High-Throughput Scientific Workflow Scheduling under Deadline Constraint in Clouds

Michelle M. Zhu[1], Fei Cao[1], and Chase Q. Wu[2]
[1]Computer Science, Southern Illinois University, Carbondale, IL 62901
[2]Computer Science, University of Memphis, Memphis, TN 38152
Email: {mzhu, fcao}@cs.siu.edu; chase.wu@memphis.edu

Abstract —Cloud computing is a paradigm shift in service delivery that promises a leap in efficiency and flexibility in using computing resources. As cloud infrastructures are widely deployed around the globe, many data- and computeintensive scientific workflows have been moved from traditional high-performance computing platforms and grids to clouds. With the rapidly increasing number of cloud users in various science domains, it has become a critical task for the cloud service provider to perform efficient job scheduling while still guaranteeing the workflow completion time as specified in the Service Level Agreement (SLA). Based on practical models for cloud utilization, we formulate a delay-constrained workflow optimization problem to maximize resource utilization for high system throughput and propose a two-step scheduling algorithm to minimize the cloud overhead under a user-specified execution time bound. Extensive simulation results illustrate that the proposed algorithm achieves lower computing overhead or higher resource utilization than existing methods under the execution time bound, and also significantly reduces the total workflow execution time by strategically selecting appropriate mapping nodes for prioritized modules.

Index Terms—Scientific workflow, workflow scheduling, cloud computing

I. Introduction

MANY/modern e-sciences produce colossal amounts of data that must be processed and analyzed by domain-specific workflows of interdependent computing modules for scientific discovery and invention. Such scientific workflows are traditionally executed on high-performance computing platforms and computational grids. With the advent of cloud computing, researchers have recognized the importance and benefit of shifting scientific workflows to cloud environments, as already evidenced in various science fields. Cloud computing offers three generic types of cloud services, namely, Infrastructure-as-a-Service (IAAS), Platform-as-a-Service (PAAS), and Software-as-a-Service (SAAS). IAAS creates virtual machines (VMs) on a physical node with full user control, as exemplified by Amazon's ES2's instances [12]. PAAS provides an execution environment for users to run applications with specific system configurations using particular programming paradigms such as Java and Python, as exemplified by Google's App Engine [13]. SAAS enables usersto run some particular software remotely without the need of installing it on their local machines. Salesforce's Database.com is one typical example that provides many application programming interfaces (APIs) for users to access the database as though it is a local database. Considering the wide variety of scientific computing modules and their disparate performance and runtime requirements, IAAS is generally considered as the best suited cloud environment for scientific workflows. A typical IAAS cloud employs a scheduler to determine the type and location of VM instances, and a pricing model to decide the cost charged to the user.

Scientific workflows can be as simple as a single module or as complex as a Directed Acyclic Graph (DAG). To facilitate execution parallelism and maximize execution efficiency, workflow modules need to be dispatched or mapped to a set of strategically selected VMs. There exists a plethora of research on the mapping and scheduling of workflow systems in clouds. However, most of these existing efforts are based on static resource models. In this paper, we consider time-varying resource availability of both computer nodes and network links upon the arrival of a user request. We believe that this model better reflects the use dynamics of a real-life cloud environment where any user can make advance reservations or utilize VM resources during the scheduling process.

In our scheduling problem, we consider two objectives from the perspective of a cloud service provider: i) satisfy the latest completion time of the entire workflow, which is typically specified as a Quality of Service (QoS) requirement; ii) maximize the system throughput to accommodate as many user requests as possible during a certain time period. The system throughput is reflected by the resource utilization rate, which is defined as the useful computing cost over the total cost including the overhead for starting up and shutting down VMs as well as the idle VM time. Although most cloud service providers charge users by hours regardless of the time spent on computing or overhead, the provider always wishes to reduce unnecessary CPU cycles spent on overhead since these wasted resources could be allocated elsewhere to meet other users' requests, especially when the cloud is heavily loaded during the peak time.

Manuscript received May 8, 2013; revised April 22, 2014.
Corresponding author email: mzhu@cs.siu.edu.
doi:10.12720/jcm.9.4.312-321

Resource utilization must be considered in the design of a scheduling algorithm to avoid early resource saturation and job request turndown.

The aforementioned cloud scheduling problem has been proven to be NP-complete [9]. We propose a heuristic workflow mapping approach, referred to as Highthroughput Workflow scheduling Algorithm with Execution time bound (HiWAE), which is a two-step procedure: In the first step, modules are topologically sorted into different layers to determine the module mapping order starting from the first layer. Each module is assigned with a certain priority value based on its computational complexity and mapped to the node that yields the lowest partial end-to-end delay (EED) as the execution time from the starting module to the current one. This mapping process is repeated until a convergence point is reached. The main goal of the second step is to improve the resource utilization rate by minimizing the overhead of VM's startup and shutdown time as well as the idle time. For example, some modules may share the same VM for reduced startup/shutdown overhead, and some VMs may be released early until the next active module arrives to save the idle time. A preliminary version of this algorithm was proposed in [1].

The rest of the paper is organized as follows. Section II, conducts a survey of workflow mapping algorithms. Section III constructs the models for scientific workflows and cloud environments to compute the workflow execution cost. Section IV proves that the EED and cost are two conflicting objectives, and optimizing both at the same time is not possible. Section V presents the algorithm details and Section VI evaluates the algorithm performance.

II. Related Works

We provide a brief description of existing workflow mapping algorithms in various environments with focus on those developed specifically for clouds.

The workflow mapping problem for minimal makespan in heterogeneous network environments has been extensively studied and is well known to be NP-hard [9]. There exist a number of heuristic algorithms [1], [2], [16], [17], [9], [18], [23] in the literature. In [18], the Heterogeneous Earliest Finish Time (HEFT) algorithm tries the mapping of each module onto all the nodes first, and then chooses the best one with the earliest finish time. Streamline [9], a scheduling algorithm originally designed for streaming data, creates a coarse-grained dataflow graph on available grid resources. In [17], an optimal algorithm is proposed to determine an optimal static allocation of modules among a large number of sensors based on an A_* algorithm. The Recursive Critical Path (RCP) algorithm takes a dynamic programming-based approach and recursively computes a critical path to minimize the EED [16]. This algorithm is used as the mapping scheme for the Scientific Workflow Automation and Management Platform (SWAMP) [19]. Condor is a specialized workload management system for compute-intensive jobs [20] and it can be used in grid environments such as Globus grid [22]. The Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for Condor jobs that manages dependencies between jobs at a higher level than the Condor Scheduler. DAGMan submits jobs to Condor in an order represented by a DAG and processes the results [21]. RCP provides a better mapping performance than the default mapping scheme employed by the Condor Scheduler [19]. However, the aforementioned algorithms either target static homogeneous environments with fully connected networks, or fail to find feasible mapping solutions when the system scales up, or adopts a simple greedy approach that oftentimes leads to unsatisfactory performance.

