Using Virtual Grids to Simplify Application Scheduling

Richard Huang[1], Henri Casanova[2], and Andrew A. Chien[1]
[1]Computer Science & Engineering and Center for Networked Systems
University of California, San Diego
{ryhuang, achien}@csag.ucsd.edu
[2]Information and Computer Sciences Department, University of Hawai`i at Manoa
henric@hawaii.edu

Abstract

Users and developers of grid applications have access to increasing numbers of resources. While more resources generally mean higher capabilities for an application, they also raise the issue of application scheduling scalability. First, even polynomial time scheduling heuristics may take a prohibitively long time to compute a schedule. Second, and perhaps more critical, it may not be possible to gather all the resource information needed by a scheduling algorithm in a scalable manner. Our application focus is scientific workflows, which can be represented as Directed Acyclic Graphs (DAGs). Our claim is that, in future resource-rich environments, simple scheduling algorithms may be sufficient to achieve good workflow performances. We introduce a scalable scheduling approach that uses a resource abstraction called a virtual grid (VG). Our simulations of a range of typical DAG structures and resources demonstrate that a simple greedy scheduling heuristic combined with the virtual grid abstraction is as effective and more scalable than more complex heuristic DAG scheduling algorithms on large-scale platforms.

1. Introduction

Efficiently scheduling the tasks of a parallel application on the resources of a distributed computing platform is critical for achieving high performance. The scheduling problem has been studied for a variety of models and assumptions and has proved to be NP-complete in most cases [1]. Consequently, researchers have developed many heuristics that exhibit polynomial time complexity and attempt to approach the optimal schedule. A popular application model for which scheduling heuristics have been developed is the "task graph" model, by which an application is represented as a weighted Directed Acyclic Graph (DAG). Nodes in the DAG represent computational tasks, and edges represent data communication among tasks. Node weights represent computational costs, and edge weights represent amounts of data to transfer between tasks. A survey of "DAG scheduling algorithms" is available in [2].

The DAG application model is particularly relevant for *scientific workflows* [3]. The last few years have seen active development and deployment of many such workflows in various domains [4-6] and these workflows require considerable amounts of computing power. Therefore, it is natural to explore the possibility of executing them on large-scale computing platforms such as grids [7]. And indeed, several efforts are underway to provide software frameworks for "grid workflows" [8]. These efforts typically focus on workflow instantiation and deployment issues and they leave the question of efficient scheduling mostly unaddressed.

In this paper we ask the question: "Are sophisticated DAG scheduling algorithms required to schedule workflows on grid platforms?" Our claim is that, in many relevant cases, simpler scheduling approaches are viable alternatives and can be preferable in practice as they are as effective, more robust, more scalable, and simpler to implement.

To "replace" the work done by sophisticated scheduling algorithms, we employ a resource abstraction called the *Virtual Grid* (VG) [9, 10]. With this abstraction, we demonstrate in simulation that a naïve greedy scheduling algorithm, which does not account for resource information nor for application information beyond task dependencies, can be as effective as a popular DAG scheduling algorithm for scheduling a workflow on large-scale grid platforms.

The rest of this paper is organized as follows. In Section 2 we define the problem, identify challenges, and highlight the limitations of current solutions. Section 3 details our approach based on the VG abstraction. Section 4 presents our experimental methodology, and Section 5

1-4244-0054-6/06/$20.00

presents our results. Section 6 discusses our results and highlights future directions.

2. Scheduling Applications in Large-Scale Environment

Users of scientific applications, and in particular of scientific workflows, are increasingly faced with situations in which they have to select appropriate compute resources among a large number of potential resources distributed over the wide-area. This is due to two factors. First, with dropping hardware prices for commodity computers, with several cluster vendors, and with the availability of open-source cluster management tools [11], it is increasingly affordable and straightforward to purchase/deploy powerful Linux clusters. Second, the development of the grid middleware infrastructure [12] makes it straightforward for users to access a wide collection of resources uniformly and securely. Additionally, with projects [13] exploring optical networks and providing high bandwidth among many clusters, there is a trend towards resource-rich environments with good network connectivity in which users can access many clusters in many institutions concurrently. Workflow applications can benefit from such environments because they are often loosely coupled and can utilize resources at multiple sites concurrently and efficiently.

With more resources to choose from, the question of effective scheduling becomes critical. In this paper we define application performance as application turn-around time, which includes the time to compute a schedule (encompassing selection of resources and selection of task-resource mappings) and the time to execute the schedule (also called makespan).

2.1. Challenges

With the explosion in the number of computing resources, the major challenge for scheduling workflows is scalability of the scheduling algorithm itself. Although of polynomial complexity, DAG scheduling heuristics may become impractical when considering large numbers of individual resources. More importantly perhaps, existing heuristics require information about individual resources and about their distances from each other over the network. Collecting and processing reasonably up-to-date such information may itself not be scalable. There is therefore a trade-off between the time spent computing a schedule (perhaps prohibitively high for a sophisticated heuristics, but low for a simple one) and the time spent executing it (arguably low for a sophisticated heuristic, but probably high for a simple one).

2.2. Current Approaches and Limitations

As seen in [2], DAG scheduling heuristics that calculate and account for the ``critical path'' of the DAG are often the most effective. The critical path is essentially the longest path in the DAG (in terms of node and edge weights), and is thus a lower bound on the overall makespan. These heuristics attempt to lower this lower bound in the hope of lowering the makespan.

