

# ParRescue: Scalable Parallel Algorithm and Implementation for Biclustering over Large Distributed Datasets

Jianhong Zhou  
Department of Computer Science  
University of Illinois at Chicago  
Chicago, IL, 60607  
jzhou13@uic.edu

Ashfaq Khokhar  
Department of Computer Science  
Department of Electrical and Computer Engineering  
University of Illinois at Chicago  
Chicago, IL, 60607  
ashfaq@uic.edu

## Abstract

*Biclustering refers to simultaneously capturing correlations present among subsets of attributes (columns) and records (rows). It is widely used in data mining applications including biological data analysis, financial forecasting, and text mining. Biclustering algorithms are significantly more complex compared to the classical one dimensional clustering techniques, particularly those requiring multiple computing platforms for large and distributed data sets. In this paper, we develop an efficient scalable algorithm, referred to as ParRescue(Parallel Residue Co-clustering), that is capable of performing biclustering on extremely large or geographically distributed data sets. ParRescue divides the cluster tasks among processors with minimal communication costs thus making it scalable over large number of computing nodes. The proposed implementation is based on an existing sequential approach that has been modified for amenable parallel implementation. The proposed ParRescue algorithm has been implemented using MPI and the performance results are reported based on executions on a 64 node Linux PC cluster connected over 100 Mbits links. The experimental results show scalable performance with near linear speedups across different data and machine sizes compared to the modified sequential algorithm.*

## 1. Introduction

Traditional clustering has proven to be useful for dividing data into disjoint groups of similar objects. It models data by its clusters and each cluster may be used to represent a concept underlying in the data set. A variety of clustering algorithms, such as K-Mean [8] and CURE [9], has been proposed and successfully applied to real-life data mining problems. Clustering techniques are good at finding

global patterns by maximizing the similarity within a class and minimizing the similarity across classes. The similarity criteria of a cluster is based upon entire one- dimension vectors of data matrix. For example, the similarity between rows is a function of the row vectors involved. The most popular similarity function is the Euclidean distance function which is famous for "dimension curse". When the dimension increases, the similarity defuses in all attributes. Therefore, the distance between two records in such cases becomes meaningless. For example, in text mining, the size of keywords set describing different documents is huge and yields sparse data matrix. In this case, clusters based on entire keywords sets may have no meaning for end users.

Biclustering is an approach that finds local patterns where a subset of records (objects) might be similar to each other based on only a subset of attributes. The goal of biclustering algorithms is to search for clusters that are tightly co-related based on some homogeneity criterions. Note that clusters can just cover part of rows or columns and overlap each other. Biclustering can generate a wide variety of clusters that captures all the significant correlation information present in a data set.

The term biclustering was first used to generate a subset of genes and a subset of conditions with high similarity score in gene expression data analysis by Cheng and Church [4]. Biclustering is also referred to as coclustering, box clustering, and subspace clustering in other application fields. Biclustering algorithms can be divided into two different sets [11]: algorithms that determine one bicluster in one iteration and algorithms that simultaneously determine a given number of biclusters. Cheng and Church [4] and Sheng et al. [3] discover one bicluster at a time, mask it with random numbers and repeat the procedure to discover a new bicluster in the second iteration. Hartigan [10] recognizes two biclusters by splicing each existing bicluster into two pieces at each iteration. FLOC [14]

randomly allocates each row(column) into a row(column) cluster and iteratively improves the quality of the biclusters. Cho et al. [5] introduce a  $k$ -mean like biclustering algorithm to discover all biclusters at the same time. First, each row(column) is randomly assigned into a row(column) cluster according to spectral approximation. Secondly, each row(column) moves to its nearest cluster based on objective function, respective squared residues, defined by Cheng and Church [4]. The move guarantees the objective function to decrease monotonically. The algorithm iterates till the objective function reaches the user defined threshold.

Apart from interesting biological applications, biclustering also has been successfully used in relevant areas such as: information retrieval and text mining, recommendation systems, target marketing and market research, database research and data mining. Some applications have huge dataset whose magnitudes are  $10^3 \times 10^6$ , while some applications have distributed datasets in different locations with the development of business globalization. All these characteristics for biclustering problem require distributed/parallel approaches. Although a significant amount of research results are reported on serial implementation of biclustering, there is still much space for improvement in the parallel solutions of this problem.

In this paper, we have developed a parallel algorithm, referred to as *ParRescue*(Parallel Residue Co-clustering), to discover biclusters in a distributed computing environment. ParRescue is based on a modified version of Cho et al.'s [5] serial algorithm that primarily works on a compressed summary  $k \times l$  matrix  $S$ , where  $k$  and  $l$  are the number of row and column clusters. The proposed ParRescue algorithm has been implemented using MPI and the performance results are reported based on executions on a 64 node Linux PC cluster connected over 100 Mbits links. The experimental results show scalable performance with near linear speedups across different data and machine sizes compared to the modified sequential algorithm.

