Estimating Resource Needs for Time-Constrained Workflows

Eun-Kyu Byun
Division of Computer Science, Korea Advanced Institute of Science and Technology
ekbyun@camars.kaist.ac.kr

Yang-Suk Kee, Ewa Deelman, Karan Vahi, Gaurang Mehta
Information Sciences Institute, University of Southern California
{yskee, deelman, vahi, gmehta}@isi.edu

Jin-Soo Kim
School of Information and Communication Engineering, Sungkyunkwan University
jinsookim@skku.edu

Abstract

Workflow technologies have become a major vehicle for the easy and efficient development of science applications. At the same time new computing environments such as the Cloud are now avaiable. A challenge is to determine the right amount of resources to provision for an application. This paper introduces an algorithm named Balanced Time Scheduling (BTS), which estimates the minimum number of virtual processors required to execute a workflow within a user-specified finish time. The resource estimate of BTS is abstract, so it can be easily integrated with any resource description language or any resource provisioning system. The experimental results with a number of synthetic workflows demonstrate that BTS can estimate the computing capacity close to the optimal. The algorithm is scalable so that its turnaround time is only tens of seconds even with workflows having thousands of tasks and edges.

1 Introduction

As high-performance distributed computing technologies advance, scientists and engineers are able to explore more complex phenomena in a variety of scientific fields. For instance, LEAD (Linked Environments for Atmospheric Discovery) [17] which orchestrates data collection and simulation experiments that forecast the formation and evolution of tornados continues to configure resources rapidly and automatically in response to weather. Likewise, the SCEC (Southern California Eartquake Center) [15] project determines which geograpic area are subject to the highest acceleration by calculating wave propagation on demand.

One of the key challenges that the applications confront in this exploration is how to transition their knowledge and legacy software to new computing environments. Some solutions include the use of higher-level application descriptions such as workflows [9], which can specify the overall behavior and the structure of applications in a platform-independent way. A workflow is often represented as a Directed Acyclic Graph (DAG) that consists of nodes and edges which represents tasks, and data and control dependencies between them. When an application is specified in this high-level manner, workflow management systems such as Pegasus [6] can target a number of execution environments and automatically transform the specifications into executable workflows that can be executed on distributed resources.

At the same time, the coordination and the provisioning of distributed resources have been challenging issues in the high-performance distributed computing community. Recent cloud computing technologies such as Amazon's EC2 [2] and 3Tera [1] enable dynamic resource provisioning taking into account of performance, cost, and other factors. Leveraging the resource specification technologies such vgDL [11] and SLA(Service Level Aggrement), users can specify a variety of resource requirements of applications. For instance, EC2 users can create a group of virtual processors with certain performance characteristics on-the-fly by selecting one out of predefined instance types even though QoS (Quality of Service) of resources, especially for network, is not yet guaranteed.

Workflow management systems can potentially benefit from compute clouds. For example, workflows can obtain consistent resources since cloud infrastructures take care of complex factors of resource management such as admission, performance, security, reliability, etc. A critical issue in the integration of workflow management and compute clouds is how to automatically estimate the number of virtual processors for given workflows because the computing capacity determines application performance and utility cost. A large number of virtual processors can reduce the execution time of parallel tasks while too many processors can cause low resource utilization, high scheduling overhead, and high cost. On the contrary, if the computing capacity is too small, the execution time of the workflow can increase, which can violate the timing requirements of applications.

In this work we focus on estimating the minimum number of virtual processors needed to complete a workflow. Specifically, we are focusing on time-constrained applications, where results are needed within a certain time frame. Note that this problem is different from conventional workflow scheduling [13, 7, 29, 25, 21, 3, 20] or cost-optimization problems [27, 28, 22, 16, 5], which aim at minimizing the application's runtime when a set of resources is given.

More importantly, this computing capacity estimate should be neutral so that it is independent of target language, resource environments, and detailed specifications of resources. As a solution, we propose a heuristic algorithm named as Balanced Time Scheduling (BTS), which estimates the minimum number of virtual processors required to execute a workflow within a given deadline. Our algorithm has several benefits when making resource allocation plans. BTS can utilize the idle time of resources allocated already instead of allocating additional resources by adjusting the start time of tasks on non-critical paths. In consequence, BTS can execute a workflow with fewer resources than the approaches based on conventional workflow scheduling techniques. The time complexity of BTS algorithm is small enough so it scales well even for workflows with thousands of tasks and edges. The experiments with synthetic workflows demonstrate the efficiency of our algorithm with respect to cost and performance.