Several efforts have been devoted to scheduling workflows in cloud environments [5]-[8], [4]. In [5], Hoffa *et al.* compared the performance and the overhead of running a workflow on a local machine, a local cluster, and a virtual cluster. Haizea is a lease management architecture [7], which implements leases as VMs, leverages the ability to suspend, migrate, and resume computations, and provides the leased resources in a customized application environment. In [8], Vockler *et al.* discussed the experience of running workflows and evaluating their performance as well as the challenges in different cloud environments. In [6], Figueiredo *et al.* made an effort to create a grid-like environment from cloud resources to ensure a higher level of security and flexible resource control. In [4], Yu *et al.* proposed a cost-based workflow scheduling algorithm that minimizes execution cost while meeting the deadline for completing the tasks.

Several job scheduling policies including Greedy (First Fit) and Round Robin algorithms are used in open-source cloud computing management systems such as Eucalyptus [10]. Queuing system, advanced reservation and preemption scheduling are adopted by OpenNebula [11]. Nimbus uses some customizable tools such as PBS and SGE [14]. The Greedy and Round Robin are heuristics that select adaptive physical resources for the VM to deploy without considering the maximum usage of the physical resource. The queuing system, advanced reservation and preemption scheduling also do not consider any balanced overall system utilization. Pegasus Workflow Manage System is a more advanced workflow scheduling algorithm [15], which maps a workflow onto the cloud to generate an executable workflow using a clustering approach, where short-duration modules are grouped as a single module to reduce data transfer overhead and the number of VMs created. The rank matching algorithm in [24] features a scheduling strategy that ranks each module's possible mapping nodes and selects the one with the lowest cost as the mapping result.

Our work aims to achieve the maximum utilization of cloud resources while guaranteeing the QoS required by individual users. Although many techniques have been proposed to meet these two objectives separately, the research efforts in tackling both problems at the same time are still very limited.

III. ANZALYTICAL MODELS

We construct the analytical cost models for the workflow task graph and the underlying cloud computer network graph to facilitate a mathematical formulation of the delay-constrained mapping optimization problem.

A. *Workflow and Cloud Models*

We model the workflow of a distributed computing application as a directed acyclic graph $G_{wf} = (V_{wf}, E_{wf})$, $|V_{wf}| = N$, where the vertices represent computing modules, i.e. $V_{wf} = \{u_1, u_2, ..., u_N\}$ with u_1 and u_n being the start and end modules, respectively. The dependency between a pair of modules (u_i, u_j) is represented as a directed edge $e_{i,j}$ with weight w_{ij} being the size of data transferred from module ui to module u_j . Module uj receives a data input w_{ij} from each of its preceding modules u_j and performs a predefined computing procedure whose complexity is modeled as a function $c_{u_j}(\cdot)$ of the total aggregated input data size z_{uj}. Note that in real scenarios, the complexity of a module is an abstract quantity, which not only depends on the computational complexity of the procedure itself but also on its implementation details. Module u_j sends a data output w_{ik} to each of its succeeding modules u_k upon the completion of execution. In general, a module cannot start its execution until all input data required by this module arrives. To generalize our model, if an application has multiple starting or ending modules, we can create a virtual starting or ending module of complexity zero and connect it to all starting or ending modules without any data transfer along the edges.

We consider a general cloud environment that supports both advance VM reservations and on-demand user requests. Thus, the resource allocation status of the cloud network is time dependent, i.e. the available computing resources on each node and the bandwidth on each network link vary over time. We model the underlying cloud network as a complete network graph $G_{cn} = (V_{cn}, E_{cn})$, consisting of a set of $|V_{cn}| = K$ computing nodes $V_{cn} = \{v_1, v_2, ..., v_K\}$ as well as a link between every pair of nodes. Node vj is featured by a normalized computing power p_{vj} based on its CPU and memory, and the network link $L_{i,j}$ from node v_i to v_j is featured by bandwidth $b_{vi,vj}$ and minimum link delay $d_{vi,vj}$.

Fig. 1 illustrates an example of three reservation requests made on one cloud node during different time slots. For example, request 1 reserves 60% of the node's capacity from t0 to t_2; request 2 reserves 20% from t_1 to t_4; request 3 reserves 40% from t_3 to t_4. The maximal available computing power of this node from t0 to t_4 is p_{v_j,t_0,t_4} = min(40%, 20%, 80%, 40%). The largest VM instance that can be allocated on v_j from time t_1 to t_n, namely $p^{VM}_{v_j,t_1,t_n}$, is computed using resources of p_{v_j,t_1,t_n}. The execution time of module u_i on node v_j during time slot t_1 and t_n is then computed as $t_{v_j,t_1}(u_i) = \frac{z_{u_i} * c_{u_i}(\cdot)}{p^{VM}_{v_j,t_1,t_n}}$, where z_{u_i} denotes the aggregated complexity-normalized input data size of module u_i. Similarly, the maximum link bandwidth along L_{v_i,v_j} during time slot t_m and t_n is $min(B_{v_i,v_j,t_m,t_n})$

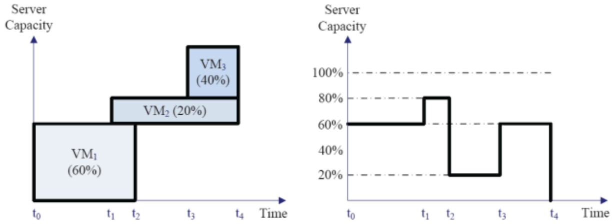

Fig. 1. Reserved requests on a single cloud node from time point t_0 to t_4

B. *Workflow Execution Cost*

The cost of running a workflow in a cloud is measured by the sum of the total time, during which VMs are running including idle and overhead time, multiplied by the corresponding VM's capacity. The time spent on mdeploying VM on a particular node v_j consists of the following components:

1) The VM startup time for selecting a virtual node and transferring a virtual image as well as the boot-up time. It is assumed to be a fixed value of t_{start}.
2) The running time for every assigned module on the corresponding VM. Suppose that a set U of modules are assigned on $VM_{vj,k}$, and start to run from time t_s and end at time t_e in a sequential manner. The running time for these modules is computed as $\frac{\sum_{u_i \in U} z_{u_i} \cdot c_{u_i}(\cdot)}{p_{VM_{v_j,k}}}$.
3) The idle time between the execution time of any two modules. When two modules run on the same VM, there could be some idle time after one module is completed and before the next module starts, calculated as $Idle(VM_{v_j,k}) = \sum_{u_i \in U}(st_i - et_{i-1})$
4) The VM shutdown time, which is also assumed to a constant of t_{shut}.

