Computing Fuzzy Rough Approximations in Large Scale Information Systems

Hasan Asfoor*, Rajagopalan Srinivasan*, Gayathri Vasudevan*, Nele Verbiest†,
Chris Cornelis†‡, Matthew Tolentino*§, Ankur Teredesai*, Martine De Cock*†
*Center for Data Science, Institute of Technology
University of Washington Tacoma, USA
Email: {asfoorhm, velamurr, gvasu, ankurt, mdecock}@uw.edu
†Dept. of Applied Mathematics, Computer Science and Statistics
Ghent University, Belgium
Email: {nele.verbiest,chris.cornelis,martine.decock}@ugent.be
‡Dept. of Computer Science and Artificial Intelligence
University of Granada, Spain
Email: chriscornelis@ugr.es
§ Intel Corporation
Email: matthew.e.tolentino@intel.com

***Abstract*—*Rough set theory* is a popular and powerful machine learning tool. It is especially suitable for dealing with information systems that exhibit inconsistencies, i.e. objects that have the same values for the conditional attributes but a different value for the decision attribute. In line with the emerging *granular computing* paradigm, rough set theory groups objects together based on the indiscernibility of their attribute values. *Fuzzy rough set theory* extends rough set theory to data with continuous attributes, and detects *degrees* of inconsistency in the data. Key to this is turning the indiscernibility relation into a *gradual* relation, acknowledging that objects can be similar to a certain extent. In very large datasets with millions of objects, computing the gradual indiscernibility relation (or in other words, the soft granules) is very demanding, both in terms of runtime and in terms of memory. It is however required for the computation of the lower and upper approximations of concepts in the fuzzy rough set analysis pipeline. Current non-distributed implementations in R are limited by memory capacity. For example, we found that a state of the art non-distributed implementation in R could not handle 30,000 rows and 10 attributes on a node with 62GB of memory. This is clearly insufficient to scale fuzzy rough set analysis to massive datasets. In this paper we present a parallel and distributed solution based on Message Passing Interface (MPI) to compute fuzzy rough approximations in very large information systems. Our results show that our parallel approach scales with problem size to information systems with millions of objects. To the best of our knowledge, no other parallel and distributed solutions have been proposed so far in the literature for this problem.**

I. Introduction

Rough set theory was introduced in the 1980s and has since become a discipline in its own right [11]. A recent scientometrics study showed an especially high rise in the popularity of rough set analysis during the last decade, with more than 80% of all the rough sets related papers in the Web of Science published from 2004 to 2013 [18]. The application domains of rough set theory span across all the traditional tasks in data mining and analysis – including feature selection, decision making, rule mining, and prediction. Free as well as commercial tools to perform data mining based on rough set analysis are available.[1]

In (fuzzy) rough set theory, the data at hand is represented as an information system, consisting of objects described by a set of attributes. Examples include an information system of patients (objects) described by symptoms and lab results (attributes), or an information system of online customers (objects) described by their demographics and purchase history (attributes). Inconsistencies arise when objects in the information system have the same attribute values but belong to a different concept (i.e. a subset of objects in the data), meaning that the available attributes are not able to discern the concept. For instance, two patients with very similar symptoms might still have been diagnosed with a different disease, and out of two customers with very similar demographics and purchase history, one might have clicked on an advertisement while the other one did not. An important question is how to take such inconsistencies that arise in training data into account when building models to predict medical diagnosis for patients, or click behavior of future customers. To tackle this problem, Pawlak's rough set theory [11] relies heavily on the construction of the so-called lower and upper approximations of concepts. These approximations are constructed based on an indiscernibility relation induced by the attributes in the information system. The original rough set theory could only handle nominal (discrete) attributes and requires discretization of continous attributes. Fuzzy rough set theory extends the original rough set theory for information systems with continuous valued attributes, and detects gradual inconsistencies within the data.

Data mining techniques based on fuzzy rough set theory have been applied successfully in many domains, including gene selection for microarray data [10], medical applications [6], credit scoring analysis [2], demand prediction [14], risk judgement [9], urban traffic flow [4], image classification

[1]See e.g. http://www.lcb.uu.se/tools/rosetta/, http://www.reduct.com/

[13], feature selection [5], and improvement of support vector machines [3] and decision trees [19].

From a scalability and compute costs perspective, in many data mining techniques based on fuzzy rough set theory, calculating the fuzzy rough lower and upper approximations is the most demanding step. Current implementations in R [12] can handle information systems of up to 100,000 objects and 1000 attributes, depending on the memory of the system they are ran on. The biggest computational bottleneck is the construction of the indiscernibility relation which, for a seemingly moderately sized information system of 1 million objects, corresponds to a similarity matrix with 10^{12} elements. The challenges are not only in the runtime needed to compute each element (grows linearly with increase in the number of attributes), but also in how to deal with the computed elements in further downstream calculations. Indeed, constructing the indiscernibility matrix is not the end goal but only an intermediate step along the way to obtain the fuzzy lower and upper approximations.

In this paper we propose the first distributed approach to computing fuzzy lower and upper approximations. It is based on the following two key insights:

- The construction of the indiscernibility matrix can be carried out in a parallel manner, making each of K compute nodes calculate n/K columns of the matrix (with n the total number of objects in the information system).
- The construction of the indiscernibility matrix does not have to be finished before (partial) computation of the lower and upper approximations can begin. Every computed element of the indiscernibility matrix can be processed further immediately, thereby eliminating the need for storing the similarity matrix in memory or on disk.

The combination of both these insights in a Message Passing Interface (MPI) based distributed algorithm[2] allowed us to efficiently calculate fuzzy lower and upper approximations of information systems with millions of rows, while a state of the art non-distributed implementation in R [12] could not handle 30,000 rows on the same hardware. The main problem with this implementation in R appears to be the inefficient memory usage, in particular the fact that the entire indiscernibility matrix is stored in memory.