The rest of the paper is organized as follows. In section 2, we present a few basic concepts of biclustering and a brief introduction of the serial algorithm. We provide a motivation for the modified sequential algorithm on which ParRescue is based. In section 3, we describe ParRescue in detail and analyze its computation and communication complexity. The experimental results are presented and discussed in section 4. Related work and conclusions are presented in section 5 and section 6, respectively.

## 2. Background

### 2.1. Problem Definition

Given a data matrix  $A$ , with set of rows  $R$  and set of columns  $C$ , where the cell  $a_{ij}$  denotes a value repre-

senting the relation between row  $i$  and column  $j$ . Let  $R = \{r_1, r_2, \dots, r_m\}$  denotes the set of rows and  $C = \{c_1, c_2, \dots, c_n\}$  denotes the set of columns of matrix  $A$ . Considering that  $I \subseteq R$  and  $J \subseteq C$  are subsets of the rows and columns, respectively,  $A_{I,J} = (I, J)$  denotes the submatrix of  $A$  which contains only the cells  $a_{ij}$  belonging to the intersection of row subset  $I$  and column subset  $J$ . A *bicluster* is a subset of rows and a subset of columns that exhibits a similar behavior. The similarity is not treated as a function of pairs of entire rows or entire columns. Instead, similarity is some specific characteristics of homogeneity, such as coherence measurement between rows and columns in the bicluster. The bicluster  $A_{I,J} = (I, J)$  is defined as a  $k \times l$  submatrix of data matrix  $A$ , where  $k$  and  $l$  are the number of rows and the number of columns in the submatrix  $A_{I,J}$ . The specific problem of biclustering is to discover a set of biclusters  $B_k = (I_k, J_k)$  from the given data matrix  $A$ , so that each bicluster satisfies some specific feature of similarity. Please read the following carefully.

### 2.2. Sequential Algorithm

We first present a high level description of the Coclustering algorithm. The Coclustering approach formulates objective functions based on minimizing two measures of squared residue that are similar to those used by Cheng and Church [4] and discovers exclusive and coherent biclusters at the same time. The Coclustering algorithm employs a "ping-pong" approach to alternatively invoke batch update and incremental update algorithms, where the incremental update refines the biclusters produced by the batch update and triggers future runs of batch update. First, spectral approximation is used to initialize row cluster indicator matrix  $R$  and column cluster indicator matrix  $C$ . After that, batch update algorithm is applied to monotonically decrease the proposed objective functions. At each iteration, the cluster indicator matrix  $C$  is updated only after determining the nearest column cluster for every column of matrix  $A$  (likewise for rows). The algorithm iterates till the objective functions converge to user tolerance factor  $\tau$ . To alleviate the problem of poor local minima in certain situations and the presence of empty clusters, Coclustering applies incremental update, a local search strategy, to move a single row or a single column from a given cluster to another by using the row (column) cluster summary array  $r_c(c_r)$ , provided the move results a decrease in the objective function. A chain of moves (move a subset of row or column) can obtain even better local minima [6]. After each row (column) batch update and row (column) incremental update, a global compressed summary matrix  $S$  is generated to reflect the finished update and to provide a new base for the next update. When the incremental update can not trigger further batch update, final set of biclusters is generated. The

main steps of the Coclustering algorithm are outlined in the Figure 1.

**Function** *Cocluster*( $A, \varepsilon, l, k$ )

**Input:**  $A$  is the data matrix,  $\varepsilon$  is the threshold of objective function,  $k$  and  $l$  are the number of row and column clusters, respectively.

**Output:** Cluster indicator matrices  $R$  and  $C$  of matrix  $A$

**begin**

```

1 initialize objval,  $R$  and  $C$ ;  $\tau \leftarrow \varepsilon \|A\|^2$ 
2 repeat
3   repeat
4     assign each row to its nearest row cluster;
5     compute a  $k \times l$  compressed summary matrix  $S$  to reflect
     the new row assignments;
6     assign each column to its nearest column cluster
7     compute a  $k \times l$  compressed summary matrix  $S$  to reflect
     the new column assignments;
8     compute a new objval;
9   until the change of objval  $> \tau$ ;
10  compute the minimal row-cluster distance  $d_c$  by using col-
    umn cluster summary array  $c_r$  for each row in matrix  $A$ ;
11  incremental update  $R$ ;
12  compute the  $S$  to reflect the new row updates;
13  compute the minimal column-cluster distance  $d_c$  by using
    row cluster summary array  $r_c$  for each column in matrix  $A$ ;
14  incremental update  $C$ ;
15  compute the  $S$  to reflect the new column updates;
16 until incremental updates can not reduce objval further;
end;

