Application-level Resource Provisioning on the Grid

Gurmeet Singh, Carl Kesselman, Ewa Deelman
Information Sciences Institute, USC
{gurmeet, carl, deelman}@isi.edu

Abstract

In this paper, we present algorithms for Grid resource provisioning that employ agreement-based resource management. These algorithms allow user-level resource allocation and scheduling of applications that are structured as a precedence-constrained set of tasks. We present a provisioning model where the resource availability in the Grid can be enumerated as a set of slots. A slot is defined as a number of processors available from a certain start time for a certain duration at a certain cost. Using a cost model that combines the cost of resource allocation and the expected application runtime, we evaluate the performance of the Min-Min and of the Genetic algorithm (GA)-based heuristics for a range of synthetic applications. We show that the GA paired with a list scheduling algorithm can obtain significantly better solutions than the Min-Min heuristic alone.

1. Introduction

Due to the autonomous and shared nature of Grid resources, the quality of service available to the user is often best effort in nature. The resource owners are interested in maximizing the utilization or the throughput of their resource while the users are mostly interested in optimizing their application performance. Due to this best effort quality of service, optimizing application performance on Grids has traditionally relied on either queue wait time predictions [1] in case of space shared systems or load predictions [2] in case of time shared systems. However, the resulting performance is only as good as the quality of the prediction. While there can be many metrics to judge the performance of the application, in this paper, we use the expected runtime as the performance metric for the application. In our study we consider workflow-based applications [3], which consist of multiple tasks with dependencies defined between them. We optimize the runtime of the workflows by provisioning resources ahead of the execution.

We use the term resource provisioning to imply creating a contract between the user and the resource owner specifying that a certain resource would be made available to the user for a certain timeframe. Resource provisioning for applications has several advantages over best effort execution. First and most importantly, it allows the user or workflow manager [4] to control the scheduling and execution of the application on the provisioned resources. This in turn enables the estimation of the application performance prior to its execution. Second, it allows the user or workflow manager to explore the space of resources to be provisioned and optimize for performance without worrying about external factors such as the workload of the resource or the policies of the resource provider. Third, it enables the adaptation of the application based on the provisioned resources in order to achieve a certain level of performance. Fourth, it allows the execution of applications that require co-allocation of resources or have other constraints on the resource requirements. Last, due to the deterministic nature of the resource availability resulting from the contractual provisioning process, mechanisms for adapting to the otherwise dynamic nature of the Grid are not essential.

However, there are likely to be costs associated with resource provisioning that justify provisioning from resource owners perspective. These costs might include an element that represents the charge for the normal usage of the resource (e.g. number of service units per processor second), but it might also include an additional element that would account for factors such as the loss of utilization for the resource provider or the increased queue wait time of best effort jobs [5, 6] due to the provisioned resources. The problem we explore in this paper is to identify from the user's perspective a set of resources to be provisioned in order to optimize a parameterized cost metric that includes the cost of provisioned resources and the application runtime (also referred to as makespan in this paper).

The resource availability in this paper is modeled as a set of resource slots. A slot represents the availability of a certain number of processors starting at a certain time for a certain duration at a certain cost from a certain resource. While the model can be easily extended to other resource types such as network and storage resources, in this paper we focus our attention to compute resources. The set of available slots at any

time can be determined by querying the Grid resources. Alternatively, the resources can publish this slot information periodically in Grid information services. This resource model is a significant departure from earlier work on application scheduling on the Grid which either considered a resource to be available in its entirety or not at all [7-9] or assumed some form of centralized control [10]. We think that given the large-scale, shared, and autonomous nature of the Grid resources, modeling the resources availability in discrete units that can be provisioned by the user is a more appropriate approach.

Much work has recently focused on using agreement-based resource management [11, 12] in order to meet the challenges of heterogeneous and autonomous resources. Using this model, the resource consumer or a broker enters into a contractual agreement with the resource provider about the availability of certain resources for a certain timeframe at a certain cost. There is ongoing work in the Grid Resource Allocation and Agreement Protocol Working Group (GRAAP-WG) [13] of the Global Grid Forum on formalizing the mechanisms and protocols for creating resource agreements. The ability to provision resources in the form of advance reservations [14] is already present in most resource management systems such as Maui [15], PBSPro [16], LSF [17] etc. Current middleware tools such as Condor Glidein [18] allow the user to execute an application schedule on the provisioned resources without any support from the resource provider. What is required is a mechanism that allows the user to discover the set of provisionable resources and a policy for pricing these resources. In the future, it is anticipated that resource provisioning would be the rule rather than the exception [11].