The remainder of this paper is structured as follows. In Section II we briefly recall preliminaries about fuzzy rough set theory and we point out scale issues. In Section III we explain how the problem that we solve in this paper is different from other problems in rough set theory for which parallel or distributed algorithms have been developed in the past. To the best of our knowledge, we are the first to present a distributed approach to calculate fuzzy lower and upper approximations, a problem that is fundamentally different in nature from problems which have been studied before. In Section IV we describe our distributed method in detail. Through a series of experiments, in Section V we show that the proposed method scales well on large information systems with millions of objects. Conclusions are presented in Section VI.

[2]Implemented in C and executed on Intel MPI. All code is available on https://github.com/WebDataScience/ParallelFuzzyRoughSetApproximationonMPI/

II. Scale issues in fuzzy rough set analysis

In fuzzy rough set analysis, the data is represented as an *information system* $(X, \mathcal{A})$, where $X = \{x_1, \ldots, x_n\}$ and $\mathcal{A} = \{a_1, \ldots, a_m\}$ are finite, non-empty sets of objects and attributes, respectively. The values of the attributes for the instances can be represented in an $n \times m$ matrix Q where $q_{i,t}$ ($i \in \{1, \ldots, n\}$ and $t \in \{1, \ldots, m\}$) is the value of attribute a_t for instance x_i. The attributes $a \in \mathcal{A}$ can be categorical or quantitative. Fuzzy rough set analysis [7] considers a notion of gradual indiscernibility between instances, modelled by a binary fuzzy relation that can be represented by an $n \times n$ matrix R. The value $r_{i,j}$ is the degree of indiscernibility or similarity between the objects x_i and x_j ($i, j \in \{1, \ldots, n\}$). The value of $r_{i,j}$ ranges between 0 (x_i and x_j are completely dissimilar) and 1 (x_i and x_j are fully indiscernible). The fuzzy indiscernibility relation matrix R can be derived from the information system $(X, \mathcal{A})$ by averaging per-attribute similarities of the objects:

$$r_{i,j} = \frac{1}{m} \sum_{t \in \{1, \ldots, m\}} f_t(q_{i,t}, q_{j,t}). \tag{1}$$

The function f_t ($t \in \{1, \ldots, m\}$) can be defined in many ways and differs for different types of attributes. In the experimental evaluation part of this paper we assume that all attributes are continuous, and therefore we use a normalized Manhattan distance based function that is the same for all attributes:

$$f_t(q_{i,t}, q_{j,t}) = 1 - \frac{|q_{i,t} - q_{j,t}|}{range(a_t)} \tag{2}$$

with the range of attribute a_t defined as the maximum minus the minimum value a_t takes in the information system.

In addition to the information system $(X, \mathcal{A})$, a fuzzy concept C is given in the form of an $n \times 1$ vector. For each $i \in \{1, \ldots, n\}$, the value c_i denotes the degree to which object x_i belongs to concept C. There may be several such fuzzy concepts to which every instance i can belong simultaneously; for example height of a person as tall and short. For brevity of explanation we focus on a single concept C. These values c_i of C range between 1 (x_i completely belongs to the concept C) and 0 (x_i does not belong at all to the concept C). The lower and upper approximations of a fuzzy concept C are defined w.r.t. an implicator and a t-norm respectively, which are relaxations of the implication and conjunction operators from boolean logic. In the experimental evaluation part of this paper we use the Kleene-Dienes implicator and the minimum t-norm, respectively defined as $x \tilde{\rightarrow} y = \max(1 - x, y)$ and $x \tilde{\wedge} y = \min(x, y)$, for x and y in $[0, 1]$. For more examples of fuzzy logical operators we refer to Bede [1]. For each instance x_j ($j \in \{1, \ldots, n\}$) its membership values to the lower and upper approximation of C (also denoted by l_j and u_j respectively) are respectively defined as:

$$l_j = \min_{i \in \{1, \ldots, n\}} (r_{j,i} \tilde{\rightarrow} c_i) \tag{3}$$

$$u_j = \max_{i \in \{1, \ldots, n\}} (r_{j,i} \tilde{\wedge} c_i). \tag{4}$$

This means that the membership value l_j of an instance x_j to the lower approximation of C is high if instances that are highly similar to it belong to a great extent to the concept C. On the other hand, the membership value u_j of an instance x_j to the upper approximation of C is high if there exist instances that are similar to x_j that also strongly belong to C.

The problem that we address in this paper is: *given a very large information system $(X, \mathcal{A})$ with n instances and m attributes, and a fuzzy concept vector C, compute the corresponding lower and upper approximation vectors L and U.* The theoretical time complexity needed to calculate the fuzzy rough lower and upper approximation membership values for all instances is $\mathcal{O}(n^2m)$, assuming that the per-attribute similarity of two objects (2) can be computed in constant time. This demonstrates the challenge to calculate these approximations for information systems with a large number of instances. Even for a dataset with 1 million instances, which seems like a fairly moderate amount, 1 trillion per-attribute similarities need to be computed. Moreover, a straightforward implementation would calculate and store the similarity matrix, which can cause memory problems as its size is $\mathcal{O}(n^2)$. If each similarity value is stored using 8 bytes; which is the typical size of a double in most architectures, then for a dataset with 1 million instances, the similarity matrix would require 8TB of memory.

III. Related work

To the best of our knowledge, we are the first to present a distributed approach to calculate the fuzzy rough lower and upper approximations in an information system. In [15], a parallel attribute reduction algorithm based on fuzzy rough set theory was proposed, but this algorithm is based on mutual information, and does not calculate the fuzzy rough lower and upper approximations explicitly. Additionally, in [8], [17], [20] rough set approaches to handle big data are presented, without explicitly calculating the lower and upper approximations. In addition, these approaches only deal with non-fuzzy rough sets; a significant distinction.

