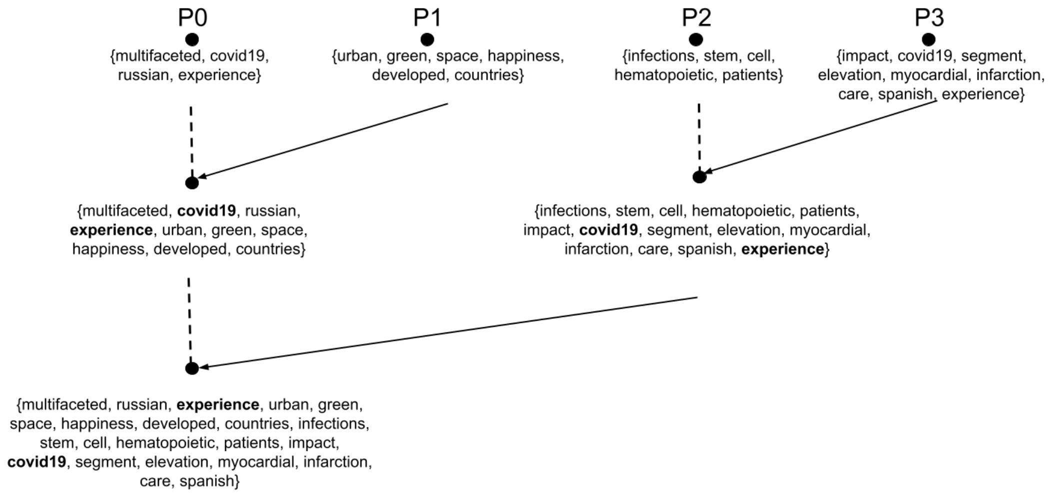
P0
{multifaceted, covid19, russian, experience}
P1
{urban, green, space, happiness, developed, countries}
P2
{infections, stem, cell, hematopoietic, patients}
P3
{impact, covid19, segment, elevation, myocardial, infarction, care, spanish, experience}
{multifaceted, **covid19**, russian, **experience**, urban, green, space, happiness, developed, countries}
{infections, stem, cell, hematopoietic, patients, impact, **covid19**, segment, elevation, myocardial, infarction, care, spanish, **experience**}
{multifaceted, russian, **experience**, urban, green, space, happiness, developed, countries, infections, stem, cell, hematopoietic, patients, impact, **covid19**, segment, elevation, myocardial, infarction, care, spanish}

Figure 3: Example of merge dictionary using reduction

After finishing building dictionary, we will broacast the dictionary to all process to build vector that represent for document. We will represent each input document in dataset as numeric vector with same dimension which is the maximum length of document in dataset. Eeach word in document will be assigned to index of its in dictionary. If length of document is smaller than maximun length, the unused element in vector will be -1. For example with final dictionary in Figure 3 and the maximum length of document is dataset is 10, the document *multifaceted covid19 russian experience* is represented as vector $[0, 15, 1, 2, -1, -1, -1, -1, -1, -1]$.

4.2 Clustering

we based on an algorithm proposed by Ajay Padoor Chandramohan [11].

Consider N data points each of them is vector and P.

1. Assign N/P data points to each processor.
2. Node 0 randomly choose K points and assigns them as cluster means and broadcast.
3. In each processor for each data point find membership using the cluster mean.
4. Recalculate local means for each cluster in each processor.
5. Globally broadcast all local means for each processor to find the global mean.
6. Go to step (3) and repeat until the number of iterations > a $max - iterator$ or number of points where membership has changed is less than a threshold %.

Split input for each process Because we scan the input folder so the number of vectors usally not in ratio with the number of processes.

```
 int start = 0;
  for (int j = 0; j < rank; j++)
  {
    if (j < number_of_element % comm_sz)
      start = start + (number_of_element / comm_sz + 1);
    else
      start = start + (number_of_element / comm_sz);
  }
  int finish = 0;
  if ((rank) < number_of_element % comm_sz)
    finish = start + number_of_element / comm_sz + 1;
  else
    finish = start + number_of_element / comm_sz;
  cout << "rank " << rank << " start, finish " << start << "," << finish << endl;
  int count = 0;
  double *local_a = new double[*local_n * *m];
  int i = 0;
  for (const auto &path : filenames)
  {
    if (count >= start && count < finish)
    {
      cout << "print from rank " << rank << " read file " << path << endl;
      double *tmp = get_array_from_file(rank, path);
      if (tmp != NULL)
        memcpy(&local_a[i * *m], tmp, *m * sizeof(double));
      i++;
      delete[] tmp;
    }
    count++;
  }
```

