A Novel Architecture for Data Mining Grid Scheduler

MEIQUN LIU[1], KUN GAO[2], ZHONG WAN[1]
[1] Culture and Communication College
Zhejiang Wanli University
No. 8, South Qian Hu Road, 315100, Ningbo, Zhejiang
P. R. China
http://www.zwu.edu.cn
[2] Computer Science and Information Technology College
Zhejiang Wanli University
No. 8, South Qian Hu Road, 315100, Ningbo, Zhejiang
P. R. China
http://www.zwu.edu.cn

Abstract: In order to improve the performance of Data Mining applications, an effective method is task parallelization. The scheduler on Grid plays an important role to management subtasks so as to achieve high performance. We introduce an additional component that we call serializer, whose purpose is to decompose the tasks into a series of independent tasks according the directed acyclic graph (DAG), and send them to the scheduler queue as soon as they become executable with respect to the DAG dependencies. The experimental result demonstrates that the architecture has good performance.

Key-Words: Scheduling Architecture, Knowledge Grid, Data Mining

1 Introduction

Generally, data mining (sometimes called data or knowledge discovery) is the process of analyzing data from different perspectives and summarizing it into useful information - information that can be used to increase revenue, cuts costs, or both. Data mining software is one of a number of analytical tools for analyzing data. It allows users to analyze data from many different dimensions or angles, categorize it, and summarize the relationships identified. Technically, data mining is the process of finding correlations or patterns among dozens of fields in large relational databases. This technology is widely used more and more.

Because of the expensive computation power and the distributed nature of data, the Knowledge Grid (K-Grid) [1] will become nature platform to implement Data Mining (DM) computation. In order to improve the performance of DM applications, an effective method is task parallelization. The scheduler on Grid [10] plays an important role to management subtasks so as to achieve high performance. In this paper, we first take an example of DM application to present a method of parallelization. The objective of this method is to decompose DM application into subtasks and then combine those subtasks to form GAG. We focus on the design of online dynamic scheduler that schedule subtasks according to the structure of the DAG.

Many efforts have already been devoted to the problem of scheduling distributed jobs in distributed environment and grid platform [2], [3] [13-18]. Many of the schedulers propose their own solution to the problem. Nevertheless, there are some characteristics of scheduling DM tasks that make the previous approaches inadequate.

First of all we lack an accurate analytical cost model for DM tasks. In the case of [18] system, the parametric, exactly known cost of each job allows the system to foresee with a high degree of accuracy which is going to be the execution time of each job. This does not hold for DM, where the execution time of an algorithm in general depend on the input parameters in a non linear way, and also on the dataset internal correlations, so that, given the same algorithm, the same set of parameters and two dataset of identical dimensions, the execution time can vary of orders of magnitude. The same can be said for other performance metrics, as memory requirement and I/O activity.

The other characteristic is that scheduling a DM task in general implies scheduling computation and data transfer [4]. Traditional schedulers typically only address the first problem, that of scheduling computations. In the case of DM, since the dataset are typically big, it is also necessary to properly take into account the time needed to transfer data and to consider when and if it is worth to move data to a

different location in order to optimize resource usage or overall completion time.

DM applications running on the K-Grid can be parallelized. Such, we can parallelize single DM application to several subtasks; several tasks may be combined to form a workflow. We call such scheduler as K-Grid Scheduler, or KGS. KGS should have the fowling feature:

1. On-line. KGS must schedule the components of DM DAGs as soon as they arrive in the system.
2. Dynamic. It must apply cost models to predict future resource status and pro-actively assign jobs to resources.
3. Adaptive. It must continuously interact with the Grid Information Service, in order to have an updated view of the system status in terms of machine and network loads.

We briefly describe here the design of KGS. A model for the resources of the K-Grid is composed by a set of hosts, onto which the DM tasks are executed, a network connecting the hosts and a centralized scheduler, KGS, where all requests arrive. The main task of KGS is to execute tasks composition. We consider that the basic components of a DM task are algorithms and datasets. They can be combined in a structured way, thus forming a DAG. In the following sections, we give a more accurate description of the mapping process, starting from the definition of models for the system architecture, the cost of DM tasks, and their execution.

Scheduling DAGs on a distributed platform is a non-trivial problem which has been faced by a number of algorithms [5]. It is crucial to take into account data dependencies among the different components of the DAGs present in the system. For this reason, we introduce in the system an additional component that we call serializer, whose purpose is to decompose the tasks into a series of independent tasks according corresponding DAG, and send them to the scheduler queue as soon as they become executable w.r.t. the DAG dependencies.

Related works: Data mining techniques are applied to achieve process improvement [19]. It can help engineers to understand the process know-how will enhance their core competiveness. Especially, the information about process improvement or product development frequently is held behind such know-how, it is called as the manufacturing intelligence. Such manufacturing intelligence will provide a positive contribution to process improvement. In order to address such issue, the authors propose an approach based on artificial neural networks (ANNs) data mining technique to mine the manufacturing intelligence. The rationality and feasibility of the proposed procedure can also be demonstrated well according to the illustrative example in this study.