The work of Zhang et al. is more closely related to the work presented in this paper. These researchers have worked on parallel approaches to calculate the lower and upper approximations in traditional (non-fuzzy) rough set theory. For instance, in [21], the authors present a MapReduce based approach to compute the lower and upper approximations in a decision system, i.e. an information system with an additional categorical decision attribute (class) that is used to approach classification problems. Calculating the equivalence classes, both w.r.t. the conditional attributes and the decision attributes, is carried out in parallel, as well as calculating the final lower and upper approximations of all decision classes. In [22], different runtime systems to use these approximations for knowledge acquisition are compared. In [16], an approach to calculate the (non-fuzzy) positive region, i.e. the union of the lower approximation of all classes is proposed.

Our work differs fundamentally from these approaches as calculating the fuzzy rough lower and upper approximations requires calculating the fuzzy similarity matrix, while the non-fuzzy lower and upper approximations are based on an equivalence relation, that can be calculated easily in one MapReduce iteration. Indeed, the maps divide chunks of objects in equivalence classes (groups with the same attribute values) and these equivalence classes are merged in the reduce step. Unfortunately, this strategy cannot be used for calculating the fuzzy similarity between objects, as all objects need to be compared against every other for each attribute value. This poses additional challenges when calculating the fuzzy rough and lower approximations, as one MapReduce round does not suffice to calculate the fuzzy similarity matrix.

The approach in [15], [22] is extended in [23] for composite rough sets, where attributes from different types can be handled. This approach seems to be suitable for our purposes at first sight, as continuous attributes can be handled. However, these composite rough sets require crisp equivalence relations for each attribute type, which means that discretization is needed to handle the continuous attributes. Our approach uses a fuzzy similarity relation to handle continuous attributes, and as a result there is no information loss that is usually associated with discretization.

IV. Computing fuzzy rough approximations in parallel

Our approach assumes the availability of a cluster with a master node and K slave nodes. The general idea is to equally distribute the computational load over the available slave nodes such that each slave node performs part of the fuzzy rough set approximations computation. Given an information system with n instances (rows), each slave node processes $\lceil \frac{n}{K} \rceil$ of the rows in the dataset, thereby generating $\lceil \frac{n}{K} \rceil$ columns of the similarity matrix, and computing $\lceil \frac{n}{K} \rceil$ of the n values that ultimately need to be aggregated per each value in the lower and the upper approximation (cfr. Equations (3) and (4)).

The flow of the overall approach is depicted in Figure 1. Computations happen on the master level (left) and on K slave nodes (right, with $K = 2$ in the picture). Each slave node k ($k \in \{1, \ldots, K\}$) is assigned a horizontal partition P_k of the data, for which it will do computations. However, it is assumed that the entire data is available to each of the K slave nodes. That is, the data is not partitioned over the slave nodes but the computations are. All P_k partitions have $\lceil \frac{n}{K} \rceil$ rows (except for the last slave node which has a partition with the remaining rows, which is possibly less than $\lceil \frac{n}{K} \rceil$).

The approach is divided into two stages – data preparation and data processing – which we describe in more detail below. In the data preparation stage, data is made available and ready to be processed by the slave nodes. In the data processing stage, upper and lower approximations are computed using the prepared data from the previous stage.

A. Data Preparation

a) Data broadcasting: At the very beginning of the data preparation stage, the master node has the instance-attribute value matrix Q as well as the class or concept vector C, each in a separate file. Before the MPI program starts, a Linux script is run at the master node to distribute the matrix Q over the slave nodes so that each slave node has a full copy of Q. This requires that each slave node has enough disk space to store the whole dataset file. By doing this, we are emulating a parallel file system. As will become clear below, even though every slave k ($k \in \{1, \ldots, K\}$) has and needs a full copy of Q, it is only responsible for the computations on its assigned partition P_k of $\lceil \frac{n}{K} \rceil$ rows of Q.

After the matrix Q is sent to all slave nodes, the Linux script triggers the MPI program to run on all slave nodes in addition to the master node. Once started, the master node