The rest of the paper is structured as follows. Section 2 describes the resource provisioning model that describes the Grid resource availability in the form of a set of slots and formulates the resource provisioning problem. Section 3 describes a retrofitted Min-Min heuristic and a genetic algorithm based heuristic for creating a resource plan for the application that optimizes a parameterized cost metric. The performance of these heuristics for a given resource availability on a number of artificially generated applications is evaluated in Section 4. The results are discussed in Section 5. Related work is presented in Section 6 followed by conclusions in Section 7.

2. Resource Provisioning Model

The Grid is comprised of R autonomous compute resources or sites. Each site r is a cluster of N_r homogeneous processors. Each site r can be queried for the possible start times and the associated cost for provisioning n processors for d duration. The query and the response are represented as:

$$E_r(n,d) = \{<s_1,n,\ d,\ c_1,f_1,r>,\ \ldots,\ <s_i,n,d,c_i,f_i,r>\ldots.\} \quad \ldots(1)$$

Note that this is no different than assuming the ability to query the possible start times of a task or a resource reservation requiring n processors for d duration from site r. Each tuple $<s_i, n, d, c_i, f_i, r>$ in the response set represents the availability of n processors for d duration starting at time s_i with a multiplicative cost of c_i and a fixed cost of f_i from the site r. Each tuple is called a slot. The total cost of using the slot is defined as $(n*d*c_i + f_i)$.

The multiplicative cost is used to specify the usage charge that is related to the amount of resource provisioned. For example, it might represent service units per processor second or dollars per processor second. This multiplicative cost might also depend on the start time similar to the existence of express queues on some Grid sites that have a faster response time but charge a higher number of service units per processor second.

The fixed cost might be used to represent the lost resource utilization or the increased response time [5] due to the fragmentation of resources resulting from allowing fixed resource reservations. It might also be used to discourage users from provisioning too many slots and instead provision resources in bigger chunks resulting in less bookkeeping for the resource provider. While a single cost factor per slot would have also served the purpose, having two separate cost factors provides better semantic description of the model.

The set of all the slots available from the site r, denoted by $E_r(.)$, can be constructed as

$$E_r(.) = \bigcup_{1\le n\le N_r} \bigcup_{1\le d\le D_r} E_r(n,d) \quad .....(2)$$

where D_r = max duration allowed at site r

However, it would overload the Grid site if every user queried for every possible value of n and d. It would be much more efficient if a single query can return the set of available slots on the site. Thus, we assume that each site r can be queried for the set of slots available on the site.

$$E_r(.) = \{<s_1,n_1,\ d_1,\ c_1,f_1,r>,\ldots\ldots,\ <s_i,n_i,d_i,c_i,f_i,r>\ldots.\} \quad \ldots(3)$$

We also assume that the sites only return the set of non-overlapping slots i.e. no two slots represent the same underlying resource and each slot can be provisioned independent of any other slot.

Alternatively, this set $E_r(.)$ can also be published using a Grid information service such as MDS [19].

A global set of resource slots: $\hat{A}$, is the set of all the resource slots available in the Grid. It can be constructed by querying each site (equation(3)) as shown below

$$\hat{A} = \bigcup_{1 \le r \le R} E_r(.) \quad ....(4)$$

The application model in this paper is a workflow consisting of compute and data transfer tasks with precedence constraints between them. The data transfer tasks are used to transfer data between the compute tasks when they are scheduled on different sites. We assume the processing capacity (processor speed) and the network latency between all the sites to be homogeneous and contention for network resources is ignored. Provisioning of network resources is not considered in these studies though it would be straightforward to include it in the set of provisionable resources. The runtime of each compute task on each of the Grid sites is known and as a result of the previous assumptions, the run time of a data transfer task is also known. The makespan is defined as the maximum completion time of any task in the application.