Data mining techniques presented in the literature are usually used for prediction and they are tested on well known benchmark problems. System identification is a practical engineering problem and an abdicative task which is affected by several kinds of modeling assumptions and measurement errors. Therefore, system identification is supported by multiple-model reasoning strategies. In [20], the author is to study the use of data mining techniques for system identification. One goal of the author is to improve views of model-space topologies. The presence of clusters of models having the same characteristics, thereby defining model classes, is an example of useful topological information. Distance metrics add knowledge related to cluster dissimilarity. Engineers are thus better able to improve decision making for system identification.

Although some research has been dedicated to the development of Knowledge Discovery in Databases (KDD) assistance mechanisms, little effort has been directed to the deployment of tools that assist humans during the KDD task definition stage. In order to satisfy this need for a KDD task definition assistance device, [21] proposes three different approaches: a) the first one is called theoretical approach and is based on concepts from the Theory of Attribute Equivalence in Databases and from Topological Spaces; b) the second employs Artificial Neural Networks to learn mappings between heterogeneous patterns and is called experimental approach; c) the third one combines the abovementioned approaches to implement what is called hybrid approach.

In this paper, we propose an effective scheduling solution for those subtasks to minimize total response time. The rest of this paper is organised as follows: in section 2, we present how to map a DM application to DAG. In section 3, we present the architecture for a K-Grid scheduler that result in the minimal response time. In section 4, we conduct experiments to evaluate the architecture. Finally section 5 concludes this paper.

2 Decomposing DM Applications to DAG

K-Grid services can be used to construct complex problem solving environments, which exploit DM kernels as basic software components that can be applied one after the other, in a modular way. A general DM task on the K-Grid can therefore be

described as a DAG whose nodes are the DM algorithms being applied, and the links represent data dependencies among the components. In this section, we present how to map DM applications to DAG.

2.1 Modeling DM Applications

We surveyed three major classes of DM applications, namely association rule mining, classification rule mining, and pattern discovery in combinatorial databases. We note the resemblance among the computation models of these three application classes.

A task is the main computation applied on a pattern. Not only are all tasks of any one application of the same kind, but tasks of different applications are actually very similar. They all take a pattern and a subset of the database and count the number of records in the subset that match the pattern. In the classification rule mining case, counts of matched records are divided into c baskets, where c is the number of distinct classes.

The similarities among the specifications of these applications are obvious, which inspired us to study the similarities among their computation models. They usually follow a generate-and-test paradigm-generate a candidate pattern, then test whether it is any good. Furthermore, there is some interdependence among the patterns that gives rise to pruning, i.e., if a pattern occurs too rarely, then so will any superpattern. These interdependences entail a lattice of patterns, which can be used to guide the computation.

In fact, this notion of pattern lattice can apply to any DM application that follows this generate-and-test paradigm. We call this application class pattern lattice DM. In order to characterize the computation models of these applications more concretely, we define them more carefully in Section 2.2.

2.2 Defining DM Applications

In general, a DM application defines the following elements.

1. A database D.
2. Patterns and a function len(pattern p) which returns the length of p. The length of a pattern is a non-negative integer. We use {} to represent zero-length patterns in association rule mining.
3. A function goodness(pattern p) which returns a measure of p according to the specifications of the application.
4. A function good (p) which returns 1 if p is a good pattern or a good subpattern and 0 otherwise. Zero-length patterns are always good.

The result of a DM application is the set of all good patterns. If a pattern is not good, neither will any of its superpatterns be. In other words, it is necessary to consider a pattern if and only if all of its subpatterns are good.

Let us define an immediate subpattern of a pattern q to be a subpattern p of q where

len(p) = len(q)-1

Conversely, q is called an immediate superpattern of p. Except for the zero-length pattern; all the patterns in a DM problem are generated from their immediate subpatterns. In order for all the patterns to be uniquely generated, a pattern q and one of its immediate subpatterns p have to establish a childparent relationship (i.e., q is a child pattern of p and p is the parent pattern of q). Except for the zero-length pattern, each pattern must have one and only one parent pattern. For example, in sequence pattern discovery, *FRR* can be a child pattern of *FR*; in association rule mining, {2, 3, 4} can be a child pattern of {2, 3}; and in classification rule mining, (C = c1)^(B = b2)^(A = a1) can be a child pattern of (C = c1)^(B = b2).

2.3 Solving DM Applications

Having defined DM applications as above, it is easy to see that an optimal sequential program that solves a DM application does the following:

1. generates all child patterns of the zero-length pattern;
2. computes goodness(p) if all of p's immediate subpatterns are good;
3. if good(p) then generate all child patterns of p;
4. applies 2 and 3 repeatedly until there are no more patterns to be considered.

Because the zero-length pattern is always good and the only immediate subpatterns of its children is the zero-length pattern itself, the computation starts on all its children, which are all length 1 patterns. After these patterns are computed, good patterns generate their child sets. Not all of these new patterns will be computed-only those whose every immediate subpattern is good will be.

