2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

Stretch Out and Compact: Workflow Scheduling with Resource Abundance

Young Choon Lee and Albert Y. Zomaya
Centre for Distributed and High Performance Computing, School of Information Technologies
The University of Sydney, NSW 2006 Australia
Email: {young.lee, albert.zomaya}@sydney.edu.au

***Abstract*—Resource abundance is apparent in today's multi-core era. Workflow applications common in science and engineering can take great advantage of such ample resource capacity. Yet, existing workflow scheduling algorithms have been mostly designed on the traditional and contrasting premise of scarce resource capacity relative to what applications require. In this paper, we address the problem of workflow scheduling exploiting resource abundance not only for performance, but also for resource efficiency. We first present the critical-path-first scheduling algorithm, which efficiently *stretches out* the schedule to proactively preserve critical path length, the shortest possible time of completion. We then develop an algorithm to *compact* the schedule for resource efficiency. The schedule is compacted by rearranging tasks making use of idle/inefficiency slots present in the schedule due to precedence constraints (synchronization). We have compared the performance of our scheduling algorithm with three previous algorithms and evaluated the efficacy of our schedule compaction algorithm by applying it to those four scheduling algorithms including our own. Experiments were conducted in a simulated environment with various real-world scientific workflows. Results show that our scheduling algorithm achieves a great performance to resource usage ratio. Further, our schedule compaction algorithm reduces resource usage by 33% on average ranging from 11.3% to 90.7% with no makespan increases.**

***Keywords*-workflow scheduling; resource efficiency; multi-core machine; cloud computing;**

I. Introduction

The scale and capacity of computer systems have increased dramatically in the past decade particularly with the prevalence of multi-core processors. Traditionally, scheduling algorithms for parallel/workflow applications assume the number of concurrently runnable tasks in a workflow application overwhelms the number of resources in the target system. Grids might be an exception; however, when heavy data dependencies are involved they are a less appealing option. Today, a single system of small/ medium size organization easily houses hundreds of processor cores; and this is contrasting to the traditional premise.

Workflow applications are common in science and engineering, and their structure/composition is known in advance. A workflow application consists of coarse-grain and precedence-constrained tasks, each of which can be a service, application, or module. Notable examples include LIGO [4], [2], Montage [3] and CyberShake [1], [6]. These applications typically run repeatedly with different sets of data and/or parameters. The execution of a workflow requires the coordination of its tasks across multiple resources. Resource abundance for such an execution provides a great opportunity for performance improvement. However, the exploitation of resource abundance is a double-edged sword in that the performance improvement from such exploitation may be achieved at the expense of resource efficiency.

Although existing workflow scheduling algorithms can work with the current state of resource abundance, their resource usage is excessive often leaving a non-negligible number of poorly utilized resources due primarily to the non-uniformity in workflow structure (Figure 1). The number of resources to be used (resource limit) can be set to avoid/alleviate such inefficiency. However, the identification of most appropriate resource limit is very difficult, if not impossible, and the impact of resource limit on makespan (increase) is non-deterministic. Dynamic resource provisioning as in [10], [9] might be an alternative; however, its application is largely limited to public clouds with resource elasticity and hourly pricing. The poor resource utilization—sourced from uneven widths of different levels in a workflow—within the hour still remains.

In this paper, we address the problem of workflow scheduling exploiting resource abundance and present two scheduling solutions. We first present the critical-path-first (CPF) scheduling algorithm, which efficiently stretches out the schedule aiming to proactively preserve critical path length, CPL. We then develop an algorithm to compact the output schedule to minimize resource usage with no makespan increase. With a particular scheduling algorithm, the schedule for a workflow without a resource limit delivers the minimum makespan possible by the algorithm; however, the schedule exhibits many idle/inefficiency slots since it is maximally stretched out. The compaction of schedule is essentially made possible using those inefficiency slots. Our schedule compaction algorithm is capable of exploiting both explicit and "implicit" inefficiency slots—not apparent in the schedule due to subsequently assigned tasks. Our approach of *stretching out and compact* is unique in the following aspects:

- The proactive preservation of CPL is sought by the "batch" scheduling of tasks along critical path at the beginning with the exploitation of resource abundance.
- Scheduling flexibility in partial schedule is explicitly taken into account for the conservative use of resources.

978-0-7695-4996-5/13 $26.00 © 2013 IEEE
DOI 10.1109/CCGrid.2013.55

- Implicit inefficiency slots are defined and used for schedule compaction for the first time to the best of our knowledge.

We have evaluated our solutions in terms of makespan, resource usage, cost efficiency and scheduling time. Experiments have been conducted in a simulated environment with five real-world scientific workflows. Results show that our scheduling algorithm achieves a great performance to resource usage ratio compared with three previous scheduling algorithms. Further, our schedule compaction algorithm reduces resource usage by 33% on average ranging from 11.3% to 90.7% with no makespan increases.

II. Models

In this section, we give a description of workflow application and system models.