Hence, the total resource cost for workflow G_{wf} is:

$$TC(G_{wf}) = \sum_{i=1}^{N} z_{u_i} \cdot c_{u_i}(\cdot) + \sum_{j=1}^{K_G} \sum_{k=1}^{M_{v_j}} p_{VM_{v_j,k}} \cdot (t_{start} + Idle(VM_{v_j,k}) + t_{shut}) \quad (1)$$

where N is the total number of modules in a workflow, K_G denotes the total number of nodes that have been allocated for the workflow, and M_{vj} denotes the total number of VMs that have been set up on node v_j .

The Utilization Rate (UR) is defined as:

$$UR = \frac{\sum_{i=1}^{N} z_{u_i} \cdot c_{u_i}(\cdot)}{TC(G_{wf})} \qquad (2)$$

which measures the efficiency of the cloud resource utilization excluding the VM overhead. Obviously, the cloud provider always desires to maximize this ratio, i.e. reduce the cost to improve the resource utilization rate, which leads to a higher system throughput. For convenience, we provide a summary of the notations used in the cost models in Table I.

TABLE I. NOTATIONS USED IN THE ANALYTICAL COST MODELS.

Parameters	Definitions
$G_{wf} = (V_{wf}, E_{wf})$	the computation workflow
N	the number of modules in the workflow
u_i	the i-th computing module
$e_{i,j}$	the dependency edge from module u_i to u_j
w_{ij}	the data size transferred over dependency edge $e_{i,j}$
z_{u_i}	the aggregated input data size of module u_i
$c_{u_i}(\cdot)$	the computational complexity of module u_i
st_i	the start time of module u_i
et_i	the end time of module u_i
$G_{cn} = (V_{cn}, E_{cn})$	the cloud network
K	the total number of nodes in the cloud
v_j	the j-th computer node
v_s	the source node
v_d	the destination node
p_{v_j}	the total computing power of node v_j
$p^{VM}_{v_j,t_1,t_n}$	the maximal percentage of computing power of VM on node v_j from t_1 to t_n
$L_{i,j}$	the network link between nodes v_i and v_j
b_{v_i,v_j,t_1,t_n}	the bandwidth of link $L_{i,j}$ from t_1 to t_n
d_{v_i,v_j}	the minimum link delay of link $L_{i,j}$
t_{start}	the time spent on setting up a virtual machine for the workflow running environment on a node
t_{shut}	the time spent on shutting down a virtual machine
$t_{v_j}(u_i)$	the execution time of module u_i running on node v_j
K_G	the total number of nodes that have been allocated for the workflow
$VM_{v_j,k}$	the k-th VM on the j-th node
M_{v_j}	the total number of VMs on the j-th node
$pVM_{v_j,k}$	the computing power of $VM_{v_j,k}$

Minimum End-to-end Delay (MED) is an important performance requirement in time-critical applications especially for interactive operations. Our mapping objective is to select an appropriate set of virtual nodes to set up VM instances for module execution to achieve MED. The utilization rate can be improved by cutting down the VM startup, shutdown and idle time. Our approach chooses the mapping scheme that results in a higher UR under the same End-to-End Delay (EED) constraint. Once a mapping schedule is determined, EED is calculated as the total time incurred on the critical path (CP), i.e. the longest execution path from the source module to the destination module.

IV. PROBLEM FORMULATION

A schedule S with the maximum resource utilization rate may be obtained by simply mapping all the modules onto one node. However, such a schedule S usually has a much longer EED than the optimal one.

We first consider a bi-objective scheduling problem to minimize the EED and maximize the utilization rate (or to minimize the total overhead). These two objectives are conflictive and cannot be achieved at the same time, as stated in Theorem 1.

Theorem 1: The bi-objective problem of minimizing the EED and maximizing the utilization rate is nonapproximable within a constant factor.

Proof: We consider a simple instance of the problem that involves only three modules, u_1, u_2 and u_3, and two computing nodes, v_1 and v_2, whose computing powers have a relationship $p_{v1} = kp_{v2}$. We assume a constant VM startup time SA and shutdown time SU. The computational complexity of modules u_1 and u_2 has a relationship $c_{u1}(\cdot) = kc_{u2}(\cdot)$, and they provide two input datasets to u_3. The link bandwidth between these two nodes is a constant $b_{v1,v2,t} = B_{v1,v2}$ without any other transfer task scheduled. The data size transferred from u_2 to u_3 is represented by $z_{12} = mB_{v1,v2}$. Assume that $z_{u1} = z_{u2}$ and the data transfer time is small enough to be ignored compared with the module running time. Note that data transfer in cloud environments is fast and such cost is typically not included in the user bill. There exist two feasible solutions S1 and S2:

1) S1 is optimal for EED: The modules u_1 and u_3 are scheduled on node v_1, and module u_2 is scheduled on node v_2. Two independent virtual nodes can start up simultaneously. The EED of S1 is calculated in Eq. 3 where $c_{u1}(\cdot) = kc_{u2}(\cdot)$ and $p_{v1} = kp_{v2}$. As u_1 and u_2 are independent, they can run in parallel on two different virtual nodes, and thus only the latest running time needs to be counted for the EED. The efficiency resource (ERC), which is the useful cost for running the workflow (i.e. user payload), is computed in Eq. 4 . The utilization rate of S1 is calculated in Eq. 5.

