Resource Minimization for Real-Time Applications Using Computer Clouds

Hao Wu, Xiayu Hua, Zheng Li and Shangping Ren
Illinois Institute of Technology
10 W 31st street, 013
Chicago, IL, USA
{hwu28, xhua, zli80, ren}@iit.edu

***Abstract*—In this paper, we address the resource minimization problem for DAG-based real-time applications using computer clouds to: (1) guarantee the satisfaction of a real-time application's end-to-end deadline; (2) ensure the number of computers allocated to the application is minimized; and (3) under allocated resources, minimize the application's makespan. We first give lower and upper bounds for resources needed to guarantee the satisfaction of a real-time application's deadline. Based on the bounds, we develop a heuristic algorithm called minimal slack time and minimal distance (MSMD) algorithm that finds the minimum number of computers needed to guarantee the application's deadline and schedules tasks on the allocated resources so that the application's makespan is minimized. Our experimental results show that the MSMD algorithm can guarantee applications' end-to-end deadlines with less resources compared with other heuristic scheduling algorithms existed in the literature. In addition, under the minimal allocated resources, the MSMD algorithm can, on average, reduce an application's makespan by 10% of its deadline.**

I. Introduction

The advancement of computer and network technology has brought the world into a new era of computer clouds. The "pay-as-you-go" business model and the service oriented models allow users to have "unlimited" resources if needed and free from infrastructure maintenance and software upgrades. Cloud services are currently among the top-ranked high growth areas in Computing and are seeing an acceleration in enterprise adoption with the worldwide market predicted to reach more than $131b in 2013 [19], [1]. Many different types of applications have been deployed on clouds. For instance, both Argonne National Laboratory and Fermi National Accelerator Laboratory provide their cloud platforms for scientific applications [10], [5], real-time applications such as online media streaming applications [15], interactive real-time e-learning [2], and online banking systems [18] are also seeking opportunities to utilize computer clouds.

Although the new cloud technology and the "pay-as-you-go" business model have brought new opportunities to many application domains, there are many technical challenges yet to be solved for service providers. One of the major issues faced by all service providers is, under their limited resources, how to guarantee the quality of services provided to their users and at the same time make maximum profit. There are two main routes to reach this goal, i.e., via maximizing system throughput or minimizing operational cost. Researchers have made significant efforts to maximize throughput by maximizing resource utilization [13] and minimizing applications' makespan [20], [4], [14], and to minimize operational cost by reducing power/energy consumption [12], [17]. However, the research on maximizing system throughput often focuses on when the number of resources are fixed and known. For instance, for the application makespan minimization problem that many researchers have been working on [20], [4], [14], their solutions are based on the assumption that the available resources are given.

However, in a cloud environment, the number of computers can be dynamically configured based on the user's demand. For a real-time application, meeting the application's deadline requirement is critical, but there is no incentive to finish the application earlier. On the other hand, from service provider's perspective, reducing applications' makespan cannot only save computer power consumption cost, but also can increase the system's throughput. Hence, if the cloud can guarantee the application's deadline requirement with minimal number of resources and reduce application's makespan without increasing the minimal number of resources, both application clients and the computer cloud service providers will benefit the most.

The work presented in this paper addresses the resource minimization issue for real-time applications using computer clouds. In particular, for a given distributed real-time application with an end-to-end deadline constraint and a computer cloud with an *unspecified* number of computers, decide the number of computers needed and a schedule on each computer to guarantee: 1) the application's end-to-end deadline is satisfied, 2) the number of computers needed for executing the application tasks is minimized, and 3) under the minimized number of computers, the application's makespan is minimized.

The remainder of the paper is organized as follows: Section II discusses related work. In Section III, we first introduce the models and terms, then formally define the resource minimization problem the paper is to address. Section IV presents an analysis to quickly calculate the resource bounds needed to guarantee a DAG-based real-time application's end-to-end deadline. Section V gives a heuristic algorithm to solve the problem defined in Section III. Experimental evaluations are presented in Section VI. We conclude and point out future

The research is supported by in part by NSF under grant number CAREER 0746643 and CNS 1018731. We would like to thank Daniela Martinez for proof reading the paper and reviewers for their constructive and valuable comments that have improved the paper.

work in Section VII.

II. Related Work

The essence behind resource minimization and application makespan minimization problems can be drilled down to a task scheduling problem which is proven to be NP-complete when there are more than two computers [7]. Thus, many heuristic approaches have been proposed. List scheduling is one of the basic approaches used for makespan minimization and it has a $(2-\frac{1}{m})$ approximation to the optimal makespan [8], where m is the number of processors. The idea of list scheduling is to list tasks in an order and then schedule tasks from the ordered list. Hence, ordering the task list becomes critical when designing list based algorithms.

Researchers have made significant efforts on ordering tasks and have developed many list scheduling based heuristic algorithms to solve application makespan minimization problems [6], [20], [4]. A well-known list scheduling based algorithm is the Coffman-Graham (CG) algorithm [6]. The CG algorithm takes a set of partially ordered tasks and assigns task priorities based on their order. The CG schedules the task with the highest priority in the list to the computer that has the earliest available time at the time of scheduling. When there are only two homogeneous computers, the CG scheduling algorithm is proven to be optimal [6].

