On Workflow Scheduling for End-to-end Performance Optimization in Distributed Network Environments

Qishi Wu[1], Daqing Yun[1], Xiangyu Lin[1], Yi Gu[2], Wuyin Lin[3], Yangang Liu[3]

[1] Department of Computer Science
University of Memphis
Memphis, TN 38152
{qishiwu,dyun,xlin}@memphis.edu

[2] Department of Management, Marketing, Computer Science, and Information Systems
University of Tennessee at Martin
Martin, TN 38238
ygu6@utm.edu

[3] Atmospheric Sciences Division
Brookhaven National Laboratory
Upton, NY 11793
{wlin,lyg}@bnl.gov

Abstract. Next-generation computational sciences feature large-scale workflows of many computing modules that must be deployed and executed in distributed network environments. With limited computing resources, it is often unavoidable to map multiple workflow modules to the same computer node with possible concurrent module execution, whose scheduling may significantly affect the workflow's end-to-end performance in the network. We formulate this on-node workflow scheduling problem as an optimization problem and prove it to be NP-complete. We then conduct a deep investigation into workflow execution dynamics and propose a Critical Path-based Priority Scheduling (CPPS) algorithm to achieve Minimum End-to-end Delay (MED) under a given workflow mapping scheme. The performance superiority of the proposed CPPS algorithm is illustrated by extensive simulation results in comparison with a traditional fair-share (FS) scheduling policy and is further verified by proof-of-concept experiments based on a real-life scientific workflow for climate modeling deployed and executed in a testbed network.

Keywords: Scientific workflows, task scheduling, end-to-end delay.

1 Introduction

Next-generation computational sciences typically involve complex modeling, large-scale simulations, and sophisticated experiments in studying physical phenomena, chemical reactions, biological processes, and climatic changes [13, 25]. The processing and analysis of simulation or experimental datasets generated in these scientific applications require the construction and execution of domain-specific workflows consisting of many interdependent computing modules[1] in distributed network environments such as Grids or clouds for collaborative research and discovery. The network performance of such scientific workflows plays a critical role in the success of targeted science missions.

The computing workflows of scientific applications are often modeled as Directed Acyclic Graphs (DAGs) where vertices represent computing modules and edges represent execution precedence and data flow between adjacent modules, and could be managed and executed by either special- or general-purpose workflow engines such as Condor/DAGMan [1], Kepler [23], Pegasus [12], Triana [11], and Sedna [31]. In practice, workflow systems employ a static or dynamic mapping scheme to select an appropriate set of computer nodes in the network to run workflow modules to meet a certain performance requirement. No matter which type of mapping scheme is applied, it is often unavoidable to map multiple modules to the same node (i.e. node reuse) for better utilization of limited computing resources, leading to possible concurrent module execution. For example, in unitary processing applications that perform one-time data processing, workflow modules may run concurrently if they are independent of each other[2]; while in streaming applications with serial input data, even modules with dependency may run concurrently to process different instances of the data. Of course, concurrent modules on the same node may not always share resources if their execution times do not overlap completely due to their misaligned start or end times. The same is also true for concurrent data transfers.

Generally, in the case of concurrent module execution, the node's computing resources are allocated by kernel-level CPU scheduling policies such as the round-robin algorithm to ensure fine-grained fair share (FS). Similarly, a network link's bandwidth is also fairly shared by multiple independent data transfers that take place concurrently over the same link through the use of the widely deployed TCP or TCP-friendly transport methods. Such system-inherent fair-share scheduling mechanisms could reduce the development efforts of workflow systems, but may not always yield the best workflow performance, especially in distributed environments. We shall use an extremely simplified numerical example to illustrate the effect of on-node workflow scheduling.

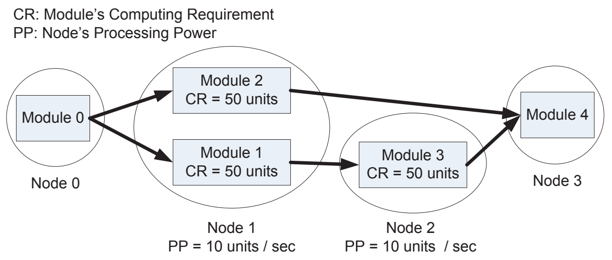

Fig. 1: A simple numerical example illustrating the effect of scheduling.

[1] Workflow modules are also referred to as tasks/subtasks, activities, stages, jobs, or transformations in different contexts.

[2] Two modules in a workflow are independent of each other if there does not exist any dependency path between them.

As shown in Fig. 1, a workflow consisting of five modules is mapped to a network consisting of four nodes. For simplicity, we only consider the execution time of Modules 1, 2, and 3. With a fair-share scheduler, the workflow takes 15 seconds to complete along the critical path (i.e. the longest path of execution time) consisting of Modules 0, 1, 3, and 4. However, if we let Module 1 execute exclusively ahead of Module 2, the completion time of the entire workflow is cut down to 10 seconds. This example reveals that the workflow performance could be significantly improved if concurrent modules on the same node are carefully scheduled.

The work in this paper is focused on task scheduling for *Minimum End-to-end Delay* (MED) in unitary processing applications under a given workflow mapping scheme. We formulate this on-node workflow scheduling problem as an optimization problem and prove it to be NP-complete. Based on the exact *End-to-end Delay* (ED) calculation algorithm, *extED* [17], we conduct a deep investigation into workflow execution dynamics and propose a *Critical Path-based Priority Scheduling* (CPPS) algorithm that allocates the node's computing resources to multiple concurrent modules assigned by the given mapping scheme to the same node to minimize the ED of a distributed workflow. We also provide an analytical upper bound of the ED performance improvement when the critical path remains fixed during workflow scheduling. The proposed CPPS algorithm is implemented and tested in both simulated and experimental network environments. The performance superiority of CPPS is illustrated by extensive simulation results in comparison with a traditional fair-share scheduling policy, and is further verified by proof-of-concept experiments based on a real-life scientific workflow for climate modeling deployed and executed in a testbed network.

The rest of the paper is organized as follows. In Section 2, we conduct a survey of related work on both workflow mapping and scheduling. In Section 3, we provide a formal definition of the workflow scheduling problem under study. In Section 4, we design the CPPS algorithm. In Section 5, we implement the proposed algorithm and present both simulation and experimental results. We conclude our work in Section 6.

2 Related Work

In this section, we provide a brief survey of related work on workflow optimization. There are two aspects of optimizing distributed tasks to improve the performance of scientific workflows: i) assigning the component modules in a workflow to suitable resources, referred to as workflow mapping[3], and ii) deciding the execution order and resource sharing policy among concurrent modules on computer nodes or processors, referred to as workflow/task scheduling. Both problems have been extensively studied in various contexts due to their theoretical significance and practical importance [7,28,30].