Distance between two vector The distance between two vectors was calculated by euclidean distance. In our implementation, we use OpenMP *for* directive and clause *reduction(+: s)* to process this function in parallel;

```
 double distance(double x[], double y[], int l_n)
{
  double s = 0;
#pragma omp parallel for num_threads(nThreads) schedule(static,nChunk) reduction(+ : s)
  for (int i = 0; i < l_n; i++)
    s += (x[i] - y[i]) * (x[i] - y[i]);
  return sqrt(s);
}
```

k-mean loop:

1. **Initial global mean**: The global mean init from master process $m * k$ first examples in which:

    - M is the dimention of input vector.
    - K is number of cluster

    and after that the global will be broadcast to other processes.

```
     double global_mean[m * k];
  memset(global_mean, 0, sizeof global_mean);
  if (rank == 0)
    for (int i = 0; i < k; i++)
      for (int j = 0; j < m; j++)
      {
        global_mean[i * m + j] = local_a[i * m + j];
      }
  int iterator = 0;
  double old_global_mean[m * k];
  // broadcast global mean
  MPI_Bcast(global_mean, k * m, MPI_DOUBLE, 0, comm);
```

2. **Stoping criteria**: The k-mean algorithmns will stop when two criteria was archived:

   - There is no change in global mean.
   - Reach the maxium iterator.

```
   int xd = 0;
  // check stop criteria
  if (iterator > 0)
  {
#pragma omp parallel for num_threads(nThreads) schedule(static,nChunk)
    for (int i = 0; i < m * k; i++)
      if (global_mean[i] != old_global_mean[i])
      {
#pragma omp critical
        xd = 1;
      }
  }
  else
    xd = 1;
  if (xd == 0 || iterator > max_iterator)
    break;
```

3. **Do not update the useless local mean vector**: When we reassign the local examples to each cluster by global mean, there is some cluster that there are no examples belong to. We should ignore these clusters, and update local in order to avoid trash numbers will be added to the global mean.

```
 for (int i = 0; i < k; i++)
   {
     if (use_cluster[i] == 0)
#pragma omp parallel for num_threads(nThreads)  schedule(static,nChunk)
       for (int j = 0; j < m; j++)
         local_mean[i * m + j] = global_mean[i * m + j];
   }
```

4. **Build the final cluster result:** In order to get the final cluster result, we should get the local example cluster from slave to master. However, the number of examples in each process is different, so we can not use *MPI_Gather*. Instead, we use another gather function named *MPI_Gatherv* which can gather array with different length.

```
 int displs[comm_sz];
  int rcounts[comm_sz];
  for (int i = 0; i < comm_sz; i++)
  {
    if (i == 0)
      displs[0] = 0;
    else
      displs[i] = displs[i - 1] + rcounts[i - 1];
    if (i < (n % comm_sz))
      rcounts[i] = n / comm_sz + 1;
    else
      rcounts[i] = n / comm_sz;
  }
  MPI_Gatherv(&d_cluster, local_n, MPI_INT, total_d_cluster, rcounts, displs, MPI_INT, 0,
              MPI_COMM_WORLD)
```

Build application In the clustering task, we implement by C++. In order to build the excutation file, we use command

```
mpic++ -std=c++17 -g -fopenmp -Wall -o kmean mpikmean.cpp
```

4.3 Finding frequent words

The Apriori is two-pass approach to solving finding frequent item (word) set problems. The key idea of this alogrithm is monotonicity which means that if set of items (words) I apperas at least s times, so does every subset.

- Pass 1: Read documents and count the occurrences of each individual item
- Items that appear greater than s times are the frequent items.
- Pass 2: Read documentes again and count only those pairs where both elements are frequent (from Pass 1)

In order to finding $k - tuples$ set of items, for each k, we construct two set of $k - tuples$. (The diagram of steps of apriori is shown in Figure 4)

- C_k is set of $k - tuples$ candidate which is contructed based on the $(k-1) - tuples$ frequent sets from the pass $k-1$
- L_k is the set of truly frequent $k - tuples$ that its occurrences in documents greater than s