In practice however, for the purpose of scheduling grid workflows, these heuristics are not used. For instance, the Pegasus grid workflow framework [14, 15] implements only the simplistic random, round-robin, or min-min [16] heuristics for scheduling workflows of the Montage astronomy application [17, 18].

There are several reasons for the lack of acceptance of more sophisticated scheduling algorithms. First, these algorithms are more complicated to implement. Second, they often require more information about the application and/or the resources, which may be difficult to obtain scalably. Third, there has been no clear demonstration that they would improve application turn-around time in practice (i.e., achieve a good trade-off between the time to compute a schedule and the time to execute it).

An open question is then "Should sophisticated scheduling algorithms be used for workflows in grid environment?" While scheduling is a common topic of discussion in the grid workflow community there is not consensus on the answer to this question. In this paper we show that although the use of sophisticated algorithms may be worthwhile, simplistic algorithms can achieve comparable or even better application turn-around time in many relevant cases, provided that an adequate resource abstraction is used for resource pre-selection

3. Scheduling Using Virtual Grids

One simple way in which to improve the scalability of a scheduling algorithm is to constrain its operation to a pre-selected set of resources. Scheduling algorithms typically perform *implicit* resource selection: they consider a large number of potential resources and compute a schedule that utilizes only a fraction of these resources in the end. On resource-rich grid platforms for workflow applications this fraction is typically minute. If instead one pre-selects a "good" subset of the potential resources and then run the scheduling algorithm, one should be able to obtain schedules with comparable makespans in significantly less time. Perhaps more importantly, if the pre-selection is reasonable, it is possible that very simple scheduling algorithms could achieve similar makespans as more sophisticated algorithms. The sophisticated algorithms should improve the makespan, but at the expense of longer time to compute the schedule and may require a wealth of information regarding resources and/or applications. The question is whether one can "get by" with simpler algorithms in practice.

3.1. Virtual Grids

A Virtual Grid (VG) provides a high-level, hierarchical abstraction of the resource collection that is needed and used by an application. This abstraction provides a clean separation of concerns between grid applications and the complexity of the grid infrastructure: the application specifies its resource needs using a high-level language, vgDL, and the Virtual Grid Execution System (vgES) finds and allocates appropriate resources.

This abstraction is implemented as part of the VGrADS project [19]. We discuss here only the concepts and features that are relevant to the following sections. More information on the vgDL language and the vgES system can be found in [9] and [10].

The salient point of vgDL is the capability for applications to specify hierarchical resource aggregates and qualitative notions of network proximity between these aggregates. vgDL contains three resource aggregates, distinguished by homogeneity and network connectivity: (i) **LooseBag**: a collection of heterogeneous nodes with possibly poor connectivity; (ii) **TightBag**: a collection of heterogeneous nodes with good connectivity; and (iii) **Cluster**: a set of well-connected nodes with identical (or nearly so) individual resource attributes. The notion of "good" is defined in term of a network latency threshold. The implicit assumption is a positive correlation between low latency and high bandwidth. For instance, in vgDL, an application can request a Cluster of 32 Opteron processors with clock rate higher than 2Ghz and more than 1GB of RAM that is "close" to a TightBag of 32 to 128 processors that have clock rates higher than 1Ghz. The tenet of the VGrADS project is that such simple and qualitative specifications fit the need of most applications in practice. Such requests are sent to the vgES system.

The vgES system constantly gathers and indexes information about available resources in an off-line manner using grid information services [20, 21]. It then uses relational database technology and efficient algorithms to select resources that match the requirements expressed in a vgDL query (see [10] for an evaluation). Note that this system operates at a higher level than grid middleware such as Globus [12] and leverages such middleware to acquire grid resources.

3.2. Scheduling with VGs

Our scheduling approach consists of pre-selecting resources by obtaining a VG corresponding to a simple vgDL description, and then scheduling the application within the resources in the VG. In essence, whereas complex scheduling algorithms explicitly (and sometimes by brute force) search for a good schedule by examining trade-offs between computation and communication explicitly, the vgES, when processing a vgDL requests, immediately bypasses undesirable branches of the search. There is no magic here: vgES does a lot of the necessary work to prune the search space. The point is that it does it very efficiently and scalably. Furthermore, vgES does this once for the entire workflow – not for each task. Thus, we anticipate that this reduced resource search space (and reduced work) and a simple greedy heuristic can in fact achieve comparable or perhaps better turn-around times for many relevant applications in practice.

4. Experimental Approach

Our goal is to investigate whether using the VG abstraction can indeed simplify the scheduling of workflow applications on large-scale platforms. We perform the following experiments. We use DAGs from a real-world grid workflow applications, Montage [17], as well as randomly generated DAGs to better understand the impact of DAG characteristics on our results. We consider a computing platform generated by a tool [22] that instantiates synthetic large-scale computing environments that are representative of current technology.

Using simulation we execute two different scheduling algorithms: a naïve greedy algorithm (which we call "simple") and a standard DAG scheduling algorithm (which we call "complex"). We execute these algorithms in three modes: (i) on the whole "resource universe" without pre-selection of resources; (ii) only on some pre-selected "top" fraction of the resources sorted by clock rate; and (iii) only on pre-selected resources that have been obtained as part of a VG. We obtain the VG by querying our vgES prototype, which has stored resource information corresponding to our synthetic computing environment. Therefore, we conduct 6 different types of experiments, as summarized in Table 1. We provide details on all the above in the following sections.