2.4 Mapping DM application to DAG

We propose to use a DAG structure called exploration DAG (E-GAG, for short) to characterize

pattern lattice DM applications. We first describe how to map a DM application to an E-GAG.

The E-GAG constructed for a DM application has as many vertices as the number of all possible patterns (including the zero-length pattern). Each vertex is labeled with a pattern and no two vertices are labeled with the same pattern. Hence there is a one-to-one relation between the set of vertices of the E-GAG and the set of all possible patterns. Therefore, we refer to a vertex and the pattern it is labeled with interchangeably.

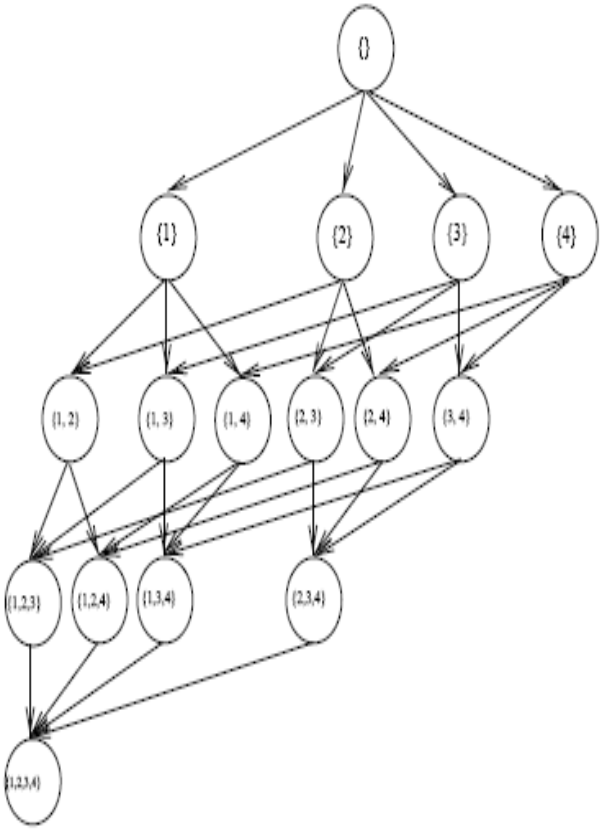

Fig.1 a complete E-GAG for an association rule mining application on the set of items {1, 2, 3, 4}.

There is an incident edge on a pattern p from each immediate subpattern of p. All patterns except the zero-length pattern have at least one incident edge on them. The zero-length pattern has an outgoing edge to each pattern of length 1. Fig.1 shows an E-GAG mapped from an association rule mining application.

3 Knowledge Grid Scheduler

3.1 Serialization Process

We consider that the basic building blocks of a DM task are algorithms and datasets. They can be combined in a structured way, thus forming a DAG. DM components correspond to a particular algorithm to be executed on a given dataset, provided a certain set of input parameters for the algorithm. We can therefore describe each DM components L with the triple:

L = (A, D, {P}).

Where,

A is the DM algorithm;

D is the input dataset;

{P} is the set of algorithm parameters.

For example if A corresponds to "Association Mining", then {P} could be the minimum confidence for a discovered rule to be meaningful. It is important to notice that A does not refer to a specific implementation. We could therefore have more different implementations for the same algorithm, so that the scheduler should take into account a multiplicity of choices among different algorithms and different implementations. The best choice could be chosen considering the current system status, the programs availability and implementation compatibility with different architectures.

Scheduling DAGs on a distributed platform is a non-trivial problem which has been faced by a number of algorithms in the past. Although it is crucial to take into account data dependencies among the different components of the DAGs present in the system, we first want to concentrate ourselves on the cost model for DM tasks and on the problem of bringing communication costs into the scheduling policy. For this reason, we introduce in the system an additional component that we call serializer (Fig.2), whose purpose is to decompose the tasks in the DAG into a series of independent tasks, and send them to the scheduler queue as soon as they become executable w.r.t. the DAG dependencies.

Such serialization process is not trivial at all and leaves many important problems opened, such as determine the best ordering among tasks in a DAG that preserve data dependencies and minimizes execution time.

Nevertheless, at this stage of the analysis, we are mainly concerned with other aspects in the system, namely the definition of an accurate cost model for single DM tasks and the inclusion of communications into the scheduling policy.

3.2 Cost Model

The following cost model assumes that each input dataset is initially stored on a single machine m_h, while the knowledge model extracted must be moved to a machine m_k. Due to decisions taken by the scheduler, datasets may be moved to other machines and thus replicated, or may be partitioned among diverse machines composing a cluster for parallel execution. Therefore, the scheduler has to take into

account that several copies (replicated or distributed) of a dataset may exist on the machines of its Grid.