The rest of this paper is organized as follows. Section 2 illustrates an example when resource capacity estimate is required and Section 3 details the proposed algorithm. The methodology and experimental results are presented in Sections 4 and 5, respectively. In Section 6, we discuss the prior studies closely related to our research. Finally, Section 7 concludes this paper with future research directions.

2 A Use Case: Pegasus on EC2

Pegasus [6] is a workflow management framework which enables the users to describe logical behavior of applications via abstract workflows, maps abstract workflows onto distributed resources through workflow planning, and uses Condor DAGMan [24] to execute tasks with fault-tolerance. Meanwhile, EC2 (Elastic Compute Cloud) is one of Amazon's Web Services components, which enables users to configure a virtual cluster on-the-fly. To instantiate a virtual cluster, users specify a VM (Virtual Machine) image named AMI (Amazon Machine Image), an instance type (out of 5 predefined platform configurations as of July 2008), the number of virtual processors.

For example, let an application require a cluster that consists of 10 Opteron processors with more than 1GB memory each. A EC2 command for this request can be

```
ec2run ami-xxxxxx -t m1.small -k gsg-keypair -n 10
```

In the above example, the user uses a small instance, the default standard instance type, which instantiates a VM image on a 32-bit platform that has 1.7 GB memory, 1 EC2 compute unit, and 160GB storage space.

EC2 charges on the basis of instance type, the number of instances, and their lifespan. Since the instance type and lifespan can be directly extracted from the user specifications of applications (e.g., task execution time and deadline), the key functionality required to integrate workflow technologies with cloud computing infrastructures is to estimate the number of virtual processors. Moreover, the number of processors is a common metric for computing power that can be used for other provisioning systems. Therefore, our main focus is on the estimate of the number of homogeneous processors.

The following is a simple use case for our resource estimator. We assume that a VM image which includes application software packages is created. The user would specify (in the Pegasus framework) application-specific knowledge about the resource requirements (e.g., processor type, memory capacity) and the application-level information (e.g., locations of executable, data, and replica) needed to run their application and the requested finish time. Then, a resource capacity estimator intercepts the resource information before the ordinary planning of Pegasus takes place; it synthesizes EC2 commandline arguments through a capacity estimate and instantiates a virtual cluster. Basically, the estimator determines the minimum number of virtual processors to complete a workflow within a certain deadline, termed the RFT (Requested Finish Time). Pegasus then can continue its normal planning process with the provisioned resources.

3 BTS Algorithm

Our algorithm is motivated by a simple idea that a task can be delayed as long as the delay does not violate its time constraints and other tasks can take advantage of the slack. We provide the resource estimate under several assumptions: a workflow is defined by a set of tasks with predicted execution time and a set of edges between tasks each with data transfer time; a *host* is defined as an independent processing unit on which a task is executed. A host is equivalent to a virtual processor of EC2, and it is connected to other hosts via network; tasks are non preemtable and can be executed on any host on-demand.

In practice, we consider three criteria in the design of the algorithm; **1) Communication cost**: When two dependent tasks are scheduled on the same host, the data transfer time between them can be ignored. This can reduce the makespan and eventually the number of hosts; **2) Overestimation**: A resource capacity can be overestimated as long as workflows can finish within a given deadline. However, we should reduce this overestimate because a tight estimate can improve resource utilization and reduce the overall resource allocation cost; **3) Scalability**: Since workflow planning is a time-consuming process and determining the minimum number of hosts for a deadline is an NP-hard problem, an algorithm should be scalable with a low time complexity.

We embody this idea through three steps; initialization which determines the valid scheduling time range of each task, task placement which determines the detailed schedule of workflow tasks, and task redistribution which reduces the number of hosts by adjusting the start time of the tasks scheduled at the placement. The following sections describe each step in details.

3.1 Initialization

The goal of the BTS algorithm is to minimize the amount of resources required by a workflow while satisfying the user-provided time constraints. For this purpose, BTS keeps track of a valid scheduling time range of each task, termed schedulable duration (SD), throughout its scheduling processes. A task can be scheduled at any time in its schedulable duration without violating the time constraints of the entire workflow.