However, the CG algorithm cannot be directly applied to DAG-based applications unless the dependencies among tasks in a DAG-based application are resolved. One commonly used approach to resolve task dependencies is to list the DAG-based application in a topological order. Another commonly used approach is to assign and list tasks by their priorities. The prioritization scheme is based on when each task finishes, i.e. counting from the bottom of the DAG-based application task graph (b_{level}), or when each task starts, i.e. counting from the top (t_{level}). Many existing makespan minimization algorithms adapted this approach as their prioritization basis. The heterogeneous-earliest-finish-time (HEFT) algorithm [20] is one of them and it uses the summation of a task's b_{level} and t_{level} values as its priority and hence provides a $O(|V|^2 m)$ list-based heuristic algorithm for minimizing the makespan.

Lee et al. improved the HEFT algorithm by taking the energy consumption into consideration and proposed an energy conscious scheduling (ECS) algorithm [11]. The ECS algorithm has a similar performance on makespan reduction compared to HEFT, but ECS reduces energy consumption by 10%. The duplication based bottom up scheduling (DBUS) algorithm [4] is another heuristic algorithm that takes the b_{level} and t_{level} approach as the basis of its prioritization method. Unlike the HEFT algorithm, the DBUS approach takes the t_{level} and an additional static top level st_{level} as the tasks' priorities. The DBUS also duplicates tasks on each machine at the scheduling phase and thus is a $O(|V|^2 m^2)$ heuristic.

The research briefly summarized above has been mainly focused on how to schedule tasks *under fixed amount of resources* to maximize the system's throughput and minimize an application's makespan, rather than to minimize the number of computers needed to guarantee a real-time application's deadline. In fact, it is possible that an application can be scheduled on n computers with the same makespan as on m $(m > n)$ computers by different heuristic algorithms.

Research on resource bound problem for DAG-based real-time applications started in the early 1960's [9] and we have obtained some results since then [16], [3]. However, neither T.C. Hu's original lower bound [9] nor Ramamoorthy's improvement [16] can be directly applied to the DAG-based application with different task execution times if tasks are not allowed to migrate among computers. Al-Mouhamed further extended and improved the resource lower bound with the consideration of heterogeneous task execution time and communication cost [3]. However, Al-Mouhamed's method to calculate the lower bound is too expensive to be applicable in practice for large scale applications and for on-line cloud applications.

In this paper, we give resource bounds that though may not be as tight as desired, they can be quickly calculated. In addition, we present a heuristic algorithm to find the minimal number of computers needed and schedule application tasks on the allocated minimal number of computers so that the applications' makespan is also minimized. The CG, HEFT, DBUS and ESC algorithms from the literature will be used as base lines to evaluate our proposed approach.

III. Problem Formalization

In this section, we first introduce models and terms the work is based on and then formulate the resource minimization problem the paper is to address.

A. Application Model

A distributed real-time application A is modeled as a weighted direct-acyclic-graph (DAG) $G(V, E)$, where each task $\tau_i \in A$ is represented by a node $v_i \in V$, the weight $w(\tau_i)$ on the node v_i represents task τ_i's worst execution time (WECT) on a unit speed computer, and an edge $(v_i, v_j) \in E$ represents dependency between τ_i to τ_j. A task can only start after all its predecessors complete. Each application is given a release time T_R and a relative end-to-end deadline T_D. Hence, the application's absolute end-to-end deadline is $T_R + T_D$.

Tasks without any predecessors or any successors are defined as *entry* tasks and *exit* tasks, respectively. For convenience, we assume each application has one entry task denoted as τ_{entry} and one exit task denoted as τ_{exit} [1]. Fig. 1 gives an example of a DAG-based application task graph, where $\tau_{entry} = \tau_0$ and $\tau_{exit} = \tau_{10}$.

B. System Model

A computer cloud is modeled as a set of networked computers, i.e. $C = \{c_1, \ldots, c_M\}$. We assume computers in the cloud are homogeneous with unit speed. Hence task execution time does not change when it is deployed to different computers in the cloud. Task executions are non-preemptive and each computer can only execute one task at any given

[1] If an application has multiple entry tasks or multiple exit tasks, we add a *virtual entry* task and a *virtual exit* task with zero execution time and connect from the virtual entry task to all actual entry tasks and from all actual exit tasks to the virtual exit task, respectively.

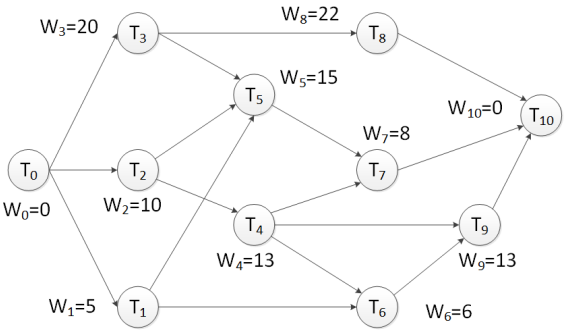


Fig. 1: An Example of DAG-Based Application Task Graph with End-to-End Deadline D=60

time. Unprocessed tasks are buffered in a task queue by the computers the tasks are assigned to.

For a given application $A = \{\tau_1, \ldots, \tau_n\}$ and a set of computers where application tasks are deployed on, a Boolean function $S(\tau_i, c_j)$ defines whether a task $\tau_i \in A$ is deployed on computer c_j. In other words, if a computer c_j is used by application A, then $\exists \tau_i \in A$ such that $S(\tau_i, c_j) = 1$. We use $N(A)$ to denote the resource set utilized by application A:

$$N(A) = \{c_j | \exists \tau_i \in A, c_j \in C \wedge S(\tau_i, c_j) = 1\} \quad (1)$$

C. Definitions

Given a DAG-based application $A = \{\tau_1, \ldots, \tau_n\}$ with release time T_R and relative deadline T_D, and its corresponding task graph $G(V, E)$, we define the following terms.