$$EED(S1) = SA + \frac{c_{u_1(\cdot)} \cdot z_{u_1}}{p_{v_1}} + \frac{z_{12}}{b_{v_1,v_2}} + \frac{c_{u_3} \cdot (\cdot) z_{u_3}}{p_{v_1}} + SU. \qquad (3)$$

$$ERC = (k+1)c_{u_2(\cdot)} \cdot z_{u_2} + z_{12} + c_{u_3}(\cdot) \cdot z_{u_3}. \qquad (4)$$

$$UR(S1) = \frac{ERC}{ERC + (k+1)(SA+SU)p_{v_2}}. \qquad (5)$$

2) S2 is optimal for the utilization rate: All the modules should be mapped to v_1, which is more powerful, to achieve a better EED with maximized utilization rate. To calculate the EED, the running time of u_1 and u_2 needs to be computed first. In the beginning, two modules need to share the computing power of v_1 until u_2 is finished. The running time for u_2 is $\frac{2c_{u_2(\cdot)} \cdot z_{u_2}}{p_{v_1}} = \frac{2c_{u_1(\cdot)} \cdot z_{u_1}}{kp_{v_1}}$ when u_1 is still in execution. When u_2 releases the node, the running

time for the remaining portion of *u₁* is $\frac{c_{u_1(\cdot)} * z_{u_1} - \frac{c_{u_1(\cdot)} * z_{u_1}}{k}}{p_{v_1}}$. Thus the EED (S2), ERC' and UR(S2) can be calculated as follows:

$$EED(S2) = SA + \frac{c_{u_1(\cdot)} \cdot z_{u_1}}{p_{v_1}} + \frac{z_{12}}{b_{v_1,v_2}} + \frac{c_{u_3}(\cdot) \cdot z_{u_3}}{p_{v_1}} + SU. \tag{6}$$

$$\begin{aligned} ERC' &= (k+1)c_{u_2(\cdot)} \cdot z_{u_2} + z_{12} + c_{u_3}(\cdot) \cdot z_{u_3} \\ &= ERC. \end{aligned} \tag{7}$$

$$\begin{aligned} UR(S2) &= \frac{ERC}{ERC + (SA+SU) \cdot p_{v_1}} \\ &= \frac{ERC}{ERC + k(SA+SU) \cdot p_{v_2}} > UR(S1). \end{aligned} \tag{8}$$

Since the transfer time is much faster than the running time, S1 has a smaller EED and also a smaller utilization rate, which contradicts our assumption on its optimality. Therefore, it is impossible to optimize both objectives at the same time. Thus, we attempt to maximize the utilization rate within the constraint of the largest acceptable EED.

We consider the following delay-constrained utilization maximization problem for workflow mapping:

Definition 1: Given a DAG-structured computing workflow $G_{wf} = (V_{wf}, E_{wf})$, and an arbitrary computer network in a cloud environment $G_{cn} = (V_{cn}, E_{cn})$ with timedependent link bandwidth and node computing power, we wish to find a workflow mapping schedule such that the utilization rate is maximized within the largest acceptable end-to-end delay constraint, i.e. the execution time bound (ETB):

$$\max_{all\ possible\ mappings} (UR_{G_{cn}}(G_{wf})), \text{ such that } EED \leq ETB. \tag{9}$$

Here, $UR_{G_{cn}}(G_{wf})$ is the product of the utilization rates of all the resources that are assigned to either run a module or transfer data as shown in Eq. 2. Apparently, a smaller number of resources yield a higher combined UR.

V. ALGORITHM DESIGN

We propose a two-step heuristic workflow mapping approach, referred to as High-throughputWorkflow scheduling Algorithm with Execution time bound (HiWAE). In the first step, modules are divided into different layers through topological sorting, which determines the module mapping order starting from the first layer. Modules are assigned with different priority values based on a combined consideration of their complexities and whether or not they are on the critical path (CP). Each module is mapped to the node that results in the lowest partial EED from the starting module to the current one. This module mapping process is repeated until the difference in EED between two contiguous rounds falls below a certain threshold. The second step improves the resource utilization rate by cutting down the VM's startup, shutdown, and idle time. Strategies used for this purpose include module grouping on the same VM to save the startup/shutdown time and resource release to save the idle time. The pseudocode of HiWAE is provided in Alg. 1.

```
Algorithm 1 HiWAE($G_{wf}, G_{cn}, t_s, ETB$)
Input: workflow task graph $G_{wf}$, cloud network graph $G_{cn}$, workflow's earliest start time $t_s$, the execution time bound ETB
Output: a task scheduling scheme with the minimum resource cost within the given execution time bound
EEDOrientedForwardMapping($G_{wf}, G_{cn}, t_s$);
DelayConstraintedBackwardMapping($G_{wf}, G_{cn}, M_{tm}, t_s, ETB$).
```

A. Step 1: Minimized End-to-End Delay (MED)

1) Construct a computing environment ${G_{cn}}^*$ with homogenous computing nodes and communication links to calculate the initial Critical Path (CP). Since our cloud environment supports in-advance resource reservations in addition to on-demand requests, the available resource capacity graph is time dependent and a set of time stamps are used to represent and track the periods when resources remain unchanged.

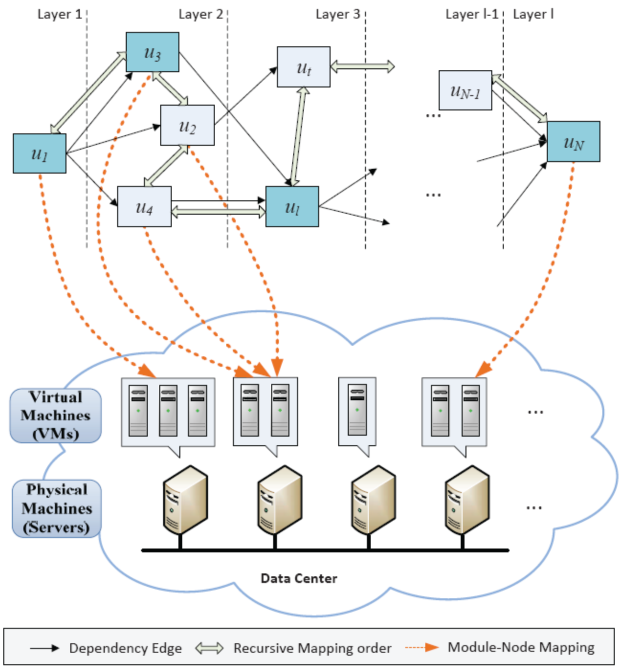

Fig. 2. Layer-ordered prioritized modules mapped to the underlying cloud.