SD is defined as $LFT(latest\ finish\ time) - EST(earliest\ start\ time)$. EST represents the earliest start time of a task when all its parent tasks finish as early as possible and LFT represents the latest finish time when all the descendants of a task are executed as late as possible until RFT. The initial values of EST and LFT of each task are calculated based on the RFT, *UpLength*, and *DnLength*; *UpLength* denotes the length of the longest execution path to a task from the entry task and *DnLength* is the length of the longest execution path from a task to the exit task as defined in the following equations.

$$\text{If } RFT < \max_{\forall task_i} \{ET_i + UpLength_i\}, \text{ stop and return false.} \quad (1)$$

$$SD_i = LFT_i - EST_i \quad (2)$$

$$EST_i = UpLength_i \quad (3)$$

$$LFT_i = RFT - DnLength_i \quad (4)$$

$$UpLength_i = \max_{\forall task_j \in P(i)} \{UpLength_j + ET_j + DTT_{j,i}, 0\} \quad (5)$$

$$DnLength_i = \max_{\forall task_j \in C(i)} \{DnLength_j + ET_j + DTT_{j,i}, 0\} \quad (6)$$

where $P(i)$: set of parent tasks of $task_i$,
$C(i)$: set of child tasks of $task_i$,
ET_i : predicted execution time of $task_i$,
$DTT_{i,j}$: data transfer time between $task_i$ and $task_j$

First, BTS checks whether RFT is valid or not as in inequality (1). If a given RFT is smaller than the minimum makespan of the workflow, BTS returns an error. The term on the right of the inequality denotes the minimum makespan which is equal to the length of the longest execution path through the workflow.

If a tasks has a long SD, it is flexible and accordingly likely to be scheduled at the time when some resources are idle. Therefore, widening a schedulable duration of a task can contribute to reducing the total number of hosts required by the workflow. Increasing the schedulable durations of the tasks on the critical path is particularly important because they are the most time-contrained. We note that the data transfer time between tasks can be eliminated when the tasks are co-located on the same resource. As a result the co-located tasks on the critical path can have more scheduling flexibility.

We implement this idea in a simple heuristic algorithm. First, $UpLength$s of all tasks are calculated by conducting a reverse depth first search starting from the exit task (if no single exit task exists, a dummy task with a runtime of zero is added and it is made dependent on all the leaf nodes of the workflow). Then, for each task in a descending order of $UpLength$, we find the child task that has the largest ($DnLength + ET + DTT$), since the path through the child task determines $DnLength$ of the current task. If any incoming edges to the child task are not zeroed, the DTT between the current task and the child task is set to zero. Then, we calculate the *DnLength* of the task using the new DTT. Note that all children of a task are visited prior to the task because the *UpLength*s of children cannot be smaller than those of their parents. After calculating $DnLength$s of all tasks, $UpLength$s are recalculated to reflect the changed DTTs. Using $UpLength$ and $DnLength$ values the initial schedulable durations for all the tasks are computed.

Figure 1 (a) shows an example workflow. Circles represent tasks and arrows show data dependencies between the tasks. *UpLength*s and *DnLength*s of tasks are summarized in 1 (b). During the calculation, DTT of four edges: (1,2), (3,4), (4,5), and (7,8) are ignored. The numbers to the left of the arrows represent the original values and those on the right the values after the task co-locating algorithm is applied. Finally, the initial values of schedulable durations of tasks are summarized in Figure 1 (c).

3.2 Task Placement

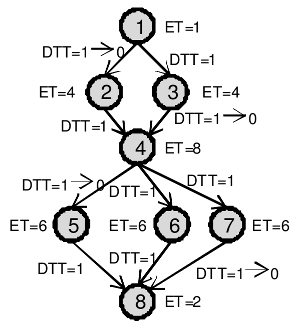
The task placement algorithm iterates over three steps until all tasks' start times are determined. 1) select the task to be scheduled, 2) find the best start time of the selected task within its schedulable duration to minimize the host requirement, and 3) update EST and LFT of all dependent tasks considering the effects of the already scheduled tasks.

The scheduling order of tasks and their placement in the schedule is determined by a set of rules. First, tasks with a narrow schedulable duration are scheduled with higher priority since tasks with a wide schedulable duration have more flexibility. Second, if multiple tasks have the same schedulable duration, a task that has more independent tasks as its children has priority since such independent tasks tend to be scheduled in the same time slot. Third, if the number of descendants is larger than that of the ancestors, a task is scheduled at the earliest time. The task is scheduled at the latest time in the opposite case because the slack created by the placement can benefit a larger number of tasks. Otherwise, either of two places is selected randomly. The following describe the algorithm in detail.

```
Initialize schedulable duration of all tasks.
Initialize NH(i) = 0 , 0 $<$ $i$ $\leq$ The number of time slots, NH(i) denotes
the number of hosts occupied by the placed tasks at the time slot $i$
While there are unscheduled tasks, repeat the following steps.
1. Pick up a task with the narrowest schedulable duration
2. Determine the starting time(x),$EST < x < (LFT - ET)$

      s.t. $\max_{i \in TimeSlot(x, x+ET)} \{NH(i)\}$ is minimized.

   $TimeSlot(a, b)$:set of time slots covering time range from a and b

3. Add 1 to $NH(i), \forall i \in TimeSlot(x, x + ET)$.
4. Update EST and LFT of all dependent tasks.

   $EST_i = \max_{\substack{\forall task_j \in P(i)-STS, \\ \forall task_k \in P(i) \cap STS}} \{ EST_j + ET_j + DTT_{j,i}, \; ST_k + ET_k + DTT_{k,i} \}$

   $LFT_i = \max_{\substack{\forall task_j \in C(i)-STS, \\ \forall task_k \in C(i) \cap STS}} \{ LFT_j - ET_j - DTT_{j,i}, \; ST_k - DTT_{k,i} \}$

   $ST_i$ : scheduled start time of $task_i$,$STS$ : set of already scheduled tasks
```