Predecessors ($pred(\tau_i)$) and Successors ($succ(\tau_i)$): For task $\tau_i \in V$, its predecessor and successor task sets are defined below:

$$pred(\tau_i) = \{\tau_j | \tau_j \in A \wedge (\tau_j, \tau_i) \in E\} \quad (2)$$

$$succ(\tau_i) = \{\tau_j | \tau_j \in A \wedge (\tau_i, \tau_j) \in E\} \quad (3)$$

Application Sequential Execution Time (T_{seq}): The sequential execution time of application A is defined as the summation of all composing tasks' execution time.

Task Earliest Start Time ($EST(\tau_i)$) and Latest Finish Time ($LFT(\tau_i)$): For a given task $\tau_i \in V$, its earliest start time and latest finish time are recursively defined as follows:

$$EST(\tau_i) = \begin{cases} T_R & \text{if } \tau_i = \tau_{entry} \\ \max\limits_{\tau_k \in pred(\tau_i)} \{EST(\tau_k) + w_k\} & \text{otherwise} \end{cases} \quad (4)$$

$$LFT(\tau_i) = \begin{cases} T_R + T_D & \text{if } \tau_i = \tau_{exit} \\ \min\limits_{\tau_k \in succ(\tau_i)} \{LFT(\tau_k) - w_k\} & \text{otherwise} \end{cases} \quad (5)$$

A task's earliest start time and latest finish time are only based on the application's task graph, its release time and relative deadline. They are independent of how tasks are assigned to computers.

Task Actual Start Time ($AST(\tau_i)$) and Actual Finish Time ($AFT(\tau_i)$): A task's actual start time ($AST(\tau_i)$) and actual finish time ($AFT(\tau_i)$) are defined as when the task is dispatched for execution and completed its execution on a computer, respectively. It can be different from its earliest start time and latest finish time. In fact, we have $AFT(\tau_i) = AST(\tau_i) + w_i$ and $AST(\tau_i) \geq EST(\tau_i)$.

Task Ready Time ($ready(\tau_i)$): A task's ready time $ready(\tau_i)$ is the latest actual finish time of all its predecessors. The definition is given below:

$$ready(\tau_i) = \max_{\tau_k \in pred(\tau_i)} \{AFT(\tau_k)\} \quad (6)$$

Task Maximal Slack Time ($mslack(\tau_i)$): For a given task $\tau_i \in V$, its maximal slack time is defined as

$$mslack(\tau_i) = LFT(\tau_i) - (EST(\tau_i) + w(\tau_i)) \quad (7)$$

Intuitively, the maximal slack time indicates how long a task can afford to wait before causing a deadline violation. For task τ_i, $mslack(\tau_i) = 0$ means τ_i must start at its earliest start time or it will cause the application to miss its end-to-end deadline.

Task Topological Level ($Lev(\tau_i)$): Given a DAG-based application A, its task τ_i's topological level $Lev(\tau_i)$ is defined as:

$$Lev(\tau_i) = \begin{cases} 0 & \text{if } \tau_i = \tau_{entry} \\ \max\limits_{\tau_k \in pred(\tau_i)} \{Lev(\tau_k)\} + 1 & \text{otherwise} \end{cases} \quad (8)$$

Critical Path P_c and Critical Path Execution Time (T_C) : For a given application task graph, a path execution time is defined as the summation of its composing task's execution time. A critical path, denoted as P_c, is a path that starts at the entry task τ_{entry}, ends at the exit task τ_{exit}, and has the longest path execution time. There may exist more than one critical paths in an application's task graph. However, by definition, every critical path has the same path execution time. We denote the critical path execution time as T_C.

Schedulable Application : For a given application A with relative deadline T_D, the application is *schedulable* if and only if its critical path execution time satisfies $T_C \leq T_D$.

Computer Earliest Available Time ($av(c)$): For a given computer c, if its totally ordered task queue is $Q_c = \{\tau_1^c, \ldots, \tau_h^c\}$, then computer c's earliest available time for a new task τ_i not in the queue can be calculated as follow:

$$av(c) = AFT(\tau_h^c) = \begin{cases} 0 & n = 0 \\ \max\{AFT(\tau_{h-1}^c), ready(\tau_h^c)\} & \\ \quad + w(\tau_h^c) & n > 0 \end{cases} \quad (9)$$

Table I gives the EST , LFT , mslack , Lev , and whether a task is on a critical path for the example task graph given in Fig. 1. The concept of task priority will be discussed in Section V.

D. Problem Formulation

Based on the models and definitions presented in the earlier subsections, we formally define the problem we are to address, i.e. resource minimization for real-time applications using computer clouds. To achieve the goal, we take two steps. The first step is to minimize the number of computers needed to guarantee the satisfaction of a DAG-based real-time application's end-to-end deadline. Once the minimal number of

TABLE I: Example Application's Task Property

Tasks	EST	LFT	mslack	CP	Levels	Priority
τ_0	0	17	17	$\surd$	0	1
τ_1	0	37	32		1	4
τ_2	0	28	18		1	3
τ_3	0	37	17	$\surd$	1	2
τ_4	10	41	18		2	6
τ_5	20	52	17	$\surd$	2	5
τ_8	20	60	18		2	7
τ_6	23	47	18		3	9
τ_7	35	60	17	$\surd$	3	8
τ_9	29	60	18		4	10
τ_{10}	43	60	17	$\surd$	5	11

computers needed is decided, our second step is to minimize the application's makespan under the minimized number of resources decided by the first step.