```

**Figure 1.** *Coclustering Algorithm.*

### 2.3. Modified Sequential Algorithm

In order to design a parallel algorithm based on the Coclustering approach, several factors need to be evaluated. First of all, the partitioning of the input data matrix should be done in such a way that it induces infrequent and least amount of inter-processor communication. Also, since input data is distributed over multiple nodes, a sequential approach inside each node with larger memory requirements becomes feasible, provided it reduces the overall computation cost. Furthermore, for incremental update algorithm (line 10-15), each row (column) moves to its nearest cluster according to the current row (column) biclusters by using the row (column) cluster summary array  $r_c(c_r)$ . The  $r_c(c_r)$  array is the same for all iterations, therefore it can be pre-calculated before the rows (columns) nearest cluster search loop to reduce crucial computation cost when the length of chain increases. Instead of generating a row (column) cluster summary array  $r_c(c_r)$  for each row (column) in each iteration, we generate a row (column) cluster summary matrix

$R_c(C_r)$  for all rows and columns in one pass.

Multiplication is a time consuming operation. Therefore, if the number of multiplications is considerably trimmed, computation cost can be further reduced. In the batch update algorithm (line 3-9), the data matrix of size  $mn$  is involved in a scalar multiplication process of type  $(ba_1 + ba_2 + \dots + ba_j)$  giving rise to  $O(mn)$  multiplications. Instead we have used  $b(a_1 + a_2 + \dots + a_j)$  formulation to reduce the number of multiplication to only  $O(nk)$ . Recall that  $m$  is the number of records in the dataset, which can be in millions or billions, and  $k$  is a relatively small constant representing the number of row clusters in the dataset.

Data Matrix  $A$  needs to be loaded in main memory during clustering process. Row-wise access to cells in matrix  $A$  is needed in row updates, while Column-wise access to cells in matrix  $A$  is deduced during column updates. In order to improve the performance of cell access for both directions, we utilize more memory to achieve high speed cell accesses. We perform a transpose  $A'$  of  $A$  before the start of the column phase. However, since  $A$  does not change during the algorithm, it need to be computed only once. This modification considerably improves the performance of cell accesses in matrix  $A$  by removing segment problems in cache memory. Although it increases memory cost by double, which is a big problem for the sequential Coclustering algorithm on limited memory computer. For ParRescue, this increment in the memory cost is shared among all the processors. Each processor experiences  $1/p$  th of the total increase in memory size.

### 3. ParRescue: Parallel Biclustering

In this section, we introduce the ParRescue algorithm to discover biclusters in a parallel/distributed computing environment. In the following we first discuss the main approach and then provide detail of the algorithm. We assume that the input data matrix is partitioned into chunks of rows. Row block  $A_{IY} = (I, Y)$  is a subset of rows defined over the set of all columns  $Y$ , where  $I = \{i_1, i_2, \dots, i_k\}$  is a subset of rows ( $I \subseteq R$  and  $k < m$ ). In a data matrix, each row denotes a record or object, and each column denotes an attribute the objects are associated with or a condition the objects satisfy. Row-wise partition is a natural pattern for geographically distributed data sets, where attributes (columns) across the nodes are the same, while records (rows) across the nodes are different. This may also imply that each local data set reflects partial characteristics of the whole data set. Coclustering alternatively invokes batch update algorithm and incremental update algorithm in a "ping-pong" manner. One "ping-pong" iteration includes: row batch update, column batch update, row incremental update and column incremental update. These four steps are based on local data matrices and global bicluster information, such as

global row cluster size and global bicluster summary matrix  $\|R^T AC\|$ . Following are the major steps of the ParRescue algorithm.

1. Each processor is assigned to a distinct row-block. Initially a processor is able to compute the assigns of each row to its corresponding row cluster locally without global bicluster information.
2. All the processors exchange their local row-cluster information.
3. Each processor has access to only  $\frac{m}{p}$  values of each column of the data matrix and thus computes partial column cluster assignment for each column. Inter-processor communication is performed to exchange this partial column cluster assignment.
4. Each processor re-computes the column cluster assignment.
5. Based on the global row and column cluster information each processor performs incremental update on row and column clusters assignment.