Table 1: Scheduling schemes in Grid environments

Scheduling Algorithm	Resources
Complex	Universe
Complex	Top Hosts
Complex	VG
Simple	Universe
Simple	Top Hosts
Simple	VG

4.1. The Montage Application

Montage is an astronomy application that creates a mosaic image of a portion of the sky on demand. Figure 1 shows the structure of a small Montage workflow. All tasks on level k have a parent task on level k-1. The top-level tasks (level 1) are not dependent on any other tasks.

For our experiments, we consider a 4469-task Montage workflow used to create a five square degree mosaic of the sky centered at M16. Table 2 shows the average runtimes of Montage tasks on a 1.5Ghz host as reported in [23]. We are interested in seeing how communication might affect scheduling. Therefore, for each Montage workflow, we vary the communication-to-computation ratio (CCR). We test ratios of 0.1, 0.5, 1.0, 2.0, and 10.0. A ratio of 1 implies equal amount of computation and communication. For each task, we calculate the size of its output file based on the computational cost, the CCR, and the maximum bandwidth in the network, which in our case is 10Gbps. For example, for a CCR of 1, we derive the appropriate file size such that the communication cost would also be 8.2 seconds. In this case, the files size would be 152MB, as it would take 8.2 seconds to transfer this on the fastest link in our synthetic platform.

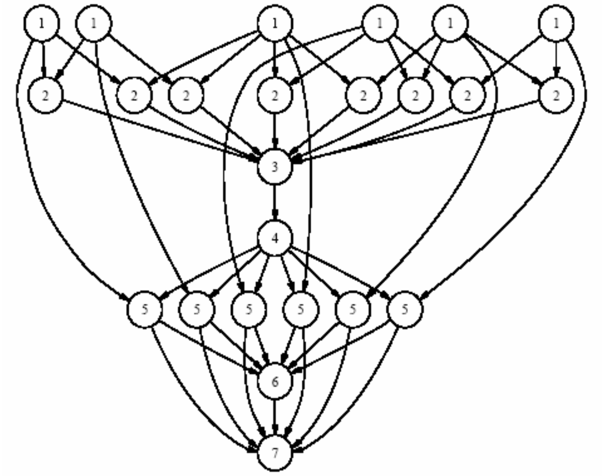

Figure 1: A small Montage workflow

Table 2: Runtime and number of tasks at various levels of the Montage workflow

Level	Task name	Number of Tasks	Runtime (in seconds)
1	mProject	892	8.2
2	mDiffFit	2633	2
3	mConcatFit	1	68
4	mBgModel	1	56
5	mBackground	892	1
6	mImgtbl	25	6
7	mAdd	25	40

4.2. Random DAGs

We also generate a collection of random DAGs following the method outlined in [2]. For each random graph, we vary its size, its mean computation cost (using a 1.5Ghz host as the reference), its communication-to-computation ratio (CCR), its parallelism, its density, its regularity, and its mean task computational cost. The parallelism parameter determines the width of the DAG; density characterizes the number of edges; regularity determines the regularity of the number of tasks at each level. Table 3 summarizes the different parameters and their corresponding values for the random DAGs we generate. See [2] for more details.

Table 3: DAG parameters and corresponding values for random DAG generation

DAG Parameter	Values	Default Value
DAG size (tasks)	44, 447, 4469, 8938	4469
CCR	0.1,0.2,1,2,10	1
Parallelism	0.1,0.2,0.5,0.8,1	0.5
Density	0.1,0.2,0.5,0.8,1	0.5
Regularity	0.1,0.2,0.5,0.8,1	0.5
Mean comp cost	1,5,40,100	40

4.3. Scheduling Algorithms

Among all the DAG scheduling algorithms surveyed and evaluated in [2] we choose the popular MCP (Modified Critical Path) algorithm [24], as it is competitive according the results in [2]. MCP is our "complex" scheduling algorithm. The pseudo code for MCP is shown in Figure 2.

```
$CP$ = length of the longest path (in terms of node weights
        and edge weights)  from the root node to the end
        node, including both these nodes
For each non-root node $N_i$ in the DAG
    $BL_i$ =  length of the longest path (in terms of node
              weights and edge weights) from node $N_i$ to the
              end node, including both these nodes
    $ALAP_i = CP - BL_i$
End For

For each node $N_i$
    $L_i$ = list of the $ALAP$ values of node $Ni$ and all its
          descendents, in ascending order
End For

Sort all $L_i$ lists in lexicographical order and
Re-Order the nodes according to this order
For each node $N_i$
    Schedule $N$i on the host that would complete its
    execution soonest
End For
```

Figure 2: Modified Critical Path (MCP) Algorithm

```
While there are still some tasks to schedule
  For each node $N_i$ whose predecessors, if any,
        have already been scheduled
    Schedule $N_i$ on the host that would start its execution
        soonest
  End For
End While
```

Figure 3: Simple Greedy Algorithm

For our "simple" scheduling algorithm we use a greedy scheduling algorithm that assigns each task to a random available host as soon as the task's dependencies have cleared. The corresponding pseudo code is shown in Figure 3.

We expect that running a more complex scheduling algorithm such as MCP on the resource universe would produce the best makespan by taking into consideration all the resources. We hope that appropriate resource pre-selection would allow a simple scheduling algorithm to achieve better trade-off between the time to compute a schedule and the time to execute the schedule, thereby leading to better turn-around time.

4.4. Resources

We are interested in scheduling applications in large-scale environments. Since we do not have immediate access to hundreds of clusters for running our experiments, we use simulation of a synthetic resource pool. We use the synthetic resource generator described in [22] that bases its models on the characteristics of 650 real-world clusters (ones registered with the Rocks project [25]). We generate 1000 clusters for a total of 33,667 hosts. While having access to 1,000 different clusters may seem far-fetched today, current trends indicate that this may be typical within 5 years.