Objective 1: Minimize the Number of Resources Needed

Given an application $A = \{\tau_{entry}, \cdots, \tau_i, \cdots, \tau_{exit}\}$ with release time T_R and relative deadline T_D, its corresponding task graph $G(V, E)$, and sufficient set of N computers, determine a subset of computers of size $M, M \leq N$, such that

$$\textbf{Objective 1:} \quad \min M$$

$$\textbf{Subject to:} \quad AFT(\tau_{exit}) \leq T_R + T_D \tag{10}$$

$$\textbf{and} \quad \forall \tau_i \in A, \sum_{j=1}^{M} S(\tau_i, c_j) = 1 \tag{11}$$

where $S(\tau_i, c_j) = 1$ if and only if task τ_i is assigned to computer $c_j \in C(A)$. The first constraint given by (10) guarantees end-to-end deadline satisfaction and the second constraint given by (11) ensures that each task can only be deployed to one computer.

Objective 2: Minimize Makespan under Allocated Resources

Once the minimum number of computers (M) needed to guarantee the application's end-to-end deadline is determined, our next task is to minimize the application's makespan on the M computers:

$$\textbf{Objective 2:} \quad \min \quad AFT(\tau_{exit})$$

$$\textbf{Subject to:} \quad \forall \tau_i \in A, \sum_{j=1}^{M} S(\tau_i, c_j) = 1 \tag{12}$$

IV. Resource Bounds

As stated in the previous section, one of our objectives is: for a given application, determine the minimum number of computers needed to guarantee the application meet its end-to-end deadline. In this section, we study the attributes of a DAG-based application and determine the bounds for the minimum number of computers needed.

Lemma 1: Given a DAG-based real-time application A, let the application's release time and relative end-to-end deadline be T_R and T_D, respectively, its sequential execution time be T_{seq}, critical path execution time be T_C, and the minimal number of computers needed to guarantee the application's end-to-end deadline be M. If the application is schedulable, i.e. $T_C \leq T_D$, then we have:

$$M \geq \lceil \frac{T_{seq}}{T_D} \rceil \tag{13}$$

□

Proof: We prove Lemma 1 by contradiction. If $T_{seq} \leq T_D$, we have $\lceil \frac{T_{seq}}{T_D} \rceil = 1$. As the application's sequential execution time is less than its deadline, i.e. $T_{seq} \leq T_D$, trivially, with $M = 1$ computer, we can guarantee the application's deadline. If $T_{seq} > T_D$, assume the minimum number of computers needed to guarantee $AFT(\tau_{exit}) \leq T_R + T_D$ is M'. Let $M' < \lceil \frac{T_{seq}}{T_D} \rceil$. Given M' computers, the best scenario is that the work load is evenly distributed to the M' computers and all tasks are executed without waiting. Under such scenario, the application's makespan is $T_R + \frac{T_{seq}}{M'}$, which is the earliest possible time the application can complete. Hence, we have $AFT(\tau_{exit}) \geq T_R + \frac{T_{seq}}{M'}$. Since M' is a positive integer, we have $M' < \frac{T_{seq}}{T_D}$, which implies $T_D < \frac{T_{seq}}{M'}$.From the conclusion $AFT(\tau_{exit}) \geq T_R + \frac{T_{seq}}{M'}$, we have $AFT(\tau_{exit}) > T_R + T_D$, contradicting the assumption that $AFT(\tau_{exit}) \leq T_R + T_D$. □

Lemma 2: Given a DAG-based real-time application A, let the application's release time and relative end-to-end deadline be T_R and T_D, respectively, its corresponding task graph $G(V, E)$, critical path execution time be T_C, level of *exit* task $Lev(\tau_{exit})$ and the number of computers needed to guarantee the application's end-to-end deadline be M. If the application is schedulable, i.e. $T_C \leq T_D$, then we have:

$$M \leq |V| - Lev(\tau_{exit}) \tag{14}$$

where $|V|$ is the number of tasks in the application. □

Proof: It is obvious that if each task is scheduled to an idle computer and each computer only executes one task, application A can finish with a makespan of T_C. Since $T_C \leq T_D$, application A can finish before its end-to-end deadline under $|V|$ computers. Based on the definition of task's topological level given in Section III, there must exists a path P_i that consists of at least $Lev(\tau_{exit}) + 1$ tasks and no two tasks are from the same level. Since path P_i must be sequentially executed, dispatching all tasks on P_i to the same computer will not affect the application's makespan. Hence, we can at least reduce $Lev(\tau_{exit})$ computers from total $|V|$ computers. As a result, $M = |V| - Lev(\tau_{exit})$ computers are sufficient to guarantee an application's end-to-end deadline. □

Combining Lemma 1 and 2, we have the following theorem.

Theorem 1: Given a DAG-based real-time application A, let the application's release time and relative end-to-end deadline be T_R and T_D, respectively, its corresponding task graph $G(V, E)$, sequential execution time be T_{seq}, critical path execution time be T_C, level of *exit* task $Lev(\tau_{exit})$ and the minimal number of computers needed to guarantee the application's end-to-end deadline be M. If the application is schedulable, i.e. $T_C \leq T_D$, we have:

$$\lceil \frac{T_{seq}}{T_D} \rceil \leq M \leq |V| - Lev(\tau_{exit}) \tag{15}$$

□

In the next section, we present a heuristic scheduling algorithm based on the theorem.