Figure 1 (d) shows the time schedule diagram of tasks after conducting the task placement in the workflow in Figure 1 (a). The x-axis denotes the elapsed time from the start of the execution of the workflow and the y-axis denotes the number of hosts occupied by the scheduled tasks. Tasks 1, 8, 2, 3, 4, 7, 5, and 6 are scheduled in this order and eventually the diagram shows that three hosts are required since tasks 5, 6 and 7 are use different hosts during time slots from 22 to 27.

The time complexity of this algorithm is $O(n(n + e + tlogt))$ where n is the number of tasks, e is the number of edges, and t is the number of time slots; the time complexity



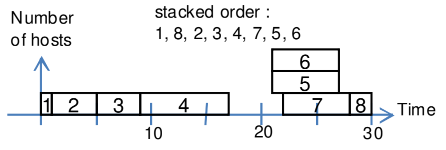
(a) Workflow

Task	UpLength	DnLength
1	0	24→23
2	2→1	19
3	2	19→18
4	7→6	10
5	16→14	3
6	16→15	3
7	16→15	3→2
8	23→22	0

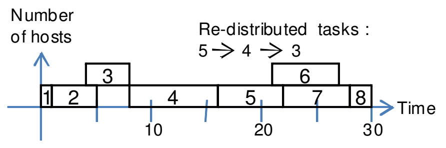
(b) UpLength, DnLength

Task	EST	LFT	SD
1	0	7	7
2	1	11	10
3	2	12	10
4	6	20	11
5	14	27	13
6	15	27	12
7	15	28	13
8	22	30	9

(c) Initial EST, LFT and Schedulable duration when RFT = 30



(d) Schedule of tasks after Task placement



(e) Schedule after Task Redistribution

Figure 1. An example of BTS algorithm

of selecting a task with the minimum time range is $(O(n))$; that of updating ESTs and LFTs of all dependent tasks is $(O(e))$; and that of selecting a time slot that minimizes the number of hosts is $(O(tlogt))$. The number of time slots is calculated by dividing RFT by unit time. BTS uses the greatest common divisor (GCD) of predicted execution time of all tasks as the unit time. Since the precision of execution time is coarse, the unit time in practice is not so small and the order of t is not much larger than the order of e or n.

3.3 Task Redistribution

The task placement technique can fail to find a global optimum for workflows with certain structures. This can be seen in Figure 1 (d), where task placement algorithm scheduled tasks with narrower schedulable duration first: task 3 is scheduled before tasks 4, 5, 6 and 7. This causes tasks 5, 6 and 7 to be scheduled on different hosts. To deal with such imbalance of resource utilization, we introduce a new

```
1. Non-propagated Redistribution
    For all tasks scheduled at time slot(s) with maximum $NH$.
    Check whether it can be scheduled at other time slot except the current time slot.
    If possible, reschedule the task and restart algorithm

2. Thrust to the earlier slots
    a. Select the earliest time slot which have maximum $NH$.
    b. Select the task(t) with the least number of ancestor tasks while satisfying
        $UpLength_t + ET_t \le ScheduledStartTime(t)$
    c. Call ThrustLeft( t, start time of the selected time slot)

3. Thrust to the later slots
    a. Select the latest time slot with maximum $NH$.
    b. Select the task(t) with the least number of descendant tasks while satisfying
        $RFT - DownLength_t - ET_t \ge Scheduled\,Finish\,Time(t)$
    c. Call ThrustRight( t, end time of the selected time slot)

4. If both 2 and 3 return fail, return NH as the estimate number of host
```

```
ThrustLeft(task t, time b)        // b: time bound of task t's finish time
    Find maximum x which satisfy $\max_{i \in TimeSlot(x, x+ET_t)} \{NH(i)\} + 1 < \max_{i \in TimeSlot(0, RFT)} \{NH(i)\}$
        where $UpLength_t \le x \le b$
    If x doesn't exist, return false
    If x < EST, $\forall p \in P(t)$, call $ThrustLeft(p, x - DTT_{p,t})$
    If at least one parent return false, return false
    Update ESTs of all child tasks and return true

ThrustRight(task t, time b)       // b: time bound of task t's start time
    Find maximum x which satisfy $\max_{i \in TimeSlot(x, x+ET_t)} \{NH(i)\} + 1 < \max_{i \in TimeSlot(0, RFT)} \{NH(i)\}$
        where $b \le x \le RFT - DnLength_t - ET_t$
    If x doesn't exist, return false
    If x < LFT, $\forall c \in C(t)$, call $ThrustRight(c, x + ET_t + DTT_{t,c})$
    If at least one child return false, return false
    Update LFTs of all parent tasks and return true
```

