DIGIDT: Distributed Classifier Construction in the Grid Data Mining Framework GridMiner-Core

Juergen Hofer
Distributed and Parallel Systems Group
University of Innsbruck
Technikerstrasse 13, 6020 Innsbruck, Austria
juergen.hofer@uibk.ac.at

Peter Brezany
Institute for Software Science
University of Vienna
Nordbergstrasse 15/C/3, 1090 Wien, Austria
brezany@par.univie.ac.at

Abstract

Grid Data Mining denotes efforts to utilize data mining and knowledge discovery techniques leveraging the large-scale computational and storage power offered by Grid infrastructures. Data preprocessing and data mining algorithms are known to be both compute and data intensive and therefore appear to be ideal pilot applications to test whether Grid toolkits hold what they promise. This paper describes results of our research efforts carried out to design and prototype a runtime environment and framework for Grid Data Mining, called GridMiner-Core. For the prototype implementation the Globus Toolkit 3 was selected as a service-oriented Grid middleware. Our prototype allows the integration and execution of data preprocessing and data mining tools, applications and algorithms in a Grid-transparent manner, i.e. the algorithm contributor can focus on knowledge discovery problems without the necessity to handle Grid specifics. Scalability of the prototype has been examined and the results are presented. A distributed classification algorithm, denoted as DIGIDT, has been designed and implemented as pilot application to demonstrate functionality of GridMiner-Core. DIGIDT utilizes well-known concepts to parallelize classification algorithms. It is designed for distributed loosely coupled systems aiming to allow the induction of global decision tree classifiers from datasets that are distributed over several sites. Scalability behavior of our DIGIDT prototype is discussed to conclude the work presented here.

1 Introduction

The GridMiner Project [1, 8] aims at the combination of state-of-the-art Grid, data mining and On-Line Analytical Processing (OLAP) technologies. Data mining and OLAP if applied in conjunction can provide highly efficient and powerful data analysis and knowledge discovery solutions. Recent advances in Grid middleware research and development brought us a bit nearer to the far-end goal of Grids that promise to provide hardly limited computational power and storage capacity. A novel system called GridMiner has been designed and prototyped that allows to use knowledge discovery techniques on top of Grid infrastructures. Efforts are undertaken to add distributed datasource mediation mechanism [3], workflow orchestration [10], OLAP technologies [6] and additional data mining pilot applications embracing classification, association rules, clustering etc.

This article reports on our efforts undertaken in two areas. First we have designed and implemented a prototype of a framework that supports data mining developers with porting their existing applications to or developing new techniques for grid environments, called GridMiner-Core. It has to be seen as an additional layer above generic grid middleware that provides functionality common to a wide range of data mining and data preprocessing applications. New data mining techniques can easily plugged into the system to extend its capabilities. The second achievement reported here is the prototype implementation of a pilot application for GridMiner-Core, a distributed decision tree classifier algorithm, that takes an approach to data placement that is well-suited for loosely-coupled distributed computing scenarios. This allows interesting scenarios and real-world applications that have high restrictions on data privacy. The prize of complete attribute-wise data privacy is then analyzed with a set of performance experiments.

The rest of this article is structured as follows. In Section 2 the GridMiner-Core framework is described with its constituents, its design, prototype implementation and the results of a set of performance experiments. We then describe in Section 3 DIGIDT, the first pilot application of GridMiner-Core, its data structures and algorithm along with a set of performance experiments, we undertook to analyze the performance trade-off.

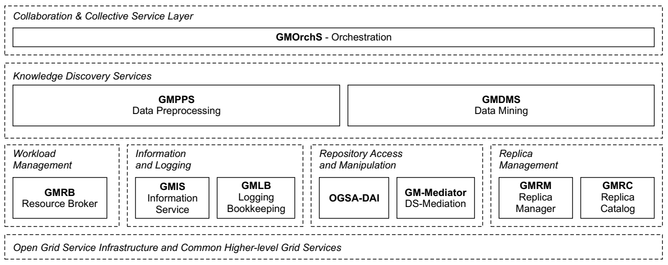

Figure 1. GridMiner Components.

2 GridMiner-Core

GridMiner-Core is a framework that supports development and runtime execution of data mining and data preprocessing applications running in Grid environments. It aims at data mining application developers that want an easy-to-use framework that supports them in either porting their existing applications or developing new applications that run in grid environments. The following paragraphs describe the design and implementation of GridMiner-Core followed by a report of the results of some experiments.

2.1 Architecture

In our prior work a set of use case scenarios for Grid data mining has been described [9] and it has been concluded that service-oriented grid architectures, such as OGSA [7] address the identified needs best. Globus 3 is the first generation of grid middleware that both is service-oriented and follows the OGSA. Although OGSI, that was the concrete realization of OGSA with Globus 3, is soon to be superseded by WSRF [4], first to be implemented by Globus 4, these two mechanisms differ foremost in implementation details and terminology. In fact the architecture described in this section is exclusively affected by a change in terminology.

