Efficient Resource Capacity Estimate of Workflow Applications for Provisioning Resources

Eun-Kyu Byun, Jin-Soo Kim
Korea Advanced Institute of Science and Technology
ekbyun@camars.kaist.ac.kr, jinsoo@cs.kaist.ac.kr

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

Abstract

Workflow technologies have become a major vehicle for the easy and efficient development of science applications. When integrating the workflow technology with the state-of-art resource provisioning technology, the challenge is to determine the amount of resources necessary for the execution of workflow. This paper introduces an algorithm named Balanced Time Scheduling (BTS), which estimates the minimum number of hosts 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 languages and resource provisioning systems. Moreover, the experimental results with a number of synthetic workflows and several real application workflows demonstrate that BTS can estimate the resource capacity close to the lower bound while 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 such as the Grid[10] advance, scientists and engineers are able to explore more complex phenomena in a variety of scientific fields such as biology, astronomy, geology, medicine, etc [15, 23, 2, 25]. One of the key challenges that they 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, 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 [8] can target a number of execution environments and automatically transform the specifications into executable workflows that can be executed on campus or 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 server-level virtualization technologies such as virtual cluster [14], Virtual Grid [18, 17], and Amazon's EC2 [1] enable dynamic resource provisioning taking into account performance, fault-tolerance, economics, and other factors. They provide resource description languages such as RSL [7], JSDL [4], ClassAd [21], Redline [22], and vgDL[17]. Using these languages, users can not only specify basic resource attributes such as processor type, memory/disk, capacity, and network bandwidth but also advanced features such as cost and reliability.

Workflow management systems can potentially benefit from the distributed resource management technologies. For example, workflows can obtain better resource provisioning since resource management technologies can acquire resources, considering complex factors such as resource dynamics, economics and availability. They can also benefit from advanced fault tolerance features provided by the resource management systems.

A critical issue in the integration of workflow management and resource management is how to make them communicate with each other. Specifically, a key question is how to automatically estimate resource requirements for given workflows. The most important attribute from the perspective of high-level workflows is the number of CPUs/hosts required, because the resource set size is an important factor in determining the makespan of workflow application and of the cost of resource allocation. If the number of resources is large, parallel execution of independent tasks can reduce the execution time while too many resources can cause low resource utilization, high scheduling overhead, and high cost. On the other hand, if the number of resources is too small, the execution time of the workflow can increase. As such, it is important to estimate the minimum amount of resources to complete a workflow within a given deadline. Note that this problem is different from conventional workflow scheduling[19, 9, 36, 33, 29, 3, 28] or cost-optimization problems[37, 35, 30, 26, 6], which aim at minimizing the application's runtime on a given set of resources.

More importantly, this resource capacity estimate should be neutral so that it is independent of target language, resource environments, and detail specifications of resources. As a solution, we propose a heuristic algorithm named as Balanced Time Scheduling (BTS), which estimates the minimum number of hosts 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 path. As a consequence, BTS can execute a workflow with fewer resources than the approaches based on conventional workflow scheduling techniques. Next, the time complexity of BTS algorithm is small enough so it is scaled well even for workflows with thousands of tasks. The experiments with synthetic workflows and several real application workflows demonstrate the efficiency of our algorithm with respect to cost and performance.

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

2 Related Work

Workflow scheduling is a well-known problem and many studies have been conducted. The main objective of these algorithms is to minimize the makespan of workflow for a given set of resources. Most algorithms rely on a list scheduling technique, which assigns ranking values to each task in a workflow and schedules each task in a descending order of ranking value. Algorithms differ in the way they calculate ranking values. List scheduling provides quite a good performance with relatively small time complexity[33, 3, 29, 24]. HEFT[33], which is one of the scheduling algorithms employed by Pegasus, is one of the most popular algorithms based on list scheduling. HEFT considers both communication and computation cost and achieves relatively good performance in general cases. For more information about workflow scheduling algorithms and their characteristics, please refer to [19, 36, 9]. In addition to the list scheduling method, other techniques such as dividing DAG into several levels [28], greedy randomized adaptive search [5], task duplication [27] and critical path first [33], are also investigated. Different from the conventional workflow scheduling techniques which aim at minimizing the makespan over limited resources, our goal is to find the minimum resource set size that satisfies a given deadline.

Another category of scheduling methods is on workflow scheduling over unbounded resources. In practice, clustering techniques [11] such as DSC [34] and CASS-II [20] 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 the application deadline on the resource amount required for the application.

Singh et al [30] and Yu et al [35] 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 [37] distributes a deadline to subgraphs and cost-optimization is performed for each subgraph. Cost optimization is also an important issue in project management. Scheduling techniques for project management [13] calculate *Float* that represents the schedulable time range of each subtask and find optimal task-resource mappings using a linear programming technique. Our algorithm also uses a notion of *schedulable duration* similar to *Float*. However, the cost-minimization techniques basically solve a problem against bounded resources.

Conceptually, our problem can be thought as a cost-minimization problem 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 the conventional cost-minimization problems and our approach is that they focus mostly on selecting a subset out of limited resources whose properties such as unit cost and available time range are known. On the contrary, our approach assumes that resource universe and selection mechanism are completely opaque.

There are only a few prior studies that have the same goal to our research. Sudarsanam et al [31] proposed a simple technique to estimate the amount of resources. They iteratively calculate the makespan and utilization for numerous resource configurations and determine the best one. Even though this approach is likely to find an optimal solution, it does not scale well with large workflows and large resource sets. Next, Huang et al [12] 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 complex shape. Due to such limitations, this approach is only useful for the specific classes of workflows. By contrast, our algorithm can be applied to any type of workflows and it does not require any empirical data since our algorithm directly analyzes the workflow structure. Finally, our algorithm can arbitrarily explore any desired finish times that are greater than the minimum execution time while Huang's approach can find the resource collection size only for the minimum execution time.

3 The Problem

The resource capacity estimate is a critical function required to bridge the gap between workflow management systems and distributed resource management systems. For instance, Pegasus [8] is a workflow management framework which enables the users to describe logical behavior of application via abstract workflows, maps abstract workflows onto distributed resources through intelligent workflow planning, and uses Condor DAGMan [32] to execute tasks with fault-tolerance. On the other hand, the Virtual Grid (VG) [18] enables users to program their resource environments using a resource description language named vgDL, specifying a temporal and spatial resource requirements via resource slots [16].