2) Call **EEDOrientedForwardMapping()** function in Alg. 2 to map all modules to underlying cloud nodes. We first compute the CP by employing the wellknown polynomial-time Longest Path (LP) algorithm, namely **FindCriticalPath()**, and then run the prioritized module mapping algorithm **PModulesMapping()** to map the workflow to the

network graph until the convergence of EED is reached, as shown in Fig. 2.

Algorithm 2 EEDOrientedForwardMapping (G_{wf}, G_{cn}, t_s)

```
Input: workflow task graph $G_{wf}$, cloud network graph $G_{cn}$, workflow's earliest start time $t_s$
Output: the temporary mapping scheme with the minimum end-to-end delay (MED)
$i = 1$;
$G_{wf}{}^*$ = mapped workflow based on $G_{cn}{}^*$;
$CP_i$=FindCriticalPath($G_{wf}{}^*$);
call $MED_i$ = PModulesMapping($G_{wf}, G_{cn}, t_s$);
update $G_{wf}{}^*$;
while $|MED_i - MED_{i-1}| \geq Threshold$ do
    $CP_i$=FindCriticalPath($G_{wf}{}^*$);
    call $MED_i$ =PModulesMapping($G_{wf}, G_{cn}, t_s$);
    update $G_{wf}{}^*$;
    $i++$;
end while
return $MED_i$.
```

Algorithm 3 PModulesMapping(G_{wf}, G_{cn}, t_s)

```
Input: workflow task graph $G_{wf}$, cloud network graph $G_{cn}$, workflow's earliest start time $t_s$
Output: the temporary mapping scheme with the best EED namely MED
for all $u_j \in$ CP do
    set $u_j$.flag = 1;
end for
conduct topological sorting and assign the priority value to each module;
dMinMED = $\infty$;
for all $u_i \in$ SortedArray do
    for all $v_j \in$ Node Set $V_{cn}$ do
        calculate the start running time for $u_i$ run on $v_j$;
        call GetPartialMED() to calculate the partial EED for $u_i$ mapped on $v_j$;
        if EED in this round is smaller than previous round then
            update mapping result for current module;
        end if
    end for
end for
```

The pseudocode of **PModulesMapping()** algorithm is provided in Alg. 3. This algorithm first conducts topological sorting to sort modules into different layers. Each module is assigned a priority value based on its computing and communication requirements. The module on the CP is given the highest priority value within the same layer. Starting from the first layer, each module is mapped onto an appropriate node with the lowest partial execution time from the starting module. A backtracking strategy is adopted to adjust the mapping of the preceding modules (i.e. pre-modules) of each newly mapped module in order to further reduce its partial EED. The remapping of any pre-module may also trigger the remapping of its succeeding modules (i.e. suc-modules) if necessary. Such back-and-forth remapping is only limited to one layer, i.e. confined within the affected area in order to control the algorithm's complexity. The shaded modules that comprise of the CP are given the highest priority in their corresponding layers. In Fig. 2, the forward order to map those modules follows u_1, u_3, u_2, u_4, u_t, ..., u_{n-1}, u_n, as marked by the dotted arrows. A new CP is computed after each round of module mapping and such mapping is repeated until the improvement of EED over the previous round is below a certain threshold.

The complexity of this iterative module mapping algorithm is $O(k \cdot l \cdot N \cdot |E_{cn}|)$ where l is the number of layers in the sorted task graph, *N* is the number of modules in the task graph, E_{cn} is the number of links in the cloud network graph, and *k* is the number of iterations where the obtained *EED* meets a certain requirement.

The above mapping procedure is illustrated in Fig. 2, where the upper part represents a DAG-structured workflow with shaded modules along the CP, and the lower part represents a cloud environment. After the topological sorting, u_1 falls in layer 1; u_2, u3 and u_4 fall in layer 2. The modules in layer 1 are mapped onto v_s first, then the modules in layer 2, and so on. For example, u_t has its pre-modules as u_3 and u_4, which are mapped onto v_3 and v_2, respectively. The mapping strategy that leads to the lowest partial EED is chosen for that module. We assume that the inter-module communication cost within the same node is negligible as the data transfer within the same memory is typically much faster than that across a network. Since the resource capacity is time dependent in a cloud environment, instead of calculating one partial EED for each possible mapping, we calculate K (i.e. the number of time slots for one cloud node) possible partial EEDs.

After we map the downstream layer, we adjust its upstream layer's modules depending on its current mapping result. For example, in the above case, u_t is mapped to v_t. We need to adjust its pre-modules u_3 and u_4. During the adjustment process, we also need to calculate the partial EED. Instead of calculating the EED from the source module to the adjusted module, we calculate the partial EED from the source module to its latest finished suc-module.

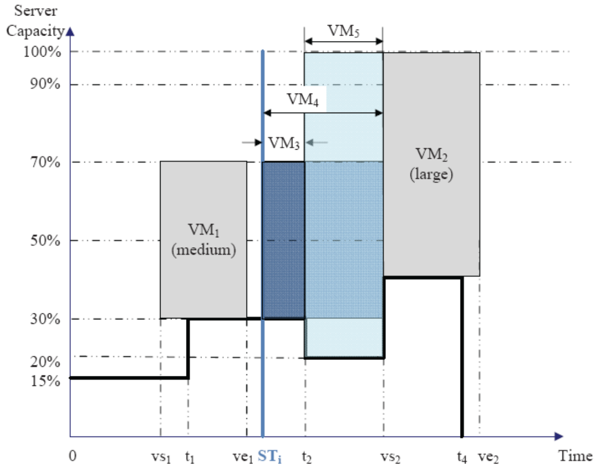

Fig. 3. Map module u_i with start running time ST_i on a cloud node with three possible VMs instances in forward mapping.

This module mapping process is essentially a dynamic programming process. Let us define $u_j \in pre(u_i)$ as the $MN(u_j)$ as u_j's mapping node. We have the following recursive Eq. 10 leading to the minimal $EED(u_i, v_k)$ for the forward mapping.

Similarly, we define $u_l \in suc(u_i)$ as the set of u_i's sucmodules. We also define a recursive equation to update $EED(u_i)$ as in Eq. 11 for the backward mapping.