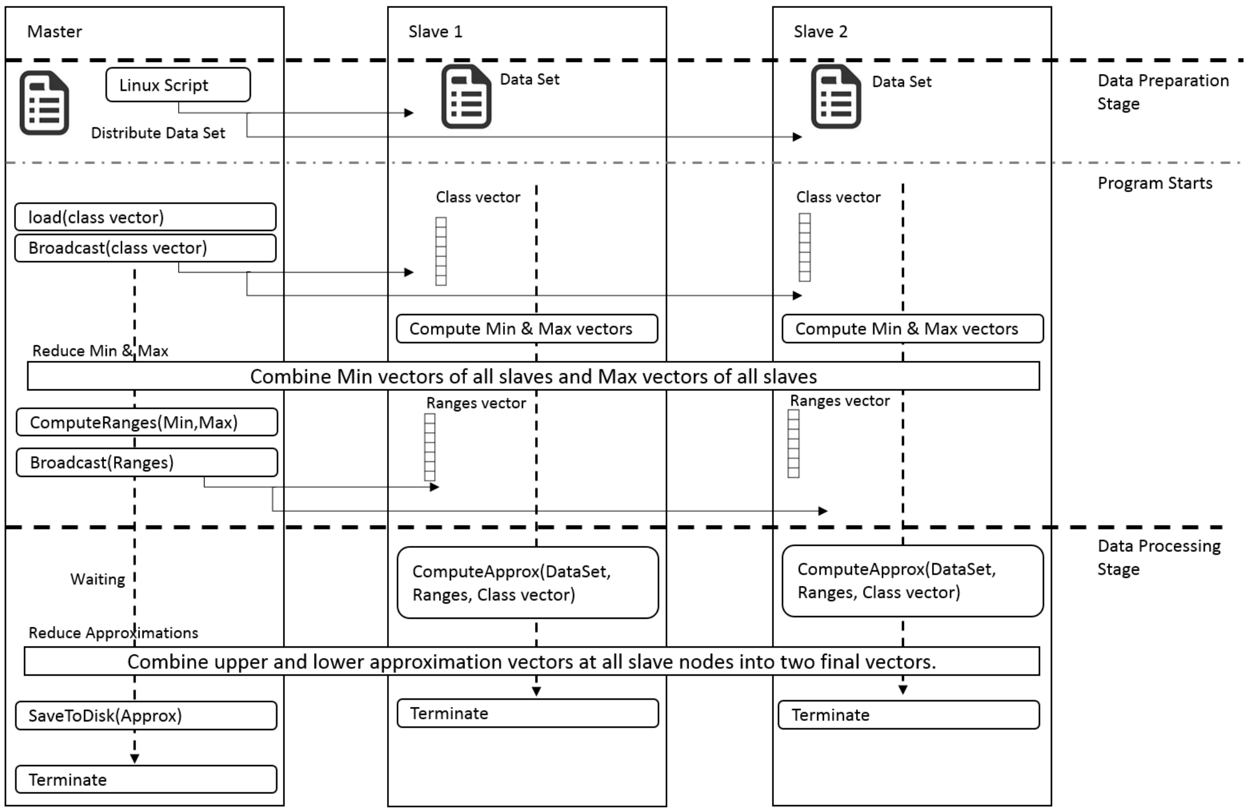

Fig. 1. This figure summarizes the overall flow of our distributed approach to computing fuzzy lower and upper approximations. The approach has two stages, namely data preparation and data processing. The data preparation stage starts with a Linux script that distributes the dataset, i.e. the instance-attribute matrix, to all slave nodes. Then the master node loads and broadcasts the concept or class vector. After that, each slave node computes the minimum and maximum values of each attribute for a subset of the rows, and then the master node reduce the minimum and maximum vectors created by the slaves into two final vectors. Then the master computes the attribute ranges given the minimum and maximum vectors and finally broadcasts the ranges vector. This concludes the data preparation stage which gets the data ready to be processed. At the start of the data processing stage, each slave node has the dataset, the ranges vector and the class vector as input, uses these to produce two intermediate vectors: one for the upper approximation and another one for the lower approximation. Finally the intermediate vectors get reduced at the master producing the final upper and lower approximation vectors. At this point, all slave nodes terminate while the master saves the final approximation vectors to disk and then terminates.

loads the class vector C from the disk and broadcasts it to all slave nodes.

b) Attribute range computation: After that, the range of every attribute – which is needed in the denominator of Equation (2) – gets computed in a parallel way. To this end, each slave node k ($k \in \{1, \ldots, K\}$) creates two vectors $\min^k$ and $\max^k$, each of size m, with m the number of attributes in the information system. The objective of these two vectors is to keep track of the minimum values and maximum values of each attribute in the dataset seen so far. Recall that each slave node k is responsible for a partition P_k of instances. Below, we denote the the index set of these instances (rows) by I_k. Initially, within each slave node k, all values in $\min^k$ are set to $+\infty$ and all values in $\max^k$ are set to $-\infty$. Then, the slave node k iteratively goes through all instances $i \in I_k$ in its partition P_k and updates the minimum and maximum values for all attributes $t \in \{1, \ldots, m\}$ so that

$$\min_t^k \leftarrow \min(q_{i,t}, \min_j^k) \tag{5}$$

$$\max_t^k \leftarrow \max(q_{i,t}, \max_j^k) \tag{6}$$

After each slave k is done populating its $\min^k$ and $\max^k$, they get reduced to two vectors $\min^f$ and $\max^f$ at the master node so that, for $t \in \{1, \ldots, m\}$,

$$\min_t^f = \min_{k \in \{1,\ldots,K\}} \min_t^k \tag{7}$$

$$\max_t^f = \max_{k \in \{1,\ldots,K\}} \max_t^k \tag{8}$$

Then, the master node computes the range of each attribute and stores the result in the ranges vector so that, for $t \in \{1, \ldots, m\}$,

$$\text{range}_t = \max_t^f - \min_t^f \tag{9}$$

Finally the master node broadcasts the ranges vector to all slave nodes.

B. Data Processing

In this stage, each slave node k ($k \in \{1, \ldots, K\}$) has the matrix Q, the class vector C and a vector with the range of each attribute. Recall that each slave k is responsible for a horizontal partition P_k of Q, and that we denote the index set of these instances by I_k. The goal of our algorithm is a lower approximation vector L and an upper approximation vector U, with their values l_j and u_j as defined in Equations (3) and (4), for $j \in \{1, \dots, n\}$. Note that (3) and (4) can be rewritten as

$$l_j = \min_{k \in \{1,\dots,K\}} \min_{i \in I_k} (r_{j,i} \tilde{\rightarrow} c_i) \quad (10)$$

$$u_j = \max_{k \in \{1,\dots,K\}} \max_{i \in I_k} (r_{j,i} \tilde{\wedge} c_i). \quad (11)$$

The computation of

$$l_j^k = \min_{i \in I_k} (r_{j,i} \tilde{\rightarrow} c_i) \quad (12)$$

$$u_j^k = \max_{i \in I_k} (r_{j,i} \tilde{\wedge} c_i) \quad (13)$$

for $k \in \{1, \dots, K\}$, can be done fully in parallel on each of the K slave nodes. Since we need these values for all $j \in \{1, \dots, n\}$, this involves each slave node k constructing the $\lceil \frac{n}{K} \rceil$ columns from the similarity matrix that correspond to the index set I_k. However, during calculation of (12) and (13), each value $r_{j,i}$ of the similarity matrix is immediately combined with c_i, eliminating the need to store (entire columns of) the similarity matrix.