A simple motivating scenario is one where users run their applications via Pegasus while using the resources dynamically provisioned by Virtual Grid. The user would specify the 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 when they submit the workflow to the system. Then, a resource capacity estimator intercepts the resource information before the ordinary planning of Pegasus takes place and synthesizes a description through a capacity estimate. For example, the estimator can generate the following vgDL description when the user needs a cluster consisting of 1 Ghz Opteron processors with 2 GB memory each and want to finish the application within 1 hour.

```
// 00/00/0000@00:00:00 - slot start time, 00:00:00 - slot duration
// node = [ ... ] - node definition
vgdl = ClusterOf(node)[ 10 ] <03/31/2008@12:00:00, 01:00:00> {
   node = [ (Processor == "Opteron") && (Clock >= 1024) && (Memory >= 2048) ]
}
```

In this example, the host requirements are embedded into the node definition and the requested finish time is converted into the slot duration while the size of cluster is determined automatically by the estimator. Once the estimator submits this description, the Virtual Grid execution system allocates resources and then Pegasus can continue its normal planning process with the resources. Note that acquiring appropriate resources is up to resource selection systems while runtime schedulers at the execution phase take care of the dynamics in execution time and data transfer time over actual resources.

The estimator determines the minimum number of hosts to complete a workflow within a certain deadline, termed the RFT(Requested Finish Time). 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* means an independent processing unit on which only one task is executed at the same time and it is connected to other hosts via network; Tasks are non preemtable and can be executed on any host on-demand.

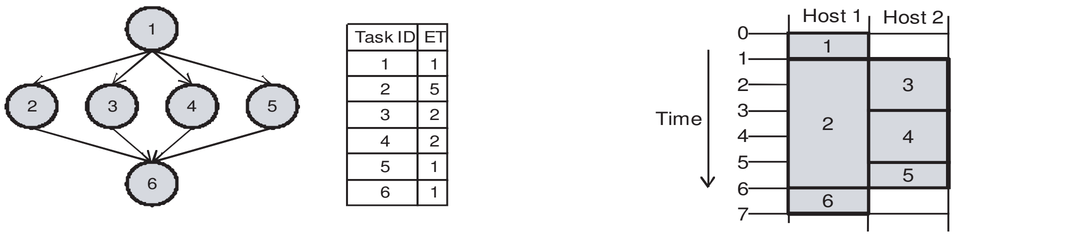
In practice, we consider three criteria in the design of the algorithm; **1) Communication cost**: When two dependent tasks are scheduled on the same host, the data transfer time between them can be ignored. This can reduce the makespan and eventually the number of hosts; **2) Overestimation**: A resource 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.

4 BTS Algorithm

4.1 Key Idea

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 that other tasks can take advantage of the slack. Figure 1 shows how this simple idea can reduce the number of resources required to execute a workflow. The structure of workflow and the execution time of each task are shown in

Figure 1 (a). For simplicity, we ignore data transfer time between tasks in this example. A simple resource capacity estimate is to have 4 hosts to exploit the maximum parallelism of the workflow. Then, we can finish this workflow in 7 time units, which is the sum of execution times on the longest path (i.e., task 1, 2, and 6). In the meantime, task 3, 4, and 5 use their resources only during a partial period of time and are idle in the remaining time. Therefore, we can delay task 3, 4, and 5 arbitrarily as long as they can finish by the finish time of task 2. For instance, we can execute task 3, 4, and 5 sequentially on a single host because the sum of their execution times is equal to the execution time of task 2. Accordingly, we can schedule all tasks with only two resources as shown in Figure 1 (b).



Task ID	ET
1	1
2	5
3	2
4	2
5	1
6	1

<a> Workflow and execution time

<b> Optimal scheduling for minimum makespan

Figure 1. An example of workflow with 6 tasks. ET is the execution time of the task.

4.2 Algorithm Details

Our algorithm embodies this idea through three steps; task clustering, task placement, and task redistribution. The task clustering step selects pairs of tasks with direct data dependencies to remove their data transfer costs. Then, the task placement step determines the detailed schedule of workflow tasks. Finally, the task redistribution step reduces the number of hosts by adjusting the start time of the tasks scheduled at the placement step. The following sections describe each step in details.

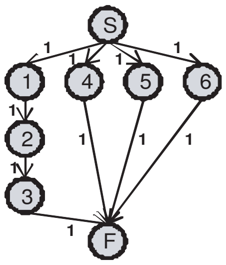
4.2.1 Task Clustering

When two tasks connected by an edge are scheduled onto the same resource, the data transfer time between them can be ignored and this eventually lessens the execution time of the critical path of workflow. The goal of task clustering is similar to that of unbounded scheduling techniques such as DSC[34]. Since this problem is known NP-hard, we use a simple and fast approximation algorithm. Before we explain the details of the algorithm, we first define two key terms used in this paper.

$$UpLength_i = \max_{\forall task_j \in P(i)} \{UpLength_j + ET_j + DTT_{j,i}, 0\} \tag{1}$$

$$DownLength_i = \max_{\forall task_j \in C(i)} \{DownLength_j + ET_j + DTT_{j,i}, 0\} \tag{2}$$

where $P(i)$: set of parent tasks of $task_i$, $C(i)$: set of child tasks of $task_i$,



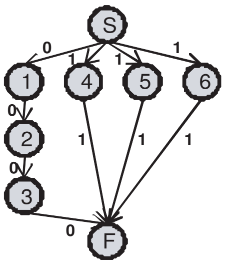
(a) Workflow

Task ID	UpLength
S	0
1	2
2	4
3	6
4	2
5	2
6	2
F	8

(b) UpLength before task clustering

Task ID	DownLength
S	4
1	3
2	2
3	1
4	2
5	2
6	2
F	0

(c) DownLength



(d) Workflow after task clustring

Task ID	UpLength
S	0
1	1
2	2
3	3
4	2
5	2
6	2
F	4

(e) UpLength after task clustering

Figure 2. Example steps of task clustering phase.

$$ET_i : \text{predicted execution time of } task_i,\ DTT_{i,j} : \text{data transfer time between } task_i \text{ and } task_j$$

The *UpLength* denotes the length of the longest path to a task from the entry task (denoted S–this task is artificially placed as the root of a workflow), which is the earliest start time of the task when all parent tasks finish as early as possible. We can calculate the *UpLength* value of each task by conducting a depth first search starting from the exit task (denoted F–this task is artificially placed as the last task of the workflow and depends on all the workflow leaves). On the other hand, the *DownLength* denotes the length of longest path from a task to the exit task. Once the *UpLength*s of all tasks are calculated, we visit tasks in a descending order of *UpLength* and calculate *DownLength*. For each task, we find the child of the task that has the largest *(DownLength + ET + DTT)*. If the child task is not clustered with its other parents, the two tasks are clustered and DTT between the tasks is set to 0. Then, we calculate the *DownLength* of the task. 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. Finally, *UpLength*s are recalculated to reflect the updates of clustering.