When network resources were still scarce in early years, workflow modules were often mapped to homogeneous systems such as multiprocessors [6, 22]. As distributed computing platforms such as Grids and clouds are rapidly developed and deployed, research efforts have shifted to workflow mapping in heterogeneous environments to

[3] The workflow mapping problem is occasionally referred to as a scheduling problem in the literature. In this paper, we designate it specifically as a mapping problem to differentiate it from the workflow/task scheduling problem under study.

utilize distributed resources at different geographical locations across wide-area networks [8, 9, 28]. However, several of these studies assume that computer networks are fully connected [30] or only consider independent tasks in the workflow [10]. These assumptions are reasonable under certain circumstances, but may not sufficiently model the complexity of real-life applications in wide-area networks. In real workflow systems, a greedy or similar type of approach is often employed for workflow mapping. For instance, Condor implements a matchmaking algorithm that utilizes classified advertisements (ClassAds) mechanism for mapping user's applications to suitable server machines [27]. Most existing mapping algorithms are centralized, but decentralized methods are desired for higher scalability. More recently, Rahman *et al.* proposed an approach to workflow mapping in a dynamic and distributed Grid environment using a Distributed Hash Table (DHT) based d-dimensional logical index space [26].

Considering the limit in the scope and amount of networked resources, a reasonable workflow mapping algorithm inevitably results in a mapping scheme where multiple modules are mapped to the same node, which necessitates the scheduling of concurrent modules. On-node processor scheduling has been a traditional research subject for several decades, and many algorithms have been proposed ranging from proportional, priority-based, to fair share [18, 19], to solve scheduling problems under different constraints with different objectives. Most traditional processor scheduling algorithms consider multiple independent processes running on a single CPU with a goal to optimize the waiting time, response time, and turnaround time of individual processes, or the throughput of the system. The multi-processor and multi-core scheduling on modern machines has attracted an increasing amount of attention in recent years [21, 24].

Our work is focused on a particular scheduling problem to achieve MED of a DAG-structured workflow in distributed environments under a given mapping scheme. A similar DAG-structured project planning problem dated back to late 1950's and was tackled using Program/Project Evaluation Review Technique and Critical Path Method (PERT/CPM) [14, 20]. PERM is a method to analyze the involved tasks in completing a given project and identify the minimum time needed to complete the total project. CPM calculates the longest path of planned activities to the end of the project, and the earliest and latest that each activity can start and finish without making the project longer. The workflow scheduling problem differs from the project planning problem in that the computing resources are shared among concurrent modules and the execution time of each individual module is unknown until a particular task scheduling scheme is determined. This type of scheduling problems are generally NP-complete. Garey *et al.* compiled a great collection of similar or related multi-processor scheduling problems in [15], which were tackled by various heuristics such as list scheduling and simple level algorithm [16].

In practical system implementations, concurrent tasks are always assigned the same running priority since the fair share of CPU cycles is well supported by modern operating systems that employ a round-robin type of scheduling algorithms. However, in this paper, we go beyond the system-inherent fair-share scheduling to further improve the end-to-end performance of scientific workflows through the use of an on-node scheduling strategy taking into account the global workflow structure and resource sharing dynamics in distributed environments.

3 Workflow Scheduling Problem

In this section, we construct analytical cost models and formulate the workflow scheduling problem, which is proved to be NP-complete.

3.1 Cost Models

We model a workflow as a Directed Acyclic Graph (DAG) $G_w = (V_w, E_w)$, $|V_w| = m$, where vertices represent computing modules starting from the start module w_0 to the end module w_{m-1}. A directed edge $e_{i,j} \in E_w$ represents the dependency between a pair of adjacent modules w_i and w_j. Module w receives a data input of size z from each of its preceding modules and performs a predefined computing routine, whose computing requirement (CR) or workload is modeled as a function of the aggregate input data sizes. Module w sends a data output to each of its succeeding modules after it completes its execution. In this workflow model, we consider a module as the minimal execution unit, and ignore the inter-module communication cost on the same node. For a workflow with multiple start/end modules, we can always convert it to this model by adding a virtual start/end module with $CR = 0$ and connected to all the original start/end modules with data transfer size $z = 0$.

We model an overlay computer network as an arbitrary weighted graph $G_c = (V_c, E_c)$, consisting of $|V_c| = n$ nodes interconnected by $|E_c|$ overlay links. A normalized variable PP_i is used to represent the overall processing power of node v_i without specifying its detailed system resources. Link $l_{i,j}$ from v_i to v_j is associated with a certain bandwidth $b_{i,j}$. It is assumed that the start module w_0 serves as a data source on the source node v_s without any computation to supply all initial data needed by the application and the end module w_{m-1} performs a terminal task on the destination node v_d without any further data transfer.

Based on the above models, we can use an existing mapping algorithm to compute a mapping scheme under the following constraints on workflow mapping and execution/transfer precedence [17]:

- Each module/edge is required to be mapped to only one node/link.
- A computing module cannot start execution until all its required input data arrive.
- A dependency edge cannot start data transfer until its preceding module finished execution.

A mapping scheme $M : G_w \to G_c$ is mathematically defined as follows:

$$M : G_w \to G_c = \begin{cases} w \to c, \forall w \in V_w, \exists c \in V_c; \\ l(c_{i'}, c_{j'}) \in E_c, \begin{cases} \text{if } w_i \to c_{i'}, w_j \to c_{j'}, e(w_i, w_j) \in E_w, \\ 0 \leq i, j \leq |V_w| - 1, 0 \leq i', j' \leq |V_c| - 1. \end{cases} \end{cases} \tag{1}$$

Once a mapping scheme is obtained, we can further convert the above workflow and network models to virtual graphs as follows: each dependency edge in the workflow is replaced with a virtual module whose computational workload is equal to the corresponding data size, and each mapped network link is replaced by a virtual node whose processing power is equal to the corresponding bandwidth. This conversion facilitates workflow scheduling as we only need to focus on the module execution time. We use t_w^s

Table 1: Parameters in the cost models and algorithm design.

Parameters	Definitions
$G_w = (V_w, E_w)$	a computing workflow
m	the number of modules in the workflow
w_i	the i-th computing module
w_0	the start computing module
w_{m-1}	the end computing module
$e_{i,j}$	the dependency edge from module w_i to w_j
z	the data size of dependency edge e
CR_i	the computing requirement of module w_i
$G_c = (V_c, E_c)$	a computer network
n	the number of nodes in the network
v_i	the i-th network of computer node
v_s	the source node
v_d	the destination node
PP_i	the processing power of node v_i
$l_{i,j}$	the network link from node v_i to v_j
$b_{i,j}$	the bandwidth of link $l_{i,j}$

and t_w^f to denote the execution start and finish time of module w. For convenience, we tabulate the notations used in the cost models in Table 1, some of which will be used in the algorithm design.

3.2 Problem Definition

Since the start time of a module in the workflow depends on the end time of its preceding module(s), even independent modules assigned to the same node may not always run in parallel. Therefore, task scheduling is only applicable to concurrent modules that run simultaneously at least for a certain period of time. We formally define the On-Node Workflow Scheduling (ONWS) problem as follows:

Definition 1. (***ONWS***) *Given a DAG-structured computing workflow $G_w = (V_w, E_w)$, a heterogeneous overlay computer network $G_c = (V_c, E_c)$ and a mapping scheme $M : G_w \to G_c$, we wish to find a job scheduling policy on every computer node with concurrent modules such that the mapped workflow achieves:*

$$\text{MED} = \min_{\text{all possible job scheduling policies}} (T_{\text{ED}}) . \qquad (2)$$

3.3 NP-Completeness Proof

We first transform the original ONWS problem to its decision version, referred to as ONWS-Decision, with a set of notations commonly adopted in the definitions of traditional multi-processor scheduling problems in the literature.