In our implementation, at the beginning of the data processing stage, the master node is waiting for each slave node to complete its computations and produce l_j^k and u_j^k, for all $j \in \{1, \dots, n\}$. In order for a slave node to do that, it loads its assigned partition P_k and keeps it in memory throughout the execution of the program. In addition, for each loaded row $q_{i,.}$ in P_k ($i \in I_k$), each value $q_{i,t}$ ($t \in \{1, \dots, m\}$) in that row gets divided by ranges$_t$, i.e. by the previously computed range of the attribute a_t. Then, the slave node loops over each of the rows in the matrix Q that it has on disk and does the following (we can refer to this loop as the outer loop; it ranges over $j \in \{1, \dots, n\}$):

1) Read row $q_{j,.}$ from the dataset file, and divide each value $q_{j,t}$ ($t \in \{1, \dots, m\}$) in that row by ranges$_t$, i.e. by the previously computed range of the attribute a_t.
2) For each row $q_{i,.}$ in P_k, the following steps are carried out (we can refer to this as the inner loop; it ranges over $i \in I_k$):
    a) Given row $q_{j,.}$ from the dataset file and row $q_{i,.}$ from P_k, compute $r_{j,i}$ as defined by equation 1. Note that we do not need to divide by the range here because this has been taken care of at an earlier step.
    b) Compute the values $r_{j,i} \tilde{\rightarrow} c_i$ and $r_{j,i} \tilde{\wedge} c_i$.
    c) Update the values l_j^k and u_j^k:

$$l_j^k \leftarrow \min(l_j^k, r_{j,i} \tilde{\rightarrow} c_i)$$
$$u_j^k \leftarrow \max(u_j^k, r_{j,i} \tilde{\wedge} c_i)$$

Once all the K slave nodes have completed the above steps, the master node aggregates all computed values into

$$l_j = \min_{k \in \{1,\dots,K\}} l_j^k \quad (14)$$

$$u_j = \max_{k \in \{1,\dots,K\}} u_j^k \quad (15)$$

for all $j \in \{1, \dots, n\}$. These vectors L and U represent the final lower and upper approximations respectively. At this point, all slave nodes terminate while the master node is saving these two vectors to disk and then terminates.

Pseudocode for the master and slave programs is presented in Algorithm 1 and 2. The execution of the outer loop, i.e. lines 26-36 in Algorithm 2, can be optimized by running multiple iterations in parallel by creating a separate thread for each iteration of the loop. The number of concurrent iterations is determined by a parameter passed to the program.

Algorithm 1 Master Program

```
1: INPUT: pathToClassvector, $m$, $n$, $K$
2: DECLARE: ranges, min, max, $l$, $u$
3: classvector ← loadFromFile(pathToClassvector)
4: broadcast(classvector)
5: ReduceFromSlaves($\min^1, \dots, \min^K, \max^1, \dots, \max^K$)
6:     for $t$ from 1 to $m$ do
7:         $\max_t^f \leftarrow \max_{k \in \{1,\dots,K\}} \max_t^k$
8:         $\min_t^f \leftarrow \min_{k \in \{1,\dots,K\}} \min_t^k$
9:     end for
10: EndReduce
11: for $t$ from 1 to $m$ do
12:     $\text{range}_t \leftarrow \max_t^f - \min_t^f$
13: end for
14: broadcast(range)
15: wait for slaves to finish
16: ReduceFromSlaves($l^1, l^2, \dots, l^K, u^1, u^2, \dots, u^K$)
17:     for $j$ from 1 to $n$ do
18:         $l_j \leftarrow \min_{k \in \{1,\dots,K\}} l_j^k$
19:         $u_j \leftarrow \max_{k \in \{1,\dots,K\}} u_j^k$
20:     end for
21: EndReduce
22: save $L$ and $U$ to disk
23: Terminate
```

While the main contribution of this work is the MPI implementation, we did consider using the MapReduce paradigm using Apache Spark to develop a comparative solution. To this end, we viewed the indiscernibility matrix R as originating from a matrix-multiplication-like combination of the matrix Q with its transposed matrix Q^T, and similarly, we viewed the lower and upper approximation vectors L and U as matrix-multiplication-like combinations of R with C (with the element-wise multiplication replaced by computations for the similarity values or fuzzy logical operators respectively). Following a traditional MapReduce setting common for large matrix operations, we split matrix Q into blocks using both horizontal and vertical partitioning logic. In doing so we caused the Spark implementation to be severely memory constrained while storing the (key, value)-pairs of the indiscernibility matrix and due to the number of RDDs resulting due to the block partitions. In a future work we hope to address this issue further.

V. Experimental results

A. Experimental setup

We tested the scalability of our distributed algorithm on 11 synthetically generated datasets, with the number of instances (rows) varying from 10,000 to 10 million, and the number of attributes (columns) varying from 10 to 50. All attribute values are randomly generated numbers between 0 and 1000. The concept vectors are generated in the same way, but with randomly generated values between 0 and 1 to comply with

Algorithm 2 Slave Program

```
1:  INPUT: dataset, $m$, $n$, $K$
2:  partitionSize $\leftarrow 1 + (n/K)$
3:  DECLARE: $\text{min}^k, \text{max}^k, l^k, u^k$
4:  initialize($\text{min}^k, +\infty$)
5:  initialize($\text{max}^k, -\infty$)
6:  initialize($l^k, 1$)
7:  initialize($u^k, 0$)
8:  startRow $\leftarrow (k-1) \cdot$ partitionSize
9:  endRow $\leftarrow$ startRow $+$ partitionSize
10: receive broadcasted classvector
11: for $i$ from startRow to endRow do
12:     $q_{i,.} \leftarrow$ getRowFromDataSet(dataset,$i$)
13:     for $t$ from 1 to $m$ do
14:         $\text{max}^k_t \leftarrow \max(\text{max}^k_t, q_{i,t})$
15:         $\text{min}^k_t \leftarrow \min(\text{min}^k_t, q_{i,t})$
16:     end for
17: end for
18: reduce $\text{max}^k$ and $\text{max}^k$ to the master node
19: receive broadcasted ranges
20: for $i$ from startRow to endRow do
21:     $q_{i,.} \leftarrow$ getRowFromDataSet(dataset,$i$)
22:     for $t$ from 1 to $m$ do
23:         $q_{i,t} \leftarrow q_{i,t}/range_t$
24:     end for
25: end for
26: for $i$ from 1 to $n$ do
27:     $q_{j,.} \leftarrow$ getRowFromDataSet(dataset,$i$)
28:     for $j$ from 1 to $m$ do
29:         $q_{j,t} \leftarrow q_{j,t}/range_t$
30:     end for
31:     for $i$ from startRow to endRow do
32:         simValue $\leftarrow$ similarity$(q_{j,.}, q_{i,.})$
33:         $l^k_j \leftarrow \min(l^k_j, \text{simValue} \tilde{\rightarrow} c_i)$
34:         $u^k_j \leftarrow \max(u^k_j, min(\text{simValue} \tilde{\wedge} c_j, c_i))$
35:     end for
36: end for
37: reduce $u^k$ and $l^k$ to the master node
38: Terminate
```

Node	Cores/Node	Speed/Core (GHz)	Memory (GB)
MPI master	16	2.67	16
slave nodes 1 to 7	48	2.70	62

TABLE I. HARDWARE SPECIFICATIONS OF THE CLUSTER

the definition (see Section II). As an implicator and t-norm we used respectively the Kleene-Dienes implicator and the minimum t-norm (see Section II).