The global information in the above algorithm is communicated to all the processors using an all-to-all broadcast and global binary associative reduction operations. While the proposed ParRescue algorithm has been presented and implemented as a message passing parallel program, it is based on the well-known divide-and-conquer strategy and can be easily adapted to shared memory and coarser grid based distributed computing paradigms.

### 3.1. ParRescue Algorithm: Details

This section describes the proposed ParRescue algorithm in detail using an SPMD programming model. The ParRescue algorithm is described in detail in Figure 2. Assume there are  $p$  processors in the distributed system. We further assume that the data matrix  $A$  is partitioned into  $p$  subsets and the subset assigned to processor  $i$  is denoted  $A_i$ . In local batch update and incremental update (line 5, 10, 17, 23), each processor generates a local row cluster indicator matrix  $R_i$  and a column cluster indicator matrix  $C_i$  for the partial data matrix  $A_i$ . In (line 6, 11, 18, 24), an all-to-all communication is used to generate the global compressed summary matrix  $S$  which reflects the finished update and provides a base for the new update. For column batch update, the global column distance matrix  $D_c$  is collected (line 7-8) to generate consistent global column clusters for all processors. When the object function converges to user defined threshold  $\tau$ , ParRescue exits batch updates and invokes incremental updates (line 13). For row incremental update, the global minimal row-cluster distance  $d_r$  should

be chosen from all local minimal distance  $\bar{d}_r$  and the local row cluster indicator matrix  $R_i$  is updated when the global minimal row-cluster distance  $d_r$  is calculated by processor  $i$  (line 14-16). For column incremental update, the global row cluster summary matrix  $R_c$  is generated (line 19-20) to compute the global minimal column-cluster distance  $d_c$  for all processors. If the incremental updates decrease the object function then it is helpful to trigger a new batch update to reduce the objective function furthermore. Otherwise, the algorithm reaches a optimal bicluster solution (line 25). Each processor outputs the local bicluster information in a file. Since each processor has the same column clusters guaranteed by ParRescue, the total bicluster patten is simply produced by merging row clusters of these files according to the same row cluster labels.

### 3.2. Performance Analysis

We briefly remark on the computation complexity of the sequential and parallel algorithms in this section. Based on a  $m \times n$  data matrix  $A$ , we generate a bi-cluster model with  $k$  row clusters and  $l$  column clusters. For the sequential Coclustering algorithms (both original and modified), the asymptotic computation complexity is  $O(mnk)$  for row batch update,  $O(mnl)$  for column batch update,  $O(mkl)$  for row incremental update and  $O(nkl)$  for column incremental update. Thus the complexity is  $O((mn)(k+l) + (kl)(m+n))$  per iteration. The overall complexity of the algorithm is:

$$T_{seq} = O(t((mn)(k+l) + (kl)(m+n))) \quad (1)$$

where  $t$  is the number of iterations. Since  $m, n \gg k, l, t$ , the complexity can be simplified to  $O(mn)$ .

For the proposed ParRescue algorithm (outlined in Figure 2, assume there are  $p$  processors in use, each processor contains a  $\frac{m}{p} \times n$  data matrix  $\bar{A}$ . The total time ( $T_p$ ) on a  $p$  processor system is:

$$T_p = T_{computation} + T_{communication} \quad (2)$$

$T_{computation}$  is  $O(\frac{mn}{p}k)$  for row batch update,  $O(\frac{mn}{p}l)$  for column batch update,  $O(\frac{m}{p}kl)$  for row incremental update and  $O(nkl)$  for column incremental update. Thus the complexity is  $O((\frac{mn}{p})(k+l) + (kl)(\frac{m}{p} + n))$  per iteration. Similarly, for each processor, the overall complexity is  $O(t((\frac{mn}{p})(k+l) + (kl)(\frac{m}{p} + n)))$  where  $t$  is the number of iterations. Since  $m, n \gg k, l$ , the  $T_{computation}$  is simplified to  $O(t(\frac{(k+l)mn}{p}))$ .

The communications between processors are based on cluster summary information, not on the basic data values. After each cluster update, each processor generates a  $k \times l$  local compressed matrix  $\bar{S}$ . The global compressed matrix  $S$  needs to be updated on all processors so that all processors perceive the update. Total communication cost for this

---

**Algorithm** *ParRescue*( $i, A_i, \varepsilon, k, l, R_i, C_i$ )

**Input:**  $i$  is the processor ID,  $A_i$  is the partial data matrix on processor  $i$ ,  $\varepsilon$  is the threshold of objective function,  $k$  and  $l$  are the number of row clusters and the number of column clusters, respectively

