

## Distributed Data Mining in Desktop Grids

*Daniela Barbalace*

barbalace@si.deis.unical.it

*Università della Calabria, Rende (CS), Italy*

*Claudio Lucchese*

c.lucchese@isti.cnr.it

*ISTI-CNR, Pisa, Italy*

*Carlo Mastroianni*

mastroianni@icar.cnr.it

*ICAR-CNR, Rende (CS), Italy*

*Salvatore Orlando*

orlando@dsi.unive.it

*Department of Computer Science, University of Venice, Italy*

*Domenico Talia*

talia@deis.unical.it

*Università della Calabria, Rende (CS), Italy*



CoreGRID Technical Report  
Number TR-0141

June 17, 2008

Institute on Knowledge and Data Management  
Institute on Architectural Issues: Scalability, Dependability,  
Adaptability

CoreGRID - Network of Excellence

URL: <http://www.coregrid.net>

# Distributed Data Mining in Desktop Grids

Daniela Barbalace

barbalace@si.deis.unical.it  
Università della Calabria, Rende (CS), Italy

Claudio Lucchese

c.lucchese@isti.cnr.it  
ISTI-CNR, Pisa, Italy

Carlo Mastroianni

mastroianni@icar.cnr.it  
ICAR-CNR, Rende (CS), Italy

Salvatore Orlando

orlando@dsi.unive.it  
Department of Computer Science, University of Venice, Italy

Domenico Talia

talia@deis.unical.it  
Università della Calabria, Rende (CS), Italy

*CoreGRID TR-0141*

June 17, 2008

## 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.

---

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

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 manpower. Two of the popular volunteer computing platforms available today are BOINC and XtremWeb.

BOINC [2] 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 [4][7] 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 network 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 [8] and Weka4WS [10] 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  $\mathcal{I} = \{a_1, \dots, a_M\}$  be a finite set of *items* or *singletons*, and let  $\mathcal{D} = \{t_1, \dots, t_N\}$  be a dataset containing a finite set of *transactions*, where each transaction  $t$  is a subset of  $\mathcal{I}$ . We call  $k$ -itemset a set of  $k$  items  $I = \{i_1, \dots, i_k \mid i_j \in \mathcal{I}\}$ . Given a  $k$ -itemset  $I$ , let  $\sigma(I)$  be its *support*, defined as the number of transactions in  $\mathcal{D}$  that include  $I$ . Mining all the frequent itemsets from  $\mathcal{D}$  requires to discover all the itemsets having a support greater or equal to a given minimum support threshold  $\bar{\sigma}$ . We denote with  $\mathcal{L}$  the collection of frequent itemsets, which is indeed a subset of the huge search space given by the power set of  $\mathcal{I}$ .

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

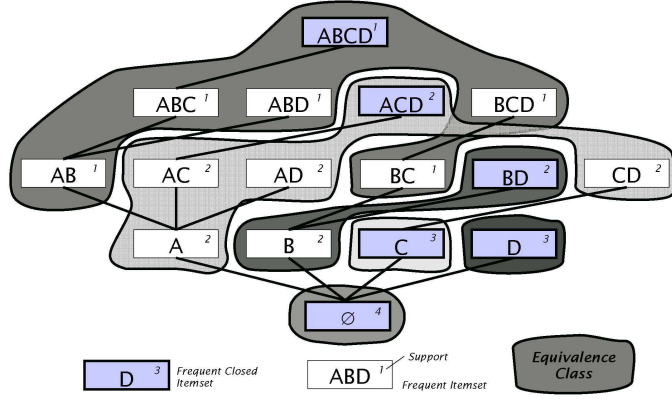


Figure 1: Lexicographic spanning tree of the frequent itemsets, with closed itemsets and their equivalence classes, mined with  $\bar{\sigma} = 1$  from the dataset  $\mathcal{D} = \{\{B, D\}, \{A, B, C, D\}, \{A, C, D\}, \{D\}\}$ .

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  $\mathcal{L}$  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  $\mathcal{C}$  of closed itemsets [11] 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:  $f(T) = \{i \in \mathcal{I} \mid \forall t \in T, i \in t\}$ ,  $g(I) = \{t \in \mathcal{D} \mid I \subseteq t\}$

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 [9], 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  $\mathcal{L}^1$ . During a second scan, a vertical bitmap representing the dataset is built by considering frequent items only. The resulting bitmap has size  $|\mathcal{L}^1| \times |\mathcal{D}|$  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  $g(X)$ <sup>1</sup>. Initially,  $X = c(\emptyset)$ , and  $g(X) = \mathcal{D}$ . 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  $J = \langle X, g(X), \mathcal{D} \rangle$  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  $|\mathcal{L}_1|$  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.