Figure 2 illustrates how to calculate *UpLength* and *DownLength* of a workflow with 8 tasks through task clustering. The execution time of each task and the data transfer time between tasks is uniformly 1 time unit. First, *UpLength*s for all tasks are calculated in (b). Then, *DownLength*s are calculated in the order of task F, 3, 2, 1, 4, 5, 6, and S in (c) while task pairs (3,F), (2,3), (1,2), and (S,1) are clustered as shown in (d). Finally, *UpLength*s are recalculated (Figure 2 (e)).

4.2.2 Task Placement

A *schedulable duration* of a task can be defined by EST(Earliest Start Time) and LFT (Latest Finish Time). Through the task clustering process, we can have the maximum schedulable duration of each task since *UpLength* means the EST when all ancestors of task are executed as early as possible and $(RFT - DownLength)$ means the LFT when all descendants of task are executed as late as possible. Then, in the task placement step, we determine when is the best time to schedule each task in its *schedulable duration*, considering the timing constraints of its dependents and the resource utilization.

As shown in Figure 3, task placement calculates the *schedulable duration* of tasks, determines the scheduling order of

```
1. Check whether RFT is greater than or equal to the length of critical path. If not, set RFT as the length of critical path
2. Initiate EST and LFT for all tasks.    EST = UnLength,    LFT = RFT - DownLength
3. While there is unscheduled tasks
    a. Pick up a task with the narrowest schedulable duration.    Schedulable duration = LFT - EST
    b. Determine the starting time(x) between EST and LFT – ET such that $\max_{i \in TimeSlot(x,x+et)} \{NH(i)\}$ is minimized
       NH(i) : the number of tasks already scheduled in time slot i , TimeSlot(a,b):set of time slots covering time range between a and b
    c. Add 1 to the number of host of time slots selected in step 2-b
    d. Update LFT of all ancestor tasks and EST of all descendant tasks
```

Figure 3. Algorithm description of task placement phase.

tasks, and finds the best start time of each task to minimize resource capacity. First, the task placement algorithm checks if RFT is valid by comparing it to the length of the critical path. Then, it initializes EST with *UpLength* and LFT with $(RFT - DownLength)$. A task can be scheduled anytime in its schedulable duration, the safe time range in a way as to not violate the time constraints of (forward and backward) dependent tasks. Whenever the start time of a task is determined, EST and LFT of all dependent tasks are updated by equation 3 and 4. The second term of the max functions in both equations means the time bound restricted by scheduled tasks whose start and finish times are already set. Basically, the schedulable duration of unscheduled tasks become narrower as more tasks are scheduled.

$$EST_i = \max_{\forall task_j \in P(i)-STS, \forall task_k \in P(i) \cap STS} \{EST_j + ET_j + DTT_{j,i}, ST_k + ET_k + DTT_{k,i}\} \tag{3}$$

$$LFT_i = \max_{\forall task_j \in C(i)-STS, \forall task_k \in C(i) \cap STS} \{LFT_j - ET_j - DTT_{j,i}, ST_k - DTT_{k,i}\} \tag{4}$$

where ST_i : scheduled start time of $task_i$, STS : set of 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 two or more tasks have the same schedulable duration, the task having fewer dependents has priority. Third, if the number of descendants is larger than that of ancestors, a task is scheduled at the earliest time because the slack of a larger number of tasks will be affected. In the opposite case, the task is scheduled at the latest time. Otherwise, either of the two places is selected randomly.

Figure 4 illustrates how the task placement technique works for Figure 2(a) when RFT is 7 and the unit time of time slots is 1. The x-axis represents scheduling time and the y-axis does the number of hosts. The table for EST and LFT has been constructed already via task clustering and task placement initialization. If a task is scheduled, a rectangle with a height of 1 and a width of ET is placed in its time slot. At first, BTS selects task 4 since $(LFT - EST)$ of all tasks are equal and task 4 has the least number of dependent tasks. Task 4 is scheduled at the earliest time slot within its schedulable duration since the

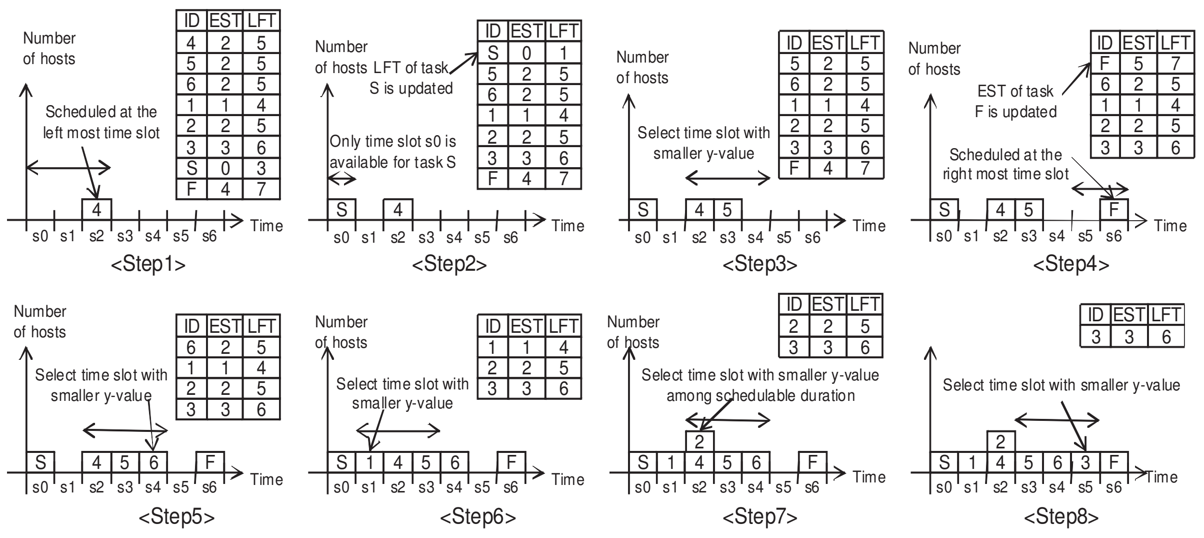

Number of hosts
Time
s0 s1 s2 s3 s4 s5 s6
ID EST LFT
Scheduled at the left most time slot
<Step1>
LFT of task S is updated
Only time slot s0 is available for task S
<Step2>
Select time slot with smaller y-value
<Step3>
EST of task F is updated
Scheduled at the right most time slot
<Step4>
<Step5>
<Step6>
Select time slot with smaller y-value among schedulable duration
<Step7>
<Step8>