**Output:**  $R_i$  and  $C_i$  are local cluster indicator matrices for row and column of partial data matrix  $A_i$

**begin**

- 1 Initialize  $objval$ ,  $R_i$  and  $C_i$ ;  $\tau \leftarrow \varepsilon \|A\|^2$
- 2 **repeat**
- 3   **repeat**
- 4     locally assign each row to its nearest row cluster;
- 5     locally compute a  $k \times l$  compressed summary matrix  $\bar{S}_i$  to reflect the new row assignments;
- 6     broadcast the  $\bar{S}_i$  to other processors, gather other's  $\bar{S}_j (j \neq i)$  and generate global  $S$ ;
- 7     locally compute a  $n \times l$  column distance matrix  $D_c^i$ ;
- 8     broadcast the  $D_c^i$ , gather other's  $D_c^j$  and sum to the global  $D_c$ ;
- 9     globally assign each column to its nearest column cluster;
- 10     locally compute a  $k \times l$  compressed summary matrix  $\bar{S}_i$  to reflect the new column assignments;
- 11     broadcast the  $\bar{S}_i$  to other processors, gather other processor's  $\bar{S}_j (j \neq i)$  and generate the global  $S$ ;
- 12     Compute a new  $objval$ ;
- 13   **until** the change of  $objval > \tau$ ;
- 14   locally compute the minimal row-cluster distance  $\bar{d}_r^i$  by using  $m/p \times l$  column cluster summary matrix  $C_r^i$  for all rows in data matrix  $A_i$ ;
- 15   broadcast the  $\bar{d}_r^i$  to other processors, gather other's  $\bar{d}_r^j (j \neq i)$  and generate the global  $d_r$  from them;
- 16   incremental update row clusters indicator matrix  $R_i$  when the global  $d_r^i$  is generate by processor  $i$ ;
- 17   locally compute a  $k \times l$  compressed summary matrix  $\bar{S}_i$  to reflect the new row updates;
- 18   broadcast the  $\bar{S}_i$  to other processors, gather other's  $\bar{S}_j (j \neq i)$  and the generate global  $S$ ;
- 19   locally compute a  $n \times k$  row cluster summary matrix  $R_c^i$ ;
- 20   broadcast the  $R_c^i$ , gather other's  $R_c^j (j \neq i)$  and sum to the global  $R_c$ ;
- 21   compute the global minimal column-cluster distance  $d_c$  by using  $n \times k$  global  $R_c$  for all columns in data matrix  $A_i$ ;
- 22   incremental update  $C_i$ ;
- 23   locally compute a  $k \times l$  compressed summary matrix  $\bar{S}_i$  to reflect the new column updates;
- 24   broadcast the  $\bar{S}_i$  to other processors, gather other's  $\bar{S}_j (j \neq i)$  and generate the global  $S$ ;
- 25 **until** incremental updates can not reduce  $objval$  further;

**end;**

**Figure 2. ParRescue Algorithm.**

collective operation is  $O(pkl)$ . For row batch update, the global size of the row clusters is updated with  $pk$  communication cost. While for column batch update, each processor only knows partial column data, in order to generate consistent column clusters within the global data matrix, each process needs global column data. The local column data is an accumulative column distance matrix  $D_c$  with  $n$  columns and  $l$  column clusters, instead of the basic column data which is a vector with length  $\frac{m}{p}$ . For row incremental update, the global minimal row-cluster distance should be chosen from all local minimal distances by  $p$  communication between processors. For column incremental update, the same as column batch update, it needs global column data by broadcasting local column data. The column data is an accumulative row cluster summary matrix  $R_c$  with  $n$  columns and  $k$  row clusters. The total communication cost of this step is  $O(pnk)$ . Therefore, the communication cost per iteration is  $O(pnl + pnk + pk + p + 4pkl) = O(pn(k+l))$  where  $k, l, p \ll m, n$ . Thus the total time  $T_{communication}$  is  $O(t(pn(k+l)))$ .

The whole time cost for ParRescue is:

$$T_p = O\left(t\left(\frac{(k+l)mn}{p}\right)\right) + O(t(pn(k+l))) \quad (3)$$

From the above cost analysis, the communication cost  $O(t(pn(k+l)))$  is usually negligible relative to the computation cost  $O\left(t\left(\frac{(k+l)mn}{p}\right)\right)$  when  $m \gg n$ , which is a common case in many applications. The computation of ParRescue decreases with the increase in the number of processors. However, this achievement accompanies with the increase in the communication overhead.