<sup>1</sup>The input should also include the set of items used to calculate closures, which is not described here because of space constraints. Please refer to [9].

Among the many approaches to solve this problem, an interesting one is [5]. 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  $\mathcal{L}^1 \times \mathcal{L}^1$ . 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.

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 they 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 [1]) 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*.

The algorithm works as follows: when a data mining application must be executed, a set of job adverts are generated by the *manager* node. Each job advert specifies the items that must be included in the frequent itemsets to be mined.

An available miner issues a *job query* to retrieve one of these job adverts in the manager node. Job queries can be delivered directly to the manager node or, if the location of this node is not known (for example, if several data mining applications are concurrently running), they can travel the network through the super-peer interconnections. When a job advert is found that matches the job query, the related job is assigned to the requesting miner. The miner is also

informed, through the job advert, about the data that it needs to execute the job. The required input data can be the entire data set stored in the data source, or a subset of it.

The miner does not download data directly by the data source, but issues a *data query* to discover a data cacher. This query can actually discover several data cachers: each of these sends an ack to the miner. Then the miner selects the nearest data cacher (or the most convenient data cacher according to a given strategy) and gives it the responsibility to retrieve the input data set from the data source, or from another data cacher that has already downloaded the data. After retrieving the input data, the cacher stores it, then passes it to the miner for job execution. Of course, in the future the data cacher will be able to provide the data set to other miners that request it, and to other data cachers.

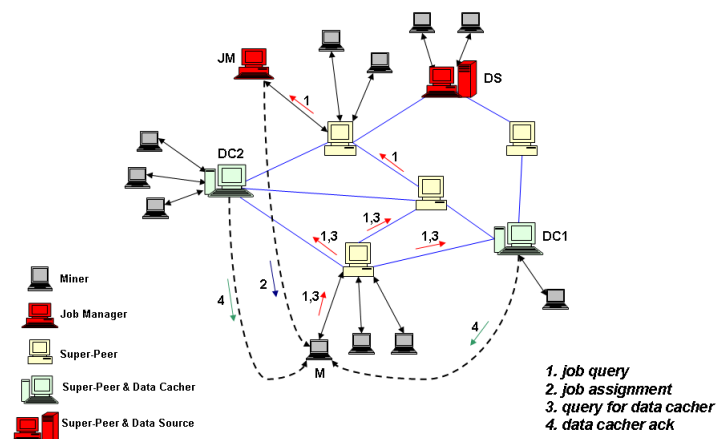


Figure 2: Caching algorithm in a sample super-peer network (1/2).

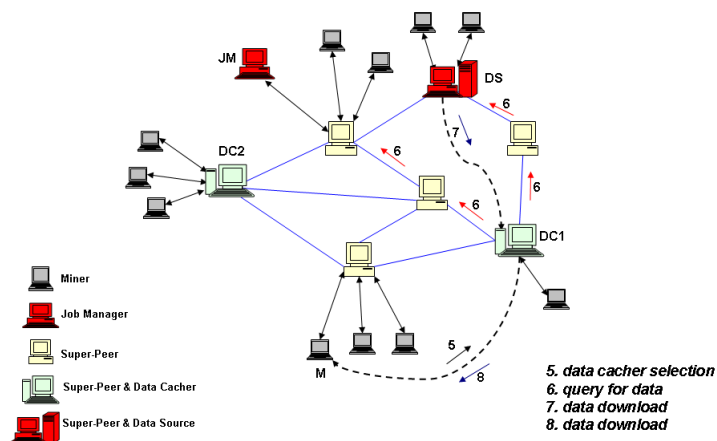


Figure 3: Caching algorithm in a sample super-peer network (2/2).