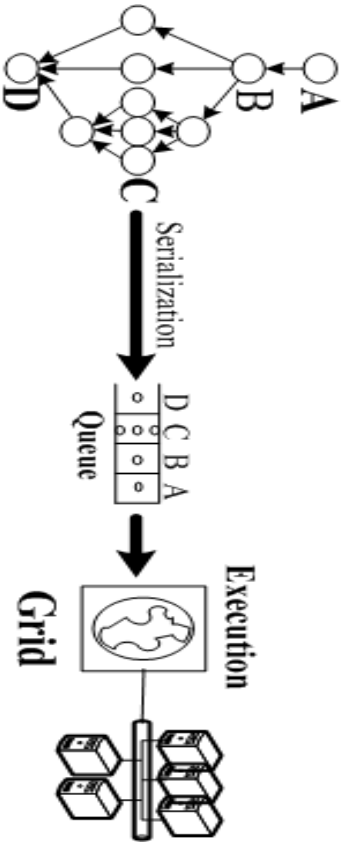

Fig. 2 Serializer

3.2.1 Sequential Execution

Suppose that the whole dataset is stored on a single machine m_h. Task t_i is executed sequentially by a code running on machine m_j, with an execution time of e_{ij}. In general we also have to consider the communications needed to move D_i from machine h to machine m_j, and the further communications to move the results $|a_i(D_i)|$ to machine m_k. The total execution time is thus:

$$E_{ij} = |D_i| / b_{hj} + e_{ij} + |\alpha_i(D_i)| / b_{jk}$$

Of course, the relative communication costs involved in dataset movements are zeroed if either h=j or j = k.

3.2.2 Parallel Execution

Task t_i is executed in parallel by a code running on a cluster c_{lJ}, with an execution time of e_{iJ}. In general, we have also to consider the communications needed to move D_i from machine m_h to cluster c_{lJ}, and to move the results $|a_i(D_i)|$ to machine m_k. The total execution time is thus:

$$E_{iJ} = \sum_{m_t^J \in cl_J} \frac{|D_i| / |cl_J|}{b_{ht}} + e_{iJ} + \sum_{m_t^J \in cl_J} \frac{|\alpha_i(D_i)| / |cl_J|}{b_{tk}}$$

Of course, the relative communication costs are zeroed if the dataset is already distributed, and is allocated on the machines of c_{lJ}.

3.2.2 Performance Metrics

E_{ij} and E_{iJ} are the expected total execution times of task t_i when no load is present in the system. When load is present on machines and networks, scheduling will delay the start and thus the completion of a task. In the following we will analyze the actual completion time of a task for the sequential case. A similar analysis could be done for the parallel case.

Let C_{ij} be the wall-clock time at which all communications and sequential computation involved in the execution of t_i complete. To define C_{ij} we need to define the starting times of communications and computation. Let s_{hj} be the start time of communication needed to move the input dataset from machine h to machine j, let s_j be the start time of the sequential execution of task t_i on machine j, and, finally, let s_{jk} be the start time of communication needed to move the knowledge result model extracted from machines j to machine k. From the above definitions:

$$C_{ij} = (s_{hj} + \frac{|D_i|}{b_{hj}}) + \delta_1 + e_{ij} + \delta_2 + \frac{|\alpha_i(D_i)|}{b_{jk}} = s_{hj} + E_{hj} + \delta_1 + \delta_2$$

Where,

$$\delta_1 = s_j - (s_{hj} + \frac{|D_i|}{b_{hj}}) \geq 0$$

And

$$\delta_2 = s_{jk} - (s_j + e_{ij}) \geq 0$$

So, if A_i is the arrival time of task t_i, and t_i is the only task in execution on the system, then the optimal completion time of the task on machine m_j is:

$$\overline{C_{ij}} = A_i + E_{ij}$$

Suppose that $m_{\bar{j}}$ is the specific machine chosen by our scheduling algorithm for executing a task t_i.

Let

$$C_i = C_{\bar{i}\bar{j}}$$

And

$$\overline{C_i} = \overline{C_{\bar{i}\bar{j}}}$$

Let T be the set of tasks to be scheduled.

The makespan for the complete scheduling is defined as $\max_{t_i \in T}(C_i)$, and measures the overall throughput of the system.

3.3 Predicting DM Tasks Execution Time

DM application computation times depend on many factors: data size, specific mining parameters provided by users and actual status of the Grid etc. Moreover, the correlations between the items present in the various transactions of a dataset largely influence the response times of DM applications. Thus, predicting its performance becomes very difficult.

Our application runtime prediction algorithms operate on the principle that applications with similar characteristics have similar runtimes. Thus, we maintain a history of applications that have executed along with their respective runtimes. To estimate a given application's runtime, we identify similar applications in the history and then compute a statistical estimate of their runtimes. We use this as the predicted runtime.

The fundamental problem with this approach is the definition of similarity; diverse views exist on the criteria that make two applications similar. For instance, we can say that two applications are similar because the same user on the same machine submitted them or because they have the same application name and are required to operate on the same size data. Thus, we must develop techniques that can effectively identify similar applications. Such techniques must be able to accurately choose applications' attributes that best determine similarity. Having identified a similarity template, the next step is to estimate the applications' runtime based on previous, similar applications. We can use several statistical measures to compute the prediction, including measures of central tendency such as the mean and linear regression.