Definition 2. (***ONWS-Decision***) *Given a set T of tasks t, each having a length(t) and executed by a specific processor p(t), a number $m \in Z^+$ of processors, partial order $\prec$*

on T, and an overall deadline $D \in Z^{+}$, is there an m-processor preemptive schedule for T that obeys the precedence constraints and meets the overall deadline?

Note that in Definition 2, the workflow in the original ONWS problem is expressed as a set of tasks with a partial-order relation and the given mapping scheme is expressed as a set of node assignments or requirements for all tasks. Also, we consider preemptive scheduling in our problem. The difficulty of ONWS-Decision is given by Theorem 1, which is proved by showing that the *m*-processor bound unit execution time (MBUET) system scheduling problem in [16], an existing NP-complete problem as defined in Definition 3, is a special case of ONWS-Decision.

Definition 3. (***MBUET***) *Given a set T of tasks, each having length(t) = 1 (unit execution time) and executed by a specific processor p(t), an arbitrary number $m \in Z^{+}$ of processors, partial order $\prec$ of a forest on T, and an overall deadline $D \in Z^{+}$, is there an m-processor schedule for T that obeys the precedence constraints and meets the overall deadline?*

Theorem 1. *The ONWS-Decision problem is NP-complete.*

Proof. Obviously, the MBUET scheduling problem is a special type of ONWS-Decision problem with $length(t)$ restricted to be 1 for all tasks $t \in T$ and partial order $\prec$ restricted to be a forest. For preemptive scheduling as considered in ONWS-Decision, a task is not required to finish completely without any interruption once it starts execution. However, if we restrict the length of all tasks to be 1 (the smallest unit of time) in the input of ONWS-Decision, each task would finish in its entirety once it is assigned to the designated processor for execution. Moreover, the partial order $\prec$ of a forest is a special DAG structure of workflow topology. Therefore, the MBUET problem polynomially transforms to the preemptive ONWS-Decision scheduling problem. Since MBUET is NP-complete [16], the general ONWS-Decision problem is also NP-complete. The validity of the NP-completeness proof by restriction is established in [15], where "restriction" constrains the given, not the question of a problem.

4 Algorithm Design

4.1 Analysis of Resource Sharing Dynamics in Workflows

Since the end-to-end delay (ED) of a workflow is determined by its critical path, i.e. the path of the longest execution time, a general strategy for workflow scheduling is to reduce the execution time of critical modules (i.e. modules on the critical path) by allocating to them more resources than those non-critical modules that are running concurrently. Ideally, the MED would be achieved if all possible execution paths from the start module to the end module have the same length of execution time.

We first present a theorem on the resource sharing dynamics among the concurrent modules assigned to the same node, which will be used in the design of our scheduling algorithm.

Theorem 2. *The finish time of the last completed module among k concurrent modules executed on the same node with a single fully-utilized processor of processing power PP is a constant, $\frac{\sum_{i=1}^{k} CR_i}{PP}$.*

Proof. Since the total workload of all k modules is fixed, i.e. $\sum_{i=1}^{k} CR_i$, and the processor is fully operating, no matter how the modules are scheduled, the total execution time remains unchanged and is bounded by the finish time of the last completed module.

To understand the significance of workflow scheduling, we investigate a particular scheduling scenario shown in Fig. 2 where the best performance improvement could be achieved over a fair-share scheduling policy with the following conditions:

- There are k modules running concurrently on the same node v_1 during their entire execution time period;
- Among k modules, one is a critical module and $k-1$ are non-critical modules, and all of them are of the same computing requirement (CR);
- Each non-critical module is the only module on its execution path (except for the start and end modules).

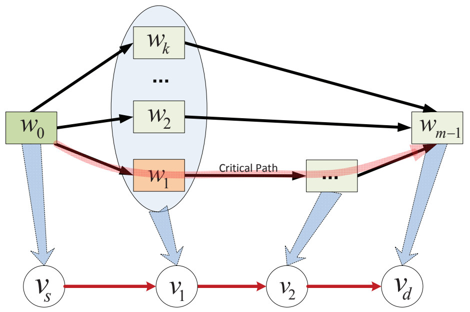

Fig. 2: A scheduling scenario that achieves the upper bound performance.

Based on the scheduling scenario depicted in Fig. 2, we have the following theorem:

Theorem 3. *The workflow's MED performance improvement of any on-node task scheduling policy with a fixed critical path over fair share is upper bounded by 50%.*

Proof. We first justify that the conditions in Fig. 2 are necessary for achieving the upper bound of the workflow's MED improvement.

Let x be the original execution time of the critical module on v_1 using the fair-share scheduling policy. If there were any non-critical modules on v_1 that finished before x, even if we let the critical module w_1 run exclusively first, it would take longer than $\frac{x}{k}$ to complete. Thus, we would not be able to reduce the execution time of the critical module to the minimum time possible, i.e. $\frac{x}{k}$. On the other hand, if there were any non-critical modules on v_1 that finished after x, from Theorem 2, we know that the total execution time on this node would be greater than x, and hence we would not be able to reduce the length of the critical path to x.

The above discussion concludes that all k modules on node v_1 must share resources during their entire execution time and finish at the same time x to achieve the maximum possible reduction on the length of the critical path. Since we consider fair share as the comparison base in Theorem 2, the CRs of all k modules must also be identical. In this scheduling scenario, if we let the critical module w_1 execute exclusively first, its finish time $t_{w_1}^f$ would be reduced from x to $\frac{x}{k}$, which is the best improvement possible with k concurrent modules on the same node.

We use y to denote the sum of the execution time for the rest of the modules on the critical path (excluding w_1). From Theorem 2, we have $max(t_{w_i}^f) = x$, $i = 2, 3, \ldots, k$, which means that the latest finish time of the last completed non-critical module would be still x. Since the new length of the critical path (i.e. MED) becomes $\frac{x}{k} + y$, the upper bound MED improvement is achieved if this new length is equal to the latest finish time of any non-critical module, i.e. $\frac{x}{k} + y = x$. It follows that $y = \frac{k-1}{k}x$. Therefore, the MED improvement over fair share is $\Delta = \frac{(x+y)-x}{x+y} = \frac{k-1}{2k-1}$, which is 1/2 or 50% as $k \to \infty$.