A. Workflow Applications

A workflow application can be represented by a directed acyclic graph (DAG), G = (V, E) comprising a set V of tasks, V = $\{v_0, v_1, ..., v_n\}$ with a set E of edges, each of which connects two tasks representing their precedence constraint or data dependency. The computation cost (execution time) of a task v_i is denoted as w_i. The communication cost from task v_i to task v_j is denoted as $c_{i,j}$. For a given task v_i, the set of its communication costs with its child tasks is denoted as C_i. The completion time of a workflow application is defined as *makespan* or schedule length. Other characteristics of DAG are described in Figure 1(a).

Among the predecessors of a task v_i, the predecessor that completes the communication at the latest time is the most influential parent (MIP) of the task denoted as $MIP(v_i)$. For example, task 5 (v_5) is MIP of task 9 (v_9), and task 7 is MIP of tasks 10, 11 and 12 (Figure 1(a)). MIP tasks determine the start time of their child tasks. The earliest start and finish times of a task v_i is defined as:

$$EST(v_i) = \begin{cases} 0 & \text{if } v_i = v_{entry} \\ EFT(MIP(v_i)) + c_{MIP(v_i),i} & \text{otherwise,} \end{cases} \quad (1)$$

$$EFT(v_i) = EST(v_i) + w_i \quad (2)$$

where v_{entry} is the entry task of a workflow application and $c_{MIP(v_i),i}$ is the communication cost/time between $MIP(v_i)$ and v_i. The latest start and finish times are then defined as:

$$LST(v_i) = LFT(v_i) - w_i, \quad (3)$$

$$LFT(v_i) = \begin{cases} EFT(v_i) & \text{if } v_i = v_{exit} \\ \min\limits_{v_k \in chdrn(v_i)} \{LST(v_k) - c_{i,k}\} & \text{otherwise} \end{cases} \quad (4)$$

where $chdrn(v_i)$ is the set of child tasks of v_i.

The actual start and finish times of a task v_i are denoted as $AST(v_i)$ and $AFT(v_i)$, and they can be different from its earliest start and finish times if the actual finish time of another task scheduled on the same resource is later than $EST(v_i)$.

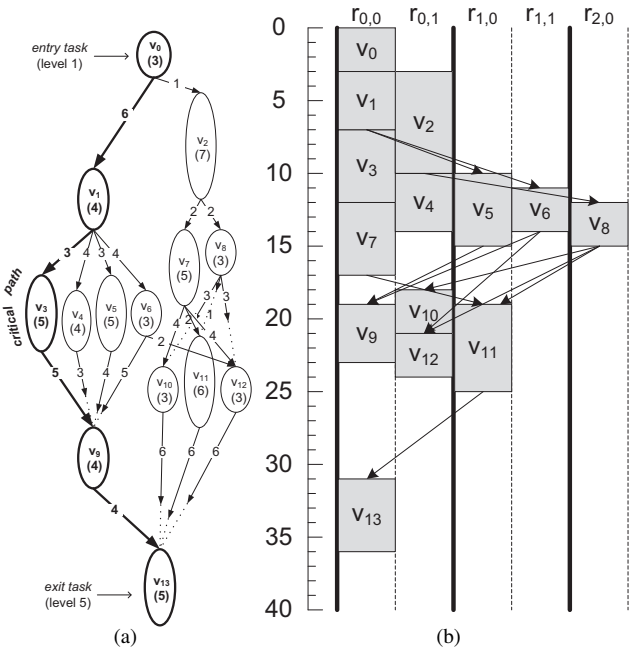

Figure 1. Workflow scheduling example in dual-core compute resources. (a) a sample workflow. (b) output schedule of CPF. The size of tasks/vertices and the length of edges are proportional to their computation and communication costs (shown in parentheses and along edges), respectively. Dotted lines in Figure 1(a) indicate waiting times for the completion of MIP tasks and these waiting times recursively contribute to allowable delays.

The difference between LFT and EFT of v_i is denoted as allowable delay or $AD(v_i)$; this is present as a result of synchronization required for child tasks of v_i.

The longest path of a graph is the critical path (shown with thick line in Figure 1(a)). CP denotes the set of tasks along the critical path. For a given DAG, the critical path determines the theoretical minimum of completion time and it is defined as the summation of computation costs of tasks in CP (or CP tasks).

B. System Model

The target system in this study consists of a set R of compute resources, R = $\{r_0, r_1, ..., r_m\}$. A resource can be a physical compute node or a virtual machine particularly in public clouds. Each resource in R consists of a set of p processing elements or (virtual) processor cores, i.e., r_i = $\{r_{i,0}, r_{i,1}, ..., r_{i,p}\}$. We assume inter-task communication cost is negligible (i.e., 0) within a single resource. Resources are assumed to be homogeneous in terms of their core count, computing power and cost.

III. Stretching Out Workflow Schedule

In this section, we begin by discussing the main idea of our scheduling algorithm and describe the scheduling flexibility of workflow application particularly in partial schedule followed by algorithm description.

A. Rationale behind CPF

The critical path first (CPF) algorithm can be characterized by its lookahead scheduling. The idea is critical path length can be proactively preserved by assigning all CP tasks on a particular resource (or CP resource) 'at the beginning'[1] and then scheduling remaining tasks in the way that the impact on the makespan of CP schedule is minimal. Thus, the scheduling is performed looking ahead to the quality of schedule. The makespan preservation is essentially facilitated by stretching out the schedule of those remaining non-CP tasks exploiting resource abundance with scheduling flexibility in workflow structure (AD and LST/LFT).