Figure 1 provides an overview on the GridMiner constituents that form the GridMiner framework. Built on top of the OGSI [5] or other generic service-oriented middleware like WSRF, the common functionality that is independent of the concrete knowledge discovery or data mining technique, is realized. The *GridMiner Information Service (GMIS)* collects and aggregates service data from all available Grid services and provides a querying interface for resource discovery and monitoring. The *GridMiner Logging and Bookkeeping Service (GMLB)* collects and persists information about scheduling decisions, resource reservations and allocations, jobs and logging and error messages. The workload management components of GridMiner-Core provide distributed scheduling and Grid resource management. The main objective is to allocate and assign resources (i.e. Grid services) to Grid applications effectively and efficiently. The *GridMiner Resource Broker (GMRB)* receives a resource specification and returns the GSHs of the Grid service that matches those requirements. The repository access and manipulation components abstract from concrete data repositories by providing a single API for accessing and manipulating various kinds of data sources to higher-level knowledge discovery services. This functionality is delegated to the GridMiner Mediator [3] and external tools such as OGSA-DAI [12]. The knowledge discovery service layer contains the core data preprocessing and data mining functionality. Due to the large variety and heterogeneity of techniques and approaches the services at this layer have to be very versatile. They provide a set of common functionality that are likely to be needed by the majority of techniques. Developers that want to add, for example, new data mining algorithms only have to derive it from the base data mining service to inherit the common functionality. Then they implement the technique or simply integrate an already existing technique into the framework. At the very top of the framework architecture the GridMiner Orchestration Service (GMOrchS) allows to compose workflows from activities (i.e. it groups several service invocations to a workflow), handles failures and interacts transparently with optional components e.g. the resource broker or the replica management components. It can be seen as a facade to other GridMiner and third-party Grid services that it unifies under a single service. Data replication, the usage of identical, redundant copies of the same logical file can reduce access latency and increase overall performance. There exists a set of third-party tools for replica management where this task is delegated to. Currently all the components described above, except the replication tools, are part of the current research in the context of the GridMiner project.

2.2 Design Characteristics

Figure 2 outlines a default collaboration scenario involving the workload management and the information service constituents. The client retrieves a factory best-fitting to his requirements formulated in terms of GridMiner-RSL. The Client application then contacts this factory to create an instance that itself interacts with replica management components and uses the GridMiner mediator or the OGSA-DAI toolkit for dataset repository access.

Service Metadata. Service metadata are not kept in a separate component but directly attached to the services or their factories. Within OGSI factories, they are utilized by service requesters to create service instances. Consequently the factories in the GridMiner Framework offer *instance independent information* required to distinct and select services of the same type. The static and dynamic informa-

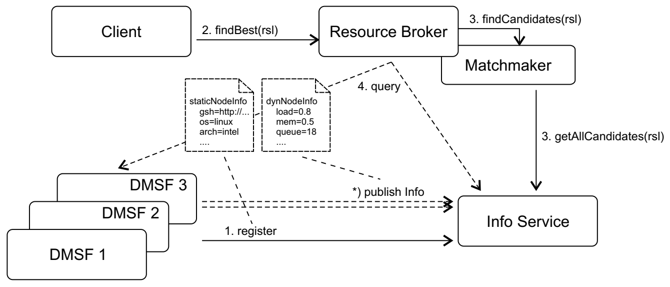

Figure 2. Service Registration and Information Publishing in GridMiner.

tion is published on registration and then regularly sent to the Info Service (see Figure 2). The origin of all *instance-dependent metadata* is the service instance itself. Once a service instance has been created, the instance publishes this information using the service data mechanisms. These include status, progress, log-messages, errors, results - all related to the action that is actually performed by the instance.

Job Submission and Control. For large datasets data preprocessing and data mining operations are likely to be long-running. The asynchronous interaction schema of knowledge discovery services in GridMiner-Core has been tailored to operations with medium- and long-time execution times. For executing a job a service instance is created, the job description is submitted and the instance executes the job. Each instance executes only one operation at a time, i.e. there is no instance-internal concurrency. In GridMiner-Core each knowledge discovery service has a *submitJob* operation with an activity parameter. This parameter is used to submit activity descriptions, which are XML documents that contain all the job-specific details and parameters. The structure and content of these activities is restricted using a known XML Schema that is published via the SDE *activitySchema* in the factory. Clients can examine these activity schemata to discover which activities are supported and use it to create the activity document. This approach has been chosen to fulfill the objective of extensibility on the one hand and as mean to deal with the manifoldness of input parameters of different data preprocessing and data mining operations on the other hand. The set of supported activities in the *activitySchema* SDE might be changed at runtime to add new techniques that were included into the system.

Instance-dependent information that is related to the job execution are published using instance service data elements. Among these are the state and progress of the job and log, warning and error messages appearing during execution. The currently executed activity is also published to allow third parties to see what active instances are running. The states of instances are defined as *ready* until an activity has been submitted and *running* that might either result in the *error* state which means the operation cannot proceed or *finished* to indicate that the operation finished successfully. In the latter case the job result can be fetched using the *getOutput* operation that replies directly with the result or transports the result to the destination using a suitable method if specified in the activity.

GridMiner Resource Specification Language. The main objective of the GridMiner Resource Specification Language (GM-RSL) is to provide a common language to describe resources. Instances of this language are created by end-users perhaps with assistance of client applications to express their requirements and constraints on the resources (i.e. services) they want to use. The first prototype implementation employs a GM-RSL language that utilizes hard-wired xml-elements for the attributes of the service data elements. This decision was made to allow rapid prototyping of GridMiner-Core. The integration of a powerful, generic language is indispensable for a production-ready version of GridMiner-Core, however not essential to early prototypes for examining the concepts in the context of this work. The next step is one of the major open issues that is to survey existing solutions, to select the best-fitting and to integrate this into GridMiner-Core.

GridMiner Data Mining Service. The GridMiner Data Mining Service (GMDMS) is the central component in GridMiner-Core. It provides the core data mining functionality that is in fact delegated to external toolkits and in this way plugged homogeneously into the system. Due to the large variety and heterogeneity of techniques and approaches for data mining the GMDMS is only a container that delegates the execution to an implementor. It is a generic service that is easy to extend.