Figure 4. A trace of task schedule graph for workflow in figure 2(a) when RFT is 6 time units.

number of ancestors is not larger than that of descendants (step 1). Due to the scheduling of task 4, LFT of task S is changed to 1 and task S must be scheduled at s0 (step 2). Next, task 5 is selected and scheduled at s3 which is the leftmost time slot with the smallest height between its EST and LFT (step 3). This placement updates EST of task F to 5. Then, BTS selects task F and schedules it at the rightmost time slot (s6) because task F has more ancestors than descendants (step 4). Remaining tasks are scheduled at time slots with the smallest height (step 5 through 8). BTS allocates another host to schedule task 2 at step 6 since all the time slots on the first host in its schedulable duration are already occupied.

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 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. For example, if the size of unit time is 10 seconds and RFT is 10 hours, the number of time slots is 3600.

4.2.3 Task Redistribution

The task placement technique can fail to find a global optimum for workflows with certain structures. For example, Figure 5 illustrates a typical workflow that repeats a series of parallel and serial executions, the scheduling result of our placement algorithm, and on optimal solution. Let the execution time be 1, the data transfer time, zero, and RFT, 10. As shown in the graphs, our placement algorithm overestimates the resource capacity by 1 because it fails to balance the resource utilization across hosts. Specifically, task 8 is randomly scheduled as late as possible because the number of ancestors is equal to the number of descendants by the placement rule 3. Accordingly, the first host does not have enough slack time to schedule the tasks at the sixth level of the workflow. However, if task 8 is scheduled a little bit earlier, as the optimal solution shows, we can have a better result.

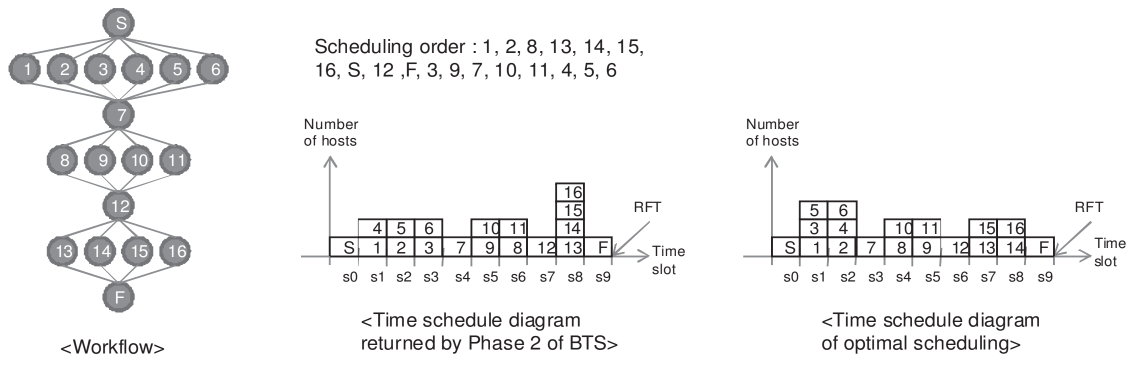

S
1 2 3 4 5 6
7
8 9 10 11
12
13 14 15 16
F
<Workflow>
Scheduling order : 1, 2, 8, 13, 14, 15, 16, S, 12 ,F, 3, 9, 7, 10, 11, 4, 5, 6
Number of hosts
RFT
Time slot
s0 s1 s2 s3 s4 s5 s6 s7 s8 s9
<Time schedule diagram returned by Phase 2 of BTS>
Number of hosts
RFT
Time slot
s0 s1 s2 s3 s4 s5 s6 s7 s8 s9
<Time schedule diagram of optimal scheduling>

Figure 5. Task schedule graph generated through the Task Placement phase and the optimal schedule for an example workflow. Each task is identical and has (ET=1, DTT=0). (RFT = 10)

Instead of introducing more complexity into the placement algorithm, we introduce another step for balancing resource utilization. This imbalance is mainly due to the placement rule that places a task as early or late as possible, depending on the number of ancestors and dependents. Hence, the main idea of task redistribution is to move tasks in the tallest time slot to adjacent time slots one by one within its schedulable duration and to see if it reduces the number of resources or not.

A high-level description of our task redistribution algorithm is shown in Figure 6. The first non-propagated redistribution step adjusts the start time of tasks in the tallest time slots without affecting other tasks. Then, Step 2 and 3 move tasks in the tallest 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.