B. Scheduling Flexibility of Workflow Applications

A task with multiple parent tasks requires synchronization (e.g., task 13 with its four parent tasks in Figure 1(a)); and this is the source of allowable delay and latest start/finish time, i.e., scheduling flexibility. The parent task with the latest finish time (task 9 for task 13) is likely to have the high priority among all its parent tasks and to be scheduled first. Other parent tasks can then afford some delay to their start time, i.e., allowable delay and latest start time. For example, the scheduling of task 12 on $r_{0,1}$ in Figure 1(b) is not possible based on its EFT (21). However, its LFT (31) enables its scheduling on $r_{0,1}$. Since the LFT of a task is derived from the schedule information of its child tasks (Equations 3 and 4) and this information is not yet available we use interim LFT (ILFT) calculated with the partial schedule at the time of scheduling the target task. For a task v_i, the ILFT is primarily determined by its *partner tasks*, more precisely their latest completion time. A task is referred to as a partner task of another task v_i if it shares any child task of v_i, e.g., tasks 6 and 8 are partner tasks of task 7 in Figure 1(a). Note that partner tasks are different from sibling tasks who share the same parent task. The latest completion time of a task v_i is defined as:

$$LCT(v_i) = \begin{cases} AFT(v_i) + \max\limits_{c_{i,j} \in C_i} \{c_{i,j}\} & \text{if } v_i \text{ is scheduled} \\ EFT(v_i) + \max\limits_{c_{i,j} \in C_i} \{c_{i,j}\} & \text{otherwise} \end{cases} \tag{5}$$

The ILST and ILFT of v_i on $r_{j,k}$ are then defined as:

$$ILST(v_i, r_{j,k}) = ILFT(v_i, r_{j,k}) - w_i, \tag{6}$$

$$\begin{aligned} ILFT(v_i, r_{j,k}) &= AFT(v_i, r_{j,k}) \\ &+ (max\{LCT(v_i, r_{j,k}), \\ &\quad max\{LCT(partner_i)\}\} \\ &- LCT(v_i, r_{j,k})) \end{aligned} \tag{7}$$

where $partner_i$ is the set of partner tasks of v_i. $LCT(v_i, r_{j,k})$ is computed with AFT as v_i is being considered to be scheduled on $r_{j,k}$. ILFT is referred to as actual latest finish time (ALFT) if this is computed with the complete schedule.

[1]Tasks 0, 1, 3, 9, and 13 are first assigned to $r_{0,0}$ in Figure 1(b).

Another scheduling flexibility we use is idle slots within the CP schedule. During the scheduling of remaining non-CP tasks, the schedule of CP tasks needs to be continuously updated (line 28 in Algorithm 1), i.e., it is loosened due to precedence constraints (communication delays). We then make use of idle slots resulted from this loosening with AD. Many of these idle slots are neither present nor used with existing CP-base scheduling algorithms. Because, these usable idle slots often appear before CP tasks whose non-CP predecessor tasks are partly scheduled[2]; and this is not possible in those existing algorithms since a task can be scheduled only after all its predecessor tasks are scheduled. Besides, those algorithms dedicate a resource to CP tasks not allowing other tasks to be scheduled in that resource.

C. The Critical Path First Algorithm

CPF first schedules CP tasks onto a dedicated resource, r^{CP} zeroing out all communication delays. It then tries to preserve the makespan of this partial schedule as much as possible exploiting resource abundance. Remaining tasks (G') are prioritized by their topological order and scheduled based on ILFT and degree of makespan increase. The set R' of candidate resources is considered for each remaining task. R' grows dynamically with resources already used plus a new (empty) resource, r^{new} (line 24) as the scheduling proceeds. Algorithm 1 presents the pseudo-code of CPF.

For each task v in G', its EFTs and ILFTs on resources in R' are computed.[3] Now a "schedulable" slot, $S(v, r)$ for v in each resource r in R' is identified. We classify a schedulable slot as a *spot-on* or *alternative* slot. $S(v, r)$ can be an idle slot for which the finish time is between $EFT(v)$ and $ILFT(v)$, and the size is enough for v, i.e., a *spot-on* slot. If no such a slot is available, $S(v, r)$ is set to the first available slot that can accommodate v with the finish time of the slot later than $ILFT(v)$, i.e., an *alternative* slot. Resource with these slots are continuously updated with best ones among them (lines 14 and 16).

A task without any direct precedence constraint with CP tasks (no CP child tasks or $chdrn(v) \cap CP == \emptyset$) is scheduled in a first-fit fashion based on ILFT except the case in which the best spot-on resource, r^* is a new resource, r^{new} (line 19). The task is assigned to the first resource that completes the execution before the ILFT of that task (line 10). Since tasks without CP child tasks do not have an immediate impact on makespan at the time of their scheduling, the focus with these tasks is more on condensing the schedule hoping to reduce resource usage.

In the meantime, a task with one of more CP child tasks is scheduled in a best-fit fashion based on makespan increase. The task is assigned to the resource on which its finish time

[2]In Figure 1(b) the idle slot between tasks 3 and 9 is incurred (i.e., the schedule is loosened) by the communication delay from parent tasks 5 and 6 of task 9; and it is used by non-CP task 7.