When scheduling and simulating the execution of workflows we ignore the architectures of the hosts and use only clock rates to determine task runtimes. We scale the reference task runtimes on a 1.5GHz host to account for lower or higher clock rates.

A survey of three topology generators [26-28] showed that none had convincing models for latencies, bandwidths, or contention. We opted for the following simple model. We use a Gaussian distribution for the latencies between clusters. The Gaussian distribution has mean of 100ms and standard deviation of 100ms. We opted for classifying the bandwidths in our synthetic resource pool into 10Gbps for intra-cluster connections and connections with latency lower than 0.5ms (e.g., within a building), 1Gbps for latencies lower than 1ms (e.g., within a campus), 622Mbps (OC12 link) for latencies lower than 40ms, and 155Mbps (OC3 link) for latencies greater than 40ms.

To run our experiments, we used an Intel Xeon 2.4GHz machine, on which all scheduled computations and VG instantiations were performed.

We have assumed that vgES has negotiated with the local resource managers such that the application has sole access to the resources in the VG for the duration of its execution; thus we do not consider resource contention. For network contention, we did not find any other convincing models, so we are using the Gaussian distribution outlined above. We have assumed no wide variance in network workloads for the duration of the application. Studying the effects of possible jitter in file transfer times is outside the scope of this paper, but these effects should impact all our scheduling methods equally anyway.

Virtual Grids. In general, one can expect that using an unlimited number of the fastest machines in a cluster will lead to the lowest application makespans. Unfortunately, the number of nodes in a cluster is limited. In fact, the fastest clusters might not always be the biggest clusters. Furthermore, it may be best to use multiple clusters provided they are not too "far" from each other.

The above is exactly the sort of trade-offs that make scheduling difficult. The VG abstraction allows users the luxury of asking for a TightBag (that is sets of heterogeneous hosts that are "close"), with a parameter to determine what "close" means. The vgES will identify such a TightBag quickly, even in large-scale environments [10]. Our approach focuses on finding an appropriate TightBag for a given DAG. The size of the TightBag (in number of hosts) should be proportional to the widest portion of the DAG to allow maximum parallelism. For instance, for the Montage workflow described in Table 2, we can write the vgDL specification shown in Figure 4, which asks for a TightBag containing between 500 and 2633 hosts, where hosts have clock rates higher than 3Ghz. We choose 2633 as the upper bound on the number of hosts in the VG as this represents the widest portion of the Montage DAG. The [rank = Nodes] statement just means that a larger TightBag is preferable. (See [10] for all details regarding vgDL.) When the resource platform does not contain the number of resources we want (2633) for a TightBag, we can specify the willingness to accept fewer resources.

In our synthetic resource environment such a request returns a VG containing 924 hosts. If a sufficiently large TightBag cannot be found however, then more complex VG structures may be required. This is an interesting question and a possible answer is to use a LooseBag of TightBags (with as few TightBags as possible so that a sufficient number of nodes are acquired). In this case, sophisticated scheduling algorithms would indeed be necessary, especially for applications that tend to be data-intensive and for which scheduling of data transfers between TightBags must be judiciously chosen. Other efforts [29] in the VGrADS project are exploring the use of such resource structures and of non-greedy scheduling algorithms to achieve high performance. We argue that, although this may not be true today, when and if resource environments become abundant with good networking connectivity among many subsets of the available clusters, then for many relevant applications (but not all), the vgES would acquire reasonably sized TightBags on behalf of the user with high probability.

```
VG = TightBagOf(nodes) [500:2633]
[rank = Nodes] {
   nodes =   [ (Clock>=3000) ]
}
```

Figure 4: vgDL used for the Montage workflow

Top Hosts (Fastest). To show that using a VG is better than just picking the fastest hosts in the resource universe, we experiment with a subset of the resource universe that consists of the fastest 2633 hosts. We run our scheduling algorithms on this subset of the hosts.

5. Results

The main result from our experiments is that, regardless of DAG size, using the VG approach with a simple scheduling algorithm is preferable. We discuss below specific results for Montage and random DAGs. We compute a lower bound on application makespan by assuming that all tasks run on hosts as fast as the fastest available host and that all data transfers take place on network links as fast as the fastest network link available.

5.1. Montage

Figure 5 and Figure 6 show results for the Montage workflow using the MCP and the greedy algorithm. Results include the time to compute the schedule, the application makespan resulting from the schedule, the time to obtain a VG when applicable, and the total application turn-around time including all of the above.

The results in Figure 5 are for the actual Montage communication costs. The intermediate files generated by different stages ranged from 300 bytes to 4 megabytes, so communication costs were relatively low. The conclusion from these results is that running the greedy algorithm on a VG achieves the best application turn-around time overall (within 8% of the ideal lower bound), if not the best makespan. The best makespan is achieved when running MCP on the whole resource universe, but this makespan comes with a prohibitive scheduling cost. Running on Top Hosts (fastest) gives good performance (if not best) because communication costs are low. Interestingly, running the greedy algorithm on the whole resource universe still outperforms running MCP on the whole universe in spite of poor makespan since the time to compute the MCP schedule is so high.