Input to GMDMS. The DataMiningPortType operation *submitJob* accepts one generic parameter, which was already discussed in the paragraph on job submission and control. On startup the data mining service factory reads a configuration file, loads the corresponding data mining technique implementations and publishes this XML-Schema via the SDE *activitySchema*. The example depicted in Figure 3 contains a simple data mining activity named *ClassificationActivity* that contains a complex XML type dataSource referencing the dataset, the minimum confidence and the index of the example label as input parameters. The DMS instance validates the received activity against the XML-Schema. If the activity is both well-formed and valid against the configuration file the DMS instance passes the activity to an internal component that

```
<ns2:ClassificationActivity
        xmlns="http://dms.types.gridminer.org"
        xmlns:ds="http://datasource.types.
                      gridminer.org"
        xmlns:ns2="http://activities.
                 types.gridminer.org"
        xmlns:xsi="http://www.w3.org/2001/
                  XMLSchema-instance"
        xsi:type="ns2:ClassificationActivity">
   <ds:dataSource >
      <ds:name>Weather</a>
      <ds:type>ARFF</a>
      <ds:file>/home/jhofer/java/weka-3.2.3/
                  data/weather.arff</ds:file>
   </ds:dataSource>
   <ns2:confidence>0.25</ns2:confidence>
   <ns2:classIndex>3</ns2:classIndex>
</ns2:ClassificationActivity>
```

Figure 3. Sample Classification-Activity

encapsulates the internal instantiation process of the algorithm implementations, denoted as *DataMiningTechniques* (DMT). Once, the DMT implementation has been created, it is started in a separate thread and executes the algorithm it realizes. The DMT is operating transparently to the outside, all interactions with the user are exclusively done using the DMS.

Output of GMDMS. The data mining techniques running within the GridMiner Framework can return the discovered structures and models in every representation form they want - even several distinct forms, depending of the input, are possible. However it is highly desirable to utilize a representation standard to reduce client implementation complexity, such as PMML [13].

2.3 Prototype Implementation

The GridMiner Framework prototype has been implemented using the Globus Toolkit 3 and is completely written in Java. SOAP over HTTP is used as default message format and transport protocol. The Framework provides an internal API used by data mining techniques for many common operations such as logging or lifecycle management. Default implementations of many interfaces can be used to bootstrap porting or implementation efforts. They provide either a default behavior that can be extended by inheritance or used following delegation principles. All services are fully OGSA 1.0 compliant Grid services that run within containers.

The Framework can be deployed either to the Jakarta Tomcat servlet container with predeployed GT3 or to the GT3 standalone container and runs within both environments. The Framework is shipped with a build system based on Jakarta Ant that allows to generate documentation, WSDL files and JAX-RPC stubs, to compile stubs and source files, package them to WAR and GAR archives and deploy these archives either to Jakarta Tomcat or the GT3 standalone container.

2.4 Scalability

A set of tests has been created to estimate the basic scalability behavior of the prototype. It involves the creation of a new DataMiningService instance using DataMiningServiceFactory. A data mining job is submitted and the instance starts the execution. Then to simulate a realistic scenario where the client itself is an application first the progress and the status SDE are queried periodically to monitor job progress. This is done until either the finished or error status are reached. Finally the instance is destroyed. The tests were executed using Sun Solaris 9 on a Sun Fire 880 with 4 Sparc CPUs at 750MHz and 8 GB main memory on a single machine without investigating network effects. All tests have been executed at least thrice and the average values have been used.

System Scalability. Figure 4 contains the results of the first experiment. The test suite was executed simulating n concurrent clients and measuring the response time at the client. The testrun curve describes the total response time for the test suite while the other curves (create, submit etc) describe the behavior of the consisting operations. For a minimum number of concurrent clients hardly any changes in the average response time can be noticed. E.g. the testrun increases for 1 client with 6.1 sec to 6.5 sec for 5 clients to 6.9 sec for 10 clients, i.e. a factor of 1.13 for 10 clients compared to the single client mode. Doubling the number of concurrent clients from 50 to 100 increases the response time by a factor of 1.4 from 9.2 sec to 12.7 sec. Above the amount of 200 clients the response times however increase more significantly, e.g. by a factor of 3.1 from 200 to 300 clients. Nevertheless regarding total response times the system scales in this test case very well up to 1000 concurrent clients looking at the 10.3 factor for 10 times more clients in the case of load increases from 100 to 1000 clients. Surprisingly the algorithm execution time (exec) and time to submit the job description (submit) decrease slightly above 750 clients that is based on thread serialization effects. While the first clients up to the upper thread limit are virtually executed concurrently the last ones find themselves in an environment with a decreasing amounts of competitors and certain operations can be executed on overall average faster. The average total response time is not influenced by this effect.

Dataset Size-Up. The results of experiment 2 can be found in Figure 5. It shows the dataset size-up behavior for a single client that is naturally dominated by the scalability of the used algorithm, in the experiment case, this is

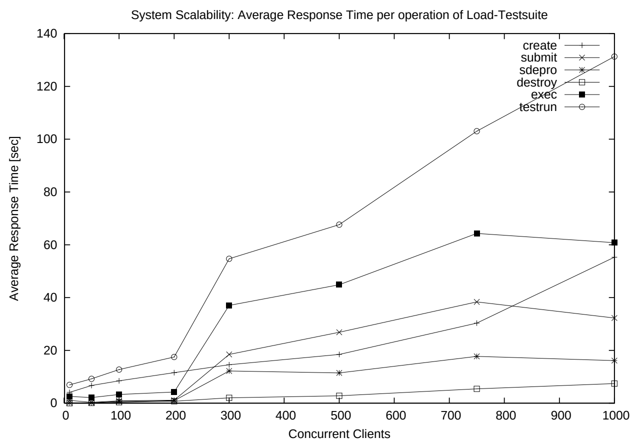

Figure 4. Response times per operation.

