Embedding Software Requirements in Grid Scheduling

Daniel M. Batista, Cesar G. Chaves, Nelson L. S. da Fonseca
Institute of Computing
State University of Campinas, Brazil
Email: {daniel,cchaves}@lrc.ic.unicamp.br, nfonseca@ic.unicamp.br

Abstract—Both grid and cloud applications may require specific software for their execution. Introducing such requirements into the DAG describing the dependencies among the tasks of an application leads to a combinatorial problem with numerous possibilities of modified DAGs. This paper introduces an algorithm to obtain a single DAG with the minimum possible makespan by reducing the need for data transfer over the network.

I. INTRODUCTION

Grid networks (Grids) have been designed to provide computational resources on a large scale for advanced science and engineering [1]. In grids, computational resources spread around the world are connected by network links composing virtual organizations for the execution of highly demanding applications. The positive experience from the utilization of grids encouraged the conception of the cloud computing paradigm [2]. Although the two paradigms differ, several mechanisms and protocols used in clouds have been inherited from grids [3]. Therefore, improvements in grid mechanisms can positively impact several clouds already deployed on the Internet.

Grid applications can be decomposed in a set of programs called tasks. Efficient scheduling of tasks to the grid resources is essential to grid operation. The grid scheduler is responsible to define how to allocate resources to each task. The specification of the requirements of an application can include software requirements [4], which demands virtual machines (VMs) to be instantiated in the grid hosts before the execution of the tasks. The instantiation of VMs impacts the utilization of the network links, since the images of the VMs must be transferred from a repository to the hosts. Besides that, the time to transfer and boot the VMs can increase the execution time (i.e. makespan) of the applications, affecting the quality of service required by users.

In [5], an approach to embed software requirements in the description of grid applications was introduced. In this approach, the scheduler does not need to be modified in order to consider the instantiation of VMs in the search for the best schedule. The problem is the need to instantiate one VM for each task, since the same image is transferred and instantiated several times, instead of being transferred and instantiated just once and shared by other tasks.

The objective of this paper is to present a new algorithm to embed software requirements in the description of grid applications. The algorithm differs from other approaches since it promotes the sharing of VMs among tasks with the same software requirement. Such sharing is pursued only when it does not affect the makespan of the applications. The algorithm is oriented to applications composed by dependent tasks and described by Directed Acyclic Graphs (DAGs). Such tasks transfer data among themselves and pose additional demands on network usage than tasks without dependencies.

The proposed approach was evaluated against two others, including the approach introduced in [5]. The DAGs modified by the proposed approach can lead to makespans that are up 9.69% lower than those given by DAGs modified by the approach in [5]. Besides that, the DAGs modified by the proposed approach do not pose additional demands on the schedulers in relation to the DAGs modified by other approaches.

This paper is organized as follows: Section II discusses related works. Section III presents the problem addressed by the proposed algorithm and the algorithm itself. Section IV explains the experiments conducted in order to evaluate the performance of the proposed algorithm. Section V presents the results obtained from the experiments, and Section VI concludes the paper.

II. RELATED WORK

In [5], a mechanism to promote the instantiation of VMs to deal with the heterogeneity of grid environments was introduced. Instead of having several hosts with all possible software configurations required by grid applications, the mechanism proposes the utilization of a VM repository. The application must then be modified to include the software requirements of the tasks. This was done by preceding each task in the DAG by a new task representing the instantiation of a VM. After the transformation, the new application can be scheduled by any grid scheduler which does not need to be modified to consider the software requirements of the applications. The problem with this proposal is that it demands great usage of network links, since each task requires the instantiation of one VM. Thus, an application with k tasks requires k images to be transferred from the repository to the grid hosts. In this paper, it is introduced an algorithm to decrease the number of instantiations and, as a consequence, to diminish the chances of network congestion. The proposal in [5] is used as a reference for comparing the performance of the proposed algorithm.
Techniques of DAG transformation were also used in [6] to schedule a specific class of applications, consisting of serial tasks partitioned in parallel tasks with requirements represented by additional precedent tasks. By doing this, the application can be scheduled by any grid scheduler. The proposal in [6] can not be used to modify DAGs of generic applications with software requirements since it is oriented to entity-level simulations applications.

The HEFT (Heterogeneous Earliest-Finish-Time) scheduler [7] is a classical scheduler largely used to evaluate the performance of new grid scheduling approaches [8] [9]. It schedules tasks considering the criterium of the longest path leading to the final task in the DAG. The HEFT is used in this paper to schedule the modified DAGs.