Fig. 3 illustrates how the partial EED is calculated for a module to be mapped on a cloud node. *VM*$_1$ and *VM*$_2$ are virtual machines that have been deployed to run some premodules. We can calculate the execution start time (ST_i) for module *u*$_i$, and then find out the time slot where ST_i is located. We check all the possible VM strategies, and select the one with the lowest partial EED. In this example, there are three possible VMs that can be allocated for *u*$_i$, namely, *VM*$_3$, *VM*$_4$ and *VM*$_5$.We calculate the execution time of *u*$_i$ on *VM*$_3$ to obtain a partial EED, then check if the execution time is shorter than the life time of *VM*$_3$. If not, we calculate the execution time on *VM*$_4$; otherwise, we calculate the execution time on *VM*$_5$. We compare the partial EED on each VM, and select the one with the lowest partial EED.

B. Step 2: Reduce VM Overhead

In the second step of this algorithm, we want to reduce the VM overhead for the workflow while still meeting the user-specified execution time bound (ETB). The overheads in a cloud include setting up, shutting down and releasinga VM as well as the VM's idle time. The goal of this step is to reduce unnecessary overheads and improve the resource utilization for higher system throughput.

```
Algorithm 4 DelayConstraintedBackwardMapping($G_{wf}$, $G_{cn}, M_{tm}, t_s, ETB$)
Input: workflow task graph $G_{wf}$, cloud network graph $G_{cn}$, the mapping result from step 1, earliest start time $t_s$, execution time bound ETB.
Output: mapping result with the lowest cost UR within ETB.
Calculate the maximal acceptable running time for each module $i$ as $MART_i$;
SortedArray = topological and priority sort;
for all $u_i \in$ SortedArray do
    SET findReuse = false;
    for all $v_j \in$ Node Set $V_{cn}$ do
        if $v_j$ has allocated VM then
            call ReuseVM() to see whether we can reuse a VM on $v_j$ ;
            if $v_j$ has reusable VM then
                update mapping result;
                break;
            end if
        end if
        call AllocateNewVM() to allocate a new VM on $v_j$;
    end for
end for
```

We provide below a brief description of **DelayConstraintedBackwardMapping()**, which is presented in Alg. 4.

$$EED(u_i) = \min_{v_k \in V_{cn}} (\max_{u_j \in pre(u_i)} (EED(u_j, MN(u_j)) + \frac{w_{ij}}{b_{j,k}}) + \frac{z_{u_i} \cdot c_{u_i}(\cdot)}{p_k}) \qquad (10)$$

$$EED(u_i) = \min_{v_k \in V_{cn}} (\max_{u_j \in pre(u_i)} (EED(u_j, MN(u_j)) + \frac{w_{ij}}{b_{j,k}}) + \frac{z_{u_i} \cdot C(\cdot)}{p_k} + \max_{u_l \in suc(u_i)} (\frac{w_{kMN(u_l)}}{b_{k,MN(u_l)}} + \frac{z_{u_l} \cdot c_{u_i}(\cdot)}{p_{MN(u_l)}})) \qquad (11)$$

1) Combine the user-specified execution time bound (ETB) with the MED calculated from Step 1 to obtain the initial maximal acceptable running time (MART) for each module. The running time is calculated as $MART_i = RT_i \cdot \frac{ETB}{MED}$
2) Perform topological sorting in a reverse direction starting from the destination module and assign the corresponding priority value for each module similar to Step 1
3) For each module u$_i$ from the last module to the first module in the reverse topological sorting list, we compare the mapping result for each possible mapping node and select the node and its corresponding VM that incurs the lowest VM overhead as the final mapping node/VM for this module. There are two cases to consider:

- If the mapping node has some allocated VMs, we then call **ReuseVM()** method to check whether or not we can reuse one of these VMs on that node. Two conditions must be satisfied when we reuse a module: a) The available VM resource should be sufficient to run the module. b) Any possible idle time should be less than the time to shut down a VM and start up a new one. If both conditions are satisfied and the partial EED to this module is less than previously found one, we update the mapping information.
- If the mapping node has no VMs or those VMs can not be reused, we call **AllocateNewVM()** to allocate a new VM for this module. The **AllocateNewVM()** is similar to **getPartialEED()**. We create a VM with the maximal allocable resource. Taking Fig. 4 as an example, we can calculate the end time of the module as ST_i. We have 3 different strategies to deploy a VM as *VM*$_1$, *VM*$_2$ or *VM*$_3$. Let *ve*$_x$ be the VM's end time and *vs*$_x$ be its start time. We calculate the running time for that module to be mapped on each VM as $\frac{z_{u_i} \cdot c_{u_i}(\cdot)}{p^{VM}_{v_j, vs_x, ve_x}}$. The allocable resource cost on a VM is $p^{VM}_{v_j, vs_x, ve_x} \cdot (ve_x - vs_x)$. We then compare the running time with the maximal running time of *MART*$_i$. If the running time is less than *MART*$_i$, this VM is acceptable and may be created. For all macceptable VMs, we compare their allocable amount of resources, and select the VM with the maximum amount of allocable resource. In this example, we would select *VM*$_1$ which has the largest area.

4) Repeat Step 3) until all modules from this workflow have been mapped.

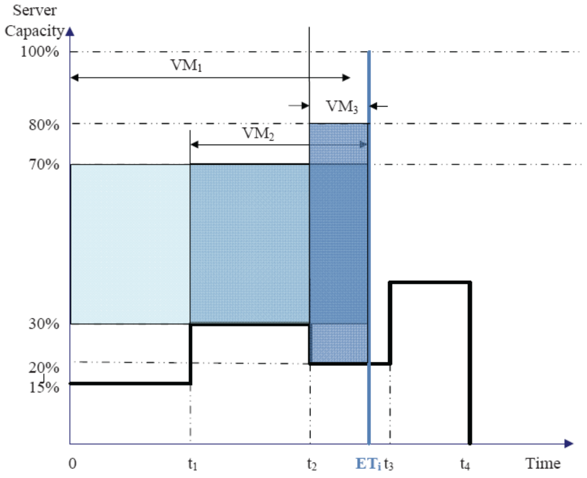

Fig. 4. Three different VMs to execute module u_i with end running time of ET_i in backward mapping.

