Anthill: A Scalable Run-Time Environment for Data Mining Applications

Renato Ferreira Wagner Meira Jr. Dorgival Guedes
Speed-DCC-UFMG
Lucia Drumond
IC-UFF

Abstract

Data mining techniques are becoming increasingly more popular as reasonable means to collect summaries from the rapidly growing datasets in many areas. However, as the size of the raw data increases, parallel data mining algorithms are becoming a necessity. In this paper we present a run-time support system that was designed to allow the efficient implementation of data-mining algorithms on heterogeneous distributed environments. We believe that the run-time framework is suitable for a broader class of applications, beyond data mining. We also present, through an example, a parallelization strategy that is supported by the run-time system. We show scalability results of two different data-mining algorithms that were parallelized using our approach and our runt-time support. Both applications scale very close to linearly with large number of nodes.

1. Introduction

Very large datasets are becoming common place in many areas. This fact is a consequence of both a continuous drop on the cost of data storage and a continuous increase in the sophistication of equipments and algorithms that collect and store such data. Analyzing these huge datasets is rapidly becoming impractical in its raw form, and data mining techniques have increased in popularity lately as means of collecting meaningful summarized information from these huge datasets. However, even the fastest sequential algorithms may not be enough to sumarize such volume of data and therefore, the development of efficient parallel algorithms for such tasks is crucial.

At the same time, Grid Computing [5] is emerging as an alternative to very expensive supercomputers. The Grid is a large distributed system created by connecting together several clusters of machines on different sites through WAN connections. The clusters are sets of homogeneous machines connected by LAN. While the potential computing power made available by the Grid is very large, exploiting this power is not trivial.

Much work has been done in developing parallel data mining algorithms over the years [11, 7]. The main limitation in all those algorithms, to the best of our knowledge, is that they have not been shown to scale well to very large number of processors. We have recently published a parallel implementation for the Frequent Itemset Mining problem for large heterogeneous distributed environments and our experimetns have shown it to scale really well [9].

In the process of creating this implementation, we generated both a parallelization strategy for a larger class of data-mining algorithms, as well as a run-time framework to support such strategy. In this paper, we focus on these two issues. The framework is refeered to as Anthill, and we show two new data mining algorithms that were implemented using the same strategy as the one for the earlier Frequent Itemset Mining problem. The two new algorithms are: k-means for clustering and ID3 for classification. Our experiments with these new applications have shown high scalability, similar to the earlier algorithm. In particular, the applications are shown to scale very close to linearly up to dozens of distributed nodes.

We believe that our framework exposes a convenient programming abstraction which is suitable for designing efficient parallel versions of algorithms in several areas besides data mining. The run-time assumes that the applications eventually run on large heterogenous distributed environments (grids). The starting point of Anthill is Datacutter [3], which is a data-flow based run-time environment for distributed architectures, but Anthill supports a richer programming model that allows a wide range of parallelizations to be efficiently implemented.

The remaining of the paper is organized as follows. In Section 2 we present Anthill run-time environment and its programming model. Section 3 describes the ID3 algorithm and its parallelization and Section 4 present some experimental results. We conclude and present some future directions in Section 6.

2. Run-time framework

In this section we describe Anthill, our run-time support framework for scalable applications on grid environments. Building applications that may efficiently exploit such environment, while maintaining good performance is a challenge. In this scenario, the datasets are usually distributed across several machines in the Grid. Moving the data to where the processing is about to take place is often inefficient. Usually, for such applications, the resulting data is many times smaller than the input. The alternative is to bring the computation to where the data resides. Success in this approach depends on the application being divided into portions that may be instantiated on different nodes on the Grid for execution. Each of these portions will perform part of the transformation on the data starting from the input dataset and until the resulting dataset.

The discussion above indicates that a good parallelization of any application in such environment should consider both data parallelism and task parallelism at the same time. Our strategy uses these two approaches together with a third approach that works over the time dimension allowing some degree of asynchronous execution of independent sub-tasks. The benefits of these three dimensions combined produces the high speedups observed in our experiments.

We based Anthill on an earlier run-time support framework for distributed environments call Datacutter, which is in turn based upon the filter-stream programming model. We now describe Datacutter and the extensions that are supported in Anthill.

2.1. Datacutter

The filter-stream programming model was originally proposed for Active Disks [1]. The idea was to create the concept of disklets or little pieces of the application computation that could be off-loaded to the processors within the disks. In that context, the disklets, or filters, are entities that perceive streams of data flowing in, and after some computation it would generate streams of data flowing out. Later, this concept was extended as a programming model suitable for a Grid environment, and a runtime system was developed that supported such model [4]. This runtime system is called Datacutter and there has been a considerable amount of effort put into various aspects of this system [8, 2].