[3]The computation is done per coarse-grain resource, not per core.

Algorithm 1 Critical Path First Scheduling

```
1: find critical path
2: assign CP tasks to r^CP // e.g., r_{0,0}
3: G' ← G−CP tasks
4: sort G' in topological order
5: r^new ← r^{CP+1}, R' ← {r^CP, r^new}, r* ← ∅, r' ← ∅
6: for v ∈ G' do
7:     compute EFT(v) and ILFT(v) with R'
8:     for r ∈ R' do
9:         find S(v, r) // schedulable slot for v
10:        if chdrn(v)∩CP == ∅ ∧ S(v, r) is spot-on then
11:            break
12:        else if S(v, r) is spot-on then
13:            compute MI(v, r) // makespan increase
14:            udate r* based on makespan increase
15:        else
16:            update r' based on AFT
17:        end if
18:    end for
19:    if (r* == r^new) ∧ (AFT(v, r*) ≥ AFT(v, r')) then
20:        if (chdrn(v)∩CP≠∅ ∧ MI(v, r*)≥MI(v, r'))
               ∨ (chdrn(v) ∩ CP==∅) then
21:            r* ← r'
22:        else
23:            r^new ← r^{new+1} // very next one to r^new
24:            R' ← R' + r^new
25:        end if
26:    end if
27:    assign v to r*
28:    update the schedule
29: end for
```

is no later than its ILFT and the makespan of the current partial schedule is best preserved (no or minimum increase). The makespan preservation check is done only if the first condition is met (line 12). For a given task with CP child tasks, makespan increase (MI) is identified by recalculating finish times of CP child tasks and CP tasks after these CP child tasks. Once the best spot-on resource, r^* is found and if it is a new resource, r^{new}, then it is compared with the best alternative resource r' (lines 19-20). The comparison is done based only on AFT if a task v is without CP child tasks, otherwise both AFTs and makespan increases are compared. This conservative resource allocation helps avoid inefficient addition of resource. Note that when the best alternative resource is selected the finish time on that resource is later than its ILFT and yet, earlier than the AFT on the best spot-on resource, i.e., r^{new}.

IV. Compacting Workflow Schedule

In this section, we describe how scheduling workflows with resource abundance also gives great opportunities for resource efficiency and show how to make use of them.

A. Finding "Inefficiency" Slots

There is inherent resource inefficiency in workflow schedule due primarily to precedence constraints (e.g., idle slots in Figure 2(a)); however, the primary source of this inefficiency can also be a great opportunity for resource efficiency. An idle slot in an output workflow schedule is defined as an "inefficiency" slot if there is a task that can be moved into it without a makespan increase. For a given workflow, inefficiency slots may only be identifiable after finishing the scheduling, and some of them are not apparent; hence, implicit inefficiency slots or simply implicit slots. The workings of this identification are described in Algorithm 2.

For a task v, all resources starting from the first resource are checked, except the resource on which v is scheduled, r_{v_i}. We first check if there is a schedulable slot (i.e., explicit inefficiency slot) at the beginning of resource ($EST(r)$, line 5). If no such a slot is found, we seek for an explicit slot (lines 12-17) and then an implicit slot (lines 18-35). The identification of implicit slots is done based on AD (i.e., $LST - AST$, line 18) as shown with dotted arrows in Figure 2(a). The *if* statement in line 18 indicates task v'' can be pushed down to accommodate v_i; however, the actual feasibility of moving v_i before v'' may not be guaranteed due to subsequently scheduled tasks. Once there is such an indication we recursively check those subsequent tasks if they too can be pushed down that eventually make a slot sufficient for v_i (the inner *while* loop). There might be more than one tasks to be pushed down. This process of checking the possibility of pushing down subsequent task ends if (1) there is no more subsequent task (line 22), (2) the inefficiency slot is realized ($t^{push} \leq 0$) as a result of accumulation of actual allowable delays ($ALST - AST$, line 25), or (3) a subsequent task cannot be pushed down enough for the remaining amount of time (t^{push}, line 27).

B. Schedule Compaction Algorithm

Essential pieces of information for our schedule compaction algorithm are LST/LFT and ALST/ALFT. Their initial values are computed with the original schedule that is input for the compaction (line 2). As the schedule constantly changes these values are also updated accordingly (line 14); more precisely, ASTs/AFTs, LSTs/LFTs and ALSTs/ALFTs of those tasks affected by a compaction operation are recalculated. The compaction starts from the back of schedule—the last resource used in the schedule—due to the typical pattern of resource efficiency deterioration.