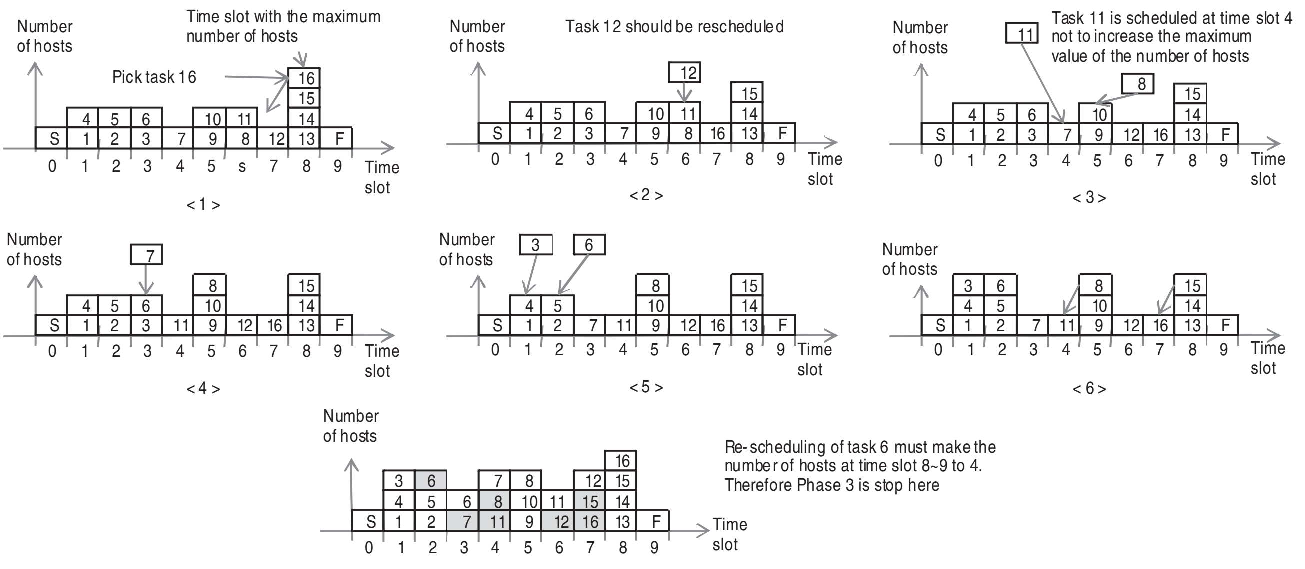
Figure 7 illustrates how our algorithm balances the tasks presented in Figure 5. Since the tallest slot is slot 8, BTS first checks if any tasks scheduled in the slot can be relocated via Non-propagated Redistribution. However, BTS cannot find any lightly loaded slots to relocate the tasks via this simple redistribution. Then, BTS tries to move tasks to the earlier time slots. BTS selects task 16 of the tallest time slot (since task 13, 14, 15, and 16 share the same ancestors and the UpLength values of them are same, task 16 is randomly selected) and moves it into the time slot 7. Since task 12 resides in the slot 7 and it is a parent of task 16, it must be relocated to slot 6. Even though this relocation increases the height of slot 6 to 3, task 12 can be safely moved to the slot since the slot is not taller than the tallest. Repeatedly, task 11 and 8 must be relocated to slot 5 in this order since they are the parents of task 12. However, the relocation of task 8 after the relocation of task 11 to slot 5 increases the height of slot 5 to 4, which makes it taller than the tallest slot. Therefore, task 8 is moved one more time into slot 4.

```
1. Non-propagated Redistribution
   a. Select time slot(s) with maximum NH.
   b. Find tasks in the time slot which can be moved to other time slot with lesser NH within its schedulable duration
   c. Repeat step b until there is no movable tasks
2. Thrust to the earlier slots
   a. Select the earliest one among time slot(s) with maximum NH.
   b. Select the task(t) with the least number of ancestor tasks while satisfying $UpLength_t + ET_t \le Scheduled\ Start\ Time(t)$
   c. Call ThrustLeft( t, start time of the selected time slot)
3. Thrust to the later slots
   a. Select the latest one(l) among time slot(s) 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 minimum x which satisfy $\max_{i \in TimeSlot(x,x+ET_t)} \{NH(i)\} + 1 < \max_{i \in TimeSlot(0,RFT)} \{NH(i)\}$   where $b \le x \le RFT - DownLength_t - ET_t$
  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 6. Algorithm description of task redistribution phase.



Number of hosts
Time slot with the maximum number of hosts
Pick task 16
Time slot
< 1 >
Task 12 should be rescheduled
< 2 >
Task 11 is scheduled at time slot 4 not to increase the maximum value of the number of hosts
< 3 >
< 4 >
< 5 >
< 6 >
Re-scheduling of task 6 must make the number of hosts at time slot 8~9 to 4. Therefore Phase 3 is stop here

Figure 7. A simplified trace of task schedule when Task Redistribution is applied to the workflow in Figure 5.

Again, task 7 in slot 4 which are the parent of task 8 is relocated to slot 3 and then task 6 and 3 are moved into slot 2. Since the relocation of task 6 and 3 makes the slot 2 taller than 3, task 3 is moved one more time into slot 1. Now, we have a new stable task schedule graph, so we can repeat the entire task redistribution from the beginning. Through the second iteration, BTS relocates task 8 and 15 to slot 4 and 7, respectively and finally it realizes that any redistribution methods cannot reduce the heights of the tallest slots, i.e., as in the last graph of Figure 7 where task 6, 7, 8, 11, 12, 15, and 16 are moved from grey boxes to white boxes, attempt to reduce the height of slot 2 make the height of slot 4 higher. In consequence, BTS concludes that it needs 3 hosts to execute the workflow in 10 time units.

5 Methodology

5.1 Target Resources

Our resource capacity estimate is a core component in the automatic synthesis of resource specifications from application workflows. This estimate is totally based on the structure and attributes of application workflows, independent of selection mechanisms or target resources while resource brokers or provisioning systems are in charge of providing the resources that satisfy the requirements. As such, we assume virtual homogeneous resources as target resources, which have identical computing power and network performance.

5.2 Application Workflows

5.2.1 Synthetic Workflows

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

- ***Fully Random Workflows (FRW):*** Workflows in this group do not have any constraints on the structure; any tasks can be connected to any other tasks. Every task is a child of the entry task and a parent of 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 the tasks only 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 tasks in the *ith* level are the parents of the tasks in the *(i+1)th* level. In the same manner to FRW, the execution time is selected via uniform random trials. We also consider two cases that the execution time of tasks in the same level is homogeneous and heterogeneous.

5.2.2 Real Application Workflow