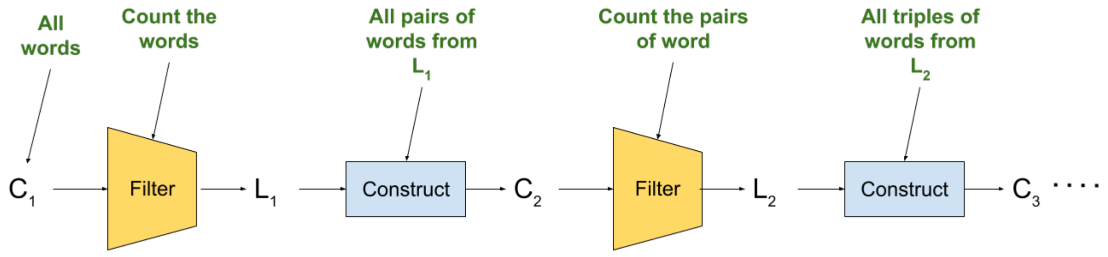

Figure 4: Apriori algorithm diagram

Base on the origin apriori, we will implement apriori in parallel version. In detail

1. Each process will read a chunk of document
2. Each process will count the local occurrence of $k - tuples$ set of words by reading its document
3. Then the global occurrence will be computed by using $MPI_AllReduce$ and filter the frequent $k - tuples$ to build L_k
4. Base on L_k, each process will construct C_{k+1} and then repeat step 1, 2, 3, 4 with $(k+1) - tuples$.

5 Performance evaluation and Result

In this section, we show the performance of the parallel-kmean algorithm in different sets and resources. We also compare the performance of two kinds of hybrid OpenMP-MPI and pure MPI parallel applications. We test our application with pure MPI version in two sizes of the problem:

- Problem 1: Number of documents is 50000.
- Problem 2: Number of documents is 100000.

node	core	process	problem 1	problem 2
1	1	1	7065.92	9224.2
2	1	2	3497.51	4778.18
4	1	4	1794.24	2646.23
8	1	8	946.459	1087.16
16	1	16	499.727	718.453

Table 1: Running time in different resources and size of problem

node	core	process	problem 1	problem 2
1	1	1	1	1
2	1	2	2.02027	1.93048
4	1	4	3.93811	3.48579
8	1	8	7.46564	8.48468
16	1	16	14.13956	12.83900

Table 2: Speedup

node	core	process	problem 1	problem 2
1	1	1	1	1
2	1	2	1.01014	0.96524
4	1	4	0.98453	0.87145
8	1	8	0.9332	1.06058
16	1	16	0.88372	0.80244