Rough sets theory as a mathematical tool to deal with uncertainty in data provides us with a sound theoretical basis to determine the properties that define similarity. Rough sets operate entirely on the basis of the data that is available in the history and require no external additional information. The history represents an information system in which the objects are the previous applications whose runtimes and other properties have been recorded. The attributes in the information system are these applications' properties. The decision attribute is the application runtime, and the other recorded properties constitute the condition attributes. This history model intuitively facilitates reasoning about the recorded properties so as to identify the dependency between the recorded attributes and the runtime. So, we can concretize similarity in terms of the condition attributes that are relevant and significant in determining the runtime. Thus, the set of attributes that have a strong dependency relation with the runtime can form a good similarity template. The objective of similarity templates in application runtime estimation is to identify a set of characteristics on the basis of which we can compare applications. We could try identical matching, i.e. if n characteristics are recorded in the history, two applications are similar if they are identical with respect to all n properties. However, this considerably limits our ability to find similar applications because not all recorded properties are necessarily relevant in determining the runtime. Such an approach could also lead to errors, as applications that have important similarities might be considered dissimilar even if they differed in a characteristic that had little bearing on the runtime.

A similarity template should consist of the most important set of attributes that determine the runtime without any superfluous attributes. A reduct consists of the minimal set of condition attributes that have the same discerning power as the entire information system. In other words, the similarity template is equivalent to a reduct that includes the most significant attributes. Finding a reduct is similar to feature selection problem. All reducts of a dataset can be found by constructing a kind of discernibility function from the dataset and simplifying it.

In the following, we present the reduct algorithm and application runtime estimation algorithm. For further detailed information see [6, 11], necessary rough set notions see [12, 14-18].

3.3.1 Heuristic Reduct Algorithm

Every entry in discernibility matrix is a set of attributes that can be distinguished by the attributes. The more frequent an attribute appears in entries of discernibility matrix, the more instance pairs can be distinguished by this attribute. So appearing frequency represents the distinguish ability of the attribute. In other words, appearing frequency implies relevance between attribute and class label. Thus attributes' frequency can be used as heuristic. We can sort attributes into ascending order by its frequency, and add first attribute (which one with highest appearing frequency) to reduct. Then we examine reduct by number of instance pairs that can be distinguished by this reduct. If threshold cannot be satisfied, next attribute is added to reduct. Do it recursively until stop criteria are satisfied.

In an optimal feature subset, feature should have high relevance with class label, and have low relevance with other features in the subset. This approach can take out irrelevant attributes, but how about redundant attributes? For example:

Attribute a has highest appearing frequency in discernibility matrix. Attribute b is completely dependents with a, and b's appearing frequency only lower than attribute a. If there is attribute c with appearing frequency lower than attribute b, but attribute c is irrelevant with attribute attribute a. According this approach a should be added in reduct firstly, and then b is added. Attribute c is added to reduct after attribute b. But in fact, it is obviously attribute b cannot provide any additional distinguish ability to the feature subset, and it should be take out from reduct.

So the above approach can not guarantee that attributes in subset have low relevance with each other. In order to solve this problem, we propose a simple but efficient method to avoid adding redundant attributes to reduct. After an attribute *ai* added to reduct, we remove all entries containing *ai* from discernibility matrix, recount attributes' appearing frequency in remained entries. Then attribute with highest new appearing frequency should be added to reduct. In fact, the remained entries represent the instances in boundary region with respect to attribute subset. According to attribute subsets, those instances cannot be distinctly classified into positive region or negative region. Thus the less boundary region attributes means the more powerful classify capacity.

On the other hand, length of entry means how many attributes can distinguish corresponding instance pair. Shorter entry implies only few attributes can distinguish corresponding instance pair. The shorter the entry is, the more important attributes in this entry are. Extremely, if entry length is 1, the only attribute contained in this entry is a member of core. So the length of entry also can be used as another heuristic information.

In our algorithm, every attribute has two properties: appearing frequency in discernibility matrix and shortest entry length. Attribute's appearing frequency is updated after a new attribute added to reduct; attribute's length is the length of shortest entry containing this attribute and it is calculated when discernibility matrix is computed. We select attribute according these two properties. The attribute with highest frequency is selected to reduct. If several attributes have same frequency, the shortest one will take precedence.

We list our algorithm as follows:

Input:

Information system: I;

Condition attribute set: A;

Decision attribute set: D;

Threshold.

Output:

A reduct of information system I: Reduct.

Initial State:

Reduct = Null, k = 1.

Step 1:

Generate discernibility matrix M, calculate frequency(ai) and length(ai).

CardM = Card(M)

Step 2:

Reduct = Core

F = A – Core

Step 3:

M = M - {m}, where intersection of entry m and Reduct is not empty.