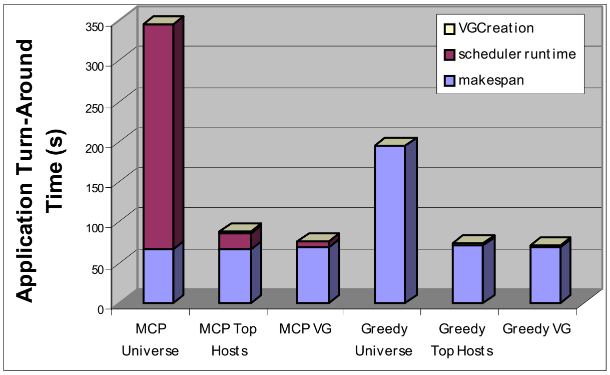

Figure 5: Running Montage workflow with actual Montage communication costs

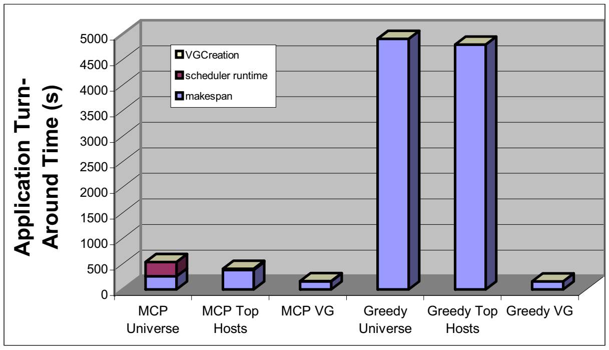

Figure 6: Running Montage workflow with equal communication and computation costs

Figure 6 shows similar results for a CCR value of 1, which is balanced communication and computation cost. Here, it is not enough to simply schedule tasks on the fastest machines as communication costs matter, and the benefits of using a VG are plain. Surprisingly, running the greedy algorithm on a VG produces a better makespan than running MCP on the resource universe. This is because MCP is just a heuristic with no guarantees. It makes greedy decisions based on the relations between tasks and the critical path, disregarding possibly harmful effects due to task dependencies. More sophisticated scheduling algorithms may or may not lead to better makespans in our experimental setting. At any rate, using a simple greedy scheduling algorithm is as effective once resources have been pre-selected.

Varying CCR. Figure 7 shows the ratio of Montage makespans as compared to running MCP on the universe, for increasing CCRs. One striking result is that when the CCR is increased, either algorithm running on the VG can construct schedule with much shorter makespans than the schedule MCP can construct on the whole resource universe.

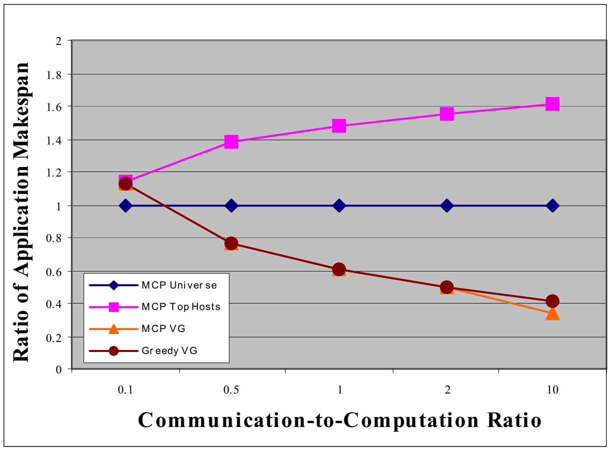

Figure 7: Ratio of Montage makespan compared to running MCP on universe while varying CCR

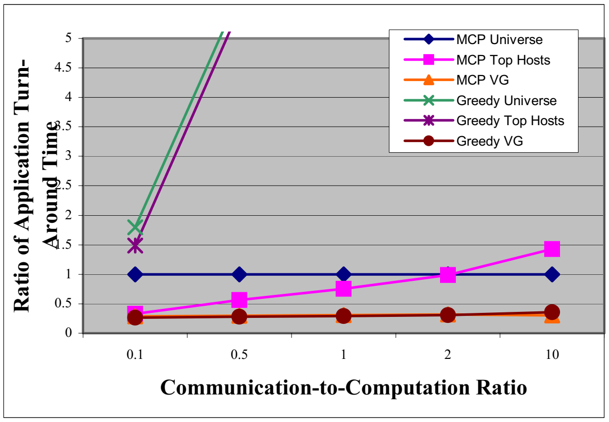

Figure 8: Ratio of Montage makespan compared to running MCP on universe while varying CCR

For most CCRs, when using the VG, no differences exist between using the greedy algorithm and MCP. Only when the CCR is very high do we notice a slight improvement in performance when MCP is used. The makespan for the greedy algorithm running on either the top hosts or the universe were 6 to 23 times longer than the MCP on universe makespan. We contend that this is not the case for many workflow applications.

We show Figure 8 to highlight the definite advantage of using the VG. When taking the scheduling time into consideration, using either algorithm running on the VG achieves application turn-around time less than 30% of the turn-around time needed to run MCP on the universe.

5.2. Random DAGs

We generate random DAGs according to the characteristics in Table 3. When varying a single parameter all other parameters take the default values shown in the table. In some cases the application turn-around time for running the greedy algorithm on the resource universe were so large that we left them out of the figures. Each data point is averaged over 10 random DAGs. The coefficients of variation for these samples were all within 3%, except for the case of running MCP on the universe, which ranged from 1% to 73%.

Varying DAG Sizes. As we vary the DAG sizes, we needed to vary the corresponding vgDLs to create different VGs for each DAG size (that is larger VGs for larger DAG widths). Expectedly, scheduling time for running MCP increases as the DAG sizes increased. However, because of the relative small sizes of the VGs compared to the universe, this increase was only marginal. Application makespan consists of the major bulk of the contribution of application makespan for MCP running on the universe. We also observed no significant makespan differences between running MCP on VG and running greedy on VG. Figure 9 shows the ratios of the application turn-around times compared to running greedy on VG. One can see that there is virtually no difference between running greedy or running MCP on VG in terms of turn-around time, especially with bigger DAGs. With smaller DAGs, because of smaller turn-around time, the difference between using MCP or greedy algorithm on the VG is magnified.