## 4. Experimental Result

### 4.1. Experimental Setup

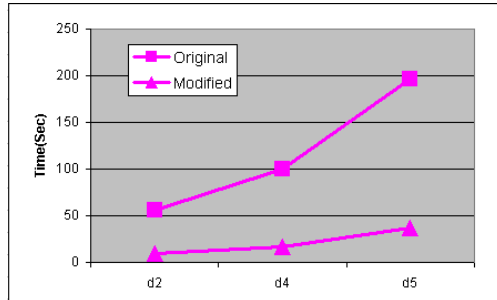
Our performance study is based on the synthetic datasets. We used nine synthetic datasets generated by the IBM data generator [1]. The Characteristics of these datasets are shown in Table 1.

All of our experiments were performed on a Linux cluster of 64 nodes connected by Myrinet fast speed network. However, the results are reported for machine sizes of upto 32 nodes. Each node consists of a 0.8GHz AMD Athlon processor with 512 MB RAM. The head node of the cluster is a dual-processor 1.5GHz AMD Athlon processor with 1GB RAM. MPICH-G2 1.2.4 is employed to implement our parallel algorithm. The operation system is Gentoo Linux 2.6.9-r9 and compiler is the GNU gcc 3.3.6.

Throughout this paper, execution time is measured in seconds and it is averaged over several iterations per cluster. We choose the number of row cluster  $k = 8$  and the number

dataset	rows	columns
d2	131,072	64
d4	262,144	64
d5	524,288	64
d17	1,048,576	64
d18	524,288	128
d19	262,144	256
d20	131,072	512
d21	65,536	1,024
d22	32,768	2,048

**Table 1. Datasets of experiments.**



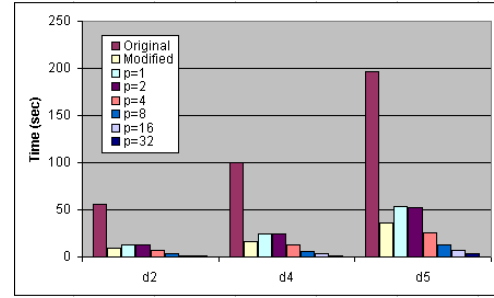
**Figure 3. Execution time of modified and original sequential *Coclustering* algorithms.**

of column cluster  $l = 4$  for all our experiments. An incremental chain with a length of 12 has been used to attain a good quality of incremental update. We used the same initial row cluster indicator matrix  $R$  and column cluster indicator matrix  $C$  for the sequential Coclustering algorithms and the ParRescue algorithm so that we compare performance based on the same input conditions. The row cluster indicator matrix is also partitioned into  $p$  row blocks. The  $i$ th block is a local row cluster indicator matrix  $R_i$  assigned to  $i$ th processor. Since column cluster matrix  $C$  is the same for all processors,  $C$  is copied to all the processors as a local row cluster matrix  $C_i$ .

#### 4.2. Analysis of Experimental Results

ParRescue is based on modified version of the *Coclustering* algorithm which facilitates its parallelization. Figure 3 shows the execution time of the first three datasets in Table 1 for modified and original versions of the sequential Coclustering algorithms. The results indicate that the modifications tremendously improved the performance of the original algorithm [5] by decreasing the execution time by a factor of 6!

We have also studied the elapsed time, the speedup and the scalability of the proposed parallel algorithm and its imple-

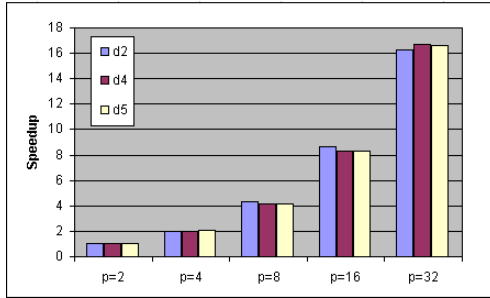


**Figure 4. Execution time of ParRescue and sequential algorithms assuming different processors and data sets.**

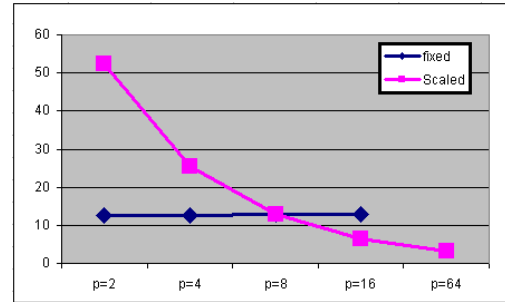
mentation assuming different machine and problem sizes. The speedup depicts the efficiency of the parallel algorithm when the number of processors varies. It is defined as the ratio ( $\frac{T_1}{T_p}$ ), where  $T_1$  is the execution time of the parallel algorithm on 1 processor and  $T_p$  is the execution time of the parallel algorithm on  $p$  ( $p > 1$ ) processor. Another interesting performance measure of a parallel algorithms is its scalability, as it captures how a parallel algorithm handles larger datasets when more processors are available.