We define the resource provisioning problem as the problem of finding a subset $\hat{a}$ of $\hat{A}$ (or in other words, an element of the power set of $\hat{A}$, $P(\hat{A})$) and a schedule $\hat{s}$ of the application over $\hat{a}$ such that the following objective function is minimized.

$$\min_{\hat{a} \in P(\hat{A})} \alpha.AC(\hat{a}) + (1-\alpha).SC(\hat{s}) \quad .....(5)$$

$$where, AC(\hat{a}) = \sum_{<..> \in \hat{a}} (n_i * d_i * c_i + f_i)$$

$$SC(\hat{s}) = \begin{cases} \psi * makespan(\hat{s}), if\ \hat{s}\ is\ complete \\ \infty, otherwise \end{cases}$$

$$0 \le \alpha \le 1, \quad 0 < \psi$$

This objective function is also called the total cost in the rest of this paper. This cost metric tries to minimize the cost of the provisioned resources while optimizing the application performance. Any subset $\hat{a}$ of $\hat{A}$ is called a resource plan. The term $AC(\hat{a})$ in the objective function is called the allocation cost of the resource plan $\hat{a}$ and denotes the total cost of all the slots included in $\hat{a}$. The term $SC(\hat{s})$ is called the scheduling cost and is used to measure the application performance. Examples of such measures can be the makespan, the reliability or the lateness of the application past a given deadline.

While, in general the scheduling cost can be used to represent any performance measure that is a function of the application schedule, in this paper, we assume the makespan as the scheduling cost ($\Psi = 1$). The term Ψ indicates the importance of the makespan in cost terms (service units per second or dollars per second). It is specified by the user and ensures that the units in equation(5) are consistent. Depending on the scheduling policy used and the resource plan $\hat{a}$ under consideration, it may not be possible to schedule the entire application over $\hat{a}$, if for example, $\hat{a}$ contains too few slots. In this case, the scheduling cost is considered infinite. Thus, the cost metric tries to find resource plans that can schedule the entire application.

The term α allows the user to specify the relative importance of the allocation cost and the scheduling cost. This model can also specify a budget and/or a deadline for the application by introducing appropriate constraints. Given the optimization problem described in equation(5), the number of possible subset of $\hat{A}$ is $2^{|\hat{A}|}$ and thus it is not possible to exhaustively examine all subsets in order to find the optimal one. In the next section, we present two heuristics that attempt to find the optimal subset $\hat{a}$ of $\hat{A}$ and a feasible schedule $\hat{s}$ of the application on $\hat{a}$ in order to minimize the total cost.

3. Algorithms for resource provisioning

While we would like to find the schedule $\hat{s}$, that minimizes the makespan of the application over the resource plan $\hat{a}$, finding such an optimal schedule is a NP-Hard problem [8]. In the case of the Min-Min heuristic, the resource plan $\hat{a}$ and the schedule $\hat{s}$ are created together. In the case of the GA heuristic, the GA is used to create the resource plan $\hat{a}$, while a list scheduling algorithm is used to create the schedule $\hat{s}$.

Min-Min heuristic

The Min-Min heuristic (Figure 1) maintains a set of allocated slots and a partial schedule of the application. At each iteration, it chooses to schedule a ready task (a task whose parents have already been scheduled) on a slot that leads to the minimum increase in the total cost. This heuristic creates the resource plan $\hat{a}$ and the schedule $\hat{s}$ incrementally. The total cost of scheduling a ready task on a slot in step 7 of the algorithm can be computed as shown in Figure 2. This heuristic is an extension of the Min-Min algorithm used for workflow scheduling in [8, 9]. However, there are a few differences. First, it works to minimize the total cost instead of only the makespan. Second, it operates on the set of ready tasks instead of the set of tasks at a particular depth in the workflow.

Due to the two min operations in step 9 and 11 of the heuristic, it is called the Min-Min heuristic. Note

that the heuristic will prefer to schedule tasks on the already allocated slots since it would lead to a lower allocation cost at each step. The heuristics does a local optimization for each task. It may not be possible to schedule a task on a slot if there isn't sufficient overlap between the runtime of the slot and the possible runtime of the task based on the completion times of the parent tasks. In addition, due to the non backtracking nature of the heuristic, it might not be able to schedule the whole application on the given set of slots $\hat{A}$, if at any stage; there is no slot in $\hat{A}$ that can schedule any task in the current set of ready tasks irrespective of the cost. If it happens, the heuristic stops at that point and returns an incomplete schedule.