For each task v_i on a resource r used in the schedule (i.e., $r \in R'$), we seek for an inefficiency slot starting from the beginning of schedule. If the slot is found the task is moved into it, and the timing information of v_i and affected tasks by this compaction is updated. Affected tasks are child and parent tasks of v_i and subsequently scheduled tasks after v^*. Note that the update on v^*'s subsequent tasks emulates the push-down of these tasks if v_i is moved into an implicit

Algorithm 2 Find Inefficiency Slot

```
1: function FINDINEFSLOT(v_i, R')
2:   v* ← ∅, r* ← ∅, R'' ← R' − r_{v_i}
3:   while r* == ∅ do
4:     r ← first resource in R''
5:     if EST(r) ≥ EFT(v_i, r) then // explicit slot
6:       r* ← r
7:       break
8:     end if
9:     for v' ∈ r do
10:      v'' ← v' + 1
11:      compute AST(v_i, r) after AFT(v', r)
12:      if AST(v_i, r) ≥ ALST(v_i, r) then
13:        break
14:      else if v'' == ∅ ∨ (AST(v'', r) − AST(v_i, r)) ≥ w_i
     then // explicit slot found
15:        r* ← r
16:        v* ← v'
17:        break
18:      else if LST(v'', r) − AST(v_i, r) ≥ w_i then
19:        t^push ← AFT(v_i, r) − ALST(v'', r)
20:        while t^push > 0 do
21:          v'' ← v'' + 1 // next task on r
22:          if v'' == ∅ then
23:            t^push = 0
24:          else if LST(v'', r)–AST(v'', r)≥t^push then
25:            t^push←t^push–(ALST(v'', r)–AST(v'', r))
26:          else
27:            break
28:          end if
29:        end while
30:        if t^push ≤ 0 then // implicit slot found
31:          r* ← r
32:          v* ← v'
33:          break
34:        end if
35:      end if
36:    end for
37:    R'' ← R'' − r
38:  end while
39:  return (v*, r*)
40: end function
```

Algorithm 3 Schedule Compaction Algorithm

```
1: Input: a schedule, SCHD for G with R'
2: compute LSTs/LFTs and ALSTs/ALFTs with SCHD
3: for r ∈ R' do // starting from the back
4:     V(r) ← tasks scheduled on r
5:     for v_i ∈ V(r) do
6:         (v*, r*) ← FindInefSlot(v_i, R')
7:         if r* ≠ ∅ ∧ v* ≠ ∅ then
8:             move v_i after v* on r*
9:         else if r* ≠ ∅ then
10:            move v_i at the beginning of r*
11:        else
12:            break
13:        end if
14:        update timing info of affected tasks
15:    end for
16: end for
17: merge partly used resources
```

inefficiency slot. For example, in Figures 2(c) and 2(d) tasks 6 and 5 are moved into implicit slots and task 9 is pushed down. If no inefficiency slots are available, the compaction with r stops since the aim of our compaction is essentially reducing the number of resources used in the schedule. That is, resource r is not completely compacted/emptied. This implies that there might be cases that some resources in the middle of schedule are partly empty, i.e., for a multi-core resource, one or more cores, but not all are empty. Partly used resources are considered for merger in a best-fit fashion once the main compaction process finishes (line 17).

In our example (Figure 2), the resultant schedule (Figure 2(d)) only uses one dual-core resource. What's more, the makespan is decreased down to 34 from 36 due to the compaction of task 11 that removes the communication cost between tasks 11 and 13. We also show that the schedule with a resource limit of 1 based on our resultant schedule (Figure 2(e)). This schedule is neither better than ours nor realistic. Such predetermination of resource limit is not quite possible in reality.

V. Experimental Results

In this section we describe the experimental setup and present experimental results. The performance of CPF is compared with three previous algorithms. Our compaction algorithm is evaluated with compacted versions of those schedules generated for the comparative evaluation of CPF.

A. Experimental Setup

Experiments were conducted in a simulated environment on an Intel 40-core machine with 4 10-core Intel 2.4GHz Xeon processors. We have run two sets of experiments each of which consists of 22,720 experiments with five real-world scientific workflows: CyberShake, Epigenomics, LIGO, Montage and SIPHT. These input workflows were obtained from the Pegasus Workflow repository.[4] There are a total of 1,420 workflows used in our experiments, i.e., 220, 440, 220, 220 and 320 for those five workflows, respectively. The size of workflow ranges from 50 to 6,000 tasks. As input workflows are described in XML format and their sizes are as large as 50MB with over 400,000 lines, their

[4]https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator.