V. Minimal Slack Time and Minimal Distance (MSMD) based Scheduling

In this section, we introduce a heuristic approach for the resource minimization problem formulated in Section III-D. The basic idea of our heuristic approach is to search for a schedule that satisfies a given application's deadline from the minimal number of resources given by (13). Once a schedule is found, the number of computers used is the least. The search for a possible schedule has two phases: task prioritization phase which is based on an application tasks' topological levels and slack time, and task scheduling phase which is based on the minimal distance between the resources' available time and the tasks' ready time.

Given an application's task graph G, Algorithm 1 outlines our heuristic approach, where Line 1 prioritizes tasks in the given application; Line 3 to Line 8 search for minimal resources needed to satisfy the deadline using the *minimal slack time and minimal distance* (MSMD) heuristic scheduling algorithm. We discuss task prioritization and the MSMD scheduling algorithm in the next two subsections.

Algorithm 1: Schedule Searching

Input : Application: $G(V, E)$
Output: A schedule satisfies $AFT(\tau_{exit}) \leq T_R + T_D$

1 $L \leftarrow prioritize(G)$// Ordered list;
2 $min \leftarrow \lceil \frac{T_{seq}}{T_D} \rceil$;
3 **do**
4 $\quad T[min] \leftarrow \{0\}$ // $av(c)$;
5 $\quad S[min] \leftarrow \{\emptyset\}$ // Computers' job queues;
6 $\quad AFT(\tau_{exit}) \leftarrow MSMD(G, T[min], S[min], L)$;
7 $\quad min++$;
8 **while** $AFT(\tau_{exit}) \leq T_R + T_D$;
9 **return** $S[min]$

A. Minimal Slack Time based Prioritization

The goal of task prioritization is to assign a priority to every task in the application. To ensure task dependency relations are not violated, we assign task priorities based on task topological levels and their minimal slack time. In particular, tasks at a lower topological level have higher priorities than tasks at a higher level; for tasks at the same topological level, tasks with a smaller slack time are given higher priorities. If two tasks at the same level have the same slack time, we arbitrarily assign one a higher priority. The task topological levels and their priorities given in Fig. 1 are shown in Table I. Tasks are then sorted by their priorities in a decreasing order and stored in a ordered list L.

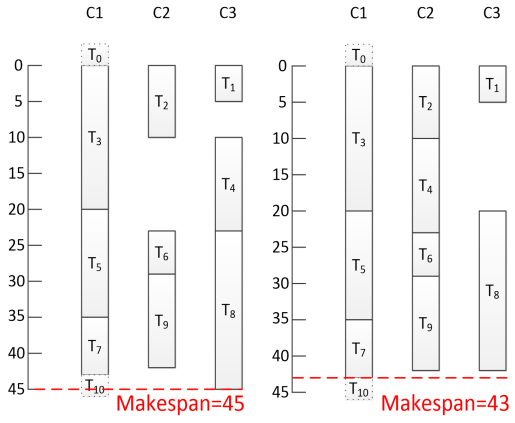
Intuitively, a leveled graph ensures that all predecessor tasks of a task τ_i are scheduled before τ_i. The minimal slack time based priority assignment ensures that the tasks that are more urgent are executed earlier. Once tasks are sorted, starting from minimal number of resources given by (13), we iteratively increase the number of computers until a schedule that meets the application's end-to-end deadline is found.

B. Minimal Slack time and Minimal Distance Scheduling Algorithm

For a given number of computers, the goal of the minimal slack time and minimal distance (MSMD) scheduling algorithm is to schedule a given application to a set of allocated computers so that the application's makespan is minimized.

As tasks on a critical path cannot be executed concurrently, assigning all critical tasks to the same computer does not increase the application's makespan. The question is how to assign tasks that are not on the critical path. One simple approach is to schedule these tasks to the computer that has the earliest available time at the time of scheduling. Again, take the application task graph given in Fig. 1 as an example, if we have three computers, based on the priority given in Table I, at time 0, as $av(c_1) = av(c_2) = av(c_3) = 0$, τ_3 is scheduled to c_1, τ_2 to c_2, and τ_1 to c_3, respectively. At time 10, $av(c_3)$ is the least and hence τ_4 is scheduled on c_3. Fig. 2(a) shows the schedule produced by the approach. We denote this approach as minimal slack and minimal available time based (MSMA) scheduling algorithm.

However, a task τ_i's start time on a computer c_j not only depends on the value of $av(c_j)$, but it also depends on its own ready time. Hence, if both computers c_j and c_k satisfy $av(c_x) \leq ready(\tau_i)$, task τ_i can be assigned to either one of them. For instance, in our application graph given in Fig. 1, τ_4 can be deployed either on c_2 or c_3. However, the decision may affect the following tasks, such as τ_8 in Fig. 2(a). If τ_4 is scheduled on c_2 rather than being scheduled on the computer with the earliest available time (c_3), τ_8 can start at time 20 which can reduce the application's makespan by 3 time units. The two different schedules are depicted in Fig. 2(b).