Recount frequency(ai) and length(ai)

Step 4:

k = Card(M)/CardM

if K <= Threshold, Stop.

Step 5:

Choose highest appearing frequency attribute f form F. If there are several attributes with same appearing frequency, choose the shortest one as f.

Reduct = reduct + {f},

F = F - {f}

Step 6:

Go to Step 3.

Fig. 3 a Heuristic Reduct Algorithm

3.3.2 Implement and Time Complexity

In order to save space, attribute sets are implemented as bit vector. Length of bit vector is equal to number of attributes. A bit of bit vector is set to 1 if corresponding attribute is contained by attribute subset, 0 otherwise. For example, in an information system I with four attributes $\{a1, a2, a3, a4\}$, a bit vector of attribute subset $\{a1, a4\}$ is represented as 1001.

Since discernibility matrix is a symmetrical matrix, there are $|U|(|U|-1)/2$ entries in matrix. To generate each entry, every attribute value of corresponding instance pair should be compared. So in step 1, the cost for generating discernibility matrix is $O(|A||U|^2)$. But in fact, there are much less entries in discernibility matrix, since entry corresponding to same class instance pair is empty. Cost for finding out highest appearing frequency attribute is $O(|A|)$, and cost for finding out shortest attribute is also $O(|A|)$. In the worst case, supposing that there are no redundant and irrelevant attribute in information system, in order to generate reduct it would recur $|A| - 1$ times. So the cost to calculate reduct is $(|A|^2|U|^2)$.

In conclusion, the total cost of our algorithm is $O(|A||U|^2+2|A|+|A|^2|U|^2)$.

So the time complexity is $O(|A|^2|U|^2)$. It is less than time complexity of [13] alogrithm ($O(|A|^3|U|^2)$).

3.3.3 Application Runtime Estimation Algorithm

Let's now look at the estimation algorithm as a whole. Its input is a history record of application characteristics collected over time, specifically including actual recorded runtimes, and a task T with known parameters whose runtime we wish to estimate.

Step 1. Partition the history into decision and condition attributes. The recorded runtime is the decision attribute, and the other recorded characteristics are the condition attributes. The approach is to record a comprehensive history of all possible statistics with respect to an application because identifying the attributes that determine the runtime isn't always possible.

Step 2. Apply the rough sets algorithm to the history and identify the similarity template.

Step 3. Combine the current task *T* with the history *H* to form a current history *HT*.

Step 4. Determine from *HT* the equivalence classes with respect to the identified similarity templates. This implies grouping into classes previous tasks in the history that are identical with respect to the similarity template. Because the similarity template generated using rough sets is a reduct, this leads to the equivalence classes consisting of previous tasks that are identical with respect to the characteristics that have the most significant bearing on the runtime. In this case, rough sets provide a basis for identifying the similarity template and finding previous tasks that match the current task by the intuitive use of equivalence classes. Thus, we integrate the process of matching the current task with previous tasks in the history into the overall process of estimating application runtime.

Step 5. Identify the equivalence class *EQ* to which *T* belongs.

Step 6. Compute the mean of the runtimes of the objects: $EQ \cap H$.

3.4 Scheduling Policy and Execution Model

We now describe how this cost model can be used by a scheduler that receives a list of jobs to be executed on the K-Grid, and has to decide for each of them which is the best resource to start the execution on.

Choosing the best resource implies the definition of a scheduling policy, targeted at the optimization of some metric. One frequent choice [7] is to minimize the completion time of each job. This is done by taking into account the actual ready time for the machine that will execute the job and the cost of execution on that machine, plus the communications needed. Therefore for each job, the scheduler will chose the machine that will finish the job earlier. For this reason in the following we refer to such policy as Minimum Completion Time (MCT).

Jobs L (A, D, {P}) arrive at the scheduler from an external source, with given timestamps. They queue in the scheduler and wait. We assume the jobs have no dependencies among one another and their interarrival time is given by an exponential distribution.

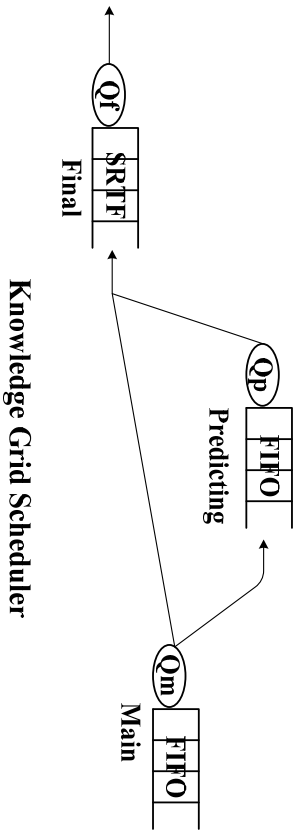

Fig. 4 The model of the scheduler

The scheduler internal structure can be modeled as a network of three queues, plotted in Figure 4, with different policies.