In Datacutter, streams are abstractions for communication which allow fixed sized untyped data buffers to be transferred from one filter to another. In a sense, it is very similar to the concept of UNIX pipes. The difference is that while pipes only have one stream of data coming in and one going out, in the proposed model, arbitrary graphs with any number of input and output streams are possible.

Creating an application that runs in DataCutter is a process referred to as decomposition into filters. In this process, the application is modeled into a dataflow computation and broken into a network of filters. At execution time, the filters that compose the application are instantiated on several machines comprising a Grid and the streams are connected from source to destination.

To execute an application, a description of the filters and the streams that connect them need to be provided to the run-time environment. With that information, a number of copies of each of the filters are instantiated on different nodes of the distributed environment. These are referred to as transparent copies of a filter.

2.2. Anthill

In this paper we presented our parallel programming model and discuss how it supports highly scalable distributed computing. Our approach is based on the simple observation that the applications can be decomposed in a pipeline of operations, which represents task parallelism.

Further, for many applications, the execution consists of multiple interations of this pipeline. The application starts with an initial set of possible solutions, and as these possibilities are passed down the pipeline, new possible solutions are created. In our experience, we noticed that many applications fit this model. Also, this strategy allows asynchronous execution, in the sense that several possible solutions are being tested simultaneously at run-time.

Our proposed model, therefore, consist of exploiting maximum parallelism of the applications by using all three possibilities discussed above: task parallelism, data parallelism and asynchrony. Because the actual compute units are copies of pipeline stages, we can have a very fine grain parallelism and since all these are happenening asynchronously, the execution will be mostly bottleneck free. In order to reduce latency, the grain of the parallelism should be defined by the application designer at run-time.

Three important issues arise from this proposed model:

1. The transparent copies mechanism allows every stage of the pipeline to be distributed across many nodes of a parallel machine and the data that goes through that stage can be partitioned across the transparent copies, which represents data parallelism.Some times it is necessary for a certain data block to reach one specific copy of a stage of the pipeline;
2. These distributed stages often have some state, which need to be maintained globally;
3. Because of the nature of the application decomposition, it can be very tricky for them to detect that the computation is finished.

These issues are discussed on the folowing subsections.

2.3. Labeled stream

The labeled stream abstraction is designed to provide a convenient way for the application to allow a customized routing of the message buffers to specific transparent copies of the receiving filter. As mentioned earlier, each stage of the application pipeline is actually executed on several different nodes on the distributed environment. Each copy of a pipeline stage is different than the other in the sense that they do data parallelism, meaning that each transparent copy will handle a distinct and independent portion of the space comprised by the stage's input data.

Which copy should handle any particular data buffer depends upon the data itself. With labeled stream we add a label to every message that traverses the stream thus creating a tuple $< l, m >$ where l is the label and m is the original message. Instead of just sending the message m down the stream, the application will now send the entire tuple $< l, m >$. Associated to each labeled stream, there is also a hash function. For every message tuple traversing the stream, the hash function is called with the tuple as parameter. The output of this hash function indicates to the system the particular copy to which that message should be delievered.

This mechanism gives the application total control of the messages. Because the hash function is called at runtime, the actual routing decision is taken individually for each message and can change dynamically as the execution progresses. This feature is convenient for it allows dynamic reconfiguration, which is particularly useful to balance the load on dynamic and irregular applications. The hash function is also a little bit relaxed in the sense that the output does not have to be necessarily return one single filter. Instead, it can output a set of filters, and in that case, a message can be replicated and sent to multiple instances. This is particularly useful for applications in which one single input data element influences several output data elements. Broadcast is one instance of this situation.

2.4. Global persistent storage

As mentioned earlier, each stage of the pipeline is distributed across many nodes on a Grid environment. Often times these stages are stateful. This means that the stage has an internal state which changes as more and more of the computation chunks is passed down the pipeline. Anthill needs a mechanism that allows the set of transparent copies of any filter to share a global state. For some applications, once the stage is partitioned across the nodes, the state variables on each transparent copy will reside locally on them. But in many cases, this state may need to be updated on different locations. This is particularly true for situations where the applications are dynamically reconfiguring itself to balance the workload, or in failure situations in which the system is automatically recovering.

If we consider the fault tolerance scenario, we add interesting features to this state. It now requires to be stable, in the sense that it has transactional property. Once a change is committed on the state, it has to be maintained. So, having multiple copies of the state on separate hosts is very important for the sake of safety.

As described above, two features are very important for any data structure maintaining the global state. First, the several data points within this state need to migrate conveniently from one filter to the other as the computation progresses. Second, it has to be stable in the sense that the data stored there need to be preserved even in the case of failures on individual hosts running filter copies. We are implementing a tuple space which is similar to Linda tuple space to maintain the state. Such structure seems very convenient for our purposes. Whenever a filter copy updates a data element, it is updated on the tuple space. A copy of it is maintained on the same filter for performance reasons, while a copy of it will be forward to another host for safekeeping. The system then allows some degree of fault tolerance in the sense that once a copy of a tuple is safely stored on a different host, the update is assumed to be committed.