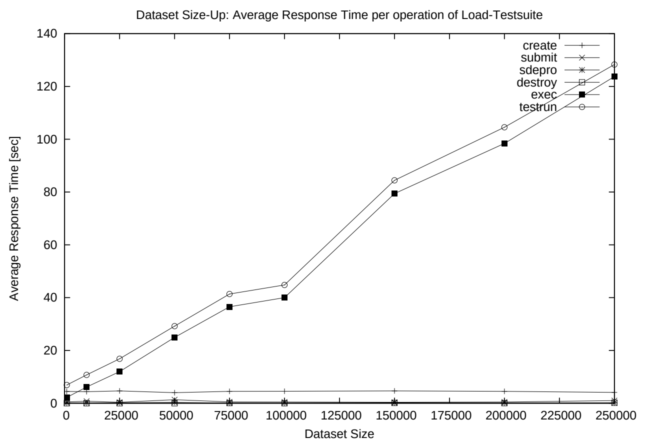

Figure 5. Dataset Size-up.

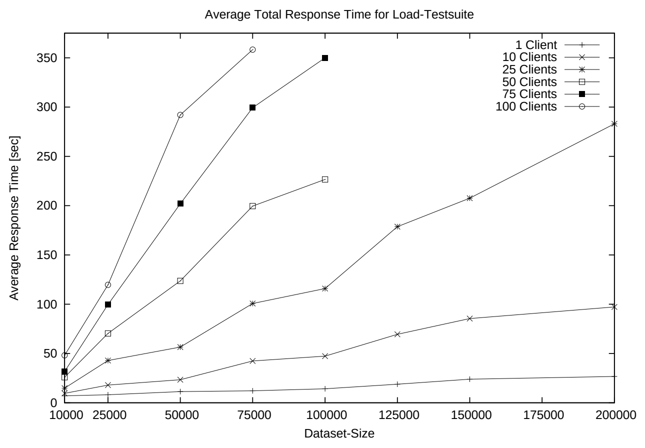

Figure 6. Average total response time.

the C4.5 implementation of the WEKA toolkit, that requires all data to stay memory resistant during the tree construction phase. The weather dataset from WEKA has been employed filled with synthetic records. As expected data size-up has no effects on the algorithm independent operations (instance creation, SDE queries, etc) in GridMiner-Core. It is vitally important to demonstrate that algorithmic load has no effects on response times of metadata querying and life-cycle management operations, so that the system keeps a high degree of responsiveness for user interaction and monitoring. The algorithm execution time (exec) and total response time (testrun) grow linearly to the dataset size by 4.2 from 10,000 to 100,000 and 7.6 from 25,000 to 250,000 records in the dataset.

Total Response Times. Finally, Figure 6 contains the results of the most comprehensive test runs. It describes the development of the average total response time for the complete execution of the testsuite per client (testrun) for dataset size-up and at different load-levels (i.e. number of concurrent clients). In this Figure we can see that the system scales along both axis well up to the system- and algorithm-dependent maximum load of 200,000 datasets in this experiment setup. Using 25 concurrent clients the response time grows from 14.8 sec for 10,000 to 283.0 sec for 200,000 records which corresponds to a factor of 19.0 for 20 times the records. If the dataset size is fixed at 100,000 records than it can be observed that the results increase from 17.9 sec for 10 clients to 119.7 for 10 times more clients that is a factor of 6.7. The C4.5 implementation of the used WEKA toolkit requires all records to stay memory-resident. Hence, for more than 50 clients the test were only executed up to 75,000 and 100,000 records because we reached the memory limits at these levels.

3 DIGIDT

Decision Trees are representations of classifiers that can be depicted as graphs [14]. The root node and the inner nodes represent tests and the outgoing edges represent the outcomes of the test. The leaf nodes are class labels indicating the group the example belongs to.

Distributed data mining denotes techniques that are performed on physically distributed repositories and/or computational resources. Considerable efforts in parallelizing classification algorithms for tightly coupled systems have been undertaken. SLIQ [11] introduced the idea of separate attribute lists written to files but still requires that a data structure, that grows in direct proportion to the number of examples in the dataset, stays memory-resident all the time. SPRINT [15], a follow-up work, extended the attribute list technique to remove this memory restriction and proposes a parallelization.

DIGIDT[1] was proposed [9] and implemented as pilot-application for GridMiner-Core that leverages concepts introduced by SPRINT but uses a different approach to dataset partitioning and workload and task assignment. DIGIDT not only partitions the workload optionally row-wise (horizontal), as described in [15] but assigns attributes to worker nodes, i.e. it performs a column-wise (vertical) partitioning. Row-wise partitioning allows very interesting applications. Projects such as the traumatic brain injury project [2] impose very strong restrictions on data security and privacy. For example the data collected at different institutions must not be combined into a universal dataset, but a global decision tree classifier should be constructed. DIGIDT supports such scenarios, where attributes or attribute groups are located at different worker nodes, each worker node works on its local data and a master node builds the global classifier. But as it will be seen subsequently this full privacy of each data sites has its prize in scale-up behavior in distributed scenarios. At each node within the decision tree, all other workers have to be barrier synchronized waiting for the split of the winning attribute. This results in a far-from-ideal scale-up as demonstrated with the experiments described later on.

3.1 The Algorithm

Phase 1 - Initialization. Figure 7 illustrates the initialization phase. The client application invokes on behalf of the user the init operation on a master data mining service that has been created using a factory. All data mining and database services register themselves within the information service and publish regularly their dynamic state information to keep the information service up-to-date. Using the activity description passed to the data mining master service it uses the resource broker to retrieve references to the resource that match the resource requirements and are selected by the employed workload distributor and scheduler. On these target nodes the master creates data mining worker services using local factories. After invocation of their *createAttributeList* operations they contact the database services, read the dataset and start with the creation of local attribute lists for each attribute they were instructed to create.

