MINING@HOME: PUBLIC RESOURCE COMPUTING FOR DISTRIBUTED DATA MINING

D. Barbalace¹, C. Lucchese², C. Mastroianni³, S. Orlando⁴, D. Talia¹

¹ DEIS, University of Calabria, Rende, Italy
² ISTI, C.N.R., Pisa, Italy
³ ICAR, C.N.R., Rende, Italy
⁴ Department of Computer Science, University of Venice, Italy

barbalace@si.deis.unical.it, c.lucchese@isti.cnr.it, mastroianni@icar.cnr.it, orlando@dsi.unive.it, talia@deis.unical.it

Abstract

Several kinds of scientific and commercial applications require the execution of a large number of independent tasks. One highly successful and low cost mechanism for acquiring the necessary compute power for these applications is the “public-resource computing”, or “desktop Grid” paradigm, which exploits the computational power of private computers. So far, this paradigm has not been applied to data mining applications for two main reasons. First, it is not trivial to decompose a data mining algorithm into truly independent sub-tasks. Second, the large volume of data involved makes it difficult to handle the communication costs of a parallel paradigm. In this paper, we focus on one of the main data mining problem: the extraction of closed frequent itemsets from transactional databases. We show that is possible to decompose this problem into independent tasks, which however need to share a large volume of data. We thus introduce a data-intensive computing network, which adopts a P2P topology based on super peers with caching capabilities, aiming to support the dissemination of large amounts of information. Finally, we evaluate the execution of our data mining job on such network.

1. Introduction

In this work we aim to explore the opportunities offered by the volunteer computing paradigm for making feasible the execution of compute-intensive data mining jobs that have to explore very huge data sets.

On the one hand, during recent years, volunteer computing has become a success history for many scientific applications. In fact, Desktop Grids, in the form of volunteer computing systems, have become extremely popular as a mean to garnish many resources for a low cost in terms of both hardware and
The use of volunteer computing platforms has grown significantly. Two of the popular volunteer computing platforms available today are BOINC and XtremWeb.

BOINC [Anderson, 2004] is by far the most popular volunteer computing platform available today, and to date, over 5 million participants have joined various BOINC projects. The core BOINC infrastructure is composed of a scheduling server and a number of clients installed on users’ machines. The client software periodically contacts a centralized scheduling server to receive instructions for downloading and executing a job. After a client completes the given task, it then uploads resulting output files to the scheduling server and requests more work. The BOINC middleware is especially well suited for CPU-intensive applications but is somewhat inappropriate for data-intensive tasks due to its centralized nature that currently requires all data to be served by a group of centrally maintained servers. BOINC was successfully used in projects such as Seti@home, Folding@home, and Einstein@home.

XtremWeb [Cappello et al., 2005][Fedak et al., 2001] is another Desktop Grid project that, like BOINC, works well with “embarrassingly parallel” applications that can be broken into many independent and autonomous tasks. XtremWeb follows a centralized architecture and uses a three-tier design consisting of a worker, a coordinator, and a client. The XtremWeb software allows multiple clients to submit task requests to the system. When these requests are dispensed to workers for execution, the workers will retrieve both the necessarily data and executable to perform the analysis. The role of the third tier, called the coordinator, is to decouple clients from workers and to coordinate tasks execution on workers.

On the other hand, due to the exponential growth of the information society, data mining applications need to deal with larger and larger amounts of data, so that, in the future, they will likely become large scale and expensive data analysis activities. However, the nature of data mining applications is very different from usual “@home” applications. First, they are not easily decomposable into a set of small independent tasks. Second, they are data-intensive, that is any sub-task needs to work on a large portion of data. These two issues make it very challenging to distribute sub-tasks to volunteer clients. In fact, neither BOINC or XtremWeb does utilize ad hoc algorithms for the propagation of large amounts of data. Nevertheless, we believe that data mining may take advantage of a volunteer computing framework in order to accomplish complex tasks that would be otherwise intractable.

In this paper we focus on the closed frequent itemsets mining problem (CFIM). This requires to extract a set of significant patterns from a transactional dataset, among the ones occurring not less than a user defined threshold.