2.5. Termination problem

In Anthill, applications are modelled as generic directed graphs. As long as such graphs remain acyclic, termination detection for any application is straight forward: whenever data in a stream ends the filter reading from it is notified. When it finishes processing any outstanding tasks it may propagate the information that the flow ended to its outgoing streams and terminate. When application graphs have cycles, however, the problem is not that simple.

It may be the case that filters operating in a cycle cannot decide by themselves whether a stream has ended or not. Remember that each filter may have any number of copies, all completely independent. Therefore, although a filter copy may have local information indicating its job is done, processing taking place in another copy may produce new data that might travel through the loop and reach the first filter again.

When such situation happens in an application in Anthill, leaving the task of detecting the termination condition to the programmer is not reasonable solution. One of the goals of the system is exactly to simplify the development of the application, what would be compromised in that case. In order to avoid that, Anthill implements a complete distributed termination detection protocol which may be relied upon by applications that require it. The protocol is implemented in the run time system, so applications need not be concerned with it. Whenever the programmer designs a filter graph

with cycles all that is required is an indication of which stream should be chosen by the run time system to insert an end-of-stream notification. The insertion of that message breaks the cycle and causes the filters to propagate the information accordingly.

We now describe in more details our termination algorithm.

2.5.1. Termination Detection Protocol For the sake of the termination protocol, the filter graph is replaced by the graph with all filter copies, where each copy is seen as connected to all copies of the filters that are connected to its filter in both directions (a copy is not connected to the other copies of the same filter, since that is a premise of the original filter-stream model). The algorithm works in rounds, when some filter copy suspects computation has completed and begins to contact its neighbors. If some filter is still computing and produces new data, that round fails and the algorithm proceeds to another round sometime in the future. If all copies of all filters agree on termination during a single round, termination is reached. The final decision is left for a process leader, responsible for collecting information from all filters.

Three types of message are then exchanged in the protocol: copies that suspect termination was reach send SUSPECT(R) to their neighbors stating they suspect termination in round R; when a copy reaches an agreement with all its neighbors, it notifies the process leader using a TERMINATE(R) message, also identifying the round number; if the leader collects TERMINATE messages from all filter copies for a same round it brodcasts a END message back to them. Although streams are unidirectional at the application level, the run-time system uses them bidirectionally and guarantee the communication channels are reliable and messages between two filter copies are always delivered in the order they were sent.

Besides the round counter, **R**, each filter copy keeps a list of the neighbors which suspect the same termination round **R** has been reached. The core of the protocol is illustrated by the extended Finite State Machine (FSM) in Figure 1. Each filter copy may be in one of two states: as long as the copy is running and/or there is data still to be read from its input streams, it remains in the *running* state. If the filter code blocks waiting for data from streams that are empty, that is an indication that it is done computing (in fact, it waits for a short interval before taking that decision, in case data is about to arrive).

While a filter is computing and/or has data in its input streams still to be read it does not propagate messages for the termination protocol. If it receives data from another filter (an application message), it removes the sender from its list of neighbors suspected of having terminated, since it is obviously computing. If it receives a SUSPECT(R') message from another node, it first checks whether the indicated round R' is the same it is in; if that is not the case and the R' round is newer (R' > R), it updates R to the new value and resets its list of suspects. After that, it adds the sender of the SUSPECT message to its list.

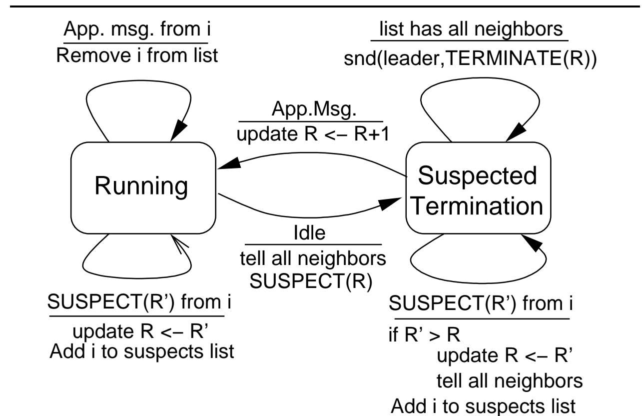

Figure 1. Extended FSM for the termination algorithm

If the run-time system in a filter copy detects it has been idle for some time (no computation taking place, and the copy is blocked waiting for data in an empty input stream), it moves to a *suspecting termination* state and notifies all its neighbors sending them a SUSPECT message with its current round number. It keeps the list of suspected neighbors it collected while in *running* state, since they were considered to be in the same termination round as itself.