(a) MSMA Approach (b) MSMD Approach

Fig. 2: A Schedule for Task Graph (Fig.1)

We introduce the concept of *distance* into our scheduling algorithm. The distance is used to indicate how close a task's *ready* time is to the computer's earliest available time. We formally define the distance ($Dis(\tau_i, c_j)$) between a task τ_i's ready time ($ready(\tau_i)$) and a computer c_j's available time ($av(c_j)$) as follows:

$$Dis(\tau_i, c_j) = \begin{cases} av(c_j) & if\ ready(\tau_i) < av(c_j) \\ ready(\tau_i) - av(c_j) & otherwise \end{cases} \quad (16)$$

Rather than schedule tasks to the computer with the earliest available time, we assign tasks to the computer with minimal distance from its ready time. Doing so will provide more chances for tasks with lower priorities to start at their earliest start time. Furthermore, it allows some non-critical tasks share the same computer with critical tasks and hence further reduces application's makespan when other computers' earliest available time become larger than the critical task computer at the time of scheduling. As shown in Fig. 2(b), with the minimal slack time and minimal distance (MSMD) approach, τ_4 is scheduled on c_2. Hence τ_8 can start at its earliest start time of 20, which in turn reduces the application's makespan to 43 compared to 45 produced by the MSMA approach.

For a given application with task graph $G(V, E)$, ordered list L, and min number of computers, the MSMD scheduling algorithm is given in Algorithm 2. In the algorithm, Line 3 to Line 4 assign tasks on the critical path to the same computer, i.e. c_0, Line 7 to Line 20 find the computer that has the minimal distance from its available time to the current task's ready time. Line 7 to Line 9 and Line 17 to Line 20 enable uncritical tasks to be scheduled on critical task computer without interfering critical tasks. The complexity of the MSMD algorithm is $O(|V|^2 m)$. We evaluate the proposed algorithm in the following section.

Algorithm 2: MSMD(G, T, S, L)

```
1  P_c ← G's critical path;
2  for i ← 0 to |L| − 1 do
3      if L[i] ∈ P_c or m = 1 then
4          Assign L[i] to S[0]
5      end
6      else
7          T[0] ← max{av(0), T[0]};
8          minDisComp ← 0;
9          distance ← Dis(L[i], 0);
10         for j ← 1 to m − 1 do
11             T[j] = av(S[j]);
12             if Dis(L[i], j) < distance then
13                 distance ← Dis(L[i], j);
14                 minDisComp ← j ;
15             end
16         end
17         Assign L[i] to minDisComp;
18         if minDisComp = 0 then
19             T[0] ← T[0] + w(L[i])
20         end
21     end
22 end
23 return AFT(τ_exit)
```

VI. Experimental Evaluations

The purpose of the experiments is to evaluate the developed MSMD algorithm by comparing it with four other algorithms published in the literature. They are the heterogeneous-earliest-finish-time (HEFT) algorithm [20], duplication-based bottom up scheduling (DBUS) algorithm [4], energy conscious scheduling (ECS) algorithm [11] and Coffman-Graham (CG) algorithm[6].

A. Experiment Settings

The DAG-based applications are randomly generated based on two indexes, i.e. the *inverse parallel index* defined as $\frac{Lev(\tau_{exit})}{|V|}$ and the *deadline tightness index* defined as $\frac{T_D}{T_C}$. The inverse parallel index is used to control the shape of the application task graph. A task graph with a low inverse parallel index indicates that many tasks can be executed in parallel and high inverse index indicates that many tasks need to be executed sequentially. The difference between an application's end-to-end deadline and its critical path execution time measures how tight the deadline is. Based on the two attributes of a task graph, we classify our applications into five different categories: tight-highly-parallel (TH), tight-low-parallel(TL), relaxed-highly-parallel (RH), relaxed-low-parallel (RH), and fully-random (FR). The classification allows us to explore how the task graph shape and the deadline tightness of the application impact the resource usage. Under each category, 800 different application task graphs are randomly generated.

B. Evaluation Criteria

One of the main objectives of our algorithm is to find the minimum number of computers needed to complete a given application's execution before its deadline. Hence, the first criterion to evaluate the performance of an algorithm is how many computers it uses to ensure an application finishes before its end-to-end deadline. We introduce the concept of *resource reduction rate* to indicate how many resources are reduced from the resource upper bound given by (14). It is defined as below:

$$\textit{Res. Red. Rate} = \frac{\text{Upper Bound - Actual Res. Used}}{\text{Upper Bound}} \tag{17}$$

The second goal of our algorithm is to minimize the makespan of an application under the given minimal resources. One way to evaluate the effectiveness of minimizing makespan is to see how much time is reduced from an application's deadline. We define *makespan reduction rate* for the evaluation. For a given application A with its release time T_R and relative deadline T_D, the makespan reduction rate is defined as:

$$\textit{MS Red. Rate} = \frac{T_R + T_D - AFT(\tau_{exit})}{T_D} \tag{18}$$

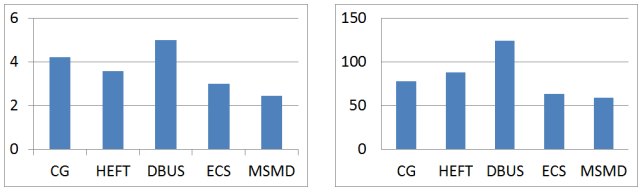
where $AFT(\tau_{exit})$ is the actual finish time of application A.

C. Compare to the Optimal Solution

Since the proposed MSMD scheduling algorithm is a heuristic algorithm, the most straightforward way to evaluate its performance is to compare with the optimal solution. We randomly generate 100 different applications. Each application has no more than ten tasks. We obtain the optimal solutions by exhaustively searching for all possible resource allocations that meet the application's deadline.