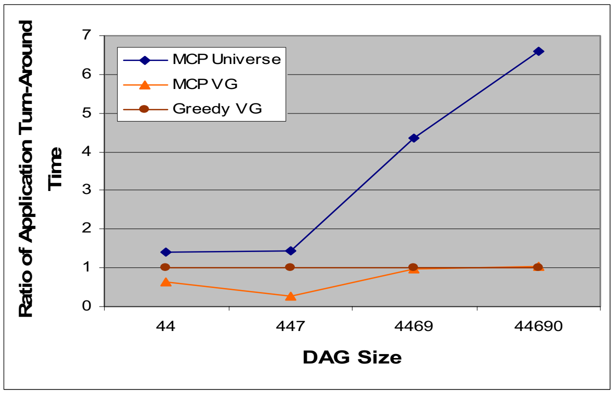

Figure 9: Varying DAG sizes for random DAGs

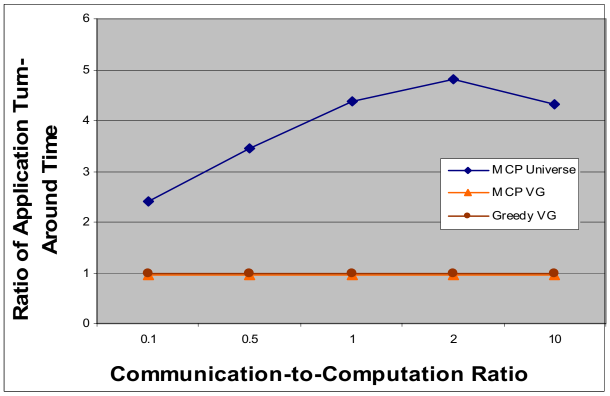

Figure 10: Varying CCR for random DAGs

Varying CCR. As with Montage, we wanted to investigate whether the greedy on VG approach would tolerate high-communication scenarios. Figure 10 shows that greedy on VG is within only 4% of results for MCP on VG for all CCR values. The performance of running greedy on the universe was between 16 and 62 times the application turn-around time for running greedy on VG.

Varying Parallelism. When the parallelism of a DAG (as defined in [2]) is 0, then the DAG is just a chain of tasks where each task depends on the previous task. Scheduling consists in finding the fastest host. When the parallelism is 1, all of the tasks can be run in parallel and scheduling consists in finding the fastest N hosts for each of the N tasks in the DAG.

Figure 11 shows results for varying DAG parallelisms. We see that at 0.5 or higher, running the greedy algorithm on the VG has comparable performance to running MCP on the VG. For parallelism of 0.8, running the greedy algorithm is actually preferable to running MCP due to MCP taking more time to compute the schedule because of the increased number of tasks at each level. However, we see the limitation of using the VG as a means for good performance when the parallelism is below 0.5. (A value of 0.5 implies that the number of tasks per stage is equivalent to the square root of the total number of tasks in the DAG.)

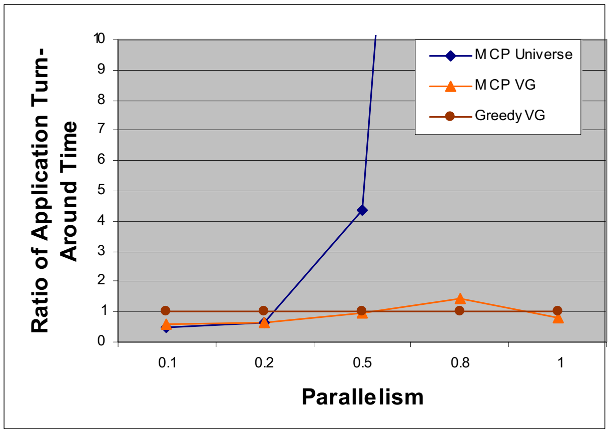

Figure 11: Varying parallelism for random DAGs

The poorer performance while running the greedy algorithm for less parallel DAGs is due to increased communication costs, or rather, the lack of opportune communications savings. Whereas MCP actively seeks to minimize communication costs by calculating the tradeoff between scheduling two tasks on the same host sequentially that would lead to longer computational time, but zero communication costs, the greedy algorithm would greedily schedule the two tasks on separate hosts whenever the second host becomes available. Of course, note that a minor modification of our greedy algorithm could alleviate this deficiency (e.g., always try to reuse a host that has been used before). Nevertheless, while the implication of Figure 11 is that when workflows are not highly parallel our approach is not effective, it is reasonable to expect that many applications will in fact have parallelism higher than 0.5 and thus not mandate anything more sophisticated than our greedy algorithm.

Varying Density. The density of a DAG determines the number of dependencies among the tasks. A density of 0.5 means that each task depends on 50% of the tasks in the previous level. Here again we found that scheduling on a VG greatly outperforms scheduling on the whole universe of resources. The application turn-around time for running MCP on the universe is 3 to 15 times more than running greedy on VG, depending on the density of the DAG. Figure 12 shows that running MCP on VG outperforms running greedy on VG in most cases. For densities higher than 0.2 the difference is below 4%, but it is up to 18% for a density of 0.1.