When in *suspecting* state, a copy keeps track of which of its neighbors are in the same round as itself and have also reached a possible termination state. As it receives SUSPECT messages from its neighbors it adds them to the list of suspects. No reply message is needed since, as that copy is alread in round R, it must have send SUSPECT messages to all neighbors when it entered that state.

If the copy receives a SUSPECT message with a larger round number R', it indicates other copies may have gone farther in their processing while that copy remained waiting for a consensus from its neighbors. It must therefore update its round counter to the new value and clear its list of suspects (since all there belonged to a previous round, now). Only the sender of the message will be added to the list at that point.

Whenever a copy has collected SUSPECT from all its neighbors for a given round, there is a widespread suspicion that termination has been reached, although that may be true for just the vicinities of that copy. At that point the process leader must be informed with a TERMINATE(R) message.

While in the *suspecting* state, application messages may still arrive; it may be the case that one of its neighbors had just been computing for a longer time before it had any data to send. When that happens, the arrival of data in a stream is bound to cause new computation to start in that filter copy, so it must give up its suspicions about termination and get back to work. At that point, it must clear its list of suspected neighbors, prepare itself for a new (future) round by incrementing its round counter, and switch back to the *running* state.

The process leader, on its turn, must keeep track of the newest termination round it has heard of (`R`). Whenever it receives a `TERMINATE(R')` message from a filter copy, it must compare `R` and `R'`: if `R'` is lower, the message may be simply discarded, since it relates to a round that is already known to have passed; if `R'` is larger than `R`, a new round was started, and `R` is not relevant anymore, so the list of terminated copies must be cleared and just the sender of that message must be added to it; finally, if they are equal, another filter copy has joined the group of processes that suspect termination was reached, so it must be added to the list of processes in termination. When that list is complete with all processes the leader can declare the aplication over. At that point it broadcast the `END` message and all processes take their final steps toward the end. In the filter stream model, that leads to the reinitialization of the terminatio protocol, and the stream selected by the user is closed, delivering an end-of-stream notification to the filter reading from it.

Since the filter graph describing an application is required to be strongly connected (although directed), the graph created by the relation of each filter copy to its neighbors will also be, specially since the *neighbor* relationship is always bidirectional. That way, when all filter copies do reach global termination, they will all be in the *suspecting* state. Since the value of the round counter grows monotonically and only gets added when a copy switches back to the *running* state, all copies are bound to converge to a same value of `R`, when termination will be agreed upon.

3. Parallelization of ID3

In this section we describe our parallel implementation of the ID3 decision tree algorithm. In a decision tree, the leaf nodes are the individual data elements. The internal nodes contain an attribute and each descending pointer encodes a possible value for the attributed mapped on the node which will distinguish the descendants. The depth of such tree is the maximum number of questions about attribute values that need to be asked about the data element in order to find one single element on the data. The basic idea of the ID3 algorithm is to use a top-down and greedy search on the data to find the most discriminating attribute on each level of the tree. For the sake of the filter definition, we distinguish three main tasks to insert a node in the decision tree:

1. For each value of each attribute, count the number of instances that have that value;
2. Compute the information gain of each attribute;
3. Find the attribute with the highest information gain.

The starting point of the ID3 algorithm is a set of m tuples, containing instances of n attributes and one out of c possible classes. Each attribute a may assume v_a values. The tree generation process is based on discriminants. Each discriminant is a test on an specific attribute that is used to divide a set of tuples in two or more subsets (depending on the number of different values that occur for the discriminant). Initially, there is no discriminant and the partition is the whole set of tuples. Then, as we find new discriminants we recursively partition the set of tuples into subsets, until all tuples in one partition belong to the same class.

The pseudo-code of the algorithm is presented in Figure 2. Lines 1–3 are the initialization of the $Partitions$, the first one being the entire database. The loop in line 4 will execute until there are no new partitions. For each of them (line 5), we select the tuples that will compose the partition q in line 6. Then we compute the information (using an entropy metric) for each attribute and instance value in lines 7–10. Lines 11-13 compute the information gain for each attribute and, in the final step, find the attribute that yields maximum information gain and inserts the corresponding

1 p.atr = None
2 p.instance = None
3 p.dataset = T
4 while ($\exists p \in Partitions$)
5     $Partitions- = p$
6     $q = \{t \in p.dataset \wedge p.atr = p.instance\}$
7     $\forall a \in Attributes$
8         $\forall v \in Values(a)$
9             $prob_c = \frac{|\{t \in q \wedge t.a = t.v \wedge t.c = c\}|}{|\{t \in q \wedge t.a = t.v\}|}$
10             $info_{a,v} = \sum_{c \in Classes} -prob_c * log(prob_c)$
11     $\forall a \in Attributes$
12         $prob_v = \frac{|\{t \in q \wedge t.a = t.v\}|}{|q|}$
13         $gain_a = \sum_{v \in Values(a)} info_{a,v} * prob_v$
14     $disc = a | gain_a isMaximum$
15     $\forall v \in Values(disc)$
16         $p.atr = disc$
17         $p.instance = v$
18         $p.dataset = q$
19         $Partitions+ = p$

