Workflow-based Tasks Scheduling on Grid

Kun Gao[1,2], Ning Zhou[3], Kexiong Chen[2], Meiqun Liu[3], Jiaxun Chen[1]
[1]Information Science and Technology College, Donghua University, P.R.C
[2]Aviation University of Air Force, P.R.C
[3]Administration of Radio Film and Television of Jilin Province, P.R.C

Abstract: - Due to the distributed nature of data and the need for high performance, it makes Grid a suitable environment for distributed data mining. Since distributed data mining applications are typically data intensive, one of the main requirements of such a DDM Grid environment is the efficient workflow scheduling. We propose an architecture for a Knowledge Grid scheduler that results in the minimal response time. The experimental result demonstrates that the architecture has good performance.

Key-Words: - Workflow, Scheduling, Data mining, Knowledge Grid

1 Introduction

The Grid has emerged recently as an integrated infrastructure for high performance distributed computation. Data mining is the process of autonomously extracting useful information or knowledge from large data stores or sets. Because of the importance of data mining and grid technologies, it is very useful to develop data mining environments on grid platforms by deploying grid services for the extraction of knowledge from large distributed data repositories. The effort that has been done in the direction of data intensive applications on the grid is the Data Grid project that aims to implement a data management architecture based on two main services: storage system and metadata management [1]. This project is not concerned with data mining issues, but its basic services could be used to implement higher-level grid services such as the ones we intend to develop. Motivated by these considerations, in [2] a specialized grid infrastructure named Knowledge Grid (K-Grid) has been proposed. This architecture was designed to be compatible with lower-level grid mechanisms and also with the Data Grid ones. The authors subdivide the K-Grid architecture into two layers: the core K-grid and the high level K-grid services. The former layer refers to services directly implemented on the top of generic grid services, the latter refers to services used to describe, develop and execute parallel and distributed knowledge discovery (PDKD) computations on the K-Grid. Moreover, the layer offers services to store and analyze the discovered knowledge.

We concentrate our attention on the K-Grid core services, i.e. RAEM (Resource Allocation and Execution Management) services. The RAEM service provides a specialized broker of Grid resources for DDM computations: given a user request for performing a DM analysis, the broker takes allocation and scheduling decisions, and builds the execution plan, establishing the sequence of actions that have to be performed in order to prepare execution, actually execute the task, and return the results to the user. The execution plan has to satisfy given requirements (such as performance and response time) and constraints (such as data locations, available computing power, storage size, memory, network bandwidth and latency). Once the execution plan is built, it is passed to the Grid Resource Management service for execution. Clearly, many different execution plans can be devised, and the RAEM service has to choose the one which maximizes or minimizes some metrics of interest (e.g. throughput, average service time).

Data mining applications running on the K-Grid can be parallelized. Such, we can parallelize single data mining application to several subtasks; several tasks may be combined to form a workflow. In this paper, we propose a workflow 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 data mining application to DAG. In section 3, we present the architecture for a Knowledge Grid scheduler that results in the minimal response time. In section 4, we conduct experiments to evaluate the architecture. Finally section 5 concludes this paper.

2 Decomposing Data Mining Application 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 Directed Acyclic Graph (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 data mining application to DAG.

2.1 Modeling Data Mining Applications

We surveyed three major classes of data mining 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 data mining application that follows this generate-and-test paradigm. We call this application class pattern lattice data mining. In order to characterize the computation models of these applications more concretely, we define them more carefully in Section 2.2.

2.2 Defining Data Mining Applications

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 data mining 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 data mining 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 Data Mining Applications

Having defined data mining applications as above, it is easy to see that an optimal sequential program that solves a data mining 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 data mining application to DAG.

We propose to use a directed acyclic graph (dag) structure called exploration dag (E-dag, for short) to characterize pattern lattice data mining applications.

We first describe how to map a data mining application to an E-dag.

The E-dag constructed for a data mining 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-dag and the set of all possible patterns. Therefore, we refer to a vertex and the pattern it is labeled with interchangeably.

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. Figure 1 shows an E-dag mapped from an association rule mining application.