III. THE DAG MODIFIER

To schedule DAGs of grid applications with software requirements, it is necessary to embed the software dependencies into the original DAG [5]. The software dependencies must be transformed into new tasks appended to the first levels of the DAG. Each new task represents the instantiation of one VM that contains the software required by the original task. The transfer of the VM image from the repository to the host also needs to be included in the new DAG. This can be achieved by creating a new entry task to represent the repository of VMs. The arcs from the repository task to the instantiation tasks represent the transfer of image file of VMs. The demands of such transfers can not be neglected since they consume network bandwidth. For example, the size of images available at the public repository described in [10] vary from 115MB to 1229MB.

Fig. 1 illustrates the process to embed the software dependencies into the original DAG as proposed in [5]. Consider the DAG with 5 tasks on the left side of the figure. The representation of the tasks have the label x:y, where x is the task ID and y is the ID of the software the task x depends on. In order to execute this DAG, it is necessary that VMs with software S1 and software S2 to be previously instantiated. The new DAG in the right side of Fig. 1 shows the modified DAG for that. The new tasks S1 and S2 represent the instantiations of the VMs requested by the original tasks, and the new task R represents the repository. The VM with software S1 will be instantiated twice, once to task 1 and once to task 2. Similarly, the VM containing the software S2 will be instantiated three times (tasks 3, 4, and 5). This new DAG can now be scheduled by any schedulers, since the scheduler does not need to differentiate the new tasks from the original ones.

The problem with the proposed approach is that there is no guarantee that the modified DAG will lead to low makespans. This happens because DAGs with software requirements can be modified in several different ways, each of them producing a new modified DAG. Fig. 2 illustrates another DAG modification valid to the original DAG of Fig. 1. Instead of having one specific instantiation to each task, the DAG modification used in Fig. 2 aggregates all the tasks with dependency on the same software so that just one instantiation is demanded. In this case, the VM containing software S1 will be instantiated in a host for the execution of task 1, that can be followed by the execution of the task 2. The VM containing software S2 can be similarly shared by tasks 3, 4, and 5.

Fig. 1. Example of the DAG modification – one VM to each task.

Fig. 2. Another DAG modification – one VM to the tasks dependent on the same software.

Each potential modified DAG can have a different makespan. For example, there are 877 possible DAGs when 7 tasks depend on the same software in the original DAG of the Montage application, a real application executed in grids [4] and represented in Fig. 3. When these DAGs are scheduled on a grid network with 400 hosts using the HEFT scheduler, the makespans produced by the different modifications vary in the interval (321, 518) seconds. The graph in the Fig. 4 plots the cumulative distribution function (CDF) of the makespans of the 877 modified DAGs. The CDF shows that 50% of the DAGs have makespans greater than 396 seconds, which is 23.55% lower than the maximum expected makespan of the application (518 seconds). This shows that the modification of the DAGs can not be realized randomly, since there is no guarantee that the makespan of the application will be close to the minimum.

The ideal DAG modification should transform the original DAG into one that demands the lowest makespan when scheduled. A naive solution to this problem is to generate all the possible modified DAGs to the original DAG, schedule them and select the one with the lowest makespan. As will be show next, this solution is computationally impracticable.

It is possible to solve the problem of the generation of all the modified DAGs by solving a problem to find all the partitions of a set. A partition of a set S is a set P of nonempty
subsets of $S$ so that $\bigcup P = S$. Each software $S_i$ requested by grid tasks can be seen as a different set containing the IDs of the tasks dependent on $S_i$. The partitions of $S_i$ gives all the possible instantiations of the VM containing the software $S_i$, each partition containing the list of tasks that require the same VM. In the examples in Fig. 1 and Fig. 2, the sets are $S_1 = \{1,2\}$ and $S_2 = \{3,4,5\}$. The partitions in the example in Fig. 1 are $\{\{1\}, \{2\}\}$ and $\{\{3\}, \{4\}, \{5\}\}$, while the partitions in the example in Fig. 2 are $\{\{1,2\}\}$ and $\{\{3,4,5\}\}$.

Although the number of partitions of a set of size $n$ can be calculated by the Bell number [11] (Equation 1), the search for all the partitions of a set leads to an algorithm with exponential computational complexity [12]. So, even if the grid scheduler runs fast, it is not practical to generate all the possible DAGs, find their schedules and select the one with the lowest makespan.

$$B_0 = 1$$

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

Algorithm 1 presents the proposed approach to modify a DAG with software requirements. The aim is to reduce the utilization of the network links and, at the same time, avoid increasing the makespan of the applications. This is done by searching the tasks with the same software dependency on the paths of the DAG, so that there will be one instantiation of VM to each software dependency on a path.