We first examine the elapsed time and the speedup of the ParRescue algorithm. We compare the elapsed time of the sequential Coclustering algorithm, modified sequential Coclustering algorithm and that of ParRescue on a cluster of 1, 2, 4, 8, 16 and 32 processors. Figure 4 shows the execution time on various datasets. We can see that ParRescue has much better performance than the original sequential algorithm. The performance of ParRescue on a single processor is lower than the modified Coclustering algorithm due to extra communication overhead. When the number of processors is greater than 2, the total execution time of ParRescue substantially decreases with the ratio of the number of used processors increases regardless of the dataset size. In Figure 5, speedups obtained for different datasets are given. As the figure demonstrates, ParRescue scales very well yielding speedups (using modified sequential algorithm as the base case) of 1.02 (2 processor on dataset  $d2$ ) to 8.30 (32 processor on dataset  $d5$ ). These speedups are linear with respect to the the number of processors. In fact the speedup is super-linear with respect to the original sequential algorithm.

Another factor that limits the speedups of ParRescue algorithm is the scalability. We test the scalability of ParRescue with fixed problem size and the scalability with scaled problem size. For fixed problem size, we keep the size of data matrix per processor fixed while increasing the number of processors. The results are shown in Figure 6. For



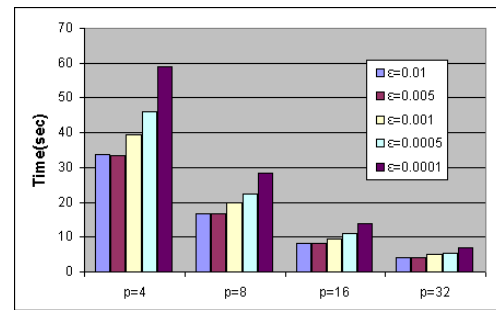
**Figure 5. Speedup of ParRescue algorithm for different data sets assuming different machine sizes.**



**Figure 6. Scalability of ParRescue with respect to fixed and scaled problem sizes.**

this case, we allocated  $655,636 \times 64$  data matrix per processor. We started with  $655,636 \times 64$  on 2 processors up to  $655,636 \times 64$  on 16 processors. It can be seen from the results that the ParRescue algorithm shows stable performance pattern from 2 processors to 16 processors and can use an increased number of processors efficiently. But when the number of processor increases to 8, the performance of ParRescue decreases from 12.5 seconds to 13 seconds due to an increase in the communication overhead. For the scaled problem size case, we tested the efficiency, a measure of how much the processors power can be effectively exploited to decrease the execution time. The experiments reported in the paper are based on dataset *d17* using different machine sizes. We observe that, given a processor set, as the size of the data matrix per processor increases, the efficiency increases as well. For the scaled problem size, from Figure 6, we can see a decrease of 1/16 in the execution time considering a change from 4 to 64 processors. Beyond 64 processors the curves flattens due to the limited size of the problem, there is no benefit of additional computing power. Moreover, the communication overhead that increases with the number of processors limits the performance. Since the communication overhead is ignorable for ParRescue, provided there is enough local computation load in each processor, the scalability in the case of the scaled problem size is better.

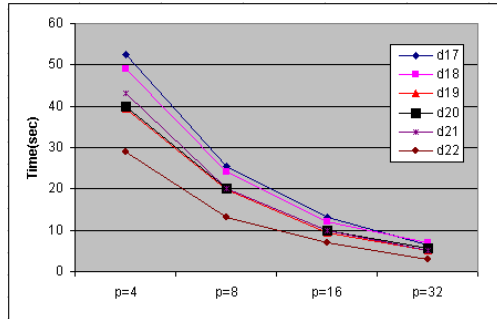
Next, we discuss the influence of changing minimum objective function threshold  $\tau$  on the performance of ParRescue. The change of  $\tau$  directly influences the number of iterations of batch updates. The smaller the value of  $\tau$ , more the number of iterations algorithm needs to reach the tighter converge condition and vice versa. This behavior is confirmed by results in Figure 7. In the chart, we use results from dataset *d19* with the minimum objective function threshold varying from 0.01 to 0.0001. We would like to point out that ParRescue needs more iterations in batch update to reach the converge threshold when  $\tau = 0.0001$  while it costs less



**Figure 7. Influence of changing minimum threshold.**

time when  $\tau = 0.001$ . It still keeps the good scalability with the number of the processors. Similar results have been observed for other datasets.