1. Initialize the set of allocated slot A_S as empty.
2. While the complete application is not scheduled
3. Let V_S = the set of scheduled tasks
4. Let V_R = the set of un scheduled ready tasks
5. For each $v \in V_R$
6. For each slot $a \in \hat{A}$
7. Let T_v^a = total cost if v is scheduled on a
8. End-for
9. Let T_v^x be the min cost over all slots for v.
10. End-for
11. Let T_y^x be the min over all v for V_R.
12. Schedule task y on slot x. Add x to A_S if it does not belong to A_S. Remove y from V_R and add to V_S.
13. Update V_R by adding any child task of y that might have become ready now.
14. End-While
15. Return A_S and the application schedule on A_S.

Figure 1. The Min-Min heuristic

Let $A_S' = A_S \cup a$.
Schedule the task v on the slot a on a processor that will lead to its minimum completion time.
$V_S' = V_S \cup v$
Total cost of scheduling v on slot a =
$\alpha.AC(A_S') + (1-\alpha).SC(\text{schedule of } V_S')$

Figure 2. Computing the total cost in step 7 of the Min-Min heuristic.

The time complexity of the Min-Min algorithm is $O(|V|^2.|\hat{A}|.n_A)$ where $|V|$ is the number of tasks in the application, $|\hat{A}|$ is the cardinality of $\hat{A}$ and n_A is the maximum number of processors in any slot in $\hat{A}$.

Genetic Algorithm (GA)

In the genetic algorithm (called GA in the rest of this paper), we create an initial population of certain number of randomly generated unique subsets of $\hat{A}$. For each individual in the population, a schedule of the application is created on the slots in the individual using the list scheduling algorithm in Figure 3.

1. Create a topologically ordered list of tasks in the application where parent tasks precede the child tasks.
2. While list not empty
3. Remove the first task from the list and schedule it on a slot that leads to the minimum finish time of the task.
4. End-While

Figure 3. A list scheduling algorithm.

The time complexity of the scheduling algorithm is $O(|V|.|\hat{A}|.n_A)$ where $|V|$ is the number of tasks in the application, $|\hat{A}|$ is the cardinality of $\hat{A}$ and n_A is the maximum number of processors in any slot in $\hat{A}$. It is possible that this algorithm might not schedule any task on a slot in an individual in the population. The total cost of the individual is computed only based on the slots that have at least one task scheduled on them. Such a slot is called a scheduled slot. The total cost of the individual is computing using equation(5) with the allocation cost determined by the scheduled slots and the scheduling cost by the makespan of the schedule. We do not prune the unscheduled slots from the individual since they might become useful later as a result of the crossover operations.

The GA goes through a number of iterations. During an iteration, each individual is mated with another individual and creates two offsprings. This is called a crossover. The first offspring inherits slots that exists only either in the first parent or the second (similar to the XOR binary operation). The second offspring inherits slots that are either present in both the parents or in neither of them (similar to the XNOR operation). This doubles the population size. The selection of the crossover operators was motivated by the observation that they lead to a greater number of unique individuals in the population. However, there still might be duplicates in the population as a result of the crossover. After removing the duplicates, we compute the total cost of each new member of the population and reduce the population to the previous size by retaining the least total cost individuals in the population. The algorithm stops after certain number of iterations and the solution selected is the individual with the least total cost in the current population.

4. Evaluation

We evaluate the performance of the Min-Min and the GA heuristic by comparing the total cost of the solution produced by each of them on a given set of global slots, $\hat{A}$ and a given application. While we use a number of synthetically generated applications, the global set of slots, $\hat{A}$ remains the same (we change the granularity of slots, but it still represents the same resource ensemble). We have implemented a parametric task graph generator that can generate applications with a given total number of tasks and a given depth. The number of tasks at each depth is determined randomly so that the total number of tasks equals the required number. Each compute task has two predecessors and the runtime of the task is uniformly distributed with an average of 100 seconds. The runtime of the data transfer tasks is also uniformly distributed with an average of 100/CCR (computation to communication ratio). All the compute tasks in the application are serial (uniprocessor) tasks.

