Critical-Path and Priority based Algorithms for Scheduling Workflows with Parameter Sweep Tasks on Global Grids

Tianchi Ma and Rajkumar Buyya
Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
Email:{tcma,raj}@cs.mu.oz.au

***Abstract*— Parameter-sweep has been widely adopted in large numbers of scientific applications. Parameter-sweep features need to be incorporated into Grid workflows so as to increase the scale and scope of such applications. New scheduling mechanisms and algorithms are required to provide optimized policy for resource allocation and task arrangement in such a case. This paper addresses scheduling sequential parameter-sweep tasks in a fine-grained manner. The optimization is produced by pipelining the subtasks and dispatching each of them onto well-selected resources. Two types of scheduling algorithms are discussed and customized to adapt the characteristics of parameter-sweep, as well as their effectiveness has been compared under multifarious scenarios.**

I. Introduction

The emergence of Grid computing has largely expanded the scale of scientific applications [1][2]. Due to the involvement of multiple organizations and institutions, scientific experiments can be conducted in collaboration in a distributed environment and share resources in a coordinated manner. Among the programming models designed for Grid computing, combining dependent tasks into workflow systems [3] has received attention in recent past.

Related research efforts on workflow management have focused on how to define and manipulate the workflow model [4], compose distributed software components and data/documents together [5][6], as well as how to reduce the global execution time or to fully utilize available resources to achieve stated objectives [7][8]. However, within Grid scale scientific applications, the presence of repeated tasks have motivated introduction for mechanism to group and manage such tasks. Let us consider the following scenario:

Suppose a data-analysis application consists of three steps. Firstly, the experiments will gather data from a remote sensor array. Each sensor in the array will be required to collect data under the configuration of specific parameters. Then, the result data will be staged to several data process centers. These centers will filter the raw data to structured data, according to the locally deployed knowledge databases on associated computational resources. The remote filtering operation needs to be divided into smaller tasks and carried out in parallel – staging it as one workload can cause overloading of both the selected computational resource and the database. At last, the filtered data will be gathered to a scientific computing visualization cluster for post-processing (e.g. visualization).

Now the scientist is going to establish a workflow in order to automate the experiment. From the designer's point of view, the simplest definition is to assign each step as a single task, hence a (pipeline) task-graph representing the workflow would be:

$$Collecting(A) -> Filtering(B) -> Visualization(C)$$

The task A and B are collections of one or more subtasks representing repeated execution of a job. In this case, operations within data collection and filtering tasks are performed over different sets of data. Such individual tasks are called *parameter-sweep* tasks [9].

A characteristic of parameter-sweep task is that there is no inter-communication between its subtasks. However, it introduces a new challenge as it requires optimization of the execution of such parameter-sweep tasks within workflow. In a workflow application scenario shown in Figure 1, it is clear that a subtask in task B need not wait for all the results from task A to be generated before it's starting. A subtask in B can be launched once a certain portion of the results from A is available. In this manner, the makespan of the application execution could be reduced substantially.

In this paper, we focus on the mechanisms and algorithms that implement the *fine-grained* optimization mentioned above. We discuss both the static and dynamic scheduling methodologies, while extending two well-known algorithms and customizing them to achieve the best effectiveness under task graphs with parameter-sweep tasks. The extended algorithms, called xDCP and pM-S, are derived from the existing Dynamic Critical Path (DCP)[10] and Master-Slave (M-S)[11] algorithms respectively. The xDCP extends the original DCP in order to support the scheduling among resources with different capabilities. The pM-S tries to give higher priority to the subtask-graph which is estimated to complete earlier. Finally, we analyze the proposed algorithms by comparing their behaviors under different experimental scenarios.

The rest of the paper is organized as follows. A formal definition of the optimization problem is presented in section II. Section III will discuss about the possible optimization toward parameter-sweep tasks in workflow. Experiments and data are given in section IV. Section V discusses the related work. Section VI presents the conclusion with the direction for future work.

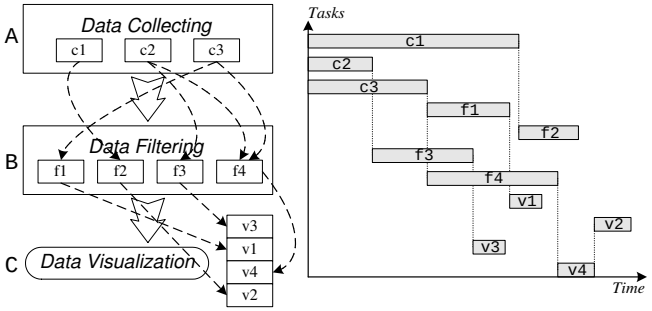

Fig. 1. Optimizing the example parameter-sweep workflow.