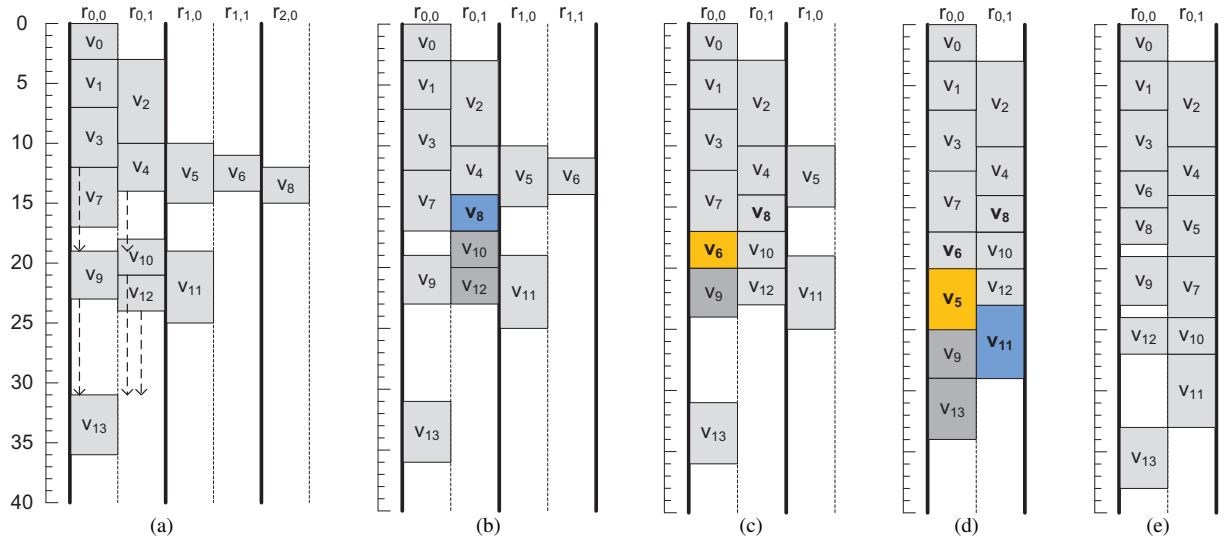

Figure 2. Workflow scheduling example in dual-core compute resources. (a) the initial output schedule of CPF for the workflow in Figure 1(a). (b) an intermediate schedule after the compaction with task 8. (c) an intermediate schedule after the compaction with task 6. (d) the final schedule after the compaction with tasks 5 and 11. (e) output schedule of CPF with a resource limit of 1 (with 2 cores). Tasks in blue and orange indicate the actual compaction in explicit and implicit slots, respectively. Those in dark gray are ones affected by the compaction.

preprocessing (reading and parsing) takes non-negligible amount of time. Thus, we converted these XML documents into plain text equivalents with some preprocessing. We used four different resource sizes (#cores), 1, 2, 4 and 8. In all our experiments no resource limit is imposed. The inter-node bandwidth is assumed to be 1 Gbps.

Three previous scheduling algorithms used in our comparisons are Dynamic Critical Path (DCP) [8], Critical-Path-on-a-Processor (CPOP) [12] and a greedy algorithm based on earliest finish time (EFT). They were chosen due to their CP base scheduling nature as in CPF, except EFT.

B. Evaluation Metrics

Experimental results are plotted and discussed in terms of makespan, the number of resources, cost efficiency and scheduling time. The quality of makespan is evaluated based on CPL. The granularity of resource is at the physical compute node or processor level, not the core level. The cost efficiency is interpreted as the performance to cost ratio. However, the cost efficiency of an algorithm is not directly comparable with that of another due to the incompatibility between two performance metrics, i.e., makespan and resource usage. Although the preference to a schedule in practice is likely to be determined by the user utility function, we do not adopt any particular utility function in this study. Rather, we use the marginal cost efficiency based on the cost efficiency of CPF. For a given workflow, the marginal cost efficiency of the schedule by a particular algorithm is defined as the summation of differences (in percentage) of the makespan and resource usage from those of the schedule by CPF. Scheduling time is the algorithm running time.

Table I
SCHEDULING PERFORMANCE

application (CPL)	algo	makespan (sec)	#res.	marginal cost eff.
CyberShake (241.55)	EFT	567.26	21.0	-10%
	CPF	566.22	20.9	-
	CPOP	683.86	18.7	-25%
	DCP	567.26	20.8	2%
Epigenomics (20873.0)	EFT	20878.10	40.5	-30%
	CPF	20876.20	31.2	-
	CPOP	34634.10	40.5	-43%
	DCP	20878.20	30.6	2%
LIGO (1396.16)	EFT	1397.94	16.2	-5%
	CPF	1397.75	15.4	-
	CPOP	1420.44	16.2	-6%
	DCP	1398.27	15.6	-1%
Montage (206.95)	EFT	211.59	42	0%
	CPF	211.68	42	-
	CPOP	230.69	41.3	-7%
	DCP	211.59	42	0%
SIPHT (5166.30)	EFT	5169.11	135.5	-16%
	CPF	5169.12	117.2	-
	CPOP	7256.05	135.5	-56%
	DCP	5169.12	107.3	8%

C. Results

We present results obtained with the resource size of 8 in this paper due to the space limit and the similarity to results with other resource sizes. All results are averaged.

Table I shows results without schedule compaction. As schedules are maximally stretched out without resource limit makespans are very close to CPL with an exception of

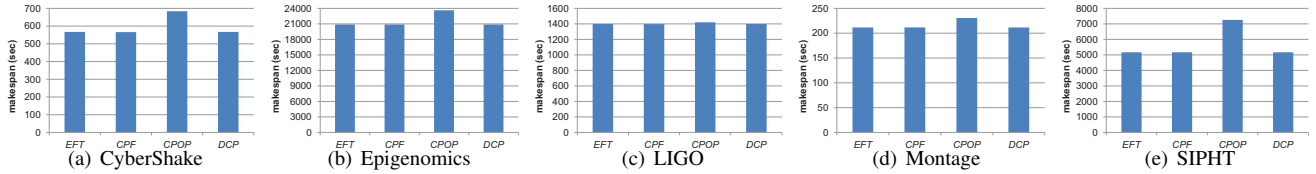

Figure 3. Average makespans.