This can either be performed on a single processor or on a parallel system. If using a single processor the operation is straightforward. The dataset is read, the attribute list is written to disk and for continuous attributes a disk-based sorting algorithm is applied. The initialization phase can be parallelized at each computing site. In this case a row-wise data placement uniformly distributing the workload to the n available processors is used, hence each processor is working on $1/n$ of the total amount of records in the dataset. Each processor then creates his part of the attribute lists. As already mentioned for continuous attributes the lists have to be sorted which implies that if n processors are used a parallel sorting algorithm has to be applied. After successful creation, each data mining working service notifies the master service that is responsible for the coordination.

[1]DIGIDT stands for Distributed Grid-enabled Induction of Decision Trees

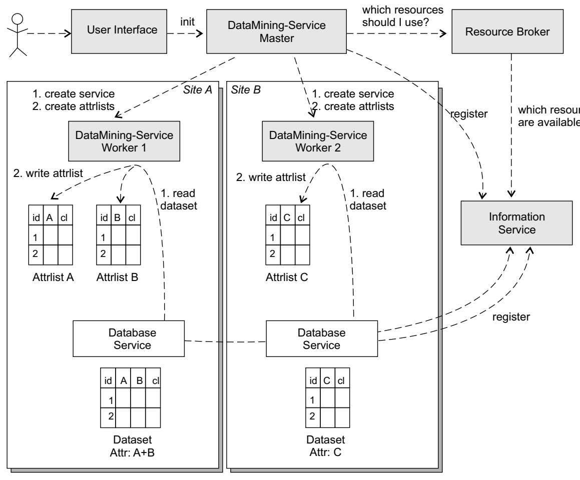

Figure 7. Phase 1 of DIGIDT.

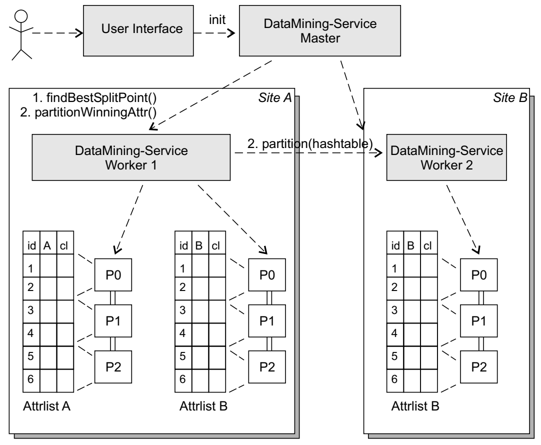
User Interface
init
DataMining-Service Master
1. findBestSplitPoint()
2. partitionWinningAttr()
Site A
Site B
DataMining-Service Worker 1
2. partition(hashtable)
DataMining-Service Worker 2
P0
P1
P2
Attrlist A
Attrlist B

Figure 8. Phase 2 of DIGIDT.

Phase 2 - Tree Construction. In the growth phase the tree is constructed by recursively partitioning the data until each partition is either pure or sufficiently small. In each iteration all possible splits are evaluated for all attributes to find the global best splitting point. The current set of examples is then further partitioned into two subsets applying the splitting test. Figure 8 depicts phase 2 of DIGIDT. Each participating site, here A and B, perform the *findBestSplitPoint*

records	atl1	atl2	atl3	atl4	hash	atlg1	atlg2	atlg3	atlg4
1e4	0.16	0.13	0.13	0.15	0.06	0.42	0.32	0.32	0.40
5e5	0.81	0.62	0.62	0.74	0.34	2.20	1.70	1.70	2.10
1e5	1.60	1.20	1.20	1.50	0.70	4.30	3.40	3.40	4.10
5e5	8.20	6.40	6.40	7.60	3.80	22.00	18.00	18.00	21.00
1e6	17.00	13.00	13.00	15.00	7.60	45.00	36.00	36.00	43.00
5e6	87.00	69.00	69.00	81.00	43.00	235.00	191.00	191.00	225.00
1e7	175.00	139.00	139.00	163.00	172.00	944.00	769.00	769.00	906.00

Table 1. Used disk space for DIGIDT data structures.

operation that scans for each attribute the attribute lists to search for its local best splitting point. The applied mechanism is identical with the algorithm proposed by SPRINT and involves the updating of the count matrix during the scan over the attribute list. The data mining master service collects all these results and notifies the worker service that has the global best result by invocation of *partitionWinningAttr*. The site holding the winning attribute list starts with the partitioning by simply applying the split test on each of the containing examples and moving them to the new attribute lists. Unfortunately the other attribute lists cannot be split without further information. For this again likewise SPRINT a probe structure (hash table) is created with the ids of each example that indicates to which child the example belongs. This hash table is created and sent to all other sites in chunks. Each non-winning attribute list can then be partitioned using the hash table chunk by chunk. A proposed optimization is that the hash table is built out of the record ids (they unambigously identify each example in the training set) of the child with fewer records. On loosely-coupled systems like the one described here this optimization is vitally important because the communication cost of broadcasting the hash table for dataset partitioning is a significant scale-up determinant. As described before and illustrated in Figure 8, the attribute list processing including the evaluation of the best splitting point and the partitioning can be done in parallel similar to the initialization phase. First the examples in the attribute lists are evenly distributed over the n available processors at each site for each attribute. Again continuous attributes have to be presorted using a parallel sorting algorithm. Then each processor starts with the splitting point evaluation for his private subset of the attribute lists speeding-up the processing to $1/n$th of the time. For categorical attributes after each processor finishes the n count matrices are collected and the global count matrix is calculated by a coordinating processor that then determines the best split point for the current attribute. In case of continuous attributes the initialization of the count matrices is more complex because all processor histograms have to be initialized with the sum of all values of all processor histograms of lower rank. Consequently these statistics have to be exchanged between all processors before the attribute list scan starts.