VI. Performance Evaluation

We implement the proposed HiWAE algorithm in C++ on a Windows 7 desktop PC equipped with Intel Core i7 CPU of 2.66 GHz and 8.0 GB memory. In the experiments, we compare our algorithm's end-to-end delay and resource utilization rate with Min-min and Max-min heuristics adapted for workflows [24]. The threshold can be set dynamically according to different rules, e.g. the difference is less than 2% of MED1 or the decreasing speed approaches zero, etc. A brief description of the two heuristics are as follows:

- **Min-min heuristic:** When a module is ready to execute (i.e. it has received all input data from all of its preceding modules), the resource resulting in the minimum partial EED can be determined (assuming that a new VM with the maximal allocable resource is allocated for each module). After calculating the minimum partial EED values for all such ready-toexecute modules, the module with the least minimum partial EED value is selected for immediate scheduling. This is done iteratively until all the modules have been mapped. The intuition behind this heuristic is that each iterative step incurs the least EED increase with the hope that the final EED is minimized.
- **Max-min heuristic:** The first step of this heuristic is exactly the same as the Min-min heuristic. In the second step, the module with the maximum minimum partial EED value among all the ready-toexecute modules is selected for immediate scheduling. The intuition behind this heuristic is that by giving preference to the longer modules (in terms of execution time), there is a hope that the shorter modules may be overlapped with the longer ones on other resources [24].

TABLE II. Workflow Cases used in the Cloud Mapping Experiments

Index of Test Case	1	2	3	4	5	6	7
# of Modules	10	20	40	60	80	100	200
# of Dependency Edges	21	36	88	120	156	215	420

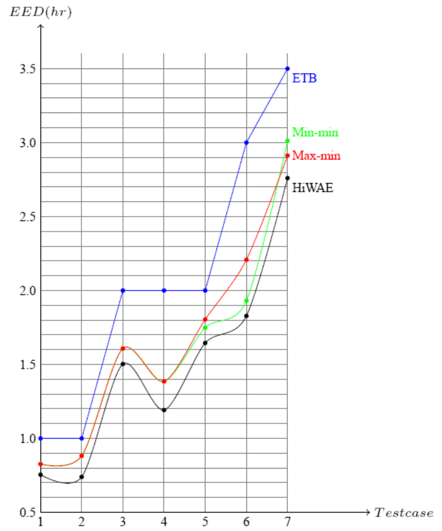

Fig. 5. Comparison of EED among different scheduling algorithms.

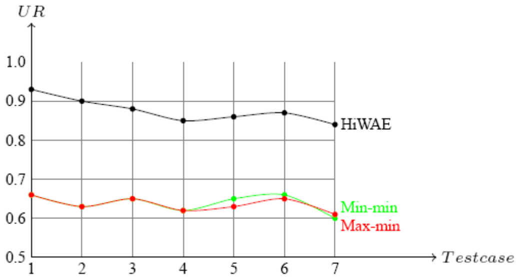

Fig. 6. Comparison of the utilization rate among different scheduling algorithms.

We run these three mapping algorithms on seven randomly generated workflows in a cloud network consisting of 100 nodes with CPU of 2.0 GHz. In this paper, we consider computing-intensive workflows and we assume that the data transfer time is negligible due to high bandwidth among servers. We develop a workflow generator class to generate our test workflows with varying parameters within a suitably predefined range of values according to some previous works [26], [27]: (i) the complexity of each task; (ii) the number of inter-task communications and the data transfer size between two tasks. The workflow mapping results in terms of workflow sizes, utilization rate and EED are presented in Table II and plotted in Figs. 5 and 6 respectively. These results demonstrate that the proposed HiWAE algorithm achieves a superior mapping performance over Min-min and Max-min in terms of EED and utilization rate. Particularly, we observe that HiWAE consistently achieves lower EED than the other two algorithms in comparison. This performance benefit is brought by our VM reuse strategy that minimizes the overhead including VM startup/shutdown and idle time. Moreover, as the workflow size increases, the number of modules in the same layer would also increase. Therefore, even a random selection would have a better chance to reuse a VM. In addition to the improvement in the VM overhead, the iterative step to further improve EED of the entire workflow also leads to lower EED than Min-min and Max-min because they only consider minimum partial EED for each module.

As discussed in [25], the percentage of modules assigned to their first choices is likely to be higher for Minmin than for Max-min, which results in a smaller EED. Max-min attempts to minimize the penalties incurred by running the modules with a longer execution time. For a workflow consisting of a module with a significantly longer execution time than the others, mapping this timeconsuming module to the best machine would allow concurrent execution with other shorter modules. In this case, Max-min is often preferred over Min-min as in the latter case, the shorter modules get to execute first and the longest modules get executed with many idle nodes for lower utilization rate. Thus, Max-min results in a more balanced workload across the nodes and a better EED.

Min-min and Max-min achieve similar utilization rates because they are more likely to choose the same resource for each module (as stated in the first step of Min-min and Max-min heuristics). Our algorithm achieves about 24% - 30% higher resource utilization than Min-min and Maxmin on average because of our VM reuse strategy that minimizes the overhead.

These experimental results show that the proposed Hi-WAE algorithm exhibits a better control over the execution time of a workflow compared to Min-min and Maxmin heuristics, and yields a significantly higher resource utilization rate by reducing the VM overhead during the workflow execution.

VII. Conclusion and Future Work

Many big data sciences are starting to use clouds as the major computing platform. We formulated a workflow scheduling problem in cloud environments. In general, it is of the cloud service provider’s interest to improve the system throughout to satisfy as many user requests as possible using the same hardware resources. Hence, the resource utilization rate is a very important performance metric, which, however, has not been sufficiently addressed by many existing workflow scheduling algorithms developed for clouds. Also, from the user’s perspective, one primary goal is to minimize the execution time of each individual workflow as stated in certain Quality of Service requirement.

Our mapping algorithm aims to achieve the dual goals of end-to-end delay performance and low overhead using a two-step approach. In the first step, modules are topologically sorted and mapped layer-by-layer to identify the best mapping strategy with the minimal execution time. If the final finish time is earlier than the latest finish time specified by the user, the extra allowed time delay is used to relax the mapping of modules to reduce the cost on VM setup and shutdown as well as the idle time. A backward module remapping procedure from the last layer toward the first layer is conducted to cut down the overhead. One strategy is to maximize the allocable volume of a VM to open the window for more modules to reuse it. After this backward mapping, any unused VM volume in terms of extra time is not requested. The simulation results demonstrated that our algorithm significantly reduces the VM cost compared with other representative cloud scheduling algorithms with a comparable or lower total execution time. It is of our future interest to implement and test this scheduling algorithm in local cloud testbeds and production cloud environments to support real-life largescale scientific workflows.