4.2 Critical Path-based Priority Scheduling (CPPS) Algorithm

Algorithm 1 CPPS($G_w{}'$, $G_c{}'$, f)
Input: A converted workflow graph $G_w{}' = (V_w{}', E_w{}')$, a converted network graph $G_c{}' = (V_c{}', E_c{}')$, a given mapping scheme $f : G_w{}' \to G_c{}'$.
Output: the scheduling policies on all the mapping nodes.

```
1: Calculate the execution time $T_{FS}$ for all the modules in $G_w{}'$ using a fair-share scheme;
2: Calculate the critical path $CP_{FS}$ and its length $T(CP_{FS})$ based on $T_{FS}$;
3: Calculate the execution time $T_{CPPS}$ for all the modules in $G_w{}'$ using Alg. 2, which in turn
   uses Alg. 3 to decide the priorities for all the modules;
4: Calculate the new $CP_{CPPS}$ and its length $T(CP_{CPPS})$ based on $T_{CPPS}$;
5: if $T(CP_{CPPS}) \geq T(CP_{FS})$ then
6:     return the fair-share schedule and $T(CP_{FS})$.
7: else
8:     return the new CPPS schedule and the ED $T(CP_{CPPS})$ of the mapped workflow.
```

We propose a *Critical Path-based Priority Scheduling* (CPPS) algorithm to solve the ONWS problem. The CPPS algorithm produces a set of processor scheduling schemes for all mapping nodes that collectively cut down the execution length of the critical path in the entire workflow to achieve MED.

The pseudocode of the proposed CPPS algorithm is provided in Alg. 1, which first calculates the critical path based on the fair-share (FS) scheduling, and then uses Algs. 2 and 3 to compute the new task schedule of concurrent modules on each mapping node. In most cases, the new schedule outperforms the FS schedule. However, if the new schedule does not improve the MED performance, the CPPS algorithm simply rolls back to the FS schedule.

In Alg. 2, we calculate the independent set of each module in line 8 and find the set set_0 of modules with the earliest start time in line 12. If there is only one module in set_0, this module is executed immediately; otherwise, a scheduling strategy needs to be decided. In line 18, we recalculate the percent of processing power that is allocated to those modules in set_0. In lines 18-32, a new scheduling policy is generated and applied to all concurrent modules.

Algorithm 2 extEDOnNode($G_w{}'$, $G_c{}'$, f, T_{FS}, CP_{FS})

Input: A converted workflow graph $G_w{}' = (V_w{}', E_w{}')$, a converted network graph $G_c{}' = (V_c{}', E_c{}')$, a mapping scheme $f : G_w{}' \rightarrow G_c{}'$, the execution time T_{FS} of each module under the fair-share scheme, and the critical path CP_{FS} based on T_{FS}.

Output: the new execution time T_{CPPS} of all modules and the ED $T(CP_{CPPS})$ of the mapped workflow.

```
1:  $t^s(set)$: the set of start times of all the modules in a $set$;
2:  $t^f(set)$: the set of finish times of all the modules in a $set$;
3:  $ids(w)$: the independent set of module $w$ on the same node (excluding $w$);
4:  $est(set) = \{w|\ w \in set$ and $t^s_w = \min(t^s(set))\}$, i.e. the set of modules with the earliest start
    time in a $set$;
5:  $ready(set) = \{w|\ w \in set$ and $w$ is "$ready$" to execute$\}$;
6:  $TBF(w)$: the partial workload of module $w$ to be finished;
7:  for all module $w_i \in V_w{}'$ do
8:      Find $ids(w_i)$;
9:      Set $w_i$ as "$unfinished$";
10: Set $w_0$ as "$ready$";
11: while exist "$unfinished$" modules $\in V_w{}'$ do
12:     Find $set_0 = est(ready(V_w{}'))$;
13:     for all module $w_i \in set_0$ do
14:         if $|ids(w_i)| == 0$ then
15:             Execute $w_i$, calculate $T_{CPPS}(w_i)$ and set $w_i$ as "$finished$";
16:         else
17:             Find $set_1 = \{w|w$ is "$ready$" and $w \in est(ids(w_i) \cup w_i)\}$;
18:             OnNodeSchedule($G_w{}'$, $G_c{}'$, $f$, $T_{FS}$, $T_{CPPS}$, $w_i \cup ids(w_i)$, $CP_{FS}$);
19:             Estimate $t^f(set_1)$;
20:             Find $set_2 = \{w \mid w \in ids(w_i)$ & $w \notin set_1$, $t^s(w) < \min(t^f(set_1))$, and $w$ is "$ready$"
                and "$unfinished$"$\}$;
21:             if $|set_2| > 0$ then
22:                 for all module $w_j \in set_1$ do
23:                     From $t^s(w|w \in set_1)$ to $\min(t^s(set_2))$: 1) finish part of $TBF(w_j)$ with the new
                        scheduling policy determined in line 18; 2) calculate the partial amount of
                        $T_{CPPS}$;
24:                     Update the new $t^s_{w_j} = \min(t^s(set_2))$;
25:             else
26:                 for all module $w_j \in set_1$ do
27:                     if $t^f_{w_j} == \min(t^f(set_1))$ then
28:                         Execute $w_j$, calculate $T_{CPPS}(w_j)$, and set $w_j$ as "$finished$";
29:                     else
30:                         From $t^s(w|w \in set_1)$ to $\min(t^f(set_1))$: 1) finish part of $TBF(w_j)$ with the
                            new scheduling policy determined in line 18; 2) calculate the partial amount
                            of $T_{CPPS}$;
31:                         Update the new $t^s_{w_j} = \min(t^f(set_1))$;
32:     Mark all ready modules as "$ready$";
33: Compute the CP based on the time components $T_{CPPS}(w_i)$ for all $w_i \in V_w{}'$;
34: return the ED $T(CP_{CPPS})$ of the mapped workflow.
```

Algorithm 3 OnNodeSchedule($G_w{}'$, $G_c{}'$, f, T_{FS}, $w_i \cup ids(w_i)$, CP_{FS})

Input: A converted workflow graph $G_w{}' = (V_w{}', E_w{}')$, a converted network graph $G_c{}' = (V_c{}', E_c{}')$, a given mapping scheme $f : G_w{}' \to G_c{}'$, and a module set $w_i \cup ids(w_i)$ that combines w_i and its independent set $ids(w_i)$.