We ran the experiments on an Intel cluster of 7 slave nodes with a total of 336 cores and 434GB of RAM. The CPUs used in the cluster are 64-bit Intel(R) Xeon(R) CPUs. In addition, the cluster includes an additional node that was used as a master node. Table I provides the specifications of each node. We compiled and ran the MPI code on Intel MPI 4.3. For the non-distributed R version, we used the 3.1 release of R and we ran the code on a single node of the cluster.

rows	attributes	threads	MPI (7 nodes)	R (1 node)
10000	10	46	1s	105s
10000	50	46	1s	365s

TABLE II. COMPARISON OF EXECUTION TIME (IN SEC) BETWEEN OUR DISTRIBUTED MPI BASED ALGORITHM AND THE NON-DISTRIBUTED ALGORITHM FROM THE "ROUGHSETS" PACKAGE IN R [12].

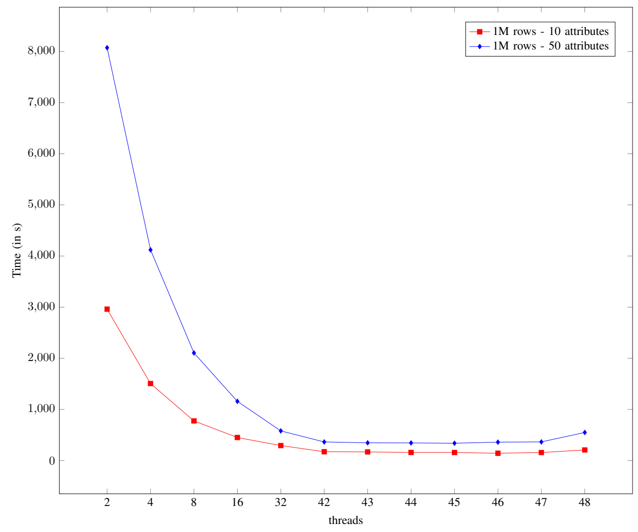

Fig. 2. Execution time (in sec) for a varying number of threads per compute node. The experiments are performed on a cluster with 1 master node and 7 slave nodes, and on two different datasets (i.e. a dataset with 1 million rows and 10 attributes, and a dataset with 1 million rows and 50 attributes). There is a linear decrease in execution time of the MPI program as the number of threads per node increases.

B. Results

We first used the implementation that comes with the "RoughSets" package in R [12], and as mentioned before we tested its limitations on one node. We could not find a distributed R version for fuzzy rough sets. This R implementation was only able to handle a dataset upto 30000 rows and 10 attributes likely because of inefficient memory usage, i.e. the fact that the algorithm keeps the entire indiscernibility matrix in memory at once. Table II compares the execution time of our distributed MPI based algorithm with the non-distributed implementation in R on a small dataset with 10,000 rows. Execution times of the MPI algorithm for datasets with 100,000, 1 million and 10 million rows are presented further in this section; no corresponding execution times for R are presented over these larger datasets because, as explained above, the R package ran out of memory.

W.r.t. our distributed MPI algorithm, we conducted four sets of experiments to observe factors that affect the scalability of our approach. Four experiments conducted focus on dataset size: number of instances, number of attributes, number of compute nodes, and number of threads per compute node. Each experiment was conducted thrice and average execution time is systematically reported.

The first set of experiments investigates the effect of an increase in the number of threads per compute node. Recall that our implementation is multithreaded for the main outer loop in the slave program, i.e. lines 26-36 in Algorithm 2. Since each iteration of this loop can be executed in parallel

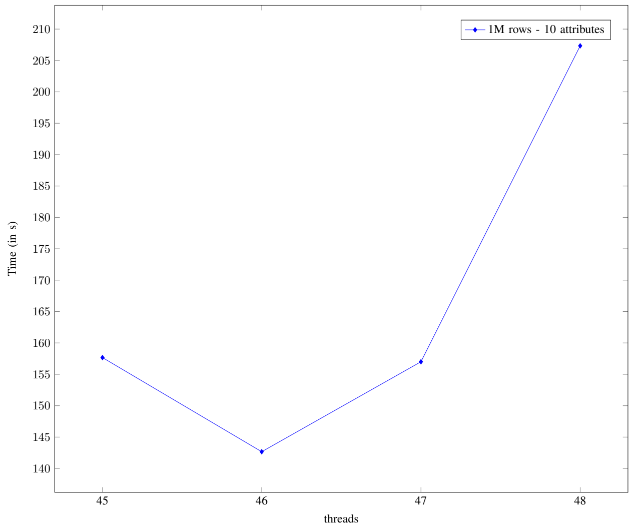

Fig. 3. Execution time (in sec) for a varying number of threads per compute node, with the number of threads approximately equal to the number of available cores. The experiments are performed on a cluster with 1 master node and 7 slave nodes, and on a dataset with 1 million rows and 10 attributes. The best performance is achieved for 46 threads.

and independently of the others, we expect to see the overall runtime decrease when more threads are used. Figure 2 validates this: the execution time decreases almost linearly with an increase in the number of threads. Comparing both curves in Figure 2 indicates that a change in the number of attributes from 10 to 50 does not change the behaviour of the curve, but only shifts the values. Focus on execution times with the number of threads approximately equal to the number of cores available per compute node (48 - a thread per core), and we notice that the best performance is achieved for 46 threads (see Figure 3) where two out of the 48 threads are given for system tasks. Hence, we fixed the number of threads to 46 for all other experiments.