**Algorithm 1** DAD modifier

**Input:** DAG $D$ with software requirements $S$ not embedded into the DAG.

**Output:** modified DAG $M$ with software requirements embedded into the DAG.

1. for each path $h \in D$ do
2. for each task $t \in h$ do
3. Include $t$ into the set $P_{h,s}$, where $S_s$ is the software dependency of $t$
4. end for
5. end for
6. Create one new entry task $R$ with weight 0
7. while there is at least one set $P$ do
8. for each software $s \in S$ at least one set $P_{h,s}$ do
9. Do $P_s$ equal to the heavier $P_{h,s}$ (The weight of $P_{h,s}$ is measured in bytes and it is calculated by adding the weights of the arcs in the path $h$)
10. Create one new task $p_s$ with weight equivalent to the boot time of the VM containing software $S_s$
11. Create one new arc from the task $p_s$ to each one of the tasks in $P_s$ with weight $\infty$ (By assigning $\infty$, the tasks will execute at the same host where the VMs will be instantiated)
12. Create one new arc from the task $R$ to the task $p_s$ with weight equivalent to the size of the VM containing software $S_s$
13. Delete the heavier $P_{h,s}$
14. end for
15. end while

When considering $m$ tasks dependent on the same VM instantiation, $m - 1$ transfers of the required VM image are avoided, which does not happen in the approach in [5] (Fig. 1).

In this way, the number of data transfers is reduced. Moreover, by aggregating only the tasks on the same path, there is no reduction of the parallelism of the application. If all the tasks dependent on the same VM were aggregated, as done in the example in Fig. 2, tasks with no data dependency would be imposed an artificial dependency, which prevent their parallel execution. This can be understood by examining the example in Fig. 2; task 3 competes with tasks 4 and 5 by the same instantiation of VM S2. This can lead to the execution of task 3 just after task 4, enlarging the ending time of task 3, and, consequently, the ending time of task 5 and the makespan of the application. By aggregating only tasks in the same path on the DAG, no new dependency among tasks is created and thus the existing parallelism in the application is not reduced.

Fig. 5 illustrates the modification of the DAG used as example when the Algorithm 1 is employed. Although tasks 3, 4 and 5 depend on software S2, they do not depend on the same VM instantiation since they are not on the same path in the DAG. In this example, tasks 4 and 5 depend on the same VM instantiation, instead of tasks 3 and 5, due to the length of the paths. It was considered that task 4 send more bytes to task 5 than does task 4 to task 3.

**IV. PERFORMANCE EVALUATION**

To evaluate the performance of the proposed algorithm, we compared it with two other ones. The Montero’s approach [5], which creates a new DAG with one VM instantiation for each task of the original DAG (Fig 1), and the “Aggregate” approach, which creates a new DAG with one VM instantiation.
Fig. 5. Another DAG modification – one VM to the tasks dependent on the same software on the same path.

for all the tasks of the original DAG that depend on the same software (Fig 2). The DAGs modified by each approach were scheduled by the HEFT scheduler. We measured the makespan of the schedule found by the scheduler and the time taken by the scheduler to find it. All the codes were implemented in C and executed in an Intel Xeon computer with 2.00GHz and 4GB of RAM running Debian GNU/Linux 5.0. A code was implemented to generated all the possible modified DAGs, so that the minimum and the maximum makespans can be computed.

Scenarios based on real grids deployed around the world were created. The application shown in this paper is the Montage application with 62 tasks (Fig. 3). The number of instructions of each task was drawn from an uniform distribution in the interval [45,53] and the number of bytes transferred between tasks were drawn from an uniform distribution in the interval [3,4]MB. The sum of all transfers were 339.5MB [1] [4] (The drawings were repeated until the sum was achieved). Two grids differing in the number of hosts were considered. The number of hosts can be either 300 or 400, which represent common sizes of real grids [13]. The network topology formed by the grid hosts was defined by the Barabasi-Albert 2 method [14], and the available bandwidth between hosts was drawn from a Pareto distribution in the interval [10,1024]Mbps. The available bandwidth from the VM repository to the grid hosts was drawn from the same distribution and interval. The average time to boot a VM was 30 seconds and the sizes of the VM images were drawn from the sizes of the distribution of the images available in [10], i.e., in the interval [115,1229]MB.

The distribution of software dependencies among the tasks of the DAG impacts the quality of the solution of the different approaches. We considered two different ways to distribute the dependencies on the DAGs. In the first one, referred as Right Path (RP), there were 5 sets of tasks on the same paths which are dependent on the same software. Four sets had 2 tasks. The fifth set had a variable size, from 1 to 7, which corresponds to the path taken at the right side of the Montage DAG. In the second distribution, referred as Bottom-Up Right-Left (BU-RL), there was just 1 set of tasks dependent on the same software, but not necessarily on the same path in the DAG. The size of the set varied from 1 to 7, which corresponds to transverse the last levels of the Montage DAG, transversing bottom-up and from right to the left. By using the two ways to distribute the dependencies on the DAG, the two opposite scenarios are considered. In the RP case, the approaches are evaluated for a variable number of tasks dependent on the same software on the same path of the DAG. In the BU-RL case, the approaches are evaluated for a variable number of tasks dependent on the same software but not on the same path of the DAG.