Figure 2. Task Redistribution algorithm

step named *Task redistribution*. The main idea of this step is to move tasks in the busiest time slot to adjacent time slots one by one within their schedulable duration and to see if this reduces the number of needed resources or not.

A high-level description of our task redistribution algorithm is shown in Figure 2. The first non-propagated redistribution step adjusts the start time of tasks in the busiest time slots without affecting other tasks. Then, Steps 2 and 3 force to move tasks in the busiest time slots to underutilized time slots. We minimize the effects on the original scheduling results of other tasks and make the results consistent if changes are required. The time complexity of this algorithm is $O(n * e * tlogt)$ in the worst case.

Figure 1(e) shows how Task redistribution complements the limitation of Task placement. Task 5 is selected since it lies in the thickest time slot and is rescheduled to start at time 16 and to finish at time 22. It causes tasks 4 and 3 also to be rescheduled. Eventually, BTS concludes that 2 hosts are required to complete the workflow within 30 time units.

4 Methodology

4.1 Synthetic Workflows

We rigorously evaluate the performance of our algorithm with randomly generated synthetic workflows. We classify the random workflows into two groups, based on their structures.

- ***Fully Random Workflows (FRW):*** Any task can be connected to any other task. For every task *a* there is a path from the entry task to task *a* and from task *a* to the exit task. We use four parameters for synthesizing this type of workflows; the number of tasks (N), the number of edges (E), range of execution time of each task (T), and data transfer time of each edge (C). Each edge connects two randomly selected tasks and the execution time of tasks are selected through random trials with a uniform distribution over given ranges.
- ***Leveled Parallel Workflows (LPW):*** Workflows in this group are structured so that only tasks in adjacent levels can have dependencies. Five parameters are used to represent these workflows: the number of tasks (N), the number of levels (L), maximum parallelism (MP), range of execution time of each task (T), data transfer time of each edge (C). MP is the maximum degree of parallelism of a workflow where the degree of parallelism of a level is the number of tasks in the level. Any task in the *ith* level can be the parent of a task in the *(i+1)th* level. The execution time is selected via uniform random trials. We also consider two cases: 1) the execution time of tasks in the same level is homogeneous and 2) heterogeneous.

4.2 Algorithm Comparison

We use existing workflow scheduling algorithms that can be used (or adapted) to estimate the needed resource capacity to evaluate the efficiency of BTS.

- ***FU:*** FU determines the number of hosts by summing up the execution times of all tasks divided by RFT. This is the resource capacity required when all hosts are fully utilized and data dependencies are ignored. Even though this algorithm does not guarantee RFT, it can calculate the lower bound on resource capacity with constant time.
- ***IterHEFT:*** HEFT [] picks the task with the largest value of $(ET + DnLength)$ and schedules it on the resource which can finish the task as early as possible. In general, the workflow makespan monotonously decreases as more hosts are used for the scheduling (up to the maximum parallelism of the workflow). Therefore, we can determine the minimum number of hosts required to finish a given workflow within a deadline by repeating HEFT-based scheduling over a growing resource set until the resulting makespan is equal to or shorter than RFT.

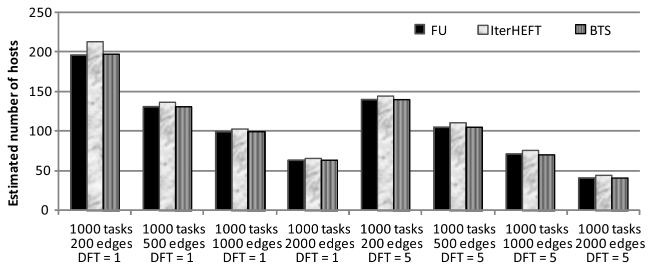


Figure 3. Comparison of three approaches on fully random workflows.