The algorithm is illustrated in Figures 2 and 3, which show the messages exchanged in a network having 7 super-peers (among which one is the data source and two are data cachers), a job manager and several miners. In the first figure, the miner *M*, available to execute a job, issues a job query (step 1) that travels across the super-peer interconnections and gets to the job manager *JM*. The job manager assigns a job to the miner and sends it the job advert that describes the job and the input data required for its execution (step 2). Afterwards, the miner issues a query to discover a close data cacher (step 3) and in this case discovers two data cachers, which advertise their presence with an ack message (step 4). The subsequent steps of the algorithm are depicted in Figure 3. The miner selects the most convenient data cacher, in this case the data cacher *DC*<sub>1</sub> (step 5). Since *DC*<sub>1</sub> does not hold the required data, it issues

a query to retrieve it from the data source or from another data cacher that has already retrieved this data (step 6). In this scenario, data is found in the data source  $DS$ . So,  $DC_1$  retrieves data from  $DS$  (step 7), stores it in the cache in order to serve future miner requests, and provides it to the miner (step 8). The miner can now execute the job, and sends the output to the job manager, through a message not shown in the figure.

The algorithm includes a number of techniques that can make execution faster, 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 7 and 8 are unnecessary. Moreover, a miner can exploit the results of discovery operations executed previously. Specifically, it needs to issue a query to discover the job manager or the data cacher only the first time that it asks to execute a job. The next times the miner can decide to directly contact the job manager and the data cacher that it has previously discovered. Finally, a miner could have the ability to cache data itself. This aspect is discussed in the following subsection.

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 [1]) 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 [3]. This model incorporates the characteristic of preferential attachment that was proved to exist widely in real networks.

The simulation scenario is summarized in Table 1.

Table 1: Simulation scenario

Scenario feature	Value
Number of super-peers, $N_{sp}$	25
Number of available miners, $N_{miners}$	250
Average number of neighbors of a super-peer	4
Size of the input data file	300 Mbytes
Number of jobs, $N_{job}$	469,200
Latency between two adjacent super-peers	100 ms
Latency between a super-peer and a local worker	10 ms
Bandwidth between two adjacent super-peers	1 Mbps
Bandwidth between a super-peer and a local worker	10 Mbps

The network contains 25 super-peers and 250 potential miners, randomly distributed among super-peers. 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 10 and 100 in the two different sets of experiments. This is perfectly reasonable, since the time needed to execute a job increases exponentially when decreasing the minimum support threshold.

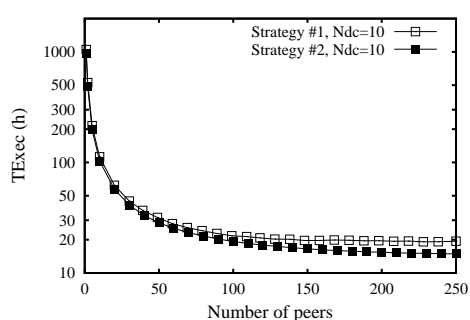


Figure 4: Overall execution time vs. the number of active miners with strategies #1 and #2 and 10 data cachers.

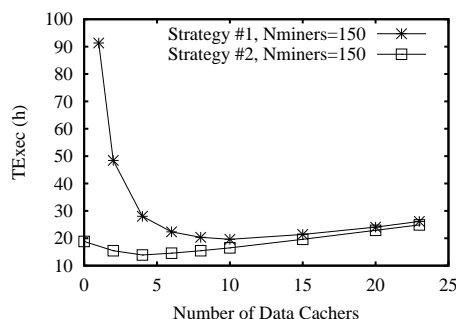


Figure 5: Overall execution time vs. the number of data cachers with strategies #1 and #2. The number of active miners is 150.

We used the factor 100 in the first set of experiments, and related results are shown in Figures 4 and 5. Figure 4 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 before executing the first job, and then reuses the data for all the following jobs.

The plot of Figure 4 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 impacts 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 5 we report the execution time obtained with the two strategies and 150 active miners, on varying the number of data cachers. With strategy #1, 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 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. With strategy #2, on the other hand, it is seen that the “optimum” number of data cachers is about 4; in fact, caching is operated directly by miners, hence there is less advantage to have nodes that are specifically dedicated to the caching of data.

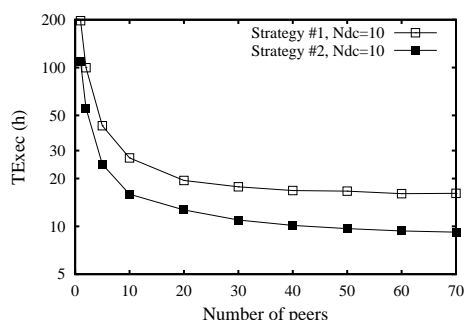


Figure 6: Overall execution time vs. the number of active miners with strategies #1 and #2 and 10 data cachers.