We neither considered DAGs with 8 or more tasks dependent on the same software (with 8 tasks, there would be generated 70,380 modified DAGs), nor considered more than 5 sets of tasks dependent on the same software, since the code that generated all the possible modified DAGs demanded more main memory than the available one.

V. NUMERICAL RESULTS

The graphs in Fig. 6 and Fig. 7 plot, respectively, the makespans when considering grids with 300 and 400 hosts to the DAGs with the RP distribution of dependencies. The grey area in the graph represents the range of possible makespans. The $x$ axis represents the number of tasks dependent on the VM with ID 62, which was randomly chosen.

![Fig. 6. Makespans in the RP case – Grid network with 300 hosts.](image)
by the Montero’s approach, due to the fact there were other 4 paths in the DAG with tasks dependent on the same VM. In this particular scenario, the makespan given by the Montero’s approach was the maximum possible one. Moreover, when there were 7 tasks dependent on the VM 62, the makespan produced by the proposed approach was only 0.55% higher than the minimum possible one.

The modified DAGs generated by the “Aggregate” approach were the same given by the proposed approach in this paper, since the tasks dependent on the same VMs were always on the same paths. Consequently, the makespan of the two approaches were the same.

Results confirm that the proposed approach can decrease the makespan of applications by decreasing the amount of data transferred via the network to instantiate the VMs.

When a grid network with 400 hosts was considered (Fig 7), the average makespan with the DAG generated by the proposed approach was 5.31% lower than that generated by using the DAG given by the Montero’s approach. The maximum difference was 9.69%. When there were 2 tasks dependent on VM 62, the scheduling with the DAG generated by the Montero’s approach produced a makespan lower than that given by the proposed approach, but the difference was only 0.38%. This happened because the HEFT scheduled some tasks of the Montero’s DAG on hosts connected by links with more available bandwidth. Results with small differences in the produced makespans are consequence of the heuristics employed by the adopted scheduler. Nevertheless, when the number of the tasks dependent on VM 62 was 1, 3 and 4, the makespan given by using the DAG generated by the proposed approach was the minimum possible one.

The graph in Fig. 8 summarizes the results obtained in the BU-RL case. Only results for the grid with 300 hosts are presented since the results for the grid with 400 hosts were quite similar.

When using the BU-RL distribution of dependencies, there were a maximum of 2 tasks dependent on the same VM on the same path. This is the reason why the makespans given by the proposed approach were not much lower than the makespans given by the Montero’s approach. However, on average, the proposed solution produced makespans 0.96% lower than those given by the makespans of the Montero’s approach. It is important to observe that when there were not a large number of tasks dependent on the same software on the same path of the DAG, the performance of the proposed approach was similar to that of the Montero’s approach.

The small graph in Fig. 8 shows that aggregating all the tasks dependent on the same VM does not lead to good solutions. The makespans values produced by using the DAG generated by the “Aggregate” approach were maximum in 6 out of 7 points in the graph. This confirms that to avoid the utilization of network resources by instantiating only 1 VM to all tasks dependent on it does not produce good results. Aggregating tasks on the same path leads to much better results as already shown.
The proposed approach demanded the lowest computational time. The execution time when the HEFT scheduled the DAG given by the proposed approach was, on average, 1.13% lower than that when the DAG given by the Montero’s approach was scheduled, and 0.07% lower than that when the DAG given by the “Aggregate” approach was scheduled.

Thus, it is possible to conclude that the proposed approach generated DAGs more efficiently scheduled than the Montero’s and the “Aggregate” approaches. Besides that, the DAGs generated by the proposed approach did not require longer execution times compared to the other two approaches.

VI. CONCLUSIONS

The modification of the DAG describing grid applications with software requirements is an alternative to the development of new schedulers. The large number of valid modifications motivates the proposal of heuristics to produce a single DAG. In this paper, it was proposed an algorithm to create new DAGs which give makespans that are smaller than those given by other existing approaches. The DAG generated by the proposed approach reduces the makespan, as well as the utilization of the network links, by sharing VMs among tasks on the same path of the DAG. Moreover, the DAGs generated by the proposed approach do not demand longer execution times than those of the other two approaches. Although the proposed approach focuses on grid applications with software requirements, it can be used to schedule cloud applications, since in this case the tasks also have software requirements met by instantiation of VMs [3].

REFERENCES