Acknowledgement

We would like to acknowledge Ms. Yang Zhao for her contributions to the preliminary design and implementation of the workflow scheduling algorithm proposed in the conference paper [1].

References

[1] M. Zhu, Q. Wu, and Y. Zhao, “A cost-effective scheduling algorithm for scientific workflows in clouds,” in *Proc. 31st IEEE International Performance Computing and COmmunication Conference*, 2012, pp. 256-265.

[2] A. Bala and I. Chana, “A survey of various workflow scheduling algorithms in cloud environment,” in *Proc. 2nd National Conference on Information and Communication Technology,* 2011, pp. 26-30.

[3] S. Zhang, X. Chen, and X. Huo, “Cloud computing research and development trend,” in *Proc. 2nd International Conference on Future Networks*, 2010, pp. 93-97.

[4] J. Yu, R. Buyya, and C. K. Tham, “Cost-based scheduling of scientific workflow applications on utility grids,” in *Proc. 1st IEEE International Conference on e-Science and Grid Computing (e-Science 2005)* , mMelbourne, Australia, Dec. 5-8, 2005

[5] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, *et al.*, “On the use of cloud computing for scientific workflows,” in *Proc. IEEE 4th International Conference on eScience*, 2008, pp. 640-645.

[6] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes, “A case for grid computing on virtual machines,” in *Proc. Distributed Computing Systems*, 2003, pp. 550-559.

[7] B. Sotomayor, K. Keahey, and I. Foster, “Combining batch execution and leasing using virtual machines,” in *Proc. 17th International Symposium on High Performance Distributed Computing*, Boston, Massachusetts, USA, June 23-27, 2008.

[8] J. Vockler, G. Juve, E. Deelman, M. Rynge, and B. Berriman, “Experiences using cloud computing for a scientific workflow application,” in *Proc. 2nd International Workshop on Scientific Cloud Computing*, 2011, pp. 15-24.

[9] B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran, “Streamline: A scheduling heuristic for streaming application on the grid,” in *Proc. 13th Multimedia Computing and Networking Conf.*, San Jose, CA, 2006.

[10] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, *et al.*, “The Eucalyptus open-source cloud-computing system,” in *Proc. IEEE International Symposium on Cluster Computing and the Grid,* 2009.

[11] Open Nebular. [Online]. Available: http://www.opennebula.org

[12] Amazon EC2. [Online]. Available: http://aws.amazon.com/ec2/

[13] Google App Engine. [Online]. Available: https://developers.google.com/app engine/

[14] Nimbus. [Online]. Available: http://nimbusproject.org

[15] E. Deelman, G. Singh, M. H. Su, J. blythe, and Y. Gil, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, pp. 219-237, July 2005.

[16] Q. Wu and Y. Gu, "Supporting distributed application workflows in heterogeneous computing environmentsm," in *Proc. 14th IEEE Int. Conf. on Parallel and Distributed Systems*, Melbourne, Australia, 2008, pp. 3-10.

[17] A. Sekhar, B. Manoj, and C. Murthy, "A state-space search approach for optimizing reliability and cost of execution in distributed sensor networks," in *Proc. Int. Workshop on Distributed Computing*, 2005, pp. 63-74.

[18] S. Topcuoglu and M. Wu, "Task scheduling algorithms for heterogeneous processors," in *Proc. 8th IEEE Heterogeneous Computing Workshop*, 1999, pp. 3-14.

[19] Q. Wu, M. Zhu, X. Lu, P. Brown, Y. Lin, Y. Gu, F. Cao, and M. Reuter, "Automation and management of scientific workflows in distributed network environments," in *Proc. 6th Int. Workshop on Sys. Man. Tech.*, 2010, pp. 1-8.

[20] Condor. [Online]. Available: http://www.cs.wisc.edu/condor

[21] DagMan. [Online]. Available: http://www.cs.wisc.edu/condor/dagman

[22] Globus. [Online]. Available: http://www.globus.org

[23] T. Ma and R. Buyya, "Critical-path and priority based algorithms for scheduling workflows with parameter sweep tasks on global grids," in *Proc. 17th Int. Symp. On Computer Architecture on High Performance Computing*, 2005, pp. 251-258.

[24] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, *et al.*, "Scheduling strategies for mapping application wormows onto the grid," in *Proc. IEEE International Symposium on High Performance Distributed Computing*, 2005, pp. 125-134.

[25] T. D. Braun, H. J. Siegel, and N. Beck, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed computing*, vol. 61, no. 6, pp. 810-837, 2011.

[26] Q. Wu, M. Zhu, Y. Gu, P. Brown, X. Lu, W. Lin, and Y. Liu, "A distributed workflow management system with case study of real-life scientific applications on grids," *Journal of Grid Computing*, vol. 10, no. 3, pp. 367-393, 2012.

[27] Q. Wu, Y. Gu, Y. Lin, and N. Ra, "Latency modeling and minimization for large-scale scientific workflows in distributed network environments," in *Proc. 44th Annual Simulation Symposium*, 2011, pp. 205-212.

Michelle M. Zhu received the Ph.D. degree in computer science from Louisiana State University in 2005. She spent two years in the Computer Science and Mathematics Division at Oak Ridge National Laboratory for her Ph.D. dissertation from 2003 to 2005. She is currently an associate professor in the Computer Science Department at Southern Illinois University, Carbondale. Her research interests include distributed and high-performance computing, remote visualization, bioinformatics, and sensor networks.

Fei Cao received the B.S. degree in software engineering from Zhejiang University, P.R. China, in 2007, the M.S. degree in computer science from California State University, Fullerton, in 2009. She is currently a Ph.D. student in the Department of Computer Science at Southern Illinois University, Carbondale. Her research interests include distributed computing and high-performance computing.

Chase Q. Wu received the B.S. degree in remote sensing from Zhejiang University, P.R. China, in 1995, the M.S. degree in geomatics from Purdue University in 2000, and the Ph.D. degree in computer science from Louisiana State University in 2003. He was a research fellow in the Computer Science and Mathematics Division at Oak Ridge National Laboratory during 2003-2006. He is currently an Associate Professor with the Department of Computer Science at University of Memphis. His research interests include parallel and distributed computing, computer networks, and cyber security.