The set of global slots, $\hat{A}$ is created by simulating a Grid composed of 4 sites where each is a cluster of 10, 20, 30 and 40 processors respectively. The background traffic on the sites is created using a Poisson distribution. The load generator creates tasks with an exponentially distributed runtime (mean 500 seconds) and an exponentially distributed interarrival time (mean 333 seconds). The number of processors required per task in the background load is randomly distributed between 1 and the maximum available on the site. The sites implement a First Come First Serve (FCFS) policy. At any time the sites can be queried for the set of free slots ($E_r(.)$). Upon receiving the query, the site scheduler computes the schedule of the currently running and the queued tasks (using FCFS) and advertises the free slots in the schedule that is the difference between the scheduled and the total area as shown in Figure 4. Due to this the slot runtimes are also exponentially distributed with a mean of 500 second. This ensures that the tasks in the application with average runtime of 100 seconds can easily fit within the slots.

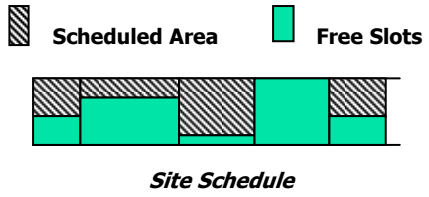

Figure 4. Site schedule and free slots.

We query the sites for free slots when the application is submitted and create the global set of slots, $\hat{A}$. The total number of slots in $\hat{A}$ is 214. Thus it is not possible to do an exhaustive search to find the optimal subset. Initially all the slots have a multiplicative cost c_i of 1 and a fixed cost f_i of 0 implying that the allocation cost is based only on the area of the slot irrespective of the start time of the slot.

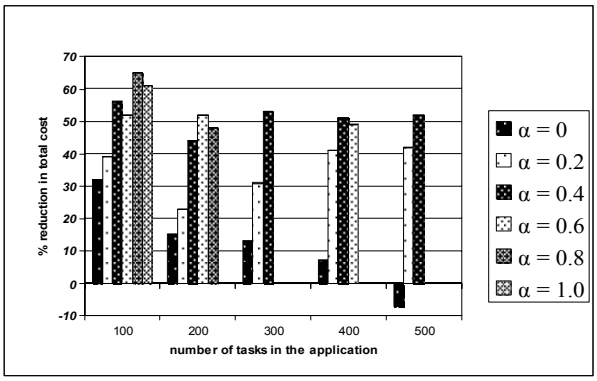

Figure 5. percent reduction in total cost using the GA heuristic over the Min-Min heuristic.

In the first experiment, we compare the total cost of the resource plan $\hat{a}$ created by the Min-Min and the GA heuristic. The size of the population size in GA is 50 and the number of iterations is fixed at 20. We create 5 applications containing 100, 200, 300, 400, and 500 tasks respectively and the depth is set of square root of n where n is the number of tasks in the application. Figure 5 shows the percent reduction in total cost of GA over the Min-Min heuristic for these applications when the value of α is varied from 0 to 1 in increments of 0.1. GA in general achieves a 25-30% reduction in the total cost over using the Min-Min heuristic. In 30% of instances, Min-Min could not complete the schedule (denoted by missing bars) as discussed in Section 3.

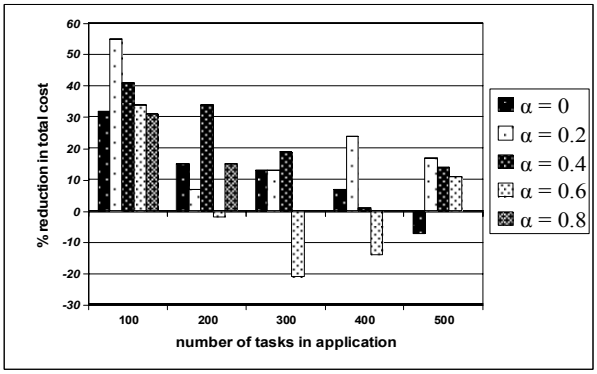

Figure 6. Slots with c_i differential.