Output: the percentage $per(w)$ of resource allocation for all modules in $w_i \cup ids(w_i)$.

```
1: $w_{cp}$: module of the critical path $cp$ (i.e. critical module);
2: $w_{non\_cp}$: module that is not on the critical path (i.e. non-critical module);
3: $per(w)$: percentage of processing power allocated to module $w$;
4: $t_{OnNode}$: the estimated execution time of a module under the new scheduling strategy;
5: $CP(w)$: the critical path (CP) consisting of module $w$;
6: $CP_L(w)$: the left CP segment from the start module to module $w$ (i.e. the CP of the left-side
   partial workflow ending at $w$);
7: $CP_R(w)$: the right CP segement from module $w$ to the end module (i.e. the CP of the right-side
   partial workflow starting at $w$);
8: $LFT(n_i)$: the latest possible finish time of concurrent modules assigned to node $n_i$;
9: $T_{exclusive}(w_i)$: execution time of $w_i$ when running exclusively on its mapping node $map(w_i)$;
10: $TBF(w)$: the partial workload of module $w$ to be finished;
11: $map(w)$: the node to which module $w$ is mapped;
12: for all modules $w_i$ in $w_i \cup ids(w_i)$ do
13:     Calculate $CP(w_i) = CP_L(w_i) + CP_R(w_i)$ based on $T_{FS}$ and $T_{CPPS}$;
14: Find $w_{cp} = \{w | CP(w) = max\{CP(w) | w \in ids(w_i) \cup w_i\}\}$;
15: Estimate execution time $T_{exclusive}(w_{cp})$;
16: Calculate the latest possible finish time $LFT(map(w_i))$ of modules mapped to $map(w_i)$;
17: bool $flag$ = false;
18: for all modules $w_i \in w_i \cup ids(w_i)$ do
19:     Estimate the amount of time $T_{tbf}$ needed to finish $TBF(w_i)$;
20:     if $w_i$ is $w_{non\_cp}$ then
21:         $\Delta t_1 = min\{LFT(map(w_i)) - T_{tbf}, T_{exclusive}(w_{cp})\}$;
22:         $\beta = |\{w_k \in CP(w_i)$ and $w_k \in \cup\{CP(w_j) | w_j \in ids(w_i)\}\}|$;
23:         if $T(CP(w_i)) + \beta \Delta t_1 \geq T(CP_{FS})$ then
24:             $flag = $ true;
25:             $\Delta t_2 = T(CP_{FS}) - T(CP(w_i))$;
26:             $t_{OnNode} = T_{tbf} + \Delta t_2$;
27:             Calculate $per(w_i)$ according to $t_{OnNode}$;
28:             mark $w_i$;
29: if $flag$ == true then
30:     for all modules $w_i \in w_i \cup ids(w_i)$ do
31:         if $w_i$ is not marked in line 28 then
32:             $per(w_i) = \left(1 - \sum_{w_i \in w_i \cup ids(w_i)}^{w_i\ marked} per(w_i)\right) \Big/ \left(\left|\{w_i \cup ids(w_i)\}\right| - \left|\{w_i | w_i\ marked\}\right|\right)$;
33: else
34:     for all modules $w_i \in w_i \cup ids(w_i)$ do
35:         if $w_i$ is $w_{cp}$ then
36:             $per(w_i)$ = 1.0;
37:         else
38:             $per(w_i)$ = 0.0;
39:     return the ID of $w_i$.
40: return $per(w_i)$ for all modules $w_i \in w_i \cup ids(w_i)$.
```

Alg. 3 performs the actual on-node scheduling, where only two cases need to be considered: 1) when $per(w_{cp})$ is 100% and other $per(w_{non\_cp})$ is 0%, 2) when all $per(w)$ are between 0 and 100%. Whenever possible, we wish to run the critical module exclusively first to cut down the length of the globe CP as much as possible. Accordingly, Alg. 2 considers the following two possible scenarios for all modules $w_i \in w_i \cup ids(w_i)$ depending on the output of Alg. 3:

1) When $per(w_{cp}) = 100\%$ and $per(w_{non\_cp}) = 0$: since Alg. 3 returns the ID of w_{cp}, we execute the critical module w_{cp} from the time point $t^s(w|w \in set_1)$ to $\min(t^s(set_2))$ exclusively; while for other modules, we suspend them to wait for w_{cp} to finish. During this period of time, w_{cp} may be entirely or partially completed. We then execute all unfinished modules (their unfinished part $TBF(w_i)$) that are mapped to node $map(w_{cp})$ in a fair-share manner until the number of concurrent modules assigned to this node changes again.
2) When all $per(w)$ are between 0 and 100%: we execute each module with its own percent $per(w_i)$ of resource allocation until the number of concurrent modules changes.

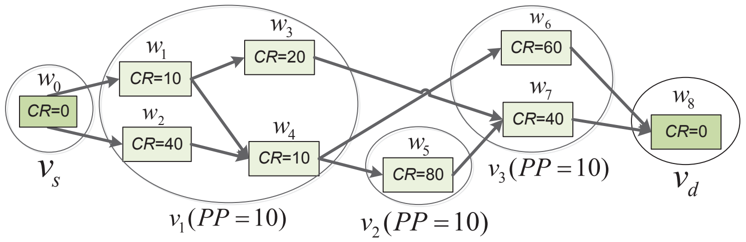

Fig. 3: A simple numerical example used for illustrating the scheduling algorithms.

We shall use a simple numerical example to illustrate the CPPS scheduling process. As shown in Fig. 3, a computing workflow consisting of nine modules $w_0, w_1, \ldots,$ and w_8 is mapped to a computer network consisting of 5 nodes v_s, v_1, v_2, v_3 and v_d. The modules' computing requirements (CR) and the nodes' processing powers (PP) are marked accordingly except for the start/end modules mapped to the source/destination nodes, whose execution time is not considered (i.e. $CR = 0$).

As shown in Fig. 4, we first compute the execution start and end time $T_{FS} : t^s_w | t^f_w$ of each module under the fair-share (FS) scheduling policy. In this scheduling scenario, the workflow takes 20 seconds to complete along the critical path (CP) consisting of modules $w_0, w_2, w_4, w_5, w_7,$ and w_8.

Now we describe the CPPS scheduling process. Fig. 5 illustrates the scheduling status when modules w_3 and w_4 are being scheduled in the CPPS algorithm. At this point of time, the left shadowed part has been scheduled by CPPS while the right shadowed part is still estimated based on FS. Note that the module start time t^s_w is always counted from the point when the module is ready to execute, not from the point when it actually starts execution. Right after w_1 finishes execution, both w_3 and w_4 are ready to execute with possible resource sharing. To decide the scheduling between them, we

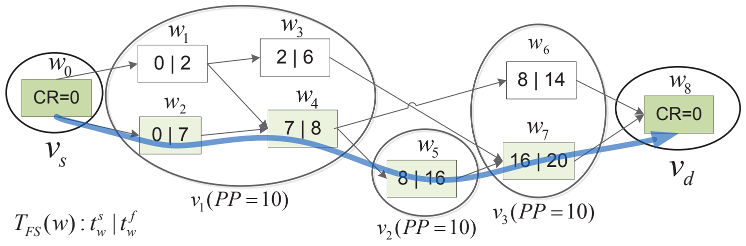

Fig. 4: The execution start time t_w^s and finish time t_w^f of each module calculated under the FS scheduling policy, listed in the form of $T_{FS}(w) : t_w^s | t_w^f$.