Figure 2. ID3 Algorithm

partitions into $Partitions$. In terms of parallelization, there are two multi-target reductions in lines 10 and 13, which identify boundaries among filters. We divide the processing into three filters. The first filter, named *Counter*, performs the operations associated with lines 4 to 9, and is responsible for counting the number of instances that each value of each attribute has. The second filter, *Attribute*, performs the operations associated with lines 10 to 12, which corresponds to computing the information gain on each of the attributes tested on the previous filter. The third, *Decision*, performs the remaining operations, which corresponds to communicating the decision of the appropriate attribute to back to the first filter where the process will continue, selecting new discriminating attributes for each of the produced classes.

In our algorithm we exploited two dimensions of parallelism: base partition and decision tree node pipelining. Base partition is achieved by running several instances of the counter filter, so each instance process a subset of the database and pipelining is when one filter is processing one node of the tree the others are processing others nodes so all nodes stay busy practically all the time. Granularity of base partition parallelism is very fine. We can assign a single instance to a filter with no changes to the code. Another source of parallelism is asynchrony. The filters may be working on multiple partitions simultaneously, in an attempt to achieve maximum efficiency.

4. Experimental Results

In this section we evaluate the implementation of two data mining algorithms in Anthill, focusing on their efficiency and scalability. The experiments were run on a 16 node cluster of PCs, connect using switched Fast Ethernet. Each node is a 3GHz Pentium IV with 1GB main memory, running Linux 2.6.

4.1. ID3

We start with an evaluation of our ID3 implementation, as described earlier. In these experiments, we run the *Decision* filter alone on a separate node. The other nodes run both *Counter* and *Attribute* filters.

To evaluate our algorithm, we used synthetic datasets that are described in [10]. In particular we used two classification functions with different complexity: functions 2 and 7. Function 2 is simpler and produces smaller decision trees when compared with function 7. In table 1 we show the characteristics of the datasets generated for these two functions. The notation Fx-Ay-DzK is used to denote a dataset with function x, containing y attributes and $z * 1000$ instances.

Dataset	DB Size (MB)	No. Levels	Max Leaves/Level
F2-A32-D1000K	172	8	5612
F7-A32-D750K	129	8	8195

Table 1. Dataset characteristics

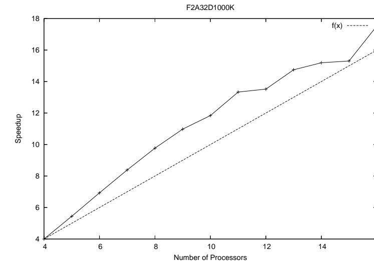

Figure 3. Speedup F2-A32-D1000K

We start by analyzing the speedups. Figures 3 and 4 show the speedups for datasets F2 and F7, respectively. We can observe that the executions of the F2 dataset scale better than those using F7, and both show superlinear behavior. F2 scales better because it is simpler and demands less memory, which seems to affect the speedup significantly. In order to understand the superlinear speedups, we performed a detailed analysis of the processor cache usage as we change the number of processors. We used PAPI (Performance Application Programming Interface [6]) to measure the number of cache misses for each configuration. Figures 5 and 6 shows that there is a substantial drop on the number of cache misses as processors are added for the execution, which is expected, and explains the superlinear behavior. For instance, in figure 5 we notice that increasing the number of processors from 4 to 14 (a factor of 3.5), resulted in a reduction of the total number of cache misses by a factor of 11.

We now focus our analysis on three criteria, to demonstrate how the various aspects of Anthill collaborate to the observed speedups.