Jobs arrive in the main queue Q_m, where predicting jobs are generated and appended to the predicting queue Q_p. Both queues are managed with a FIFO policy. From Q_p jobs are inserted into the system for execution. Once predicting is completed, the job in inserted in the final queue Q_f , where it is processed for the real execution. Since the scheduler knows the duration of jobs in Q_f , due to the prior predicting, Q_f is managed with a Shortest Remaining Time First (SRTF) policy in order to avoid light (interactive) jobs being blocked by heavier (batch) jobs.

The life-cycle of a job in the system is the following:

1. Jobs arrive in the main queue Q_m with a given timestamp. They are processed with FIFO policy. When a job is processed, the scheduler generates a predicting job and put this request in the predicting queue, with the same timestamp.

2. If a job in Q_m has parameters equals to that of a previously processed job, it is directly inserted into the final queue Q_f , with the same timestamp.

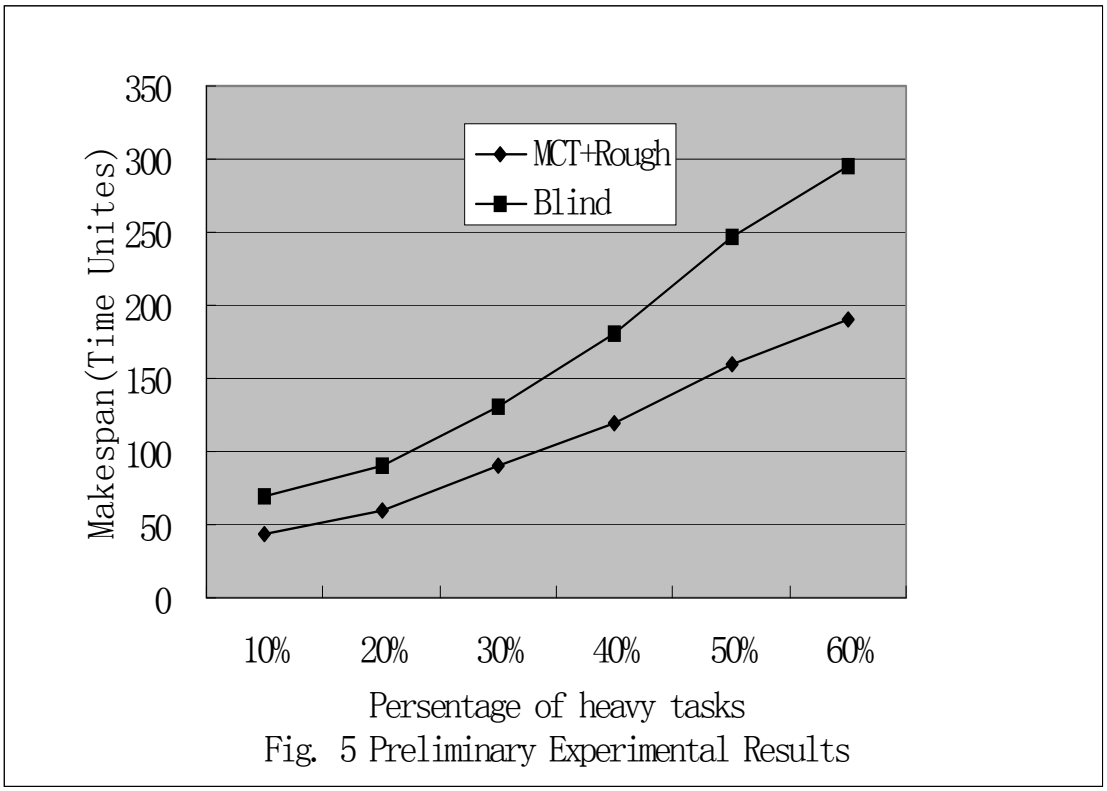

Fig. 5 Preliminary Experimental Results

3. If the predicting job has finished, it is inserted in the Q_f queue, and timestamp given by the current time. Every time a job leaves the scheduler, a global execution plan is updated, that contains the busy times for every host in the system, obtained by the cost model associated to every execution.

4. Every time a job has finished we update the global execution plan.

5. When predicting is successfully finished, jobs are inserted in Q_f , where different possibilities are evaluated and the best option selected. Jobs in Q_f are processed in an SRFT fashion. Each job has an associated duration, obtained from the execution of predicting.

4 Experiment Results

We adopted the MCT (Minimum Completion Time) [8, 9] +rough set approach to validate that our hypothesis is feasible and efficient. The mapper does not consider node multitasking, and is responsible for choosing the schedule for computations involved in the execution of a given task, but also of starting tasks and checking their completion. The MCT mapping heuristics is very simple. Each time a task is submitted, the mapper evaluates the expected ready time of each machine. The expected ready time is an estimate of the ready time, the earliest time a given resource is ready after the execution of jobs previously assigned to it. Such estimate is based on both estimated and actual execution times of all the tasks that have been assigned to the resource in the past. To update resource ready times, when computations involved in the execution of a task complete, a report is sent to the mapper. The mapper then evaluate all possible execution plans for other task and chooses the one that reduce the completion time of the task. To evaluate our MCT scheduler that exploits rough set as a technique for performance prediction, we designed a simulation framework that allowed us to compare our approach with a Blind mapping strategy, which does not base its decisions on performance predictions at all. Since the blind strategy is unaware of predicted runtime, so it scheduled tasks according the principle of FCFS (first come first serve).