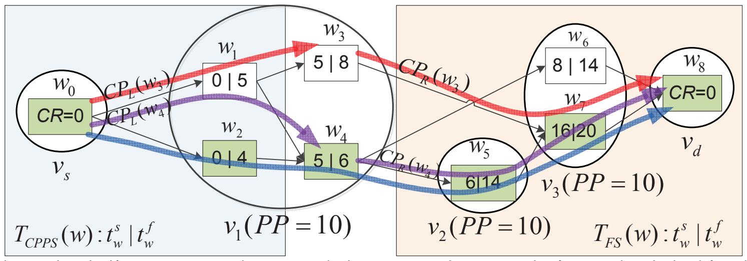

Fig. 5: The scheduling status when modules w_3 and w_4 are being scheduled in the CPPS algorithm. The left shadowed part has been scheduled by CPPS while the right shadowed part is estimated based on FS.

need to compute the longest (critical) path that traverses each of them by concatenating the left and right segments of its critical path. For w_3, the length of its left CP segment (w_0, w_1, and w_3), denoted as $CP_L(w_3)$, is 5 under CPPS, while the length of its right CP segment (w_3, w_7, and w_8), denoted as $CP_R(w_3)$, is 8 (including its own execution time), which is estimated using the FS-based measurements in Fig. 4. Similarly, for w_4, the lengths of its left and right CP segments are 5 under CPPS and 13 based on FS as shown in Fig. 4, respectively. Since $CP(w_4) = CP_L(w_4) + CP_R(w_4) = 18$ is longer than $CP(w_3) = CP_L(w_3) + CP_R(w_3) = 13$, w_4 is to be executed exclusively first. It is estimated that the length of $CP(w_3)$ would not exceed the length of the original global CP_{FS} based on FS, which is 20, even if we let w_4 run to its completion with exclusive CPU utilization, so in this case, we assign the entire CPU to w_4 until it finishes; otherwise, w_4 would execute exclusively until a certain point and then share with w_3 in a fair manner such that the length of $CP(w_3)$ does not exceed the length of the original global CP_{FS} based on FS. The new execution time of w_3 and w_4 is updated in Fig. 5. This scheduling process is repeated on every mapping node until all the modules are properly scheduled.

5 Performance Evaluation

We evaluate the performance of the proposed CPPS algorithm in both simulated and experimental network environments in comparison with the fair-share scheduling policy, which is commonly adopted in real systems.

5.1 Simulation-based Performance Comparison

Simulation Settings In the simulation, we implement the *CPPS* algorithm in C++ and run it on a Windows 7 desktop PC equipped with Intel(R) Core(TM)2 Duo CPU E7500 of 2.93GHz and 4.00GB memory.

We develop a separate program to generate test cases by randomly varying the problem size denoted as a three-tuple $(m, |E_w|, n)$, i.e. m modules and $|E_w|$ edges in the workflow, and n mapping nodes in the network. For a given problem size, we randomly vary the module complexity and data size within a suitably selected range of values, and create the DAG topology of a workflow as follows: 1) Lay out all the modules sequentially; 2) For each module, create an input edge from a randomly chosen preceding module and create an output edge to a randomly chosen succeeding module (note that the start module has no input and the end module has no output); 3) Repeatedly pick up a pair of modules on a random basis and add a directed edge from left to right between them until we reach the given number of edges.

Given the workflow structure and the number of mapping nodes with randomly generated processing power, we randomly generate the mapping scheme but with some specific topological constraints. In observation of the topological structures of real networks such as ESnet [2] and Internet2 [3], we first topologically sort all modules and then map the modules from each layer of the workflow to the nodes that are within the proximity of the corresponding layer in the network. In the same layer with multiple modules/nodes, a greedy approach is adopted for node assignment.

Note that in the network, we do not need to specify the number of links as a parameter because there must exist a link between two mapping nodes where two adjacent modules are mapped to ensure the feasibility of the given mapping scheme. A random bandwidth is then selected from a suitable range of values and assigned to each link. Since other links without any dependency edges mapped to will not affect the simulation results, they are simply ignored.

Simulation Results To evaluate the performance of the proposed CPPS algorithm, we randomly generate 4 groups of test cases with 4 different numbers of nodes, i.e. $n = 5, 10, 15,$ and 20. For each number of nodes (i.e. each group), we generate 20 problem sizes, indexed from 1 to 20, by varying the number of modules from 5 to 100 at an interval of 5 with a random number of edges.

For each problem size $(m, |E_w|, n)$, we generate 10 random problem instances for workflow scheduling with different module complexities, data sizes, node processing powers, link bandwidths, and mapping schemes, and then calculate the average of the performance measurements under both fair-share (FS) and CPPS scheduling. The MED performance improvement or speedup of CPPS over FS, defined as $\frac{T_{FS} - T_{CPPS}}{T_{FS}} \times 100\%$, is tabulated in Table 2. Note that for the problem size indexed by 1 with $m = 5$, when $n > m$ (i.e. $n = 10, 15,$ and 20), the mapping scheme maps each module one-to-one to a different node without any resource sharing, and hence the scheduling is not applicable (marked by "–" in the table); so are the cases for the problem size indexed by 2 with $m = 10$ and $n = 15$ and 20, and the problem size indexed by 3 with $m = 15$ and $n = 20$. The average performance improvement measurements together with their corresponding standard deviations are plotted in Fig. 6.

Table 2: MED improvement percentage of CPPS over FS.

Prb	Num of Mods	MED Improvement Percentage (%)			
Idx	m	n=5	n=10	n=15	n=20
1	5	0.4023	–	–	–
2	10	9.3800	0.0834	–	–
3	15	12.2844	4.1384	1.5431	–
4	20	12.3537	6.9193	9.2536	5.2417
5	25	12.8543	7.9172	11.3095	5.3134
6	30	13.6095	11.3572	11.7418	5.7544
7	35	14.0034	11.7431	13.4203	7.8624
8	40	15.4378	12.8595	13.5076	8.9894
9	45	15.3575	13.0498	13.8334	11.4692
10	50	18.8860	14.2790	11.1567	12.6608
11	55	17.8824	12.8271	13.1352	14.1371
12	60	14.6911	14.6843	14.1821	12.6241
13	65	17.3636	20.5909	16.7078	13.6390
14	70	17.1467	17.5231	17.9146	15.1107
15	75	15.9394	19.3904	16.6014	13.4181
16	80	21.3327	18.2815	18.5939	13.8302
17	85	17.8155	22.2571	18.8757	17.4691
18	90	19.0516	19.2820	17.6146	17.1279
19	95	18.7070	19.0163	17.6803	16.8793
20	100	20.2888	19.0798	18.4336	19.3523

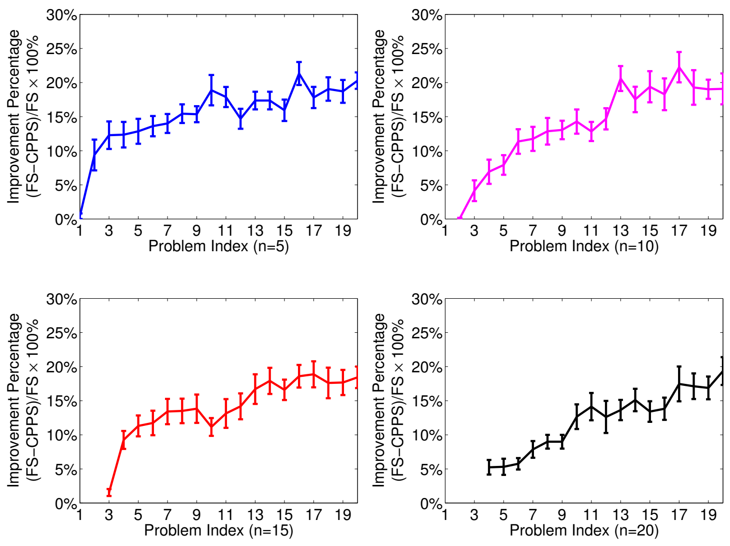

Fig. 6: MED improvement of CPPS over FS.