In the next experiment (Figure 6), the multiplicative cost factor c_i is randomly generated between 0 and 2 for all the slots in the global set $\hat{A}$. This represents a more realistic situation where the resource cost is expected to vary based on the desired start time (e.g. day time costs more than night time)

and the current workload. While the GA still outperforms Min-Min in most of the cases, the difference is less pronounced and in some cases, the Min-Min performs better. The reason is that while the Min-Min does an explicit optimization involving the allocation cost, the GA depends on the randomly generated population and the crossover operators to do the allocation cost optimization (the list scheduling only tries to minimize the makespan).

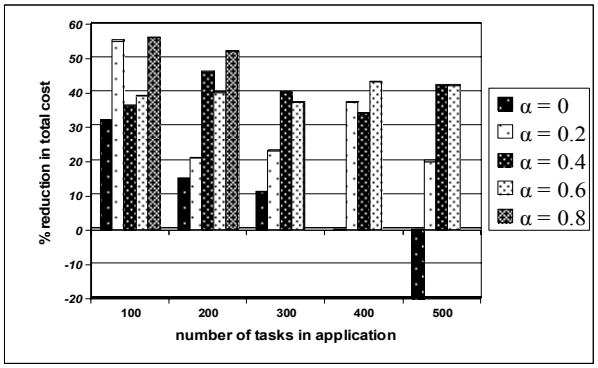

Figure 7. GA with modified list scheduling and differential c_i.

In light of the above observations, we repeated the same experiment after modifying the list scheduling algorithm to operate with the total cost as the minimization metric instead of the finish time in step 3 of the algorithm. Comparing the results in Figure 7 with the previous results in Figure 6 shows that using the modified list scheduling algorithm with the GA improves its performance significantly. In this case, the GA performs better because the modified list scheduling also does optimization of the allocation cost along with the makespan.

In the next experiment, we change the slot granularity i.e. each rectangular piece of free area as shown in Figure 4 is broken down into multiple slots with one processor each (c_i = 1). At this finer granularity the global set of slots, $\hat{A}$, contains 2981 slots for representing the same resource ensemble.

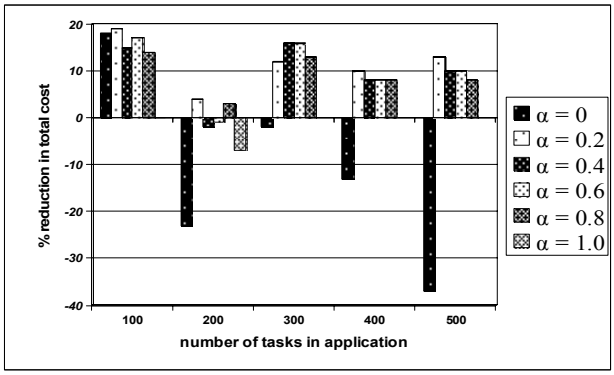

Figure 8. Slots at a finer granularity.

GA is paired with the unmodified list scheduling heuristic in order to make the results comparable to Figure 5. Figure 8 shows that with the slots being at a finer granularity, the percent reduction in total cost with GA decrease to about 5-10% on average and the Min-Min is able to find a complete solution in a large percentage of instances. One of the reasons for the bad performance of Min-Min in Figure 5 was that due to the higher allocation costs associated with the coarser slots, the Min-Min tended to pick up smaller slots at the expense of a higher makespan or an incomplete schedule.

The better performance of the GA heuristic comes at the expense of a longer runtime. While the Min-Min heuristic on average took less than half the time taken by GA for the coarse granularity experiments, it was an order of magnitude faster than GA for the fine granularity ones. However, this longer time to find a better solution would be more than offset by the reduction in total cost due to the GA heuristic. In addition, GA can also be used for deadline and budget constrained optimization by rejecting solutions where the makespan exceeds the deadline or the total cost exceeds the given budget.

We also experimented with changing the population size and the number of iterations of the GA heuristic. The population size seems to have a greater impact on the performance of the heuristic than the number of iterations for the crossover operators that we have used in these experiments.

5. Discussion

While we have ignored contention for networking resources while developing the application schedule on the provisioned resources, it does not take away from the generality of the results since we can treat the data transfer similarly to a compute task for which the networking resources can be provisioned. The application tasks in this case can be classified as either compute or transfer, which require different types of resources. Provisioning interfaces for network resources in the form of protocols such as RSVP [20] are already in existence and widely used to provide guaranteed quality of service to multimedia-type of applications.