MCP was able to achieve better application performance as the number of dependencies decreased because it was able to schedule some of the tasks on the same hosts as their parents, particularly tasks that have one parent task. As the number of dependencies decreases, unlike the greedy algorithm, MCP can increasingly optimize the communication costs.

Varying Regularity. Regularity quantifies the distribution of the number of tasks per level in the DAG. A regularity of 1 means that all levels have the same number of tasks. The lower the granularity the higher the variance in the numbers of tasks per level. Here again, using a VG is preferable to using the whole resource universe. Figure 13 shows that with the appropriate VG, running a greedy algorithm can create a schedule with makespans more than ten times shorter than running MCP on the universe when the DAG is highly irregular. Performance is more than fifty times better (not shown) when compared to greedy running on the whole universe of resources. We see that for any regularity type, the greedy algorithm running on the VG performs within 3% of MCP running on the VG.

Varying Mean Computational Cost. Varying the mean computational cost makes very little difference between running the greedy algorithm or running MCP on the VG, as seen in Figure 14. Here again, using a VG greatly outperforms using the whole resource universe.

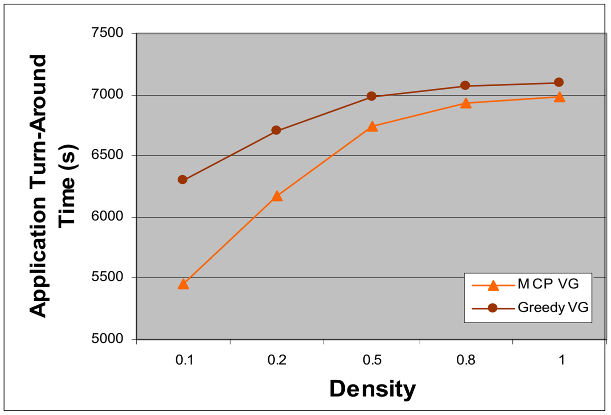

Figure 12: Varying density for random DAGs

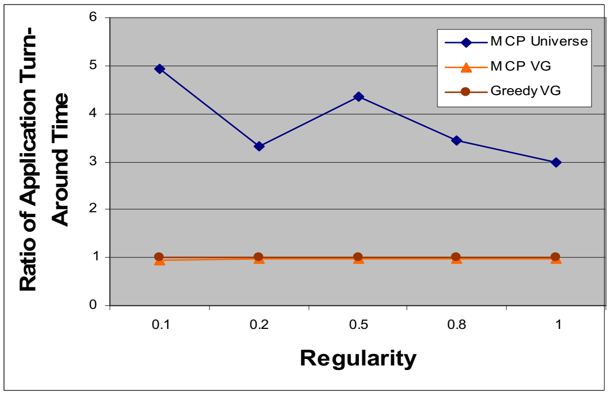

Figure 13: Varying Regularity of number of tasks per stage in the DAG

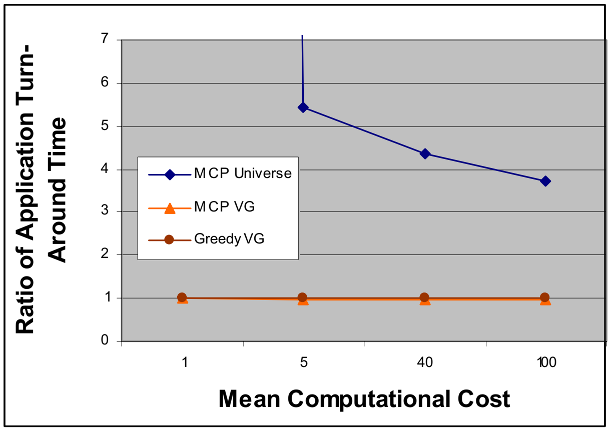

Figure 14: Varying mean computational costs for random DAGs

5.3. Summary

Montage DAG. Our results show that under various CCRs the greedy algorithm on a VG achieves the comparable or better turn-around times than using more sophisticated algorithms such as MCP.

Random DAGs. In almost all of the scenarios we tested, the greedy algorithm running on the VG perform within 4% of MCP running on the VG, both of which greatly outperforms either running on the resource universe. The only limitations we found for using the greedy algorithm on the VG occurs when the DAG is very sparse, either due to low parallelism or low number of dependencies among the tasks.

6. Summary and Impact

In this study, we have addressed the question of whether sophisticated DAG scheduling algorithms are needed to schedule workflows on grid platforms. We have considered two scheduling algorithms: (i) MCP, a popular DAG scheduling algorithm that accounts for node and edge weights in the DAG and for the characteristics of the heterogeneous underlying resources in terms of compute power and network connectivity; and (ii) a greedy algorithm that accounts only for task dependencies and is oblivious to node and edge weights and to resource capabilities. We have used simulation to demonstrate that, by using the virtual grid abstraction, the greedy algorithm leads to performance that is either better or within a few percents of that of MCP in many cases that are relevant to practice.

The above result was confirmed for DAGs from a real-world application as well as for random DAGs, and holds even for DAGs that exhibit high CCR ratios. We found that our approach does not perform well when the DAGs are sparse, either because of small amount of parallelism or small number of dependencies. We contend that in practice DAGs from real-world scientific workflows are rarely so sparse that our approach would be ineffective.