We calculate the standard deviation of the number of computers needed by different algorithms from the optimal solutions found by exhaustive search. As shown in Fig. 3(a), the MSMD algorithm has the minimum standard deviation from the optimal solution. This implies that the MSMD algorithm can guarantee applications' end-to-end deadlines with the number of computers that is close to the optimal.

We also calculate the makespan standard deviation, since heuristic algorithms may need more computers to guarantee applications' end-to-end deadlines than the optimal solution. When extra computers are used for scheduling, it is possible that the applications' makespan are smaller than the makespan produced by the optimal solution. Hence, the calculation of makespan standard deviation only considers the cases when the heuristic algorithms use the same number of computers as the exhaustive search algorithm. As shown in Fig. 3(b), the MSMD algorithm has the smallest standard deviation from the optimal solution on both minimum number of computers needed and applications' makespan. It indicates that the performance of the MSMD algorithm is close to optimal.



(a) Min Number of Computers (b) Makespan

Fig. 3: Standard Deviation Comparison

D. Comparison among Different Heuristic Algorithms

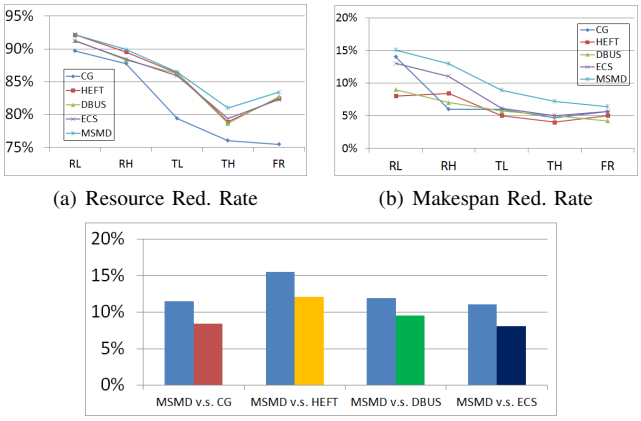
In the previous subsection, as we need to find the optimal solutions through exhaustive search, both sample size and the application size are limited. In this section, we extend the test scale. In particular, we categorize applications into five categories based on the application's inverse parallel index and deadline tightness, and randomly generate 800 test cases for each category and apply the five different algorithms, i.e. CG, HEFT, DBUG, ECS, and our MSMD, to these test cases. The test results are depicted in Fig. 4 and summarized in Table II.

TABLE II: Overall Performance Comparison

Alg.	MS Red. Rate	Res Red. Rate
HEFT	6%	82.5%
DBUS	6%	82.7%
CG	8%	75.5%
ECS	8%	82.2%
MSMD	10%	83.4%

From Table II, it is not difficult to see that the makespan reduction rate of the MSMD algorithm is 10% which is the highest among all five algorithms. The MSMD also has the highest resource reduction rate (83.4%). Although the HEFT, DBUS and ECS algorithms have high resource reduction rates (82.5%,82.7% and 82.2%, respectively) that are close to the MSMD algorithm, their average makespan reduction rates are only 6%, 6% and 8%, respectively. The Coffman-Graham (CG) algorithm has a good makespan reduction rate, but it has a low resource reduction rate. Figure 4 gives more details.

In particular, Fig. 4(a) depicts the comparison of the resource reduction rate. The MSMD has the highest resource reduction rates on all five different application categories. When the applications have relaxed deadlines (RH or RL), all five algorithms can schedule applications with much fewer computers than the lower resource bound given by (14). As



(a) Resource Red. Rate (b) Makespan Red. Rate

(c) Makespan Red. Rate under Same No. of Comp.

Fig. 4: Comparisons

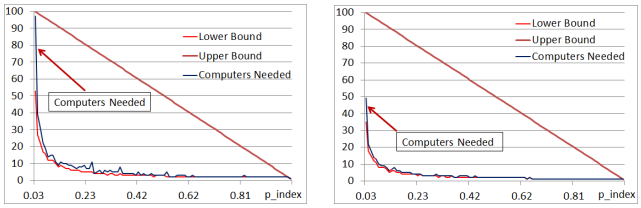
the deadlines become tight (TL or TH), all five algorithms need more computers to schedule the applications. However, the number of computers needed by all five algorithms are still much fewer than the resource upper bound given by (14).

Fig. 4(b) shows the average makespan reduction rates resulted from the different algorithms under different application categories. Again, the MSMD algorithm always results in a higher makespan reduction rate compared with other algorithms. When the applications' deadlines are relax ($\frac{T_D}{T_C} \geq 1.5$), the MSMD algorithm is the most effective algorithm for makespan reduction. On average, it reduces application's makespan by about 13% and 15% for RH and RL types of applications, respectively. Even when the applications' deadlines are tight ($1 \leq \frac{T_D}{T_C} < 1.5$), the MSMD is still the most effective makespan reduction algorithm. On average, it has 3% more makespan reduction than the other four algorithms.

However, as all five algorithms are heuristic scheduling algorithms, it is possible that the five algorithms use different number of computers to schedule the same application. Algorithms that use more computers may schedule the application with a smaller makespan than the algorithms that use less computers. Hence, the overall makespan reduction rate given in Fig 4(b) may not fully reflect algorithms' performances on makespan reduction. Fig 4(c) illustrates the detailed comparison of makespan reduction rates between MSMD and each of the other four algorithms when they are using the same number of computers to schedule the application. It clearly indicates that on average MSMD can reduce application's makespan by 3% more compared with the other four algorithms when using the same number of computers.