The slots in the experiments were created using a simple mechanism i.e. create a FCFS schedule and take the difference between the scheduled area and the total availability. This was done to reduce the computational complexity of the experiments. A Grid site in practice might use a more complicated mechanism to create this set of slots. However, the presented heuristics are not affected by how the slots are created. Also the

computational complexity of the algorithms depends on the number of available slots irrespective of the number of resources in the Grid.

While, we conducted experiments using a single set of slots representing a resource ensemble with various granularities, in practice, the set of slots available on the Grid sites would change over time (even while the slot selection is being done) due to factors such as the early or late completion of running or queued tasks, changing workload and other factors. In our future work, we plan to study algorithms that can do reprovisioning of resources in the face of changing resource availability for optimizing the total cost and dealing with failures. However, it would require a more sophisticated resource model that includes the cost for changing or terminating a provisioning agreement.

6. Related Work

While there has been considerable work in application-level scheduling on Grids [8, 9], there has been little work on resource provisioning for applications. In these works, provisioning is done implicitly for each task as a by-product of scheduling and the cost of allocation is never considered. In application scheduling on time-shared systems [2] there is no explicit provisioning operation. There has also been some work on deadline and budget constrained scheduling of applications on Grids [21]. However, the application model is a bag of tasks and the Grid resources are represented in terms of a processing capacity that is available immediately at a certain cost. Our resource model is richer in that each Grid site can provide resources in discrete quantities having a spatial and temporal dimension with associated costs. Recently, agreement-based resource management [11] has been suggested for meeting the challenges of heterogeneous and autonomous resources in the Grid. However, the focus has been on developing the protocols and the framework that allow the users to specify their requirements and create agreements. While these are essential first steps, we need algorithms that can determine the set of resources to be provisioned for an application using this resource management framework.

Resource provisioning is conceptually similar to overlay metacomputing [22] that creates a user-level aggregation of distributed computing systems. This aggregation is also sometimes called as a Virtual resource [23] or a Virtual Grid [24]. A user-level metacomputer is created in [22] by submitting placeholders to the various resource queues. A placeholder can be considered similar to a resource slot with an uncertain start time. However, the focus of that work is on maximizing the throughput of a user-level queue of jobs rather than optimizing the performance of an application. A system for executing a workflow on such a group of placeholders is described in [25]. However, the emphasis is on the mechanisms for specifying and executing workflows. Furthermore, it tries to obtain a minimal schedule of the workflow on the set of placeholders but how the resource requirement (e.g. number of processors and duration) of the placeholders is determined is not clear.

In [23], the user requests a set of resources to be co-allocated. The resource management system queries the availability of the Grid resources and gets a set of time slots that is similar to our approach. The combination of all possible slots is created in order to find the best one. As we have mentioned in Section 2, this might not be a feasible approach if the number of slots is large. In [24], the authors use relational database technology to find a set of resources that satisfy the user constraints and are available right away. Instead of assuming that each Grid resource or site is either available immediately for the user or not, we assume that the availability of a Grid resource can be discretized in the form of a set of resource slots. This provides a richer resource model. In both [23] and [24], the user provides a high level resource requirement whereas in our case the input from the user is an application specification. In our work, we also solve the application scheduling problem.

7. Conclusion and Future Directions

In this paper, we described and proposed solutions to the resource provisioning problem for applications structured as precedence-constrained set of tasks. We described a model for resource provisioning and compared the performance of two algorithms that can operate in this model. The results show that a genetic algorithm paired with a simple list scheduling algorithm can obtain significantly better solutions than the Min-Min heuristic. While the problem in this paper has been studied in the context of a Grid, the problem of provisioning resources to meet an anticipated demand is more general and is seen in areas such as Material Requirements Planning (MRP) [26] in operations research.

In the introduction, we claimed that resource provisioning is likely to result in better performance than the best effort approach due to the deterministic nature of provisioned resources and user based application scheduling. In the future, we intend to substantiate this claim with a thorough study comparing the application performance with and without resource provisioning. We also plan to study

provisioning policies for resource providers that allow both provisioned and best effort quality of service.

8. Acknowledgements

This work is supported by NSF under Cooperative Agreement number CCR-0331645 (VGrADS).