In addition to synthetic workflows, we use two real workflows: Montage [15] and EMAN [23]. Montage creates custom image mosaics of the sky on demand and consists of four major tasks: re-projection, background radiation modeling, rectification and co-addition. EMAN is a suite of scientific image processing tools aimed primarily at the transmission electron microscopy community. The structure of these workflows are depicted in Figure 8. These workflows are a type of LWP. We observed that many workflows of real applications including SCEC [25] and LEAD [2] have similar structures.

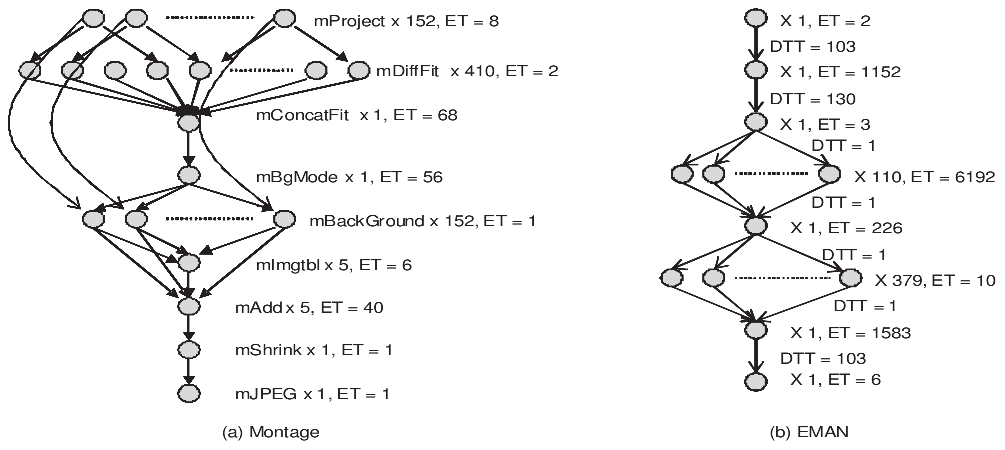


Figure 8. Workflows of Montage and EMAN application.

5.3 Metrics

BTS algorithm estimates the resource capacity required to finish a workflow application within a RFT. Since we employ a heuristic approach to solve NP-hard problems, the first metric of interest to evaluate algorithms is the overestimated amount of resources, compared to the optimal solution. Second, as advanced workflow planning infrastructures enable exploring complex application workflows and dynamic resource provisioning systems facilitate large-scale distributed resources transparently, the scalability of algorithm is critical to adapting to emerging computing environments.

5.4 Algorithms in Comparison

The existing workflow scheduling algorithms can be also used to estimate the resource capacity. We evaluate the efficiency of our algorithm by comparing to three possible approaches.

- ***FU (Full Utilization without dependencies):*** 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, this algorithm can calculate the lower bound of resource capacity with constant time.

- ***DSC (Dominant Sequence Clustering) :*** DSC[34] is one of the most popular algorithms for scheduling over unbounded resources. The main goal of DSC is to minimize makespan, assuming there is no resource limit. In particular, DSC returns the makespan, the number of clusters, a mapping of tasks to clusters, and task execution sequences inside the clusters. We can use the number of clusters as the number hosts required when RFT is equal to the minimum makespan. In DSC, all tasks have its own cluster at first and tasks are merged with one of its parents' clusters if the merging does not increase the makespan. The order of tasks to be merged is determined by the priority value calculated dynamically using EST and DownLength.

- ***IterHEFT (iterative search with HEFT algorithm):*** HEFT[33] is one of workflow scheduling algorithms which calculate the makespan of workflow over given hosts. HEFT picks the task with the largest value of $(ET + DownLength)$ 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 scheduling over a growing resource set until the resulting makespan is equal to or shorter than RFT. We can reduce the search time by setting the initial resource set size by the number of hosts calculated by FU. Due to iterations, the time complexity of this approach is $O(n^4)$ in the worst cases.

6 Experiments

6.1 Synthetic Fully Random Workflows

We evaluate the quality of BTS with respect to turnaround time and the estimated number of hosts, comparing to three other approaches. The resource capacity estimate of four approaches with a variety of synthetic random workflows is shown in Figure 9. We present 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 in the Figure. The minimum execution time of algorithms is the longest path from the entry task (S) to the exit task (F).

BTS achieves a slightly better estimate than IterHEFT while it outperforms DSC because DSC focuses on minimizing the makespan, not on minimizing the number of clusters. Considering the estimate of FU being the low bound, both BTS and

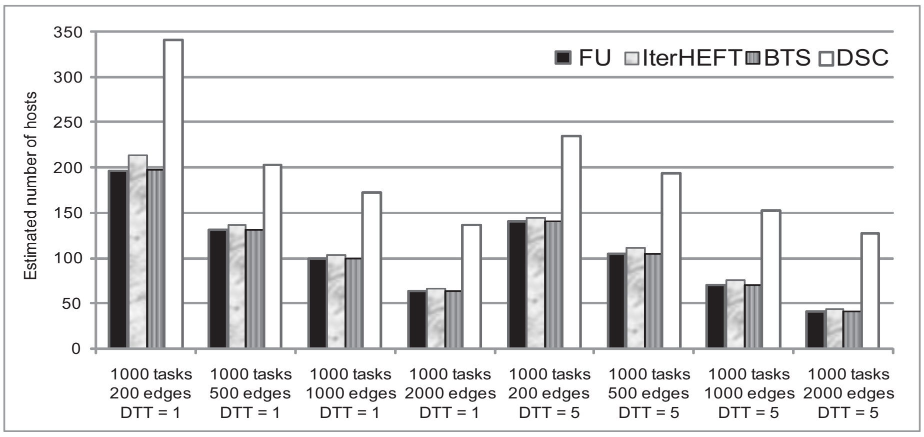


Figure 9. Comparison of four estimation methods for minimum execution time of fully random workflows. DTT is the data transfer time between tasks.

IterHEFT achieve good quality of estimate for fully random workflows. The reason BTS achieves slightly better estimate than IterHEFT is that HEFT schedules tasks as early as possible so that it can make the idle time fragmented while BTS more flexibly determines task schedule in schedulable duration to minimize the number of hosts. For example, Figure 10 illustrates how different the task schedules of BTS and HEFT are. The workflow consists of 6 tasks and their execution times are shown in Figure 10 (a). We assume that the data transfer times are zero and the deadline of workflow is 5. As in Figure 10 (b), HEFT answers that the makespan is 6 with two hosts and consequently IterHEFT will say that 3 hosts are required to meet the deadline. This is because task 3 is scheduled at the earliest available time of host 1 even though it can be delayed to time 2 without extending the makespan. As a result, the time period from 0 to 1 of host 1 is wasted and task 2 is scheduled after task 3 is finished. By contrast, BTS finds more efficient task schedule since it knows that task 3 can be delayed by time 5.

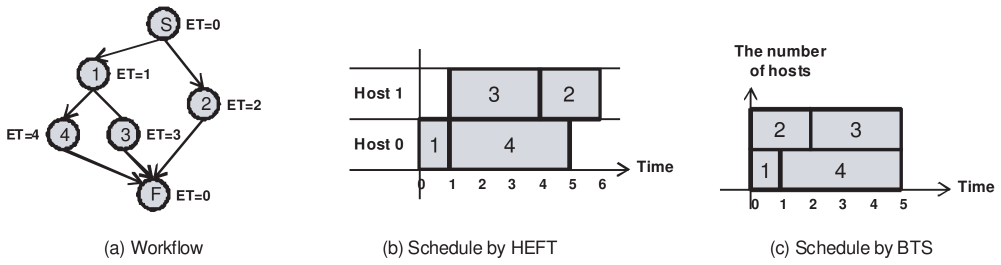


Figure 10. An example case of overestimation of HEFT.