5 Experiments

5.1 Synthetic Fully Random Workflows

We evaluate the quality of BTS with respect to the turnaround time and the estimated number of hosts, comparing to the other two approaches. The resource capacity estimate against a variety of synthetic random workflows is shown in Figure 3. Each bar represents the average estimate of 30 workflows. The execution time of tasks is randomly selected from 2 to 10 time units. The average data transfer time of each edge is 1 time unit for the four groups on the left and 6 time units for the four groups on the right.

First, BTS achieves good quality of estimate for fully random workflows–close to optimal in that the estimate of FU is considered the lower bound. Moreover, BTS performs better than IterHEFT. The reason is that HEFT schedules tasks as early as possible so that it can make the idle time fragmented while BTS determines the task schedule in a more flexible way in schedulable duration in order to minimize the number of hosts.

Next, Table 1 summarizes the turnaround time of IterHEFT and BTS measured on a PC having a 3Ghz CPU and 2GB RAM. The results show that BTS is very efficient. BTS takes less than 1 minute even with large workflows having thousands of tasks and edges while IterHEFT takes more than one hour. In summary, BTS outperforms IterHEFT with respect to quality and cost.

5.2 Synthetic Leveled Parallel Workflows

We evaluate BTS and IterHEFT against leveled parallel workflows consisting of tasks with heterogenous execution time. The results are shown in Figure 4. Each workflow has 1,000 tasks and 10 levels and the execution time of each task ranges from 3 to 10 for the graphs at the top and from 7 to 10 for the graphs at the bottom. Data transfer time is 1 for all cases. BTS achieves a similar quality of estimate to IterHEFT. The main reason is that the tasks of leveled parallel workflows at one level cannot share time slots with tasks at different levels while the fully random workflow can have tasks with wide schedulable durations. The two graphs on the top show that the number of hosts is less than the maximum parallelism when the RFT is equal to the minimum makespan because tasks with short execution time (e.g., 3 time units) at a level can be scheduled onto the same hosts while the tasks with long execution times at the same level are running.

Table 1. Estimate time(seconds)

Workflow complexity	IterHEFT	BTS
1000 tasks, 1000 edges	9.2	1.2
5000 tasks, 2000 edges	84.4	7.6
5000 tasks, 5000 edges	3914	36.1

Due to space limitations we do not present the results of the evaluation of BTS and iterHEFT for LPW workflows with homogeneous execution times. However, both algorithms estimate exactly the same number of needed hosts with BTS performing at least as well as iterHEFT.

6 Related Work

The main objective of workflow scheduling algorithms is to minimize the makespan of a workflow for a given resource set. Many algorithms were developed [13, 29, 7] such as list scheduling [25, 3, 21, 18], dividing a DAG into several levels [20], greedy randomized adaptive search [4], task duplication [19] and critical path first [25]. Different from the conventional approaches which aim at minimizing the makespan over limited resources, our goal is to find the minimum resource set that satisfies a given deadline.

Another category is workflow scheduling over unbounded resources. In practice, clustering techniques [8] such as DSC [26] and CASS-II [14] return the amount of resources required to minimize the makespan as well as the resulting schedules. To reduce the makespan, the clustering algorithms remove the data transfer between tasks with data dependency by scheduling them onto the same cluster. Similar to the conventional workflow scheduling algorithms, their main focus is to minimize makespan. Therefore, they cannot be used to explore the effect of application deadline on the resource amount required for application.

Singh et al [22] and Yu et al [28] used a genetic algorithm to find optimal task-resource mappings. Singh's approach minimizes both cost and makespan at the same time while the Yu's approach minimize only cost for a given deadline. Time Distribution approach [27] distributes a deadline to subgraphs and cost-optimization is performed for each subgraph. Conceptually, our problem can be thought as cost-minimization with time constraint over unbound resources. That is, our objective is to find a mechanism to estimate the minimal resource set required for successful workflow execution. The major difference between conventional cost-