9. References

1. Downey, A.B., *Using Queue Time Predictions for Processor Allocation* in *Proceedings of the Job Scheduling Strategies for Parallel Processing* 1997 Springer-Verlag. p. 35-57
2. Berman, F., et al., *Adaptive computing on the Grid using AppLeS.* Parallel and Distributed Systems, IEEE Transactions on, 2003. **14**(4): p. 369-382.
3. Deelman, E., et al., *Mapping Abstract Complex Workflows onto Grid Environments.* Journal of Grid Computing, 2003. **1**(1): p. 25-39.
4. Deelman, E., et al., *Pegasus: A framework for mapping complex scientific workflows onto distributed systems.* Scientific Programming, 2005. **13**(3): p. 219-237.
5. Cao, J. and F. Zimmermann. *Queue scheduling and advance reservations with COSY.* in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. 2004.
6. Smith, W., I. Foster, and V. Taylor. *Scheduling with advanced reservations*. in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. 2000.
7. Huang, R., H. Casanova, and A.A. Chien. *Using Virtual Grids to Simplify Application Scheduling.* in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. 2006.
8. Mandal, A., et al. *Scheduling Strategies for Mapping Application Workflows onto the Grid*. in *The 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*. 2005.
9. Jim Blythe, S.J., Ewa Deelman, Yolanda Gil, Karan Vahi, Anirban Mandal, Ken Kennedy. *Task Scheduling Strategies for Workflow-based Applications in Grids*. in *IEEE International Symposium on Cluster Computing and the Grid*. 2005. Cardiff, UK.
10. Wieczorek, M., R. Prodan, and T. Fahringer, *Scheduling of scientific workflows in the ASKALON grid environment* SIGMOD Rec. , 2005 **34** (3): p. 56-62
11. Czajkowski, K., I. Foster, and C. Kesselman, *Agreement-based resource management.* Proceedings of the IEEE, 2005. **93**(3): p. 631-643.
12. Andreozzi, S., et al. *Agreement-Based Workload and Resource Management*. in *e-Science and Grid Computing, 2005. First International Conference on*. 2005.
13. *The Grid Resource Allocation and Agreement Protocol Working Group*, in *https://forge.gridforum.org/projects/graap-wg*.
14. MacLaren, J., *Advance Reservations: State of the Art*, in *Working Draft, Global Grid Forum at http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html*. 2003.
15. *Maui Cluster Scheduler*, http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php.
16. *PBSPro*, http://www.pbspro.com.
17. Zhou, S., et al., *Utopia: a load sharing facility for large, heterogeneous distributed computer systems* Softw. Pract. Exper. , 1993 **23** (12): p. 1305-1336
18. Condor_Glidein, *http://www.cs.wisc.edu/condor/glidein*.
19. Czajkowski, K., et al. *Grid Information Services for Distributed Resource Sharing*. in *10th IEEE International Symposium on High Performance Distributed Computing*. 2001: IEEE Press.
20. Zhang, L., et al., *RSVP: a new resource ReSerVation Protocol.* Network, IEEE, 1993. **7**(5): p. 8-18.
21. Rajkumar Buyya, M.M., David Abramson, Srikumar Venugopal,, *Scheduling parameter sweep applications on global Grids: a deadline and budget constrained cost-time optimization algorithm.* Software: Practice and Experience, 2005. **35**(5): p. 491-512.
22. Pinchak, C., P. Lu, and M. Goldenberg, *Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences*, in *Job Scheduling Strategies for Parallel Processing*, D.G.F.a.L.R.a.U. Schwiegelshohn, Editor. 2002, Springer Verlag. p. 205--228.
23. Roblitz, T. and A. Reinefeld. *Co-reservation with the concept of virtual resources*. in *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*. 2005.
24. Kee, Y.-S., et al. *Efficient resource description and high quality selection for virtual grids*. in *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*. 2005.
25. Goldenberg, M., P. Lu, and J. Schaeffer, *{TrellisDAG}: {A} System for Structured {DAG} Scheduling*, in *Job Scheduling Strategies for Parallel Processing*, D.G.F.a.L.R.a.U. Schwiegelshohn, Editor. 2003, Springer Verlag. p. 21--43.
26. Orlicky, J., *Material Requirements Planning*. 1975, New York: McGraw-Hill.