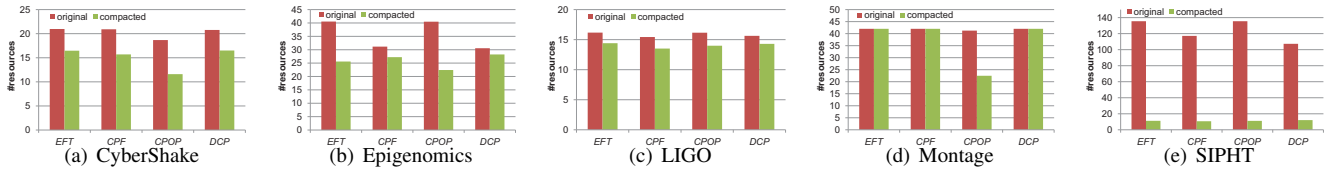

Figure 4. Average resource usage (#resources).

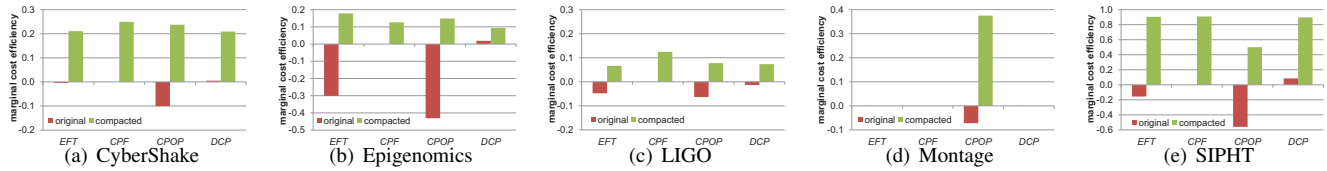

Figure 5. Average marginal cost efficiency rates.

CyberShake due to its large input sizes for entry tasks, e.g., 40GB. Note also that makespans of CPOP are often notably longer than the others due primarily to the inefficient use of CP resource—non-CP tasks are not allowed to scheduled on the CP resource even if it has schedulable slots. The average cost efficiency of EFT and CPOP are 10% and 25% poorer than CPF. DCP performs similar to CPF with its cost efficiency being 2% better than CPF; however, its scheduling time is far larger than that of CPF. DCP runs for around 40 seconds in our test environment for a large SIPHT workflow with six thousands of tasks. More specifically, scheduling time of DCP exceeds millisecond level with an SIPHT workflow of approx. 1,500 tasks or more. Scheduling time of DCP on average is an order of magnitude longer than CPF and the other two regardless of application (Figure 6).

In Figures 4 and 5, and Table II, we show the performance of our schedule compaction algorithm. Makespans in Figure 3 are original makespans before the compaction since makespan reduction—as in the example in Figure 2—was observed to be less than 1% and thus, makespans of compacted schedules are not presented here.

While resource usage reduction is mostly between 10% and 30%, that for SIPHT is much larger, 90.7% on average (Figure 4(e) and Table II). The structure of SIPHT workflow is dominated by the width of the first level consisting of a large number of small and similar size entry tasks. These tasks are scheduled on different resources consuming an excessive number of resources. However, the final schedule exhibits many inefficiency slots with large AD values with these tasks; and this enables such a significant reduction.

The overall improvement of CPOP is the largest due primarily to more inefficiency slots resulted from longer makespans compared with the other three algorithms (Figure 5). In other words, the resource inefficiency incurred with these slots provides a great opportunity for schedule compaction, and our algorithm is capable of effectively exploiting such an opportunity. Note that the largest performance improvement of CPOP should not be interpreted as the superior performance since the source of this improvement is the poor performance in its original schedules.

Table II summarizes the performance improvement of four algorithms using our schedule compaction algorithm. The efficacy of schedule compaction in resource usage reduction is apparent throughout applications without much impact of scheduling algorithm.

The average scheduling time overhead incurred by schedule compaction is 23% and specific overheads are shown with darker color bars in Figure 6.

VI. Related Work

The majority of previous studies on workflow scheduling (e.g., [11], [5], [10]) either minimize makespan within the resource capacity available or minimize cost of running workflows particularly in clouds. While the former objective is achieved using all available resources, the latter is mostly sought with minimal resource usage. These objectives are conflicting and yet, closely related to resource usage.

Y. -K. Kwok and I. Ahmad [8] propose the dynamic critical path (DCP) algorithm, which assigns a task considering the critical path of partial schedule. DCP implicitly prevents the excessive (most likely unnecessary) use of processors in that for non-CP tasks it only considers processors already used in the schedule. Despite the inherent applicability to the scheduling scenario in our study, DCP's rather excessive

Table II
EFFICACY OF OUR SCHEDULE COMPACTION ALGORITHM

application	res. usage reduction	marginal cost eff.
CyberShake	26.2%	22.7%
Epigenomics	25.4%	13.7%
LIGO	11.3%	8.5%
Montage	11.4%	9.4%
SIPHT	90.7%	80.3%
average	**33.0%**	**26.9%**

scheduling time—due to its repetitive recalculations of timing information as part of its dynamic scheduling process—makes it less attractive.