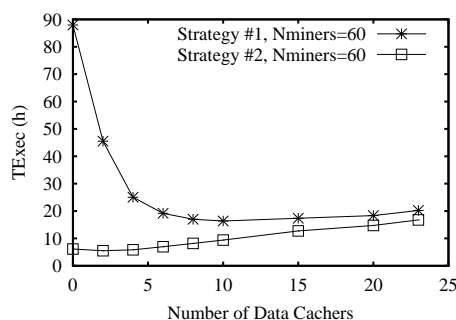


Figure 7: Overall execution time vs. the number of data cachers with strategies #1 and #2. The number of active miners is 60.

In the second set of experiments, we multiplied the execution time of every mining task by a factor 10: related results are shown in Figures 6 and 7. Figure 6 shows the overall running time on varying the number of mining peers, with strategies #1 and #2, in case of 10 data cachers. The execution time decreases from over 200 hours to about 20 hours when exploiting strategy #1 and from about 100 hours to less than 10 hours using strategy #2.

The appropriate number of active miners is about 60, while it was about 150 when using a multiplying factor equal to 100. This is a clue that more miners should be used as the computation load increases. Figure 7 reports the execution time obtained with strategy #1 and #2 and 60 active miners, on varying the number of data cachers. As in the first set of experiments, the execution time experienced with strategy #1 decreases as the number of data cachers increases from 1 to 10, then it increases. ON the other hand, if strategy #2 is adopted, it there is no advantage deriving from the use of data cachers, because caching is directly operated by miners and the computational load is not sufficiently high to justify the adoption of data cachers.

## 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 [6] and feature extraction [12], 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.

## 6 Acknowledgments

We would like to thank Ian Taylor and his colleagues at the Cardiff University for their help in defining the distributed architecture presented here.

## References

- [1] Eddie Al-Shakarchi, Pasquale Cozza, Andrew Harrison, Carlo Mastroianni, Matthew Shields, Domenico Talia, and Ian Taylor. Distributing workflows over a ubiquitous p2p network. *Scientific Programming*, 15(4):269–281, 2007.
- [2] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, 2004.
- [3] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.
- [4] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frederic Magniette, Vincent Neri, and Oleg Lodygensky. Computing on large-scale distributed systems: Xtrem web architecture, programming models, security, tests and convergence with Grid. *Future Generation Computer Systems*, 21(3):417–437, 2005.
- [5] Shengnan Cong, Jiawei Han, and David A. Padua. Parallel mining of closed sequential patterns. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 562–567, August 2005.
- [6] Souptik Datta, Kanishka Bhaduri, Chris Giannella, Ran Wolff, and Hillol Kargupta. Distributed data mining in peer-to-peer networks. *IEEE Internet Computing*, 10(4):18–26, 2006.
- [7] Gilles Fedak, Cecile Germain, Vincent Neri, and Franck Cappello. Xtremweb: A generic global computing system. In *Proceedings of the IEEE Int. Symp. on Cluster Computing and the Grid*, Brisbane, Australia, May 2001.
- [8] Rinat Khoussainov, Xin Zuo, and Nicholas Kushmerick. A toolkit for machine learning on the Grid, October 2004. ERCIM News No. 59.
- [9] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. Parallel mining of frequent closed patterns: Harnessing modern computer architectures. In *ICDM '07: Proceedings of the Fourth IEEE International Conference on Data Mining*, November 2007.

- [10] Domenico Talia, Paolo Trunfio, and Oreste Verta. Weka4WS: A WSRF-Enabled Weka Toolkit for Distributed Data Mining on Grids. In *Proc. of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005)*, Porto, Portugal, October 2005.
- [11] Rudolf Wille. Restructuring lattice theory: an approach based on hierarchies of concepts. In Ivan Rival, editor, *Ordered sets*, pages 445–470, Dordrecht–Boston, 1982. Reidel.
- [12] Michael Wurst and Katharina Morik. Distributed feature extraction in a p2p setting: a case study. *Future Gener. Comput. Syst.*, 23(1):69–75, 2007.