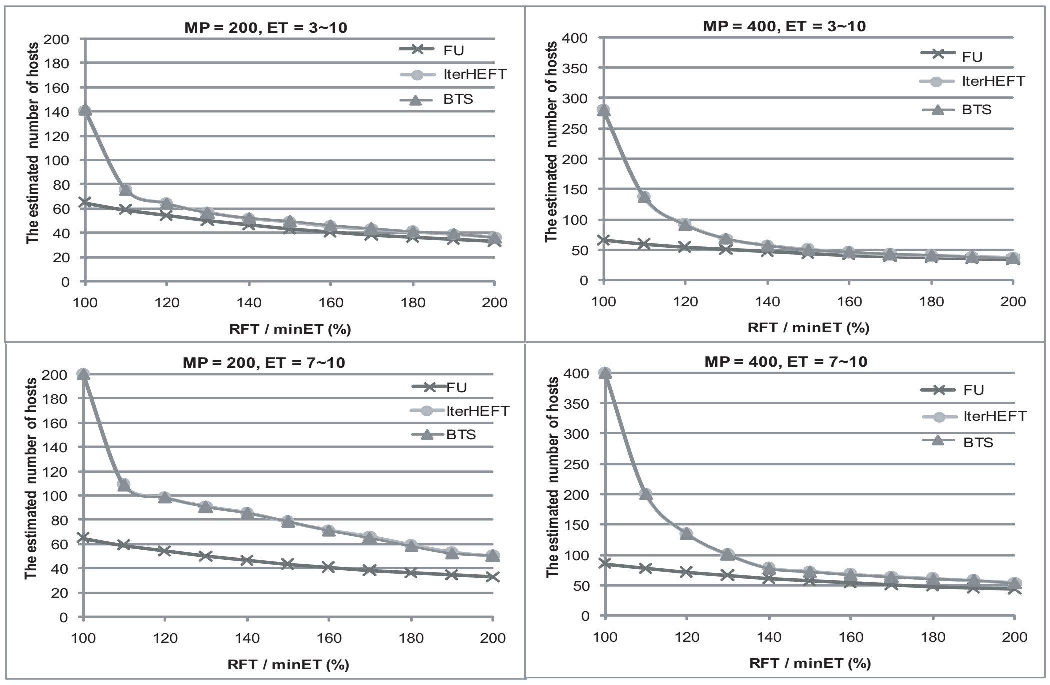


Figure 4. Comparison of BTS and IterHEFT for leveled parallel workflow with non-identical tasks.

minimization problems and our estimate is that they focus mostly on selecting a subset from a limited resource set whose properties such as unit cost and available time range are known. In contrast, our approach assumes that the resource universe and the selection mechanism are opaque.

Sudarsanam et al [23] proposed a simple technique to estimate the amount of resources. They iteratively calculated the makespan and utilization for numerous resource configurations and determined the best one. Even though this approach is likely to find an optimal solution, it does not scale well to large workflows and large resource sets. Next, Huang et al [10] proposed a mechanism for finding the minimum resource collection (RC) size to complete a workflow within minimum execution time. A RC size is determined by empirical data gathered from many sample workflows, varying the parameters such as DAG size, communication-computation ratio, parallelism, and regularity that characterize workflows. Even though this approach provides reasonable performance for workflows with similar characteristics to those of the sample workflows, it does not guarantee that its estimates are correct for arbitrary workflows. Additionally, parallelism and regularity cannot be calculated deterministically for workflows with a complex shape. Due to such limitations, this approach is only useful for specific classes of workflows. In contrast, our algorithm can be applied to any type of workflow since our algorithm directly analyzes the workflow structure. Finally, our algorithm can explore any finish times greater than the minimum execution time while Huang's approach can be applied for the minimum execution time.

7 Conclusions

In this paper, we proposed a new algorithm named BTS which estimates the minimum resource capacity needed to execute a workflow within a given deadline. This mechanism can bridge the gap between workflow management systems and resource provisioning systems. Moreover, the resource estimate is abstract and independent of the resource selection mechanism, so it can be easily integrated with any resource description language and any resource provisioning system even though our case study is conducted in the context of a compute cloud. Through our experiments with synthetic workflows, we demonstrated that BTS can estimate the resource capacity very efficiently with small overestimates, compared to the existing approaches. It also scales comparatively well, giving a turnaround time of only tens of seconds even with large workflows having thousands of tasks and edges. This short turnaround time is critical to the applications such as LEAD that need realtime adaptation.