The partial critical path (PCP) algorithm in [5] tries to minimize the cost meeting the user defined deadline of a given workflow. The cost is positively related to the computing power of service (resource in our definition). For a given workflow, the algorithm first assigns subdeadlines to CP tasks and to remaining tasks based on deadlines of CP tasks. Then, it schedules tasks in the cheapest service that can satisfy the subdeadline constraint. The assignment of subdeadlines has similarity to the calculation of ILST/ILFT in CPF. Two main differences from CPF are the deadline constraint and the fine-grain accounting of service usage without considering the number of services being used.

M. Mao and M. Humphrey [10] and C. Lin and S. Lu [9] take advantage of resource elasticity in public clouds like Amazon EC2. For a given resource, once a multiple of hours (or a predefined threshold) is being reached and no jobs/tasks are assigned to the resource, it is released (terminated). The resource acquisition and release are dynamically done to optimize their specific objective. Our work differs from [10] and [9] in the resource efficiency model. Resource efficiency in our study is optimized based on the actual resource usage rather than the number of resource hours.

The Push-Pull strategy in [7] is a supplementary technique (like our schedule compaction algorithm) for DAG scheduling algorithms. Its main goal is to further reduce the makespan of a schedule generated by a fast deterministic task scheduling algorithm. The Push-Pull strategy tends to use additional resources to achieve its goal that is contradicting with our objective of resource usage minimization.

VII. CONCLUSION

With the prevalence of multi-core processors, it is often observed that the resource capacity of a given system overwhelms resource requirements of a workflow application. In this paper, we have thoroughly studied this situation and presented our solutions. The CPF algorithm first showed stretching out the schedule with the explicit consideration of CPL known at the beginning of scheduling greatly helps make judicious scheduling decisions. Our schedule compaction algorithm demonstrated resource efficiency can be significantly improved by making use of inefficiency

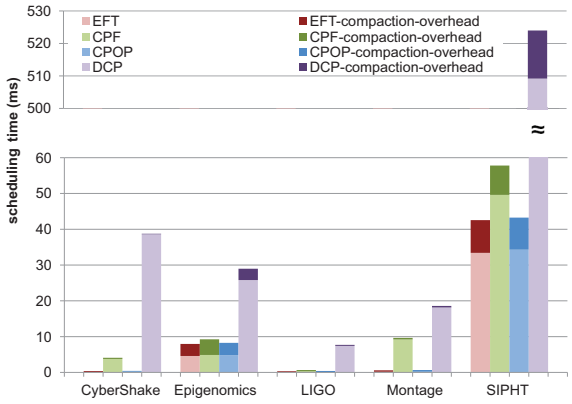

Figure 6. Scheduling time.

slots present both explicitly and implicitly. The effective exploitation of resource abundance enables the optimization of two rather conflicting objectives of makespan minimization and resource usage minimization. Our evaluation results confidently confirm this claim.

ACKNOWLEDGMENT

Professor Albert Zomaya would like to acknowledge the Australian Research Council Discovery Grant DP1097111.

REFERENCES

[1] Cybershake. http://scec.usc.edu/scecpedia/CyberShake, 2012.
[2] Ligo: Laser interferometer gravitational-wave observatory. http://www.ligo.caltech.edu/, 2012.
[3] Montage: An astronomical image mosaic engine. http://montage.ipac.caltech.edu/, 2012.
[4] ABRAMOVICI, A., ALTHOUSE, W. E., AND ET. AL. Ligo: The laser interferometer gravitational-wave observatory. *Science 256*, 5055 (1992), 325–333.
[5] ABRISHAMI, S., NAGHIBZADEH, M., AND EPEMA, D. H. J. Cost-driven scheduling of grid workflows using partial critical paths. *IEEE Trans. Parallel Distrib. Syst. 23*, 8 (Aug. 2012), 1400–1414.
[6] GRAVES, R., JORDAN, T. H., CALLAGHAN, S., DEELMAN, E., FIELD, E., JUVE, G., KESSELMAN, C., MAECHLING, P., MEHTA, G., MILNER, K., OKAYA, D., SMALL, P., AND VAHI, K. Cybershake: A physics-based seismic hazard model for southern california. *Pure and Applied Geophysics 168*, 3-4 (2010), 367–381.
[7] KIM, S. C., LEE, S., AND HAHM, J. Push-pull: Deterministic search-based dag scheduling for heterogeneous cluster systems. *IEEE Trans. Parallel Distrib. Syst. 18*, 11 (Nov. 2007), 1489–1502.
[8] KWOK, Y.-K., AND AHMAD, I. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel Distrib. Syst. 7*, 5 (May 1996), 506–521.
[9] LIN, C., AND LU, S. Scpor: An elastic workflow scheduling algorithm for services computing. In *Proceedings of the 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)* (2011), pp. 1–8.
[10] MAO, M., AND HUMPHREY, M. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011), pp. 49:1–49:12.
[11] TANAKA, M., AND TATEBE, O. Workflow scheduling to minimize data movement using multi-constraint graph partitioning. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2012), pp. 65–72.
[12] TOPCUOUGLU, H., HARIRI, S., AND WU, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst. 13*, 3 (Mar. 2002), 260–274.