3.2 Used Data Structures

DIGIDT requires the creation and usage of the following data structures. From the underlying dataset repository each site creates an *attribute list* containing $<$*record id, attribute value, class label*$>$ as suggested by SPRINT per attribute. Therefore each site i from n builds m_i attribute lists where m_i equals the number of local attributes in relation R_i. In case of Figure 7 this means that site *A* creates two attribute lists for attributes *a* and *b* whereas site *B* creates a single attribute list for its local attribute *c*. In contrast to SPRINT, the attributes lists however do not reside at the same site but may be distributed over several grid nodes depending on the dataset layout. For continuous attributes the lists have to be sorted.

In addition in DIGIDT, to each decision tree node, a *count matrix* is attached that is required for finding the best splitting points. It has two rows C_{above} and C_{below} and a column per class value for continuous attributes and in case of categorical attributes the count matrix consists of a row for each attribute value and a column per class value.

In addition to the algorithmic aspect, Figure 8 shows the workload distribution for each attribute list to p processors per computing site. For this the SPRINT parallelization technique is employed. The initial attribute list creation is performed by each processor in parallel. The resulting attribute list has to be sorted for continuous attributes. Table 3.1 puts the size of the initial attribute lists, i.e. the input to DIGIDT (in columns *atl*n) in relation to the total size of the files created during the execution. Column *hash* contains the sizes of the hash files, that are of high relevance because they need to be transferred from the worker node that split the winning node to each of the worker nodes that can start to split their loosing attributes after receiving this probe structure. In case of $1e7$ records, the sum of all hash files produced during the execution equals approximately the file size of a single numerical attribute list with 172 MB. The biggest hash files in this experiment were several MBs in size so the transfer time using a low-bandwidth connection will considerably influence the overall performance. Columns *atlg*n show the total size of all attribute lists created per worker node, i.e. per attribute. They total to several times the input file size but since they are written

records	1 machine		1 master 1 slave		1 master 2 slaves		1 master 4 slaves	
	absolute	normalized	absolute	normalized	absolute	normalized	absolute	normalized
1e5	114.93	0.98	116.91	1.00	87.81	0.75	75.44	0.65
5e5	465.48	0.84	554.20	1.00	394.97	0.71	335.67	0.61
1e6	909.59	0.82	1106.25	1.00	773.59	0.70	665.26	0.60
5e6	4410.98	0.92	4820.19	1.00	3596.07	0.75	3203.75	0.66
1e7	9013.70	0.87	10329.29	1.00	7472.79	0.72	7076.76	0.69

Table 2. Absolute and relative response times for selected experiments.

and read only locally and not exchanged with other worker nodes this is less performance relevant then the hash file size. The records have been generated based on a simple function that results in a small decision tree. A possible optimization of the current implementation is to reuse the attribute lists, marking the records to which node they belong. The current prototype implementation creates for each test a new set of attribute lists.

3.3 Prototype Implementation

A prototype of the DIGIDT algorithm has been implemented based on GridMiner-Core. The implementation is in Java and uses text files for its attribute lists. The worker nodes are realized as Grid Services in Globus 3.2 and are invoked via SOAP over HTTP. Alternatively, both best split point evaluation and attribute list splitting can be run in Java threads also in parallel within a JVM. Using this mechanism a single worker node can be used to evaluate and split several attribute lists in parallel. The master node runs either as standalone application or itself as GridMiner-Core conform data mining service.

3.4 Performance Experiments

For our performance analysis we have conducted a set of experiments on a synthetic dataset consisting of two continuous and two categorical attributes and a binary class value. The experiment environment was a workstation cluster of Sun Blade 150 Workstations running Solaris 9 interconnected via a 100Mbit Ethernet. Training set size was increased up to 10 million records. The size of the training sets is only limited by disk space. In our case, though, it turned out not to be a necessity to further increase the trainingset size because the the typical scalability behavior and limitations of DIGIDT can equally be observed in the range presented here. We repeated all experiments at least three times and the best result is being reported. Selected absolute and normalized response times are reported in Table 3.2.

Scaleup. Figure 9 and Figure 10 show the first scale-up experiment. The plotted curves are functions of the number of worker nodes (processors) (remember that there is a master node that acts as coordinator and n worker nodes performing the splitting point evaluations and splits) and shows the response time (response time is the total real time measured from the start of the program until its termination) in seconds. In Figure 9 the total dataset size was held constant at levels of $1e5$, $5e5$, $1e6$ and the four attributes were assigned to one, two and four worker nodes. The graph shows a reduction of the response time from 1106 sec with a single worker node, to 773 sec with two worker nodes, to 665 sec with four worker nodes at the 1 million record level. Although the splitting point evaluation is performed fully parallel allowing perfect scaleup, the splitting phase is far less ideal. In the splitting phase, first the winning worker node has to split the list and create the hash table, that is then sent to all other worker nodes. This process is in the current implementation sequential to the next step, where the other worker nodes split their 'loosing' attribute lists. Hence the prize of the advantages of privacy etc, described earlier in this paper are the sequential phases at each decision tree node and the small scaleup. Figure 10 plots the data of the same experiment at levels of $5e6$ and $1e7$ records showing similar behavior. The results show a reduction of the response time from 10329 sec with a single worker, to 7472 sec with two workers, to 7076 sec with four worker nodes at the 10 million record level.