We also introduce a novel data-intensive computing network, which is able to efficiently carry out our mining task by adopting a volunteer computing paradigm. The network exploits caching techniques across a super-peer net-
work to leverage the cost of spreading large amounts of data to all the computing peers.

Some previous efforts aimed at exploiting Grid functionalities and services to support distributed data mining algorithms. Grid Weka [Khoussainov et al., 2004] and Weka4WS [Talia et al., 2005] extend the Weka toolkit to enable the use of multiple computational resources when performing data analysis. In those systems, a set of data mining tasks can be distributed across several machines in an ad-hoc environment. However, they do not use any decentralized or peer-to-peer technique to improve scalability and fault-tolerance characteristics.

We used an ad hoc simulator, fed with statistics concerning a real CFIM application, in order to evaluate our data-intensive computing network. To the best of our knowledge, this is the first time that the deployment of a complex data mining task over a large distributed peer-to-peer network is shown to be effective.

2. Parallel Mining of Closed Frequent Itemset

Frequent Itemsets Mining (FIM) is a demanding task common to several important data mining applications that look for interesting patterns within databases (e.g., association rules, correlations, sequences, episodes, classifiers, clusters). The problem can be stated as follows. Let $I = \{a_1, \ldots, a_M\}$ be a finite set of items or singletons, and let $D = \{t_1, \ldots, t_N\}$ be a dataset containing a finite set of transactions, where each transaction $t$ is a subset of $I$. We call $k$-itemset a set of $k$ items $I = \{i_1, \ldots, i_k \mid i_j \in I\}$. Given a $k$-itemset $I$, let $\sigma(I)$ be its support, defined as the number of transactions in $D$ that include $I$. Mining all the frequent itemsets from $D$ requires to discover all the itemsets having a support greater or equal to a given minimum support threshold $\bar{\sigma}$. We denote with $L$ the collection of frequent itemsets, which is indeed a subset of the huge search space given by the power set of $I$.

State-of-the-art FIM algorithms visit a lexicographical tree spanning over such search space, by alternating candidate generation, and support counting steps. In the candidate generation step, given a frequent itemset $X$ of $|X|$ elements, new candidate $(|X| + 1)$-itemsets $Y$ are generated as supersets of $X$ that follow $X$ in the lexicographical order. During the counting step, the support of such candidate itemsets is evaluated on the dataset, and if some of those are found to be frequent, they are used to re-iterate the algorithm recursively.

The collection of frequent itemsets $L$ extracted from a dataset is usually very large. This makes the task of the analyst hard, since he has to extract useful knowledge from a huge amount of patterns, especially when very low minimum support thresholds are used. The set $C$ of closed itemsets [Wille, 1982] is a concise and lossless representation of frequent itemsets that has
replaced traditional patterns in all the other mining tasks, e.g. sequences and graphs.

**Definition 1**  
An itemset I is said to be **closed** iff

\[ c(I) = f(g(I)) = f \circ g(I) = I \]

where the composite function \( c = f \circ g \) is called Galois operator or closure operator, and the two functions \( f, g \) are defined as follows:

\[ f(T) = \{i \in I \mid \forall t \in T, i \in t\} \]
\[ g(I) = \{t \in D \mid I \subseteq t\} \]

The closure operator defines a set of equivalence classes over the lattice of frequent itemsets: two itemsets belong to the same equivalence class iff they have the same closure, i.e. they are supported by the same set of transactions. Closed itemsets are the maximal elements of these equivalence classes (see Fig. 2).

It comes from Definition 1 that it is not easy to find the closure of a pattern: either we need a global knowledge of the dataset or a global knowledge of the collection of frequent itemsets and their equivalence classes. For this reason, it is not easy to design a parallel CFIM algorithm.