We conduct a set of experiments to see how RFT affects the estimate quality of BTS and the results are presented in Figure 11. The X-axis represents the ratio of RFT to the minimum execution time and Y-axis does the average number of hosts estimated for 30 random workflows. Regardless of RFTs, BTS consistently estimates the resource capacity close to the lower bounds. Even though we do not present the results, we observed the similar results for workflows with different characteristics.

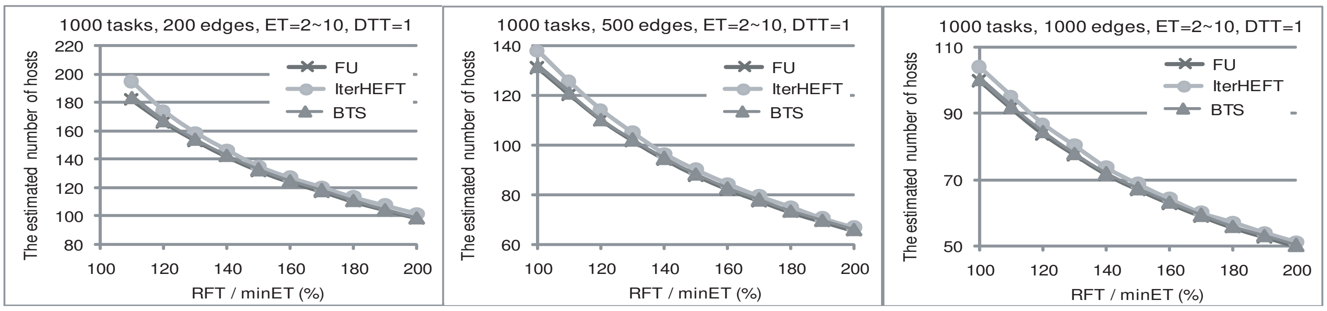


Figure 11. Estimation result for various RFTs. ET is the execution time of tasks, DTT is the data transfer time between tasks.

Figure 12 summarizes the turnaround time of three algorithms, measured on a PC having a 3Ghz CPU and 2GB RAM. The result shows that BTS takes less than 1 minute even with large workflows having thousands of tasks and edges while IterHEFT takes more than one hour. BTS is more efficient method than IterHEFT, considering both the quality and the cost of estimate.

Workflow complexity	IterHEFT	BTS	DSC
1000 nodes 1000 edges	9.2	1.2	0.2
2000 nodes 2000 edges	84.4	7.6	1.7
5000 nodes 5000 edges	3914	36.1	18.9

Figure 12. Comparison of time cost (in seconds).

6.2 Synthetic Leveled Parallel Workflows

We evaluate BTS and IterHEFT against leveled parallel workflows consisting of tasks with non-identical execution time. Each workflow has 1000 tasks and 10 levels and the execution time of each task ranges from 3 to 10 for the graphs at the top of Figure 13 and from 7 to 10 for the graphs at the bottom of the Figure. Data transfer time is 1 for all cases. As shown in Figure 13, BTS and IterHEFT produce nearly the same estimate. 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. In the meantime, 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.

Even though we do not present the results due to the space limit and the similar patterns of results, we also evaluated BTS and IterHEFT against workflows consisting of tasks with the same execution time. Both approaches estimate exactly the same number of hosts. In addition, the experiments to evaluate the effects of the maximum parallelism, data transfer time, and the number of levels on the quality of estimate showed that BTS and iterHEFT achieved almost same performance, finding near optimal solutions.

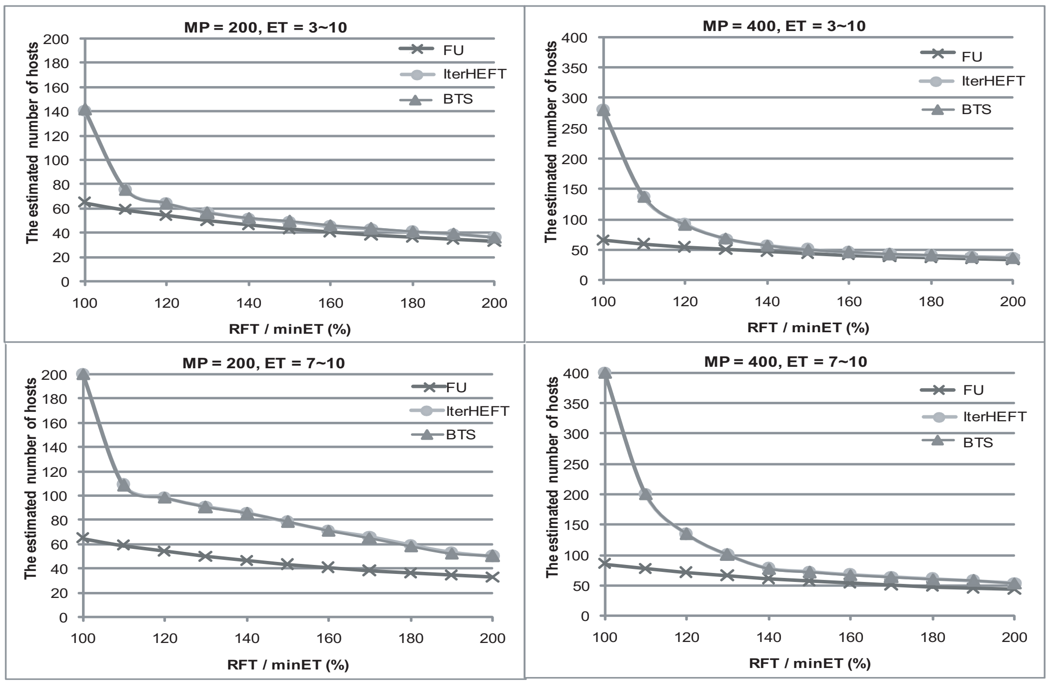


Figure 13. Comparison of BTS and IterHEFT for leveled parallel workflow with non-identical tasks.

6.3 Real Application Workflows

Finally, we evaluate our algorithm against two real application workflows: Montage and EMAN. As shown in Figure 8, these workflows are a type of LPW and the execution time of the tasks at the same level is identical. Regardless of input data, the workflows have the same structure even though the number of parallel tasks can vary. EMAN has a longer makespan than Montage; the number of time slots for Montage ranges between 200 and 400 while that for EMAN ranges between 9000 and 18000. As for the Montage workflow, we set data transfer time to 5 seconds since the size of the intermediate data produced by the application is at most several megabytes.

The resource capacity estimates for Montage and EMAN are shown in Figure 14. BTS takes 1.1 seconds for Montage and 8.2 seconds for EMAN. As expected, the quality of estimates of IterHEFT and BTS are the same because both applications have a level parallel workflow. Note that the amount of resources required for both applications dramatically decreases only with small increase of RFT. The users of these applications can save significant amount of resource consumption when the deadline is relaxed. This is particularly important when resources such as those provided by Amazon are used and where the cost is measured in dollars.

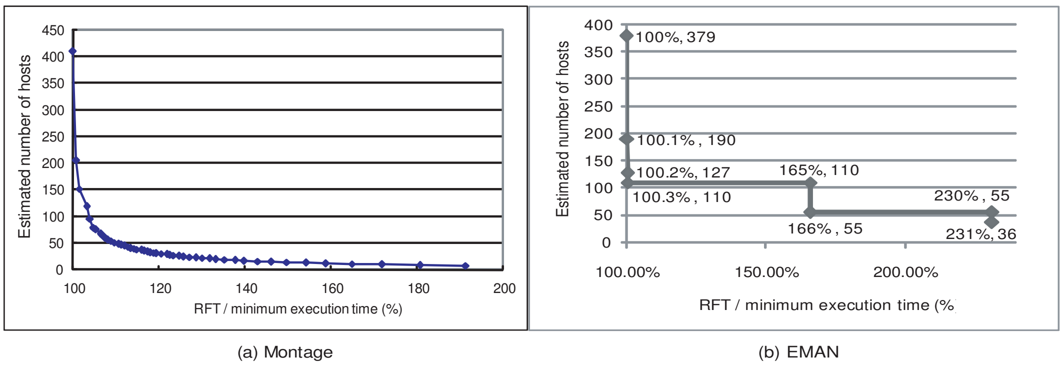


Figure 14. Estimate result of BTS for Montage and EMAN workflow (unit time = 1 second). 100% corresponds to the RFT=makespan, 200% when RFT= 2*makespan.

7 Conclusions

In this paper, we propose a new algorithm named BTS to estimate 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 such as for example the emerging cloud computing resource providers. Moreover, the resource estimate is abstract and independent to the resource selection mechanism, so it can be easily integrated with any resource description languages and resource provisioning systems. Through the experiments with synthetic and real workflows, we demonstrate 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.

In this study, we assume that all resources required for a workflow are available throughout the lifetime of application. However, holding all resources during the entire lifespan can cause the resources to be underutilized. For instance, Figure 5 shows that we need only 2 hosts from the time slot 3 through the end of execution. If we allocate 3 hosts only for first 3 time slots and then release 1 host after that, we can save 1 host, which leads to better resource utilization. Resource provisioning techniques such as Virtual Grid provide fine-grained time-based resource reservation. We believe an extension of our algorithm can exploit such advanced features of provisioning systems and enable more cost-efficient workflow execution. In the future, we plan on integrating the Pegasus workflow management system with the Virtual Grid to evaluate the approach in real settings.