The second set of experiments examines the effect of the number of available compute nodes, while keeping everything else constant. Figure 4 shows clearly that the overall execution time decreases as the computation is distributed over more nodes. Even though the magnitude of the decline becomes less as the number of compute nodes grows, Figures 2 and 4 both show that the computation of fuzzy lower and upper approximations has a strong scaling issue. The use of more hardware (nodes, threads) improves the runtime. Also note in Figure 4 that even with 1 slave node our algorithm is able to efficiently process a dataset with 1 million rows, unlike the ”RoughSets” package in R mentioned above [12] which would not allow us to get beyond 30,000 rows.

The final two sets of experiments examine the effect of the size of the dataset on the execution time. Figure 5 shows that the execution time increases approximately quadratically with the number of rows. In addition, as expected, a higher number of attributes causes to MPI program to run slower but it does not affect the general shape of the curve. Finally, Figure 6 shows that, as expected, the execution time of the MPI program grows approximately linearly with the number of attributes.

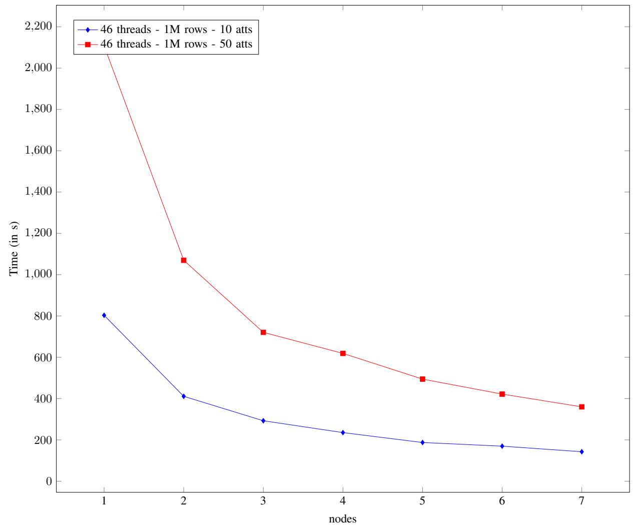

Fig. 4. Execution time (in sec) for a varying number of nodes. The experiments are performed on a cluster with 1 master node and 1 to 7 slave nodes, and on two different datasets (i.e. a dataset with 1 million rows and 10 attributes, and a dataset with 1 million rows and 50 attributes). The number of threads is kept constant at 46. The runtime decreases with the increase in number of compute nodes. Even when only 1 slave node is available, our algorithm efficiently computes lower and upper approximations of a concept vector with 1 million entries.

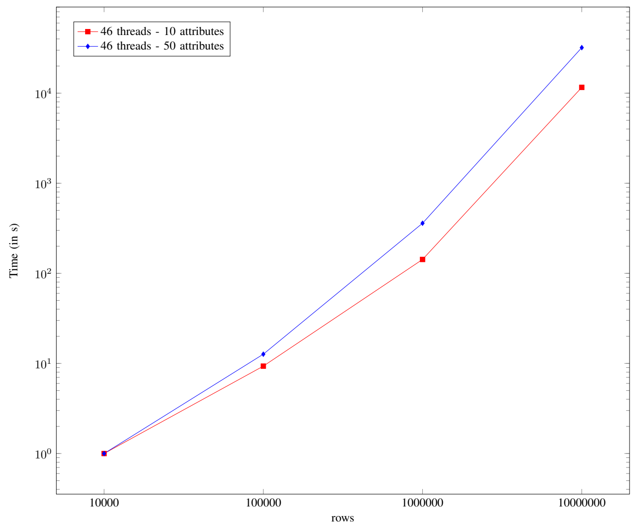

Fig. 5. Execution time (in sec) for a varying number of rows in the dataset. The experiments are performed on a cluster with 1 master node and 7 slave nodes, and on datasets with respectively 10,000, 100,000, 1 million and 10 million instances and 10 or 50 attributes. The number of threads is kept constant at 46. The execution time of the MPI program grows approximately quadratically in terms of the number of instances in the dataset.

VI. Conclusion

The operations involved in computing lower and upper approximations of concept vectors are at the core of many machine learning applications even other than fuzzy rough set theory. Current R-based, non-distributed implementations of such operations described in this paper are not scalable and bounded by memory available to keep the entire similarity matrix in-memory. The main motivation of our work is this significant computational bottleneck of construction of the

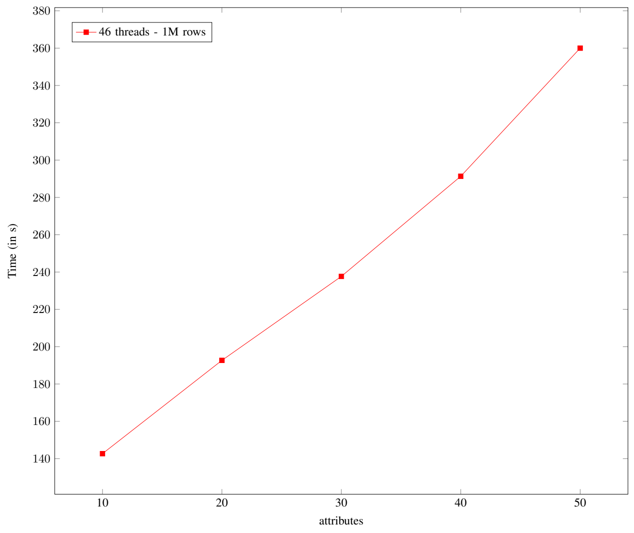

Fig. 6. Execution time (in sec) for a varying number of attributes in the dataset. The experiments are performed on a cluster with 1 master node and 7 slave nodes, and on datasets with 1 million rows and respectively 10, 20, 30, 40 and 50 attributes. The number of threads is kept constant at 46. The execution time of the MPI program grows approximately linear in terms of the number of attributes in the dataset.

indiscernibility relation, both in terms of compute time and memory requirements. In this paper we proposed a distributed algorithm for the computation of fuzzy rough approximations based on two key insights: (1) the construction of the indiscernibility matrix can be carried out in a parallel manner, which allows to reduce the compute time by leveraging multiple compute nodes and threads; and (2) every computed element of the indiscernibility matrix can be processed further immediately, thereby eliminating the need for storing the (typically large) similarity matrix in-memory or on-disk. We presented a Message Passing Interface (MPI) based implementation of this approach.