The first algorithm for mining closed itemsets in parallel, MT-CLOSED [Lucchese et al., 2007], was proposed very recently. Analogously to other CFIM algorithms, MT-CLOSED executes two scans of the dataset in order to initialize its internal data structures. A first scan is needed to discover frequent single items, denoted with \( L^1 \). During a second scan, a vertical bitmap representing the dataset is built by considering frequent items only. The resulting bitmap has size \( \vert L^1 \vert \times \vert D \vert \) bits, where the \( i \)-th row is a bit-vector representation of the tid-list \( g(i) \) of the \( i \)-th frequent item.
The kernel of the algorithm consists in a recursive procedure that exhaustively explores a subtree of the search space given its root. The input of this procedure is a seed closed itemset \( X \), and its tid-list \( g(X) \). Initially, \( X = c(\emptyset) \), and \( g(X) = D \). Similarly to other CFIM algorithms, given a closed itemset \( X \), new candidates \( Y = X \cup i \) are created according to the lexicographic order. If a candidate \( Y \) is found to be frequent, then its closure is computed and \( c(Y) \) is used to continue the recursive traversal of the search space.

Every single closed itemset \( X \) can be thought as the root of a sub-tree of the search space which can be mined independently from any other (non-overlapping) portion of the search space. Note that this is a peculiarity of MT-CLOSED. We refer to \( J = \langle X, g(X), D \rangle \) as a job description, since it identifies a given sub-task of the mining process.

Thus, it is possible to partition the whole mining task into independent regions, i.e. sub-trees of the search space, each of them described by a distinct job descriptor \( J \). One easy strategy would be to partition the search space according to frequent singletons. We would obtain \( |L_1| \) independent jobs. Unfortunately, especially with dense datasets, it is very likely that one among such jobs has a computational cost that is much higher than all the others.

Among the many approaches to solve this problem, an interesting one is [Cong et al., 2005]. First the costs of the jobs associated with the frequent singletons are estimated by running a mining algorithm on significant samples of the dataset. Then, the most expensive jobs are split on the basis of the 2-itemsets they contain.

In our setting, we are willing to address very large datasets. In this case, the large number of resulting samplings to be performed and their costs make the above strategy not suitable. Therefore, our choice is to avoid any expensive pre-processing.

First, in order to obtain a fine-grained partitioning of the search space, we will materialize jobs on the basis of the 2-itemsets in the cartesian product \( L_1 \times L_1 \). This produces a large number of jobs and sufficient degrees of freedom to evenly balance the load among workers.

Second, we will consider the opportunity to group together jobs, when their number is too large. Notice that a set of 1,000 frequent singletons results in about 500,000 jobs. Since such a large number of jobs will introduce a large overhead, we will group together \( k \) consecutive jobs, where \( k \) is a system-wide configuration parameter. We group together two consecutive jobs only if they share the same prefix. For instance, \( \{ab\} \) and \( \{ac\} \) may be grouped together, while \( \{az\} \) and \( \{bc\} \) may not.

1The input should also include the set of items used to calculate closures, which is not described here because of space constraints. Please refer to [Lucchese et al., 2007].
The reason for this constraint is given by the *partitioning optimizations* usually adopted in mining algorithm that we want to use in our caching strategies. Suppose that a job corresponds to the mining of all the itemsets beginning with the a given item $i$; then any transaction that does not contain $i$ can safely be disregarded. This technique significantly reduces the amount of data to be processed by a single job. This also explains why we only group 2-itemsets having the same prefix: we group jobs together only if the share the same projection of the data.

This data projection approach is very important in our framework. We can reduce the amount of data needed to accomplish a given job, and therefore the amount of data to be sent through the network.

3. A Data-Intensive Computing Network

We already proposed a preliminary framework for data dissemination suitable scenarios (e.g., processing of astronomical waveforms, analysis of audio files [Al-Shakarchi et al., 2007]) in which the partition of an application into independent jobs is trivial and the input dataset is the same for all the tasks. Here, the algorithm is adapted for the CFIM data mining problem, in which the specification of independent jobs is obtained through the MT-CLOSED algorithm and the input dataset may be different for different jobs.