In ParRescue data was partitioned using row-wise block partition. The number of rows and the number of columns are the main factors that influence the overall computation and communication costs, respectively. To study the influence of the size of row dimension and the size of column dimension on computation cost, we used datasets *d17* to *d22* which vary the number of rows  $m$  and the number of columns  $n$ , keeping the overall matrix size  $m \times n$  fixed to  $2^{29}$ . Figure 8 indicates that ParRescue shows better performance on dataset *d22* compared to the other datasets. Since *d22* has the smallest row size and largest column size among the datasets, it must have the smallest computation cost in the row cluster assignment phase. For all the datasets used in this figure, the execution time is approximately 5 seconds on a 32 processor cluster. We believe in this scenario the cost of computing column updates dominates the computation time.



**Figure 8. Influence of row-block size and column-block size on the execution time.**

## 5. Related Work

Although there have been numerous studies on cluster and bicluster mining, the study on parallel clustering algorithm is still limited and has mainly focused on the traditional clustering problem.

A  $k$ -mean algorithm is parallelized in [7], which is based on the message-passing model of parallel computing. This parallel  $k$ -means algorithm has nearly linear speedup. In [12], a density and grid based clustering algorithm, *pMAFIA*, is presented. It introduces an adaptive grid framework to reduce computation overload and improve the quality of clusters. *pMAFIA* also explores data parallelism and task parallelism to scale up to massive data sets and large set of dimensions. Recently, Pizzuti and Talia [13] presented a parallel clustering algorithm *P - AutoClass* to distributed memory minicomputers for Bayesian clustering. They defined a theoretical performance model of *P-AutoClass* to predict its execution time, speedup and efficiency.

In [2], a parallel biclustering algorithm, called *SPHier*, is developed to discover biclustering parallel. Different from the *ParRescue* algorithm presented in this paper, *SPHier* is based on bigraph crossing minimization.

## 6. Conclusion

In this paper, we proposed a parallel biclustering algorithm *ParRescue*. Based on the literature search we believe ours is the first parallel solution for the biclustering problem. We partitioned the data matrix into row blocks to attain reliable solution in a distributed computing framework. We exploited the compressed data to minimize the inter-processor communication. Our experimental results indicate *ParRescue* excellent parallelization performance on various data matrices and show a nearly ideal speedup. In our future work, we plan to study the parallel algorithm of

incremental biclustering and overlapped biclustering algorithms.

## References

- [1] Ibm synthetic data generation code for classification. <http://www.almaden.ibm.com/software/projects/hdb/resources.shtml>.
- [2] W. Ahmad, J. Zhou, and A. Khokhar. *Sphire: Scalable parallel biclustering using hierarchical bigraph crossing minimization*. Technical report, Multimedia lab, University of Illinois at Chicago, 2004.
- [3] Q. Cheng, Y. Moreau, and B. D. Moor. Biclustering microarray data by gibbs sampling. *Bioinformatics*, 19:ii196–ii205, 2003.
- [4] Y. Cheng and G. M. Church. Biclustering of expression data. *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology*, pages 93–103, 2000.
- [5] H. Cho and I. Dhillon. Minimum sum-squared residue co-clustering of gene expression data. *Proceedings of the 4th SIAM International Conference on Data Mining*, pages 114–125, April 2004.
- [6] I. S. Dhillon, Y. Guan, and J. Kogan. Iterative clustering of high dimensional text data augmented by local search. *Proceedings of The 2nd IEEE International Conference on Data Mining*, pages, pages 131–138, December 2002.
- [7] I. S. Dhillon and D. S. Modha. A data-clustering algorithm on distributed memory multiprocessors. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(1):24–45, January-March 2004.
- [8] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley & Sons, 2000.
- [9] S. Guha, R. Rastogi, and K. Shim. Cure: an efficient clustering algorithm for large databases. *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1(1):24–45, January-March 2004.
- [10] J. A. Hartigan. Cure: an efficient clustering algorithm for large databases. *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1(1):24–45, January-March 2004.
- [11] S. C. Madeira and A. L. Oliveira. Biclustering algorithm for biological data analysis: A survey. *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, pages 245–260, 2000.
- [12] H. Nagesh, S. Goil, and A. Choudhary. Parallel algorithms for clustering high-dimensional large-scale datasets. *Data Mining for Scientific and Engineering Applications*, 2001.
- [13] C. Pizzuti and D. Talia. *p - autoclass: Scalable parallel clustering for mining large data sets*. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):629–641, May-June 2003.
- [14] J. Yang, W. Wang, H. Wang, and P. Yu.  $\delta$ -cluster: Capturing subspace correlation in a large data set. *Proceedings of the 18th IEEE International Conference on Data Engineer*, pages 517–528, 2002.