The space for performance improvement largely depends on the given mapping scheme. In small problem sizes, or when the number of modules is comparable to the number of nodes, the modules are likely to be mapped to the nodes in a uniform manner, resulting in a low level of resource sharing, unless there exist some nodes whose processing power are significantly higher than the others in the network. Hence, in these cases, the MED improvement of CPPS over FS is not very obvious. However, as the problem size increases, more modules might be mapped to the same node with more resource sharing, hence leading to a higher performance improvement. This overall trend of performance improvement is clearly reflected in Fig 6.

5.2 Proof-of-concept Experimental Results Using Climate Modeling Workflow

Weather Research and Forecasting (WRF) The Weather Research and Forecasting (WRF) model [29] has been widely used for regional to continental scale weather forecast. It is also one of the most widely used limited-area model for dynamical downscaling of climate projection by global climate models to provide regional details. The workflow for typical applications of WRF model takes multiple steps, including data preparation and preprocessing, actual model simulation, and postprocessing. Each step could be computationally intensive and/or involve a large amount of data transfer and processing. For a specific climate research project, such procedures have to be performed repeatedly, in the context of either routine short-term weather forecast, or periodic re-initialization of model in dynamical downscaling over the length of a climate projection. Moreover, because of the chaotic nature of the atmospheric system and unavoidable errors in the input data and the imperfection of the model, ensemble approaches have to be adopted with a sufficiently large number of simulations, with slight perturbations to initial conditions or physical parameters, to enhance the robustness of the prediction, or to provide probabilistic forecast for problems of interest. Each single simulation may require a full or partial set of preprocessing and postprocessing, in addition to the model simulation. Collectively, a project in this nature may involve the execution of an overwhelmingly large number of individual programs. A workflow-based management and execution is hence extremely useful to automate the procedure and efficiently allocate the resources to carry out the required computational tasks.

Climate Modeling Workflow Structure The WRF model [4] is able to generate two large classes of simulations either with an ideal initialization or utilizing real data. In our workflow experiments, the simulations are generated from real data, which usually require preprocessing from the WPS package [5] to provide each atmospheric and static field with fidelity appropriate to the chosen grid resolution for the model.

As shown in Fig. 7, the WPS consists of three independent programs: geogrid.exe, ungrib.exe, and metgrid.exe. The geogrid program defines the simulation domains and interpolates various terrestrial datasets to the model grids. The user can specify information in the namelist file of WPS to define simulation domains. The ungrib program "degrib" the data and writes them in a simple intermediate format. The metgrid program horizontally interpolates the intermediate-format meteorological data that are extracted by the ungrib program onto the simulation domains defined by the geogrid program.

The interpolated metgrid output can then be ingested by the WRF package, which contains an initialization program real.exe for real data and a numerical integration program wrf.exe. The postprocessing model consists of ARWpost and GrADs. ARWpost reads-in WRF-ARW model data and creates output files for display by GrADS.

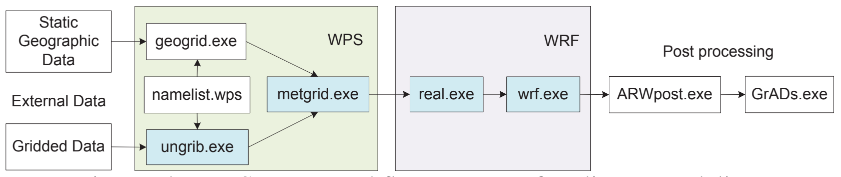

Fig. 7: The WPS-WRF workflow structure for climate modeling.

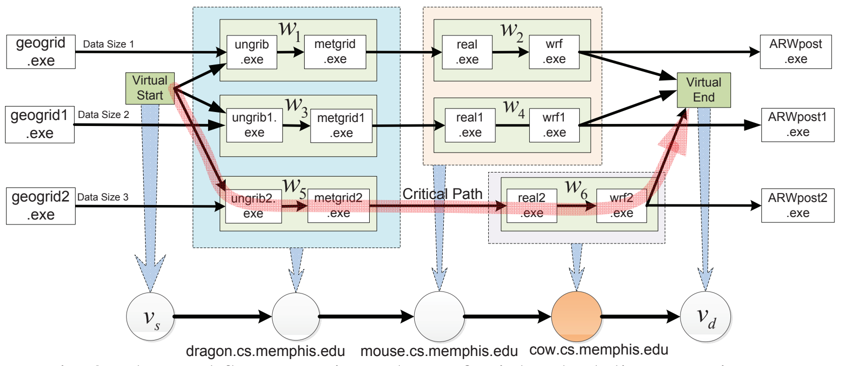

Fig. 8: The workflow mapping scheme for job scheduling experiments.

Experimental Settings and Results As shown in Fig. 8, in our experiments, we duplicate the entire WPS-WRF workflow to generate three parallel pipelines processing three different instances of input data of sizes 106.03 MBytes, 106.03 MBytes, and 740.89 MBytes, respectively. The testbed network consists of five Linux PC workstations equipped with multi-core processors of speed ranging from 1.0 GHz to 3.2 GHz. The computing requirement (CR) of each module and the processing power (PP) of each computer are measured or estimated using the methods proposed in [32]. In view of the computational complexity of each program, the scheduling experiments consider a subset of the programs in the original workflow, i.e. ungrib.exe, metgrid.exe, real.exe and wrf.exe, which serve as the main data processing routines. In the WPS part of each pipeline, we bundle ungrib.exe and metgrid.exe into one module denoted as w_1, w_3, and w_5, respectively; while in the WRF part of each pipeline, we bundle real.exe and wrf.exe into one module denoted as w_2, w_4, and w_6, respectively. We map modules w_1, w_3, and w_5 to dragon.cs.memphis.edu, modules w_2 and w_4 to mouse.cs.memphis.edu, and modules w_6 to cow.cs.memphis.edu. The virtual start and end modules are mapped to the other two machines. The preprocessing program geogrid and postprocessing program ARWpost are executed as part of the piplines, but are not considered for scheduling.

We conduct two sets of scheduling experiments on the above mapped workflow using fair-share (FS) and CPPS, respectively. In FS, each module is assigned by the system the default running priority, while in CPPS, we simply use the Linux command "nice" to adjust the level of priority to achieve coarse-grained control of execution. The module execution time and Minimum End-to-end Delay (MED) along the critical path using FS and CPPS are tabulated in Table 3. We observed that the MED performance improvement in this particular case is about 10.67%. The scheduling results in other cases with different workflow structures and mapping schemes are qualitatively similar. We would particularly like to point out that these small-scale workflow experiments with application-level coarse-grained control are conducted mainly for proof of concept. It is predictable that the performance superiority of CPPS over FS would be manifested much more significantly as the scale of computing workflows and the scope of distributed network environments continue to grow in real-life scientific applications.