References

[1] Amazon elastic compute cloud (amazon ec2). aws.amazon.com/ec2.

[2] Linked environments for atmospheric discovery. http://lead.ou.edu/.

[3] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[4] A. Anjonshoaa, F. Brisard, M. Drescher, D. Fellows, A. L. ans S. McGough, and D. Pulsipher. Job submission description language (jsdl) specification, version 1.0. *GFD-R.056, http://forge.gridforum.org/projects/jsdl-wg*, Nov 2005.

[5] J. Blythe, S. Jain, E. Deelman, T. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. In *Proceedings of IEEE International Symposium on Cluster Computing on Grid*, 2005.

[6] R. Buyya, M. Murshed, D. Abramson, and S. Venugopal. Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost-time optimization algorithm. *Software-Practice and Experience*, 35:491–512, 2005.

[7] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science, IPPS/SPDP'98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1459:62–82, 1998.

[8] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[9] F. Dong and S. G. Akl. 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.

[10] I. Foster and C. Kesselman. *The Grid: Blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., 1998.

[11] 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.

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

[13] P. M. Institute. *A Guide To The Project Management Body Of Knowledge, 3rd ed.* Project Management Institute, 2003.

[14] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. Yocumet. Sharing Networked Resources with Brokered Leases. In *Proceedings of USENIX Technical Conference*, 2006.

[15] J. C. Jacob, D. S. Katz, and T. Prince. The Montage Architecture for Grid-Enabled Science Processing of Large, Distributed Datasets. In *Proceedings of the Earch Science Technology Conference(ESTC2004)*, 2004.

[16] Y.-S. Kee and C. Kesselman. 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.

[17] Y.-S. Kee, D. Logothetis, R. Huang, H. Casanova, and A. A. Chien. 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.

[18] Y.-S. Kee, K. Yocum, A. A. Chien, and H. Casanova. Improving Grid Resource Allocation via Integrated Selection and Binding. In *Proceedings of ACM/IEEE International Conference on High Performance Computing and Communication (SC'06)*, 2006.

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

[20] J. Liou and M. A. Palis. A comparison of general approaches to multiprocessor scheduling. In *Proceedings of the 11th International Symposium on Parallel Processing*, 1997.

[21] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988.

[22] C. Liu and I. Foster. A Constraint Language Approach to Grid Resource Selection. In *Proceedings of 14th International Workshop on Research Issues on Data Engineering: Web Services for e-Commerce and e-Government Applications*, 2004.

[23] S. J. Ludtke, P. R. Baldwin, and W. Chiu. EMAN: Semiautomated Software for High-Resolution Single-Particle Reconstructions. *Journal of Structural Biology*, 128:82–97, 1999.

[24] S. V. M. Rahman and R. Buyya. 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.

[25] H. Magistrale, S. Day, R. W. Clayton, and R. Graves. The SCEC Southern California Reference Three-Dimensional Seismic Velocity Model Version 2. *Bulletin of the Seismological Society of America*, 90:S65–S76, 2000.

[26] D. A. Menasce and E. Casalicchio. 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.

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

[28] R. Sakellariou and H. Zhao. A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems. In *Proceedings of the IEEE Heterogeneous Computing Workshop*, 2006.

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

[30] G. Singh, C. Kesselman, and E. Deelman. Application-level Resource Provisioning on the Grid. In *Proceedings of Third IEEE International Conference on e-Science and Grid Computing*, 2006.

[31] A. Sudarsanam, M. Srinivasan, and S. Panchanathan. Resource estimation and task scheduling for multithreaded reconfigurable architectures. In *Proceedings of the 10th International Conference on Parallel and Distributed Systems*, 2004.

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

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

[34] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. on Parallel and Distributed Systems*, 5(9):951–967, 1994.

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

[36] J. Yu and R. Buyya. *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.

[37] J. Yu, R. Buyya, and C. K. Tham. 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.