4.1.1. Task Analysis Each task in our algorithm is associated with analyzing and determining the discriminant for

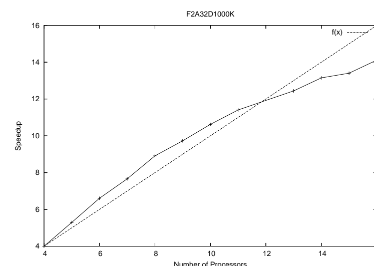

Figure 4. Speedup F7-A32-D750K

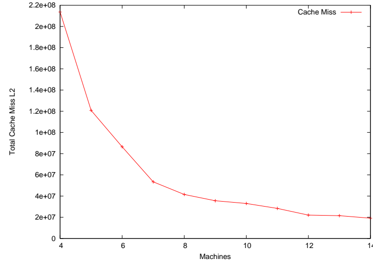

Figure 5. Total Cache Misses X Number of Processors - F2-A32-D1000K

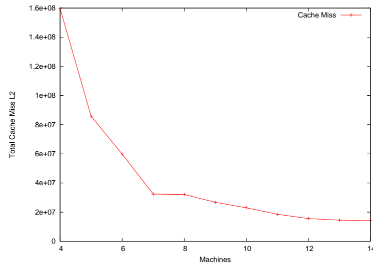

Figure 6. Total Cache Misses X Number of Processors - Speedup F7-A32-D750K

a decision tree node. Asynchrony arises by overlapping the processing of several tree nodes, which may belong to the same tree level or not. Notice that all tasks from the same tree level are independent and the parallelism in this case is trivial, and has been exploited in other contexts. We are interested in verify whether we may observe tasks from more than one level being executed simultaneously, thus exploiting all the potential parallelism present on the algorithm. We evaluate the level of asynchrony by plotting the number of active tasks from each tree level across time. Figure 7 shows the tasks behavior during the execution unning on 16 processors using F7 as input. We clearly see tasks from more than one tree level overlaping during the whole experiment (e.g., execution time 325), explaining the algorithm efficiency.

4.1.2. Filter Analysis Table 2 shows the break-down of the task execution time per filter. We consider executions from 9 to 12 nodes and F7 as the input. We observe that the majority of the processing time (over 95%) of the task occurs in the *Counter* filter.

We confirm the higher demand imposed by the *Counter* filter checking the message counters for the same configurations, as presented in Table 3, where we observe how the amount of data to be processed decreases as we go from the *Counter* to the *Decision* filter. Finally, it is interesting

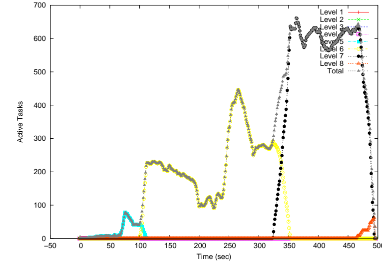

Figure 7. Active Tasks X Execution Time

Number of Processors	Counter	Attribute	Decision
9	96.17	3.36	0.46
10	95.27	4.20	0.52
11	96.13	3.30	0.56
12	96.10	3.36	0.52

Table 2. Percent of the Time in Each Filter. Test F7A32D750K.

to notice that there is a significant amount of parallelism to explore, since the elapsed time for executing the tasks is usually larger than the elapsed time in each processor (Figure **??**), which demonstrates that a larger number of processors would allow even better results.

It is interesting to note, that, despite the high variability on the demand imposed by the different filters, the application scaled very well. Other task parallelization schemes, such as pipelining, may suffer significant performance degradation as a result of such imbalance, but our application was not affected at all.

4.1.3. Filter Instance Analysis Finally, we evaluated the performance of the filter instances, in order to evaluate the imbalance that is generated by the data skewness, by the labeled stream, or even both. One basic metric for such evaluation is the variability of the execution time of each filter instance, since the execution time of the tasks may vary significantly among tasks, and comparing their times does not quantify the load imbalance among filter instances. We then calculated, for each task and for each filter, the relative standard deviation among the execution times of the filter instances. The resulting values are presented in Table 4 where we observe a slight increase on the variability

Number of Processors	Counter	Attribute	Decision
9	36785592	408649	15185
10	40872880	408649	15185
11	44960168	408649	15185
12	49047456	408649	15185

Table 3. Number of Sent Messages. Test F7A32D750K.

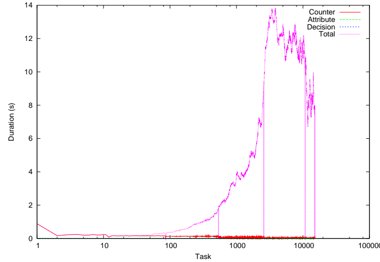

Figure 8. Elapsed time per filter and overall

Procs	Counter	Attribute
9	40.82%	28.45%
10	45.10%	28.59%
11	46.59%	29.65%
12	50.39%	29.79%

Table 4. Variability of execution times among filter instances

with the number of instances, as expected, since the load assigned to each instance is reduced, and skewness may have a stronger impact. We also observe that the relative standard deviations are quite high, although there is a compensating effect taking place, that is, the instance that works more for a given task, works less later, showing that the labeled stream presents good performance as both load distribution and balancing mechanisms.

4.2. Association Analysis

Association analysis determines association rules, which show attribute instances (usually called items) that occur frequently together and present a causality relation between them. We divide the problem of determining association rules into two phases: determining the frequent itemsets and building the rules from them. Since the first phase is much more computationally intensive, we parallelize just the first phase. In this case, the input is a set of transactions, where each transaction contains the objects that occur simultaneously, and the output is the set of items that occurs more than a frequency threshold known as support.