The impact of our finding is clear for scheduling grid workflows in practice: rather than investing time in developing and implementing sophisticated scheduling algorithms, one should *initially* implement simplistic algorithms but perform fast and appropriate resource pre-selection. The VG abstraction defined and prototyped in [9, 10, 19] provides the necessary resource pre-selection capabilities. Given that most existing grid workflow frameworks already implement simple scheduling algorithms similar to our greedy algorithm, these frameworks could just integrate and use the VG abstraction directly to ensure that many applications experience good performance. As discusses in Section 4.4, there are cases in which our approach will not suffice. If a sufficiently large TightBag cannot be found, then more complex VG structures would be required and mandate more sophisticated scheduling algorithms, especially for data-intensive applications. Other efforts in the VGrADS project [29] consider more complex VG structures and scheduling algorithms. Nevertheless, we argue that in (future) resource-rich environments, with high bandwidth between many clusters, finding a reasonably large TightBag should be possible with high probability for many relevant applications.

Another direction for future work is to explore the impact of the resource management policies. We have assumed that all resources are instantly available when needed and dedicated once acquired. However, in real-world grid platforms resource acquisitions may be delayed, denied, or revoked. Note that common sense suggests that in such a complex and time-varying environment, a simple greedy algorithm such as the one we used in this study should be more robust than and thus preferable to a more complex scheduling heuristic such as MCP.

7. Acknowledgements

The authors and research described here are supported in part by the National Science Foundation under awards NSF Cooperative Agreement ANI-0225642 (OptIPuter), NSF CCR-0331645 (VGrADS), NSF ACI-0305390, and NSF Research Infrastructure Grant EIA-0303622. Support from the UCSD Center for Networked Systems, BigBangwidth, and Fujitsu is also gratefully acknowledged.

8. References

1. Ullman, J., *NP-complete scheduling problems.* Journal of Computer and System Sciences, 1975. **10**: p. 434-439.
2. Kwok, Y.-K. and I. Ahmad, *Benchmarking and Comparison of the Task Graph Scheduling Algorithms.* Journal of Parallel and Distributed Computing, 1999. **59**(3): p. 381-422.
3. http://www.extreme.indiana.edu/swf-survey/.
4. Barish, B. and R. Weiss, *Ligo and detection of gravitational waves.* Physics Today, 1999. **52**(10).
5. Deelman, E., et al. *GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists*. in *Proceedings of the IEEE High Performance Distributed Computing*. 2002.
6. Hastings, S., et al. *Image Processing on the Grid: a Toolkit for Building Grid-enabled Image Processing Applications*. in *Proceedings of the International Symposium on Cluster Computing and the Grid*. 2003.
7. *Computational Grids: Blueprint for a New Computing Infrastructure*. 2nd ed, ed. C. Kesselman. 2003: M Kaufman Publishers, Inc.
8. Yu, J. and R. Buyya, *A Taxonomy of Scientific Workflow Systems for Grid Computing.* ACM SIGMOD Record, 2005. **34**(3): p. 44-49.
9. Chien, A., et al., *The Virtual Grid Descriptive Language: vgDL*. 2004, UCSD Technical Report CS2005-0817.
10. Kee, Y.-S., et al. *Efficient Resource Description and High Quality Selection for Virtual Grids*. in *Proceedings of the IEEE Conference on Cluster Computing and the Grid*. 2005.
11. http://rocks.npaci.edu/Rocks/.
12. http://www.globus.org.
13. http://www.optiputer.net/.
14. Deelman, E., et al. *Pegasus: Mapping Scientific Workflows onto the Grid*. in *Across Grids Conference 2004*. 2004. Nicosia, Cyprus.
15. Deelman, E., et al., *Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems.* Submitted to Scientific Programming, 2005.
16. Ibarra, O.H. and C.E. Kim, *Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors.* Journal of the ACM, 1977. **24**(2): p. 280-289.
17. Jacob, J.C., et al. *The Montage Architecture for Grid-Enabled Science Processing of Large, Distributed Datasets*. in *Proceedings of the Earth Science Technology Conference (ESTC)*. 2004.
18. Berriman, G.B., et al. *Montage: a Grid Enabled Engine for Delivering Custom Science-Grade Image Mosaics on Demand*. in *Proceedings of the SPIE Conference on Astronomical Telescopes and Instrumentation*. 2004.
19. http://vgrads.rice.edu/.
20. Wolski, R., N. Spring, and J. Hayes, *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing.* Journal of Future Generation Computing Systems, 1998. **15**(5-6): p. 757-768.
21. *Globus Monitoring and Discovery System (MDS).* http://www-unix.globus.org/toolkit/mds/.
22. Kee, Y.-S., H. Casanova, and A. Chien. *Realistic Modeling and Synthesis of Resources for Computational Grids*. in *Proceedings of the ACM Conference on High Performance Networking and Computing*. 2004.
23. Singh, G., C. Kesselman, and E. Deelman, *Optimizing Grid-Based Workflow Execution*. 2005, University of Southern California 05-851 PDF.
24. Wu, M.-Y. and D.D. Gajski, *Hypertool: A Programming Aid for Message-Passing Systems.* IEEE Transactions on Parallel and Distributed Systems, 1990. **1**(3): p. 330-343.
25. http://www.rocksclusters.org/rocks-register/.
26. Tangmunarunkit, H., R. Govindan, and S. Jamin. *Network TOpology Generators: Degree-Based vs. Structure*. in *SIGCOMM*. 2002.
27. Li, L., et al. *A First-Principles Approach to Understanding the Internet's Router-level Topology*. in *SIGCOMM*. 2004.
28. Medina, A., et al. *BRITE: An Approach to Universal Topology Generation*. in *Proceedings of MASCOTS '01*. 2001.
29. Zhang, Y., et al., *Scalable Grid Application Scheduling via Decoupled Resource Selection and Scheduling*. 2005, Rice University TR06-871.