Our algorithm exploits the presence of a super-peer network for the assignment and execution of jobs, and adopts caching strategies to make the data distribution more efficient. Specifically, it exploits the presence of different types of nodes that are available within a super-peer topology, as detailed in the following:

- the *Data Source* is the node that stores the entire data set that must be analyzed and mined.
- the *Job Manager* is the node in charge of decomposing the overall data mining application in a set of independent tasks, according to the MT-CLOSED algorithm. This node produces a *job advert* document for every task, which describes its characteristics and specifies the portion of the data needed to complete the task. This node is also responsible for the collection of output results.
- the *Miners* are the nodes that are available for job execution. A miner first issues a *job query* and a *data query* to retrieve the a job and the corresponding data.
- *Data-Cachers* are super-peers having the additional ability to cache data and the associated data adverts. Data cachers can retrieve data from the data source or other data cachers, and later provide such data to Miners.
- *Super-Peers* nodes constitute the backbone of the network. Miners connect directly to a Super-Peer, and Super-Peers are connected with one
another through a high level P2P network. Super-peers play the role of rendezvous nodes, i.e. meeting places for job or data providers and consumers. They match Miners’ queries with job and data adverts.

In order to execute our data mining algorithm, the network works as follows (see Fig. 2). A set of job adverts are generated by the Job Manager node. A job advert corresponds to the job descriptor discussed in the previous section. An available miner $M$ issues a job query (step 1), that travels across the super-peer interconnections, to the Job Manager. A matching job advert is sent back to $M$ (setp 2). Thanks to the job advert, the miner is also informed of the data necessary to complete its job. Thus, it issues a data query to discover a Data-Cacher (step 3). Since multiple Data-Cachers may answer (step 4), the miner selects the nearest one and gives it the responsibility to retrieve the required input data. In our example, the selected Data-Cacher $DC_1$ (step 5) does not hold the data neede by $M$, and issues a query to the data source $DS$ or to the other Data-Cachers (step 6). Eventually, $DC_1$ retrieves the data from $DS$ (step 7), stores it and provides it to the miner $M$ (step 8). In the future, $DC_1$ will be able to provide the same data to other miners or to other Data-Cachers. Finally, the miner $M$ executes the job.

Our implementation includes a number of techniques that can speed up the execution, depending on the state of the network and the dissemination of data. For example, in the case that the cacher $DC_1$ has already downloaded data, steps 6 and 7 are unnecessary. Also, once a Miner has discovered the Job Manager or a Data-cacher, it could decide to contact them directly without paying the cost of sending a message across the Super-Peers network. More interestingly, a miner may have the ability to store some data. In the following,
we discuss this aspect and introduce two possible caching scenarios on the miner side.

The presence of data cachers helps the dissemination of data and can improve the performance of the network. It is also useful to verify if miners themselves could give a contribution to speed up computation, in the case that they have the ability and they are willing to store some input data (in general, the public resource computing paradigm does not require hosts to store data after the execution of a job). In fact, it often happens that the input data of a job overlaps, completely or partially, with the input data of another job executed previously. Therefore, the miner could retrieve the whole data set when executing the first job, and avoid to issue a data query for the subsequent job. On the other hand, if miners have no storage capabilities, they have to download the associated input data for each job they have to execute. Therefore, two different caching strategies have been analyzed and compared:

- **Strategy #1: miners cannot store data.** The miner downloads from the data cacher only the portion of the data set that it strictly needs for job execution, and discards this data after the execution.
- **Strategy #2: miners can store data.** The miner downloads from the data cacher the entire data set the first time that it has to execute a job. Even though the miner will only use a portion of this data set, data will be stored locally and can be used for successive job executions.

Depending on the application, these simple strategies may be significantly improved. One possible approach could be to use the information present in the job adverts, in order to retrieve only those transactions of the dataset that the miner does not already store. Indeed, a wide range of opportunities is open.