Most of the association rule algorithms are based on a simple and powerful principle: for an itemset of k items to be frequent, all of its k subsets of size $k-1$ must also be frequent. Based on this principle we may easily build the itemset dependency graph, which explicits the dependencies among tasks.

Each task is divided into three partitioned reductions: counter, verifier, and candidate generator. The counter reduction just counts the number of occurrencies of a given itemset, which are forwarded to the verifier filter. The verifier filter receives the partial counts and adds them, verifying whether the itemset is frequent or not considering the whole database. Whenever the verifier finds a frequent itemset, it informs the candidate generator. The candidate generator keeps track about the itemsets found frequent so that it is able to check whether an itemset may be counted and verified, according to the task graph.

In these experiments we used different synthetic databases with size ranging from 560MB to 2.240GB, generated using the procedure described in [9]. These databases mimic the transactions in a retailing environment. All the experiments were performed with a minimum support value of 0.1%. Sensitivity analysis was conducted on data distribution, data size, and degree of parallelism.

To better understand the effects of data distribution, we distributed the transactions among the partitions in two different ways:

Random Transaction Distribution: Transactions are randomly distributed among equal-sized partitions. This strategy tends to reduce data skewness, since all partitions have an equal probability to take a given transaction.

Original Transaction Distribution: The database is simply splited into partitions, preserving its original data skewness.

We evaluate the parallel performance of the data mining application by means of two metrics: speedup and scaleup. As can be seem in Figure 9, better speedup numbers are achieved with the original transaction distribution. The reason is that the baseline time (i.e. with 8 counter filters) is much larger when the database has its original (skewed) transaction distribution. However, as we increase the number of counter filters, the execution times for different data distributions tend to approach (since the partitions get smaller, and parallel opportunities are reduced). For the scaleup experiment we increase (in the same proportion) the database size and the number of counter filters. As we can see, our parallel data mining application shows to scale very well, even for skewed databases.

In order to better understand the dynamics of the applications, we defined some metrics that may be used to understand. We will focus on the data mining algorithm, but the same applies to the vision application.

We may divide the determination of the support of an itemset into four phases:

Activation: The various notifications necessary for an itemset become a candidate may not arrive at the same time, and the verification filter has to wait until the conditions for an itemset be considered candidate are satisfied.

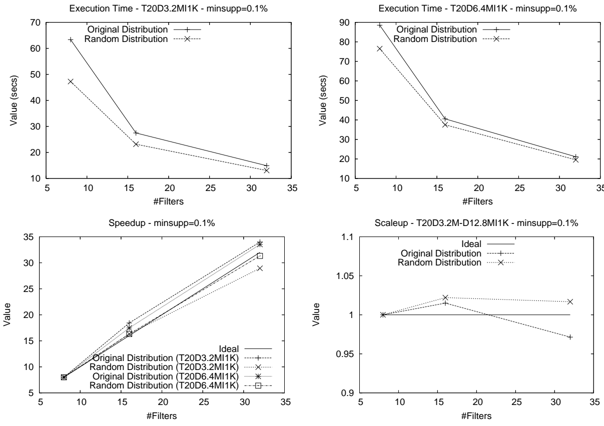

Figure 9. Parallel Performance of the Apriori.

Contention: After the itemset is considered a good candidate, it may wait in the processing queue of the counter filter.

Counting: The counter filters may not start simultaneously, and the counting phase is characterized by counter filters calculating the support of a candidate itemset in each partition.

Checking: The support counters of each counter filter may not arrive at the same time in the support checker filter, and the checking phase is the time period during which the notifications arrive.

Next we are going to analyze the duration of these phases in both speedup and scaleup experiments. The analysis of the speedup experiments explains the efficiency achieved, while the analysis of scaleup experiments shows the scalability.

In Table 5 we show the duration of the phases we just described for configurations employing 8, 16, and 32 processors. The rightmost column also shows the average processing cost for counting an itemset, where we can see that this cost reduces as the number of processors increase, as expected. The same may be observed for all phases, except for the Activation phase, whose duration seems to reach a limit around 1 second. The problem in this case is that the number of processors involved is high and the asynchronous nature of the algorithm makes the reduction of the Activation time very difficult.

Verifying the timings for the scaleup experiments in Table 6, we verify the scalability of our algorithm. We can see that an increase in the number of processors and in the size of the database does not affect significantly the measurements, that is, the algorithm implementation does not saturate system resources (mainly communication) when scaled.