Table 3: Efficiency

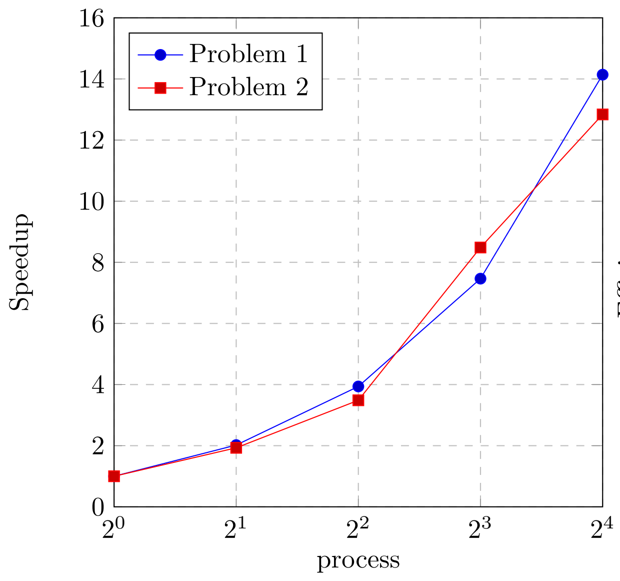

Figure 5: Speedup

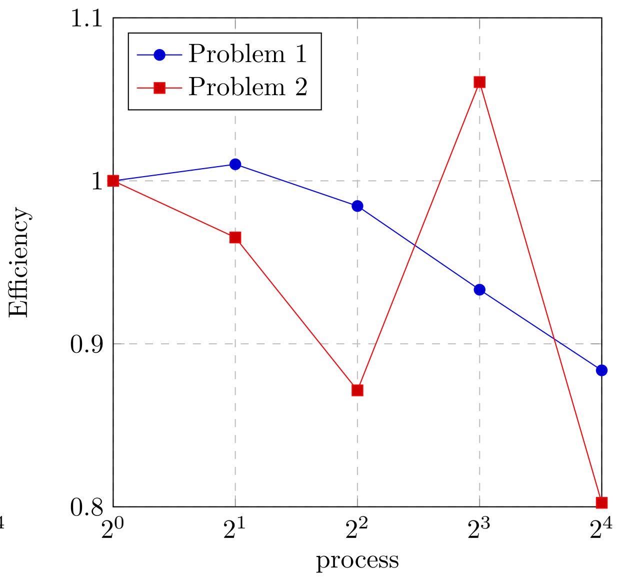

Figure 6: Efficiency

node	core(openmp threads)	process	pure mpi	hybrid
1	2	2	4590.01	9267
1	4	4	2480.65	
8	2	16	624.145	1276.64
8	4	32	537.912	1160.69
16	2	32	520.708	893.893
16	4	64	298.36	637.773

Table 4: Speedup

Output Result When after cluster 100000 documents into 20 clusters. Some frequent words in some cluster is presented in Table 5.

Cluster	1-word	2-words	3-words
0	quality knowledge association analysis work characteristics understanding evidence including participants national respiratory	{covid19 positive} {background study} {methods data} {research results}	{study coronavirus disease} {study significant results} {study analysis data} {higher results study}
12	journal findings transmission infectious evidence management social significant strategies coronavirus outbreak protein	{covid19 acute} {covid19 respiratory} {covid19 model} {respiratory syndrome}	{covid19 patients disease} {covid19 pandemic health} {respiratory syndrome acute} {respiratory acute syndrome}

13	2020 sarscov2 system cases
	respiratory acute model syndrome
	time viral present clinical
	important development treatment
		{respiratory severe}
		{respiratory syndrome}
		{respiratory, acute}
		{respiratory coronavirus}
			{2019 coronavirus disease}
			{covid19 coronavirus disease}
			{respiratory severe acute}
			{respiratory acute syndrome}

Table 5: Set of frequent words

6 Conclusion and Discussion

In this project, we implement a solution for the mentioned problems with different tasks can running in parallel using unitn's HPC cluster. The solution works properly with respect to the size of the problem (number of documents). The parallel application can process 100000 documents in an acceptable running time which we show in the performance evaluation section.

In addition, we also do scalability analysis and performance evaluation for the k-mean parallel algorithms. The result is quite intuitive. The speed up is ration with the number of processes and the efficiency decrease when the number of process rise.

Surprisingly, the result of hybrid architecture is inferior compare with the pure mpi. The reason could be :

- The design of hybrid architecture in our solution is not good enough. It can not take advantage of the shared memory model.
- The size of the problem is not sufficient for the hybrid architecture to show efficiency.

Regarding the result of the project, we can see the efficiency of the HPC cluster and parallel algorithm in front of resolving text mining problems. With powerful computation from HPC, we can mine a large amount of data that can not tackle efficiently by simple serial algorithms.

References

[1] K. Fukunaga, L. Hostetler, The estimation of the gradient of a density function, with applications in pattern recognition, IEEE Transactions on Information Theory 21 (1) (1975) 32–40. doi:10.1109/TIT.1975.1055330.

[2] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, AAAI Press, 1996, pp. 226–231.

[3] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proc. of 20th Intl. Conf. on VLDB, 1994, pp. 487–499.

[4] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, SIGMOD Rec. 29 (2) (2000) 1–12. doi:10.1145/335191.335372.
URL https://doi.org/10.1145/335191.335372

[5] M. J. Zaki, Scalable algorithms for association mining, IEEE Transactions on Knowledge and Data Engineering 12 (3) (2000) 372–390. doi:10.1109/69.846291.

[6] M. Song, Sanguthevar Rajasekaran, A transaction mapping algorithm for frequent itemsets mining, IEEE Transactions on Knowledge and Data Engineering 18 (4) (2006) 472–481. doi:10.1109/TKDE.2006.1599386.

[7] Cord-19 dataset, https://www.kaggle.com/allen-institute-for-ai/CORD-19-research-challenge.

[8] Schema of cord-19 dataset, https://www.kaggle.com/allen-institute-for-ai/CORD-19-research-challenge?select=json_schema.txt.

[9] Simple set implementation in c, https://github.com/barrust/set.

[10] Efficient mpi parallel reduction without mpi_reduce or mpi_scan (only send/recv), https://gist.github.com/rmcgibbo/7178576.

[11] K mean parallel, https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Chandramohan-Fall-2012-CSE633.pdf.

Appendix

1. Github repository: https://github.com/unitn-course-project/parallel-datamining-algorithms