4. Performance Evaluation

We used an event-based simulation framework (similar to that used in [Al-Shakarchi et al., 2007]) to analyze the performance of our super-peer protocol. In the simulation, the running times of the jobs were obtained by actually executing the serial algorithm **MT-CLOSED** on specific data, and measuring the elapsed times. To model a network topology that approximates a real P2P network as much as possible, we exploited the well known power-law algorithm defined by Albert and Barabasi [Barabási and Albert, 1999]. The bandwidth and latency between two adjacent super-peers were set to 1 Mbps and 100 ms, respectively, whereas the analogous values for the connections among a super-peer and a local miner were set to 10 Mbps and 10 ms. If during the simulation a node (e.g., a data source or a data cacher) needs to simultaneously serve multiple communications (with different miners), the bandwidth of each communication is obtained by dividing the downstream bandwidth of the server by the number of simultaneous connections.
The input dataset used to measure the running times of the various MT-CLOSED jobs is Synth2GB, which has about 1.3 millions transactions and 2.5 thousands distinct items, for a total size of 2 GB. It was produced by using the IBM dataset generator. By running the algorithm with a minimum absolute support threshold of 50,000, we obtained the info about 469,200 jobs, that were later grouped by 100 to reduce the total number of jobs. The time needed to complete the mining on a single machine was about ten hours. In order to simulate a very expensive mining task, we multiplied the running time of each job by a factor of 100. This is perfectly reasonable, since the time needed to execute a job increases exponentially when decreasing the minimum support threshold.

Figure 3(a) shows the overall running time on varying the number of mining peers by using strategies #1 and #2, in case of 10 data cachers. It is worth noting that, by using multiple distributed miners, the execution time decreases from over 1000 hours to about 20 hours when exploiting strategy #1. With strategy #2, according to which miners can store data in their own cache, the execution time is further reduced. Each miner downloads the entire data set to execute before executing the first job, and then reuses the data for all the following jobs.

The plot of Figure 3(a) also shows that when strategy #1 is adopted, an appropriate number of miners is 150, since the overall time does not decrease if additional miners are available. Of course, the “optimal” number of miners strictly depends on the problem, which impact on data sizes and job processing times.

Also the number of available data cachers has an important influence on the overall execution time. To analyze this issue, in Figure 3(b) we report the execution time obtained with strategy #1 and 150 active miners, on varying the number of data cachers. The execution time decreases as the number of data
cachers increases from 1 to 10, since miners can concurrently retrieve data from different data cachers, thus decreasing the length of single download operations. However, the execution time increases again as more than 10 data cachers are made available. The main reason is that many of these data cachers retrieve data directly from the data source, so that the downstream bandwidth of the data source is shared among a large number of connections. Results show that the time needed to distribute data to more than 10 data cachers is not compensated by the time saved in data transfers from data cachers to miners. Therefore an “optimum” number of data cachers can be estimated. This number is 10 in this case, but in general depends on the application scenario, for example on the number and length of the jobs to execute.

5. Conclusions

In order to test our volunteer network, we chose a very tough data mining task. In particular, the extraction, in a reasonable time, of all the (closed) frequent patterns from a huge database, with low minimum support. This is only feasible if we can exploit a multitude of computing nodes, like those made available by our volunteer network. Due to the features of the embarrassingly parallel tasks obtained, which require to effectively distribute large sets of similar data to the miner peers, we tested an efficient data distribution technique based on cooperating super-peers with caching capabilities. The first simulated tests of our network, for which we used parameters obtained from real runs of our data mining application, are very promising.

Our approach for distributing large amounts of data across a P2P data mining network, opens up a wide spectrum of opportunities. In fact P2P data mining a recently gained lots of interest. Not only because of the computing power made available by volunteer computing, but also because of new emerging scenarios, such as sensor networks, where data are naturally distributed, and nodes of the network are not reliable. Even if many P2P data mining algorithms, such as clustering [Datta et al., 2006] and feature extraction [Wurst and Morik, 2007], have been developed, still they suffer the cost of data dissemination. Not only our approach alleviates this cost, but it can easily deal with failure and load balancing problems. For these reasons we believe that our proposed data-intensive computing network may be a bridge towards P2P computing for other data mining applications dealing with large amounts of data, e.g. web documents clustering, or dealing with a distributed environment, e.g. analysis sensor data.

Many directions for future works are open. Among them, we can mention: (i) the adoption of more advanced strategies to disseminate and cache data in the P2P network, (ii) the use of the volunteer computing paradigm to solve even
more challenging data mining problems and (iii) the testing of the presented approach on a real distributed platform.

References