Sizeup. In the sizeup experiment the development of the response time with increasing dataset sizes has been analyzed for one, two and four worker nodes. Figure 11 plots the results. As expected the single-work curve raises steeper than in the two worker and four worker case. Again, although the dataset partitions remain private to each node, a total speedup using DIGIDT can be noticed that however decreases the more worker nodes in parallel are used. In case of 1 master and 1 worker node the curve rises from 116 sec at $1e5$ to 1106 sec at $1e6$ to 4820 sec at $5e6$ records. For 1 master and 4 worker nodes we noted 75 sec at $1e5$, 665 sec at $1e6$ and 3503 sec at $5e6$ records.

Attribute List Scan and Split time. During a random experiment run, we measured the time for finding the best split point per attribute and to split the attribute lists. The resulting plots for the scan and split operations are depicted in Figure 12 and Figure 13. The data are shown per attribute

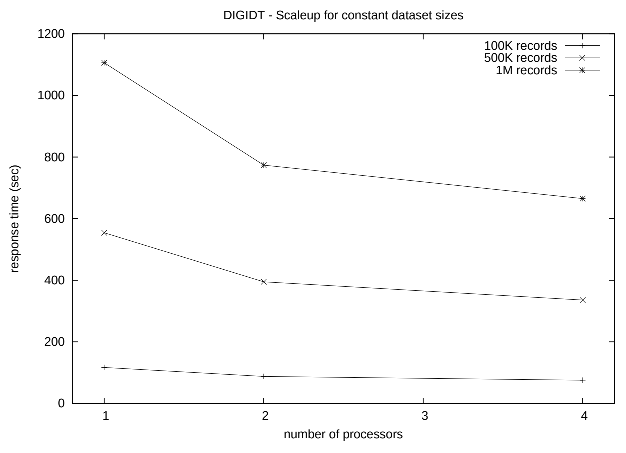

Figure 9. Scaleup for $1e5$ to $1e6$ examples.

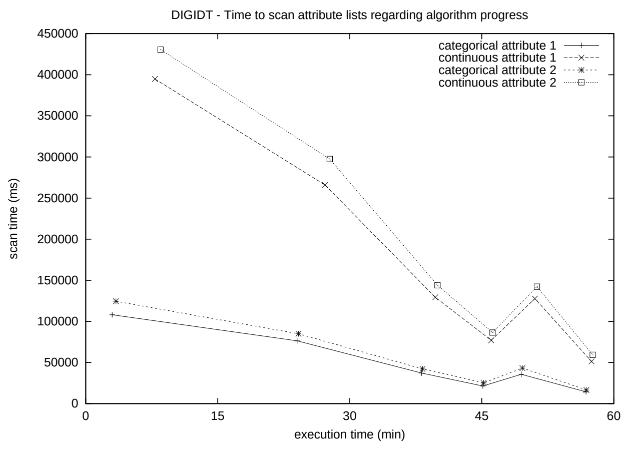

Figure 12. Split point evaluation.

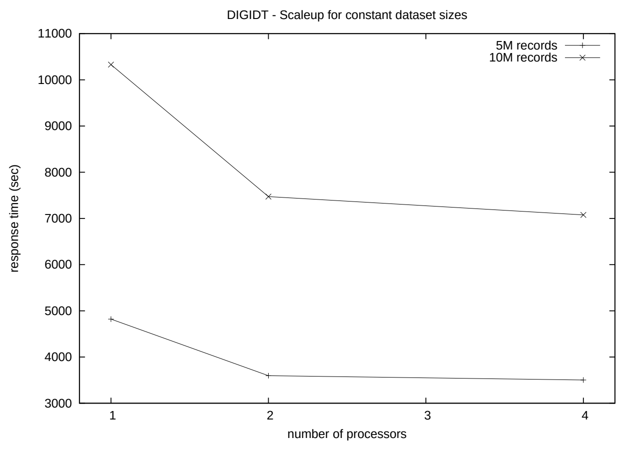

Figure 10. Scaleup for $5e6$ and $1e7$ examples.

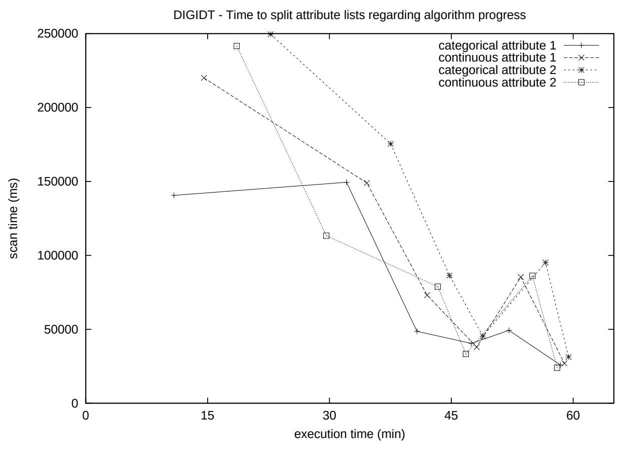

Figure 13. Attribute list split.