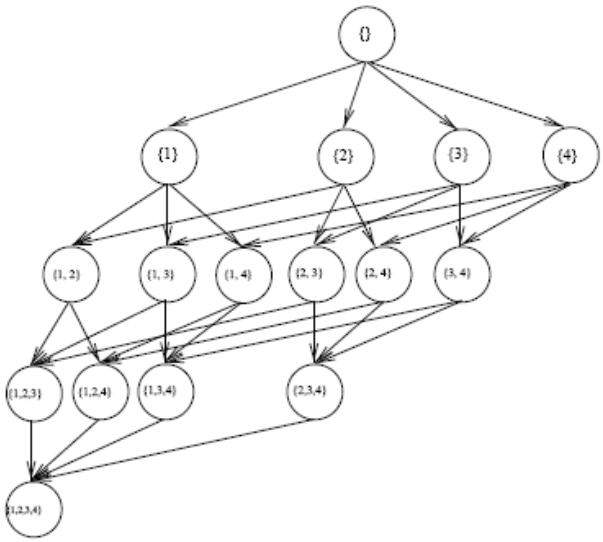

Figure 1: A complete E-DAG for an association rule mining application on the set of items {1, 2, 3, 4}.

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 data mining algorithm, D is the input dataset, and {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. See [3] for a review of them. 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 (Figure 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.

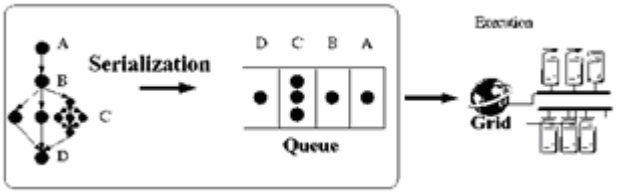

Figure 2 Serializer

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.

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.

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}.

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, $d_1 = s_j - (s_{hj} + \frac{|D_i|}{b_{hj}}) \geq 0, d_2 = s_{jk} - (s_j + e_{ij}) \geq 0$

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 $C_i = C_{\bar{ij}}$ and $\overline{C_i} = \overline{C_{\bar{ij}}}$. Let T be the set of tasks to be scheduled. The makespan for the complete scheduling is defined as $\max_{t_i \Xi T}(C_i)$, and measures the overall thoughput of the system.

3.3 Predicting DM Tasks Execution Time

Data mining 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 data mining 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.

For further detailed information see [4].

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 [5] 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.

The scheduler internal structure can be modeled as a network of three queues, plotted in Figure 3, 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.

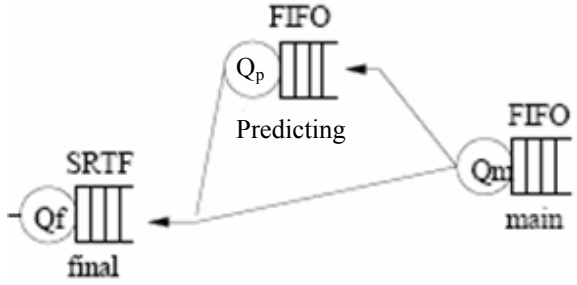

Figure 3: The model of the scheduler.

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.

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 Some Preliminary Results

We adopted the MCT(Minimum Completion Time)[6]+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 4 shows the improvements in makespans obtained by our technique over the blind one when the percentage of heavy tasks is varied.

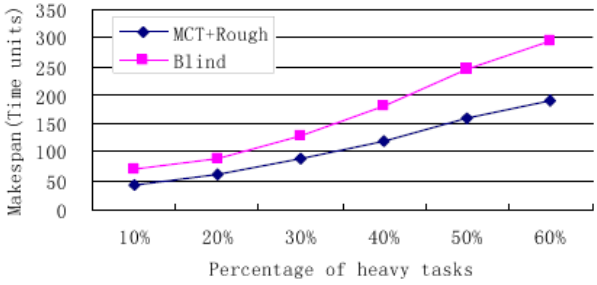

Figure 4 Preliminary Experimental Results

5 Conclusion

We propose a new solution for data mining task scheduling in Grid environment. First, we propose map a data mining application to DAG. Then, we propose a cost model for predicting the data transfer time and data mining execution time on Grid. Finally, according the priori estimation of cost, we propose the method for tasks scheduling to minimize total response time in grid environment.

References:

[1] 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.

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

[3] 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.

[4] 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.

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

[6] Tracy D. Braun, Howard Jay Siegel, Noah Beck.: A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems, Journal of Parallel and Distributed Computing, 2001