In a series of experiments on a cluster with 7 slave nodes, each with 48 cores and 62GB memory, we have shown that our approach scales well. In line with what can be expected from the theoretical runtime, we observed that the overall execution time grows approximately quadratically in the number of instances, and approximately linearly in the number of attributes in the dataset. Even when only 1 slave node is used, our algorithm can easily process a dataset with 1 million rows. Furthermore, our approach takes advantage of the availability of multiple cores and threads, demonstrating a clear decrease in execution time as the number of available slave nodes increases from 1 to 7, and an even further decrease when the number of threads per node increases from 1 to 46.

To the best of our knowledge, the approach we presented in this paper is the first distributed solution for the computation of lower and upper approximations for continuous valued large datasets in fuzzy rough set theory so far.

References

[1] B. Bede. *Mathematics of Fuzzy Sets and Fuzzy Logic*, volume 295 of *Studies in Fuzziness and Soft Computing*. Springer, 2013.

[2] A. Capotorti and E. Barbanera. Credit scoring analysis using a fuzzy probabilistic rough set model. *Computational Statistics & Data Analysis*, 56(4):981–994, 2012.

[3] D. Chen, S. Kwong, Q. He, and H. Wang. Geometrical interpretation and applications of membership functions with fuzzy rough sets. *Fuzzy Sets and Systems*, 193:122–135, 2012.

[4] N. Chen. Data-fusion approach based on evidence theory combining with fuzzy rough sets for urban traffic flow. *Research Journal of Applied Sciences, Engineering and Technology*, 6(11):1993–1997, 2013.

[5] C. Cornelis, R. Jensen, G. Hurtado Martın, and D. Slezak. Attribute selection with fuzzy decision reducts. *Information Sciences*, 180(2):209–224, 2010.

[6] J. Dai and Q. Xu. Attribute selection based on information gain ratio in fuzzy rough set theory with application to tumor classification. *Applied Soft Computing*, 13(1):211–221, 2013.

[7] D. Dubois and H. Prade. Rough fuzzy sets and fuzzy rough sets. *International Journal of General Systems*, 17:191–209, 1990.

[8] Y. Fan and C. Chern. An agent model for incremental rough set-based rule induction: a big data analysis in sales promotion. In *Proceedings of the 46th Hawaii International Conference on System Sciences*, pages 985–994, 2013.

[9] Bing Huang, Yu-Liang Zhuang, Hua-Xiong Li, and Da-Kuan Wei. A dominance intuitionistic fuzzy-rough set approach and its applications. *Applied Mathematical Modelling*, 37(12-13):7128–7141, 2013.

[10] P. Maji and S.K. Pal. Fuzzy-rough sets for information measures and selection of relevant genes from microarray data. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 40(3):741–752, 2010.

[11] Z. Pawlak. Rough sets. *International Journal of Computer Information Science*, 11:341–356, 1982.

[12] L. S. Riza, A. Janusz, C. Bergmeir, C. Cornelis, F. Herrera, D. Ślęzak, and J. M. Benítez. Implementing algorithms of rough set theory and fuzzy rough set theory in the R package "roughsets". *Information Sciences*, (in press), 2014. http://CRAN.R-project.org/package=RoughSets.

[13] C. Shang, D. Barnes, and Q. Shen. Facilitating efficient Mars terrain image classification with fuzzy-rough feature selection. *International Journal of Hybrid Intelligent Systems*, 8:3–13, 2011.

[14] B. Sun, W. Ma, and H. Zhao. A fuzzy rough set approach to emergency material demand prediction over two universes. *Applied Mathematical Modelling*, 37(10-11):7062–7070, 2013.

[15] F. Xu, L. Wei, Z. Bi, and L. Zhu. Research on fuzzy rough parallel reduction based on mutual information. *Journal of Computational Information Systems*, 10(12):5391–5401, 2014.

[16] Y. Yang and Z. Chen. Parallelized computing of attribute core based on rough set theory and MapReduce. In *Proceedings of the Conference on Rough Sets and Knowledge Technology*, pages 155–160. 2012.

[17] Y. Yang, Z. Chen, Z. Liang, and G. Wang. Attribute reduction for massive data based on rough set theory and MapReduce. In *Proceedings of the Conference on Rough Sets and Knowledge Technology*, pages 672–678. 2010.

[18] J. Yao and Y. Zhang. A scientometrics study of rough sets in three decades. In *Proceedings of the 8th International Conference on Rough Sets and Knowledge Technology*, pages 28–40, 2013.

[19] J. Zhai. Fuzzy decision tree based on fuzzy-rough technique. *Soft Computing*, 15(6):1087–1096, 2011.

[20] J. Zhang, T. Li, and Y. Pan. Parallel rough set based knowledge acquisition using MapReduce from big data. In *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 20–27, 2012.

[21] J. Zhang, T. Li, D. Ruan, Z. Gao, and C. Zhao. A parallel method for computing rough set approximations. *Information Sciences*, 194:209–223, 2012.

[22] J. Zhang, J. Wong, T. Li, and Y. Pan. A comparison of parallel large-scale knowledge acquisition using rough set theory on different MapReduce runtime systems. *International Journal of Approximate Reasoning*, 55:896–907, 2014.

[23] J. Zhang, Y. Zhu, Y. Pan, and T. Li. A parallel implementation of computing composite rough set approximations on GPUs. In *Proceedings of the Conference on Rough Sets and Knowledge Technology*, pages 240–250. 2013.