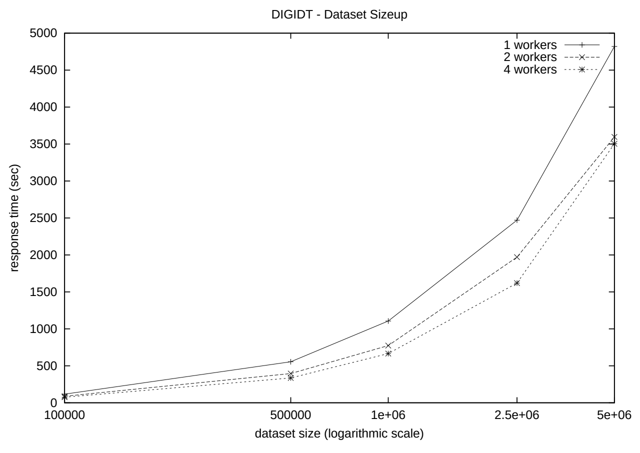

Figure 11. Dataset Sizeup

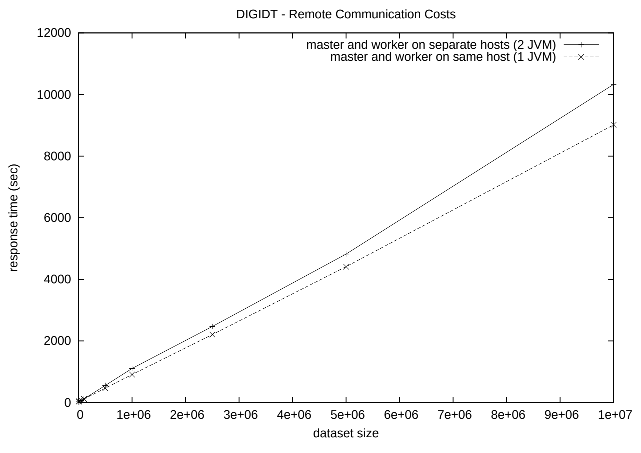

Figure 14. Remote Communication Cost.

that allows to make the important distinction between categorical and continuous attributes. With algorithm progress the times to scan and split the attribute lists decreases naturally in relation to the attribute list sizes. Figure 12 shows a huge gap between the scan times of categorical and continuous attributes. This is caused by the current implementation that uses plain text files for storing the attribute lists. While categorical attribute lists require only comparison of strings, for continuous attributes the continuous values have first to be parsed and converted from its string representation into its internal integer and double representations. This expensive conversion is responsible for the gap between the attribute list types in the plot. A possible optimization of the implementation is to use a binary format for the attribute lists that allows more efficient parsing without the need of extensive datatype conversions.

Study of Cost of Distribution. Figure 14 depicts the results of our analysis to study the cost of distributing the algorithm. The response time in seconds has been measured for the scenario where DIGIDT was executed within a single JVM on a single host to the scenario with a separated master and worker node. The difference for $1e5$ records is 116 sec to 114 sec, which means cost of 2 seconds for the SOAP/HTTP remote method invocation. These costs increase to 1013 seconds for $1e7$ examples to 10329 sec with separate hosts and 9013 sec on a single host.

4 Conclusions and Future Work

In this paper we describe the high-level architecture of GridMiner and the components it consists of. We designed and implemented a prototype of a framework for Grid data mining based on Globus 3 toolkit, called GridMiner-Core. GridMiner-Core aims at supporting developers with porting their existing data mining applications to or to assist the development of new techniques for Grid environments. A set of performance tests was carried out to analyze behavior under stress. Moreover we implemented DIGIDT as pilot application for GridMiner-Core, a distributed decision tree classifier algorithm. The observed scalability and its limitations are described.

Some optimizations both for the framework itself and DIGIDT remain open. A binary attribute list format and sending partial hash files will certainly improve performance considerably. Moreover the attribute list internal parallelism has not been implemented, but is important for a final DIGIDT product. The prototype presented must be seen in the context of the GridMiner project and is an important intermediate step towards the far-end vision denoted as Grid Data Mining. A current working prototype of our combined efforts will be demonstrated at Supercomputing 2004.

Acknowledgements This work was partly carried out by funding from 'Aurora' and 'Modern Data Analysis on Computational Grids' projects. We are thankful to Stefan Podlipnig for careful proof-reading and valuable comments during the finalisation of this paper.

References

[1] P. Brezany, J. Hofer, A. M. Tjoa, A. Woehrer, and J. Brezanyova. GridMiner: An Infrastructure for Data Mining on Computational Grids. In *Proceedings of the APAC Conference and Exhibition on Advanced Computing, Grid Applications and eResearch*, Queensland Australia, October 2003.

[2] P. Brezany, M. Rusnak, and P. Tomsich. Knowledge grid support for treatment of traumatic brain injury victims. Submitted to the Conference on High-Performance Distributed Computing, Edinburgh, Scotland, July 2002.

[3] P. Brezany, A. M. Tjoa, H. Wanek, and A. Woehrer. Mediators in the Architecture of Grid Information Systems. In *Proceedings of the Conference on Parallel Processing and Applied Mathematics*, Czestochowa, September 2003.

[4] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. From open grid services infrastructure to ws-resource framework: Refactoring and evolution 1.1, May, 3 2004.

[5] S. T. et al. Open grid services infrastructure 1.0. Gwd-r, GGF- OGSI Working Group, June 27 2003.

[6] B. Fiser, U. Onan, I. Elsayed, P. Brezany, and A. M. Tjoa. On-line analytical processing on large databases managed by computational grids. In *DEXA 2004*, Zaragoza, Spain, 30 August - 3 September 2004.

[7] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Open Grid Service Infrastructure Working Group, Global Grid Forum, June 2002.

[8] GridMiner Project. *http://www.gridminer.org/*.

[9] J. Hofer and P. Brezany. Distributed decision tree induction within the grid data mining framework gridminer-core. Technical Report GridMiner TR 2004-04, University of Vienna, Vienna, Austria, March 2004.

[10] G. Kickinger, J. Hofer, A. M. Tjoa, and P. Brezany. Workflow Management in GridMiner. In *Proceedings of the 3rd Cracow Grid Workshop*, Cracow, Poland, October 2003.

[11] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Extending Database Technology*, pages 18–32, 1996.

[12] OGSA-DAI. *http://www.ogsadai.org.uk/*.

[13] PMML. *http://www.dmg.org/pmml-v2-0.html*.

[14] J. Quinlan. Induction on decision trees. *Machine Learning*, 1:81–106, 1986.

[15] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proc. 22nd Int. Conf. Very Large Databases, VLDB*, pages 544–555. Morgan Kaufmann, 3–6 1996.