In summary, the MSMD algorithm has the highest makespan reduction rate with a minimum number of computers needed compared with other algorithms. Under the same computer resources, the MSMD algorithm can further reduce the applications' makespans by 3% of their end-to-end deadlines compared with the other four algorithms, making it the most efficient makespan reduction algorithm among the CG, HEFT, DBUS, and ECS algorithms.

From the experiments, we have also observed that the deadline tightness and the inverse parallel index are two major factors that impact the number of computers needed. However, exactly how these two factors impact the minimal resources



(a) $T_D = T_C$ (b) $T_D = 1.5T_C$

Fig. 5: Impact of Deadline Tightness on the Number of Resource Needed by MSMD

needed is not clear. In order to investigate the relationship between the minimal resources needed, deadline tightness and inverse parallel index, we perform another set of experiments. In this experiment, we create a base application that contains 100 tasks. We further create a set of applications that have the same number of tasks, the same sequential execution time, but have different task graph shapes and different critical paths. We schedule each application using MSMD algorithm under two different end-to-end deadlines, i.e., $T_D = T_C$ and $T_D = 1.5T_C$, respectively.

Fig 5 depicts the experiment results. The X-axis represents the inverse parallel index and Y-axis represents the number of computers needed. From Fig 5, we observe that an application needs almost $|V| - Lev(\tau_{exit})$ computers to guarantee its deadline only when almost all the tasks in the application can be parallelly executed and the application's end-to-end deadline is very close to the critical path execution time. The number of computers needed to guarantee an application's end-to-end deadline exponentially decreases when more tasks must be executed sequentially. When more than half of the tasks need to be sequentially executed, the number of computers needed tends to be constant regardless the tightness of the deadline.

VII. Conclusion

In this paper, we have addressed the issue of how to deploy real-time application to computer clouds so that (1) real-time application's end-to-end deadline is guaranteed, (2) number of resources allocated to the application is minimized, and (3) under the allocated minimum resources, the application's makespan is minimized. We have proven the lower and upper bounds on the number of resources needed to guarantee a given real-application's deadline. Based on the bounds, we have developed a *minimal slack time and minimal distance* (MSMD) heuristic task deployment and scheduling algorithm that finds the minimum number of resources needed to guarantee an application's deadline and also minimizes the makespan of the application under the allocated resources. The time complexity for the MSMD is only $O(|V|^2 m)$. Our experimental results have shown that the heuristic MSMD algorithm can guarantee applications' end-to-end deadline with less resources compared with other heuristic scheduling algorithms and can on average reduce applications' makespans by 10% of their deadlines under the allocated resources.

However, in our current work, we have made two assumptions: (1) computers in the cloud are homogeneous, and (2) there are no communications among application tasks. Our immediate future work is to study real-time application resource needs when these two assumptions are removed. Furthermore, it is not difficult to see from Fig. 5 that the resource upper bound we have proven is not a tight bound. Hence, another line of future work is to investigate whether the upper bound can be theoretically tightened.

References

[1] Gartner says worldwide public cloud services market to total $131 billion. http://www.gartner.com/newsroom/id/2352816/.

[2] Interactive real-time elearning. www.irmosproject.eu.

[3] M. A. Al-Mouhamed. Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *Software Engineering, IEEE Transactions on*, 16(12):1390–1401, 1990.

[4] D. Bozdag, U. Catalyurek, and F. Ozguner. A task duplication based bottom-up scheduling algorithm for heterogeneous environments. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 12–pp. IEEE, 2006.

[5] K. Chadwick. Fermigrid and fermicloud update. *In 2012 International Symposium on Grids and Clouds*, 2012.

[6] A. P. E. Coffman Jr and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[7] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. Freeman San Francisco, CA, 1979.

[8] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[9] T. C. Hu. Parallel sequencing and assembly line problems. *Operations research*, 9(6):841–848, 1961.

[10] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa. Science clouds: Early experiences in cloud computing for scientific applications. *Cloud computing and applications*, 2008, 2008.

[11] Y. C. Lee and A. Y. Zomaya. Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*, pages 92–99. IEEE, 2009.

[12] K. Liu, H. Jin, J. Chen, X. Liu, D. Yuan, and Y. Yang. A compromised-time-cost scheduling algorithm in swindew-c for instance-intensive cost-constrained workflows on a cloud computing platform. *International Journal of High Performance Computing Applications*, 24(4):445–456, 2010.

[13] Y. Z. Mengxia Zhu, Qishi Wu. A cost-effective scheduling algorithm for scientific workflows in cloud. *Proceedings of 31st IEEE International Performance Computing and Communications Conference*, 2012.

[14] M. Mezmaz, N. Melab, Y. Kessaci, Y. C. Lee, E.-G. Talbi, A. Y. Zomaya, and D. Tuyttens. A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems. *Journal of Parallel and Distributed Computing*, 71(11):1497–1508, 2011.

[15] Netflix. Netflix, 2013. http://www.netflix.com/.

[16] C. Ramamoorthy, K. Chandy, and M. J. Gonzalez. Optimal scheduling strategies in a multiprocessor system. *Computers, IEEE Transactions on*, 100(2):137–146, 1972.

[17] S. Selvarani and G. Sadhasivam. Improved cost-based algorithm for task scheduling in cloud computing. In *Computational Intelligence and Computing Research (ICCIC), 2010 IEEE International Conference on*, pages 1–5. IEEE, 2010.

[18] TEMENOS. Temenos, 2013. www.temenos.com.

[19] TheCloudMarket.com. The Cloud Market Statistic Monitor, 2012. http://thecloudmarket.com.

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