Table 3: Performance measurements using FS and CPPS.

FS		CPPS	
Module	Exec Time (sec)	Module	Exec Time (sec)
w1	31.506	w1	55.077
w2	79.893	w2	79.537
w3	31.821	w3	55.051
w4	79.728	w4	78.296
w5	43.397	w5	26.34
w6	125.551	w6	124.579
Critical Path	168.948	Critical Path	150.919

6 Conclusion and Future Work

In this paper, we formulated an NP-complete workflow scheduling problem and proposed a Critical Path-based Priority Scheduling (CPPS) algorithm. Extensive simulation results show that CPPS outperforms the traditional fair-share scheduling policy commonly adopted in real systems. We also conducted proof-of-concept experiments based on real-life scientific workflows deployed and executed in a testbed network.

However, finding a good on-node scheduling policy and finding a good mapping scheme are not totally independent. CPPS is able to improve the workflow performance over a fair-share algorithm for any given mapping scheme. We recognized that a better performance might be achieved if the interaction between the mapping and the scheduling is considered in the optimization, which will be explored in our future work.

We also plan to further refine the cost models by taking into consideration user and system dynamics and decentralize the proposed scheduling algorithm to adapt it to time-varying network environments. It is of our interest to investigate different ways to implement the scheduling algorithm (at either the application or kernel level) and compare their performances and overheads. We would also like to explore the possibilities to integrate this scheduling algorithm into existing workflow management systems and test it in wide-area networks with a high level of resource sharing dynamics.

Acknowledgments

This research is sponsored by U.S. Department of Energy's Office of Science under Grant No. DE-SC0002400 with University of Memphis.

References

1. DAGMan. http://www.cs.wisc.edu/condor/dagman
2. Energy Sciences Network. http://www.es.net
3. Internet2. http://www.internet2.edu
4. Http://www.wrf-model.org/index.php
5. WRF Preprocessing System (WPS). http://www.mmm.ucar.edu/wrf/users/wpsv2/wps.html
6. Annie, S., Yu, H., Jin, S., Lin, K.C.: An incremental genetic algorithm approach to multiprocessor scheduling. IEEE Trans. on Para. and Dist. Sys. 15, 824–834 (2004)
7. Bajaj, R., Agrawal, D.: Improving scheduling of tasks in a heterogeneous environment. IEEE Trans. on Parallel and Distributed Systems 15, 107–118 (2004)
8. Benoit, A., Robert, Y.: Mapping pipeline skeletons onto heterogeneous platforms. In: Shi, Y., van Albada, D., Dongarra, J., Sloot, P. (eds.) Int. Conf. on Computational Science, vol. 4487, pp. 591–598. Springer (2007)
9. Boeres, C., Filho, J., Rebello, V.: A cluster-based strategy for scheduling task on heterogeneous processors. In: Proc. of 16th Symp. on Comp. Arch. and HPC. pp. 214–221 (2004)
10. Braun, T., Siegel, H., Beck, N., Boloni, L., Maheswaran, M., Reuther, A., Robertson, J., Theys, M., Yao, B., Hensgen, D., Freund, R.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. JPDC 61(6), 810–837 (June 2001)
11. Churches, D., Gombas, G., Harrison, A., Maassen, J., Robinson, C., Shields, M., Taylor, I., Wang, I.: Programming scientific and distributed workflow with triana services. Concurrency and Computation: Practice and Experience, Special Issue: Workflow in Grid Systems 18(10), 1021–1037 (2006), http://www.trianacode.org
12. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M., Vahi, K., Livny, M.: Pegasus: mapping scientific workflows onto the grid. In: Proc. of the European Across Grids Conference. pp. 11–20 (2004)
13. The office of science data-management challenge (Mar-May 2004), report of the DOE Office of Science Data-Management Workshop. Technical Report SLAC-R-782, Stanford Linear Accelerator Center
14. Fazar, W.: Program evaluation and review technique. The American Statistician 13(2), 10 (April 1959)
15. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-completeness. W.H. Freeman and Company, San Francisco (1979)
16. Goyal, D.: Scheduling processor bound systems. Tech. rep., Computer Science Department, Washington State University (1976)
17. Gu, Y., Wu, Q., Rao, N.: Analyzing execution dynamics of scientific workflows for latency minimization in resource sharing environments. In: Proc. of the 7th IEEE World Congress on Services. Washington DC (Jul 4-9 2011)

18. Henry, G.: The fair share scheduler. AT&T Bell Laboratories Technical Journal (1984)
19. Kay, J., Lauder, P.: A fair share scheduler. Communications of ACM 31(1), 44–55 (1988)
20. Kelley, J., Walker, M.: Critical-path planning and scheduling. In: Proc. of the Eastern Joint Computer Conference (1959)
21. Kongetira, P., Aingaran, K., Olukotun, K.: Niagara: a 32-way multithreaded sparc processor. IEEE Micro Magazine 25(2), 2129 (2005)
22. Kwok, Y., Ahmad, I.: Dynamic critical-path scheduling: An effective technique for allocating task graph to multiprocessors. IEEE Trans. on Parallel and Distributed Systems 7(5), 506–521 (May 1996)
23. Lud*ä*scher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system. Concurrency and Computation: Practice and Experience 18(10), 1039–1605 (2006)
24. Mcnairy, C., Bhatia, R.: Montecito: A dual-core, dual-thread itanium processor. IEEE Micro Magazine (2005)
25. Mezzacappa, A.: SciDAC 2005: scientific discovery through advanced computing. J. of Physics: Conf. Series 16 (2005)
26. Rahman, M., Ranjan, R., Buyya, R.: Cooperative and decentralized workflow scheduling in global grids. Future Generation Computer Systems 26, 753–768 (2010)
27. Raman, R., Livny, M., Solomon, M.: Resource management through multilateral matchmaking. In: Proc. of the 9th IEEE Int. Symp. on High-Perf. Dist. Comp. (August 2000)
28. Ranaweera, A., Agrawal, D.: A task duplication based algorithm for heterogeneous systems. In: Proc. of IPDPS. pp. 445–450 (2000)
29. Skamarock, W., Klemp, J., Dudhia, J., Gill, D., Barker, D., Duda, M., Huang, X., Wang, W., Powers, J.: A description of the advanced research wrf version 3. Tech. Rep. NCAR/TN475+STR, National Center for Atmospheric Research, Boulder, Colorado, USA (June 2008)
30. Topcuoglu, H., Hariri, S., Wu, M.: Performance effective and low-complexity task scheduling for heterogeneous computing. IEEE TPDS 13(3) (2002)
31. Wassermann, B., Emmerich, W., Butchart, B., Cameron, N., Chen, L., Patel, J.: Workflows for e-Science: Scientific Workflows for Grids, chap. Sedna: a BPEL-based environment for visual scientific workflow modeling, pp. 427 – 448. Springer, London (2007)
32. Wu, Q., Datla, V.: On performance modeling and prediction in support of scientific workflow optimization. In: Proc. of the 7th IEEE World Congress on Services. Washington DC (Jul 4-9 2011)