In this study, we assumed that each workflow task is serial. Therefore, this estimate can be applied to applications such as parameter sweep studies. We understand that many science applications include data-parallelism and that a workflow task can be an MPI-like parallel task. We are working on extending the BTS algorithm for this class of applications by defining the task occupying *n* hosts. In addition, we assumed that all resources required for a workflow are available throughout the lifetime of application. However, holding all resources during the entire lifespan can cause resources to be underutilized. Resource provisioning techniques such as the Virtual Grid [12] provide fine-grained time-based resource reservation over heterogenous resources. We believe that an extension of our algorithm can exploit such advanced features of provisioning systems and enable more cost-efficient workflow execution. Finally, BTS doesn't consider failures of resources which may cause deadline misses. This problem may be handled by overprovisioning or computation restarts, but both may cause more resource costs. We will expand BTS to handle such behaviors. In the future, we plan on evaluating the algorithm on the cloud and integrating the Pegasus workflow management system with the Virtual Grid to evaluate the approach in real settings.

8 Acknowledgments

This work was funded by NSF under grants CCF-0725332 and CNS-0615412.

References

[1] 3Tera - Utility Computing for Web Applications. http://www.3tera.com.

[2] Amazon elastic compute cloud (amazon ec2). http://aws.amazon.com/ec2.

[3] T. L. Adam et al. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[4] J. Blythe et al. Task scheduling strategies for workflow-based applications in grids. In *Proceedings of IEEE International Symposium on Cluster Computing on Grid*, 2005.

[5] R. Buyya et al. Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost-time optimization algorithm. *Software-Practice and Experience*, 35:491–512, 2005.

[6] E. Deelman et al. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[7] F. Dong et al. Scheduling algorithms for grid computing: State of the art and open problems. *Technical Report, 2006-504, School of Computing, Queens University, Kingston, Ontario*, Jan 2006.

[8] A. Geras. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, 1992.

[9] Y. Gil et al. Examining the challenges of scientific workflows. *IEEE Computer*, Dec 2007.

[10] R. Huang et al. Automatic Resource Specification Generation for Resource Selection. In *Proceedings of ACM/IEEE International Conference on High Performance Computing and Communication (SC'07)*, 2007.

[11] Y.-S. Kee et al. Efficient Resource Description and High Quality Selection for Virtual Grids. In *Proceedings of ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGRID'05)*, 2005.

[12] Y.-S. Kee et al. Grid Resource Abstraction, Virtualization, and Provisioning for Time-targeted Applications. In *Proceedings of ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGRID'08)*, 2008.

[13] Y. K. Kwok et al. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[14] J. Liou et al. A comparison of general approaches to multiprocessor scheduling. In *Proceedings of the 11th International Symposium on Parallel Processing*, 1997.

[15] H. Magistrale et al. The SCEC Southern California Reference Three-Dimensional Seismic Velocity Model Version 2. *Bulletin of the Seismological Society of America*, 90:S65–S76, 2000.

[16] D. A. Menasce et al. A framework for resource allocation in grid computing. In *Proceedings of the 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications System*, 2004.

[17] B. Plale et al. CASA and LEAD: Adaptive Cyberinfrastructure for Real-Time Multiscale Weather Forecasting, 2006.

[18] M. Rahman et al. A dynamic critical path algorithm for scheduling scientific workflow applications on global grids. In *Proceedings of IEEE International Conference on e-Science and Grid Computing*, 2007.

[19] S. Ranaweera et al. A task duplication based scheduling algorithm for heterogeneous systems. In *Proceedings of 14th International Parallel and Distributed Processing Symposium (IPDPS00)*, 2000.

[20] R. Sakellariou et al. A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In *Proceedings of the IEEE Heterogeneous Computing Workshop*, 2006.

[21] G. C. Shih et al. A Compile-Tim Scheduling Heuristics for Interconnection-Constrained Heterogeneous Processor Architecture. *Transaction on Parallel and Distributed Systems*, 4(2):75–87, 1993.

[22] G. Singh et al. Application-level Resource Provisioning on the Grid. In *Proceedings of Third IEEE International Conference on e-Science and Grid Computing*, 2006.

[23] A. Sudarsanam et al. Resource estimation and task scheduling for multithreaded reconfigurable architectures. In *Proceedings of the 10th International Conference on Parallel and Distributed Systems*, 2004.

[24] C. Team. The directed acyclic graph manager, 2002. http://www.cs.wisc.edu/condor/dagman.

[25] H. Topcuoglu et al. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[26] T. Yang et al. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Trans. on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[27] J. Yu et al. Cost-based scheduling of Scientific Workflow Applications on Utility Grids. In *Proceedings of the 1st IEEE International Conference on e-Science and Grid Computing*, 2005.

[28] J. Yu et al. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming Journal*, 14(3-4):217–230, 2006.

[29] J. Yu et al. *Workflow Schdeduling Algorithms for Grid Computing.* Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, May 2007. Technical Report, GRIDS-TR-2007-10.