The simulated environment is composed of fifteen machines installed with GT3. Those machines have different physical configurations, operating systems and bandwidth of network. We used histories with 500 records as the condition attributes for estimation applications runtime. Data Ming tasks to be

scheduled arrive in a burst, according to an exponential distribution, and have random execution costs. Datasets are all of medium size, and are randomly located on those machines. Figure 5 shows the improvements in makespans obtained by our technique over the blind one when the percentage of heavy tasks is varied.

5 Conclusion and Future Work

In order to improve the performance, we propose a method that makes Data Mining applications parallelization in dynamic Grid environment. For this target, we introduce an additional component that we call serializer, whose purpose is to decompose the tasks into a series of independent tasks according the DAG structure, and send them to the scheduler queue as soon as they become executable with respect to the DAG dependencies. The experimental result demonstrates that the architecture has good performance.

In the future, we will use the technique to build a wide Web Service interface, so that some applications can visit this interface to obtain the performance parameter which they needed for enhancing the performance of system.

References:

[1] D. Talia and M. Cannataro. Knowledge grid: architecture for distributed knowledge discovery. Communications of the ACM, 2002.

[2] H. J. Siegel and A. Shoukat. Techniques for mapping tasks to machines in heterogeneous computing systems. Journal of Systems Architecture, 2000.

[3] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing, 1998.

[4] S. Parthasarathy. Towards network-aware Data Mining. In Workshop on Parallel and Distributed Data Mining, held along with IPDPS01, 2001.

[5] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. Journal of Parallel and Distributed Computing, 59(3):381–422, 1999.

[6] Kun Gao, Youquan Ji, Meiqun Liu, Jiaxun Chen, Rough Set Based Computation Times Estimation on Knowledge Grid, Lecture Notes in Computer Science, Volume 3470, July 2005, Pages 557 – 566.

[7] H. J. Siegel and A. Shoukat. Techniques for mapping tasks to machines in heterogeneous computing systems. Journal of Systems Architecture, 2000.

[8] M. Maheswaran, A. Shoukat, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In 8th HCW, 1999.

[9] H. J. Siegel and Shoukat Ali. Techniques for Mapping Tasks to Machines in Heterogeneous Computing Systems. Journal of Systems Architecture, (46):627–639, 2000.

[10] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: towards an architecture for the distributed management and analysis of large scientific datasets. Journal of Network and Computer Applications, (23):187–200, 2001.

[11] Kun Gao, Kexiong Chen, Meiqun Liu, Jiaxun Chen, Rough Set Based Data Mining Tasks Scheduling on Knowledge Grid, Lecture Notes in Computer Science, Volume 3528, May 2005, Pages 150 – 155

[12] J. Komorowski , et al., Rough Sets: A Tutorial, Rough-Fuzzy Hybridization: A New Trend in Decision Making , S.K. Pal and A. Skowron, eds., Springer-Verlag, pp. 3-98, 199

[13] Keyun Hu, lili Diao and Chunyi Shi: A Heuristic Optimal Reduct algorithm. 22nd Intl. Sym. on Intelligent Data Engineering and Automated Learning (IDEAL2000), Hong Kong, (2002)

[14] X.Hu, Knowledge discovery in databases: An attribute-oriented rough set approach, Ph.D thesis, Regina university, 1995.

[15] J.Starzyk, D.E.Nelson, K.Sturtz, Reduct generation in information systems, Bulletin of international rough set society, volume 3, 1998.

[16] S.K.Pal, A.Skowron, Rough Fuzzy Hybridization-A new trend in decision-making, Springer, 1999.

[17] Witten,I,H., and Eibe,F., "Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations", Morgan Kauffman, 1999.

[18] Keyun Hu, lili Diao and Chunyi Shi: A Heuristic Optimal Reduct algorithm. 22nd Intl. Sym. on Intelligent Data Engineering and Automated Learning (IDEAL2000), Hong Kong, (2002)

[19] Hsies, K.L., Applying data mining techniques into achieving process improvement, WSEAS Transactions on Systems, Volume v 5, Issue 12, December 2006, p 2774-2780, ISSN 1109-2777

[20] Saitta, S., Raphael, B., Smith, I.F.C., Data mining for decision support in multiple-model system identification, WSEAS Transactions on

Systems, Volume 5, Issue 12, December 2006, p 2795-2800, ISSN 1109-2777

[21] Goldschmidt, R., Passos, E., Vellasco, M., Intelligent assistance in KDD task definition, WSEAS Transactions on Systems, Volume 4, Issue 10, October 2005, Pages 1676-1686, ISSN: 1109-2777