Proc	Activation	Contention	Counting	Checking	Processing
8	2.741046	5.564751	9.412093	8.469050	0.001645
16	1.264842	2.058052	4.893773	4.691232	0.000759
32	1.229330	0.273229	1.129718	1.986129	0.000369

Table 5. Speedup Experiments: Profiling of Itemset Processing

Proc	Activation	Contention	Counting	Checking	Processing
8	2.741046	5.564751	9.412093	8.469050	0.001645
16	2.628118	5.538353	9.349371	8.403360	0.001596
32	2.439369	5.021002	10.311501	8.906631	0.001594

Table 6. Scaleup Experiments: Profiling of Itemset Processing

5. Clustering

Cluster analysis partitions and determines groups of objects that are similar regarding a user-given similarity criteria. In this section we discuss the parallelization of a popular clustering algorithm, k-means.

The algorithm is based on the concept of centroids, which represent the objects that compose the cluster. Each iteration of the algorithm assigns each object to the closest centroid, updating its value properly. The algorithm ends when no object changes cluster or a maximum number of iteractions is reached.

Since there is a single task that determines clusters, there is no task graph. We express the algorithm using two reductions: assigner and centroid calculator. The assigner holds the objects that must be clustered based on the centroids and determines which centroid is the closest to each object. The list of objects assigned to each centroid is then sent to the centroid calculator that recalculates the centroids.

Our clustering evaluation is based on two synthetic datasets, containing 400,000 and 800,000 points to be clustered. Each point has 50 dimensions. We performed experiments evaluating both the scaleup and the speedup. The scaleup is linear and close to 1, that is, the application scales perfectly. The execution times and respective speedups are shown in Figure 10, where we can see the time per iteraction of the algorithm for the two datasets and the speedup when varying the number of processors. In both cases, the speedup is almost linear for when clustering the 400,000-point dataset and is super-linear for the 800,000-point dataset. This super-linear behavior comes from the reduction of memory requirements as we increase the number of processors.

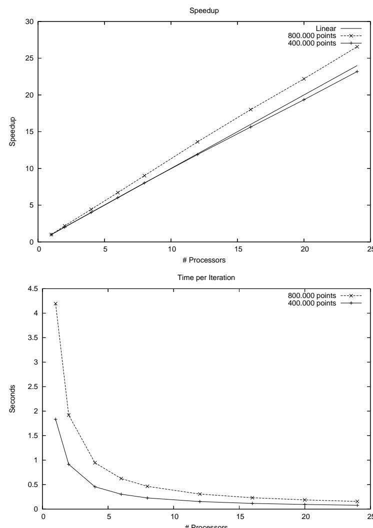

Figure 10. Parallel Performance of the k-Means Algorithm.

6. Conclusions and Future Work

In this paper we have described a run-time support framework who was developed to support the efficient implementation of a significant class of applications on heterogeneous distributed environments. We have also shown our parallelization strategy, which is the approach used to perform the decompostion on the applications. We believe that the approach can be applied for a large class of applications.

Our experimental results have shown that two applications designed using our strategy and our run-time support scale almost linearly to large number of nodes. Our experiments, however, where only limited to the number of compute nodes we had available for experimentation. In fact, we see no reason why the applications should keep the same behavior for much larger configurations.

Our run-time support system is a significant evolution on top of Datacutter. In particular, we have implemented the labelled stream infrasture and the termination detection algorithm. We still do not have an actual implementation of the stable storage for the distributed state.

For futyre work, we are in the process of incorporating fault tolerance on Anthill. A robust implementation of the distributed state is a requirement for such system.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Satlz. Active disks: Programming model, algorithms and evaluation. In *Internation Conference on Architectural Support for programming Languages and Operating Systems (ASPLOS VIII)*, pages 81–91. ACM Press, Oct 1998.

[2] M. Beynon, C. Chang, U. atalyrek, T. Kur, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz. Processing large-scale multi-dimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859, 2002.

[3] Michael Beynon, Renato Ferreira, Tahsin M. Kurc, Alan Sussman, and Joel H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, pages 119–134, 2000.

[4] Michael Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Design of a framework for data-intensive wide-area applications. In *Heterogeneous Computing Workshop (HCW)*, pages 116–130. IEEE Computer Society Press, May 2000.

[5] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[6] Philip Mucci. The performance API PAPI. White Paper of the University of Tennessee, March 2001.

[7] Clark F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21(8):1313–1325, 1995.

[8] Matthew Spencer, Renato Ferreira, Michael Beynon, Tahsin Kurc, Umit Catalyurek, Alan Sussman, and Joel Saltz. Executing multiple pipelined data analysis operations in the grid. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.

[9] A. Veloso, W. Meira Jr., R. Ferreira, D. Guedes, and S. Parthasarathy. Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. In *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, 2004.

[10] M. Zaki, C. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, page 198, Washington, DC, USA, 1999. IEEE Computer Society.

[11] Mohammed Javeed Zaki and Ching-Tien Ho. Large-scale parallel data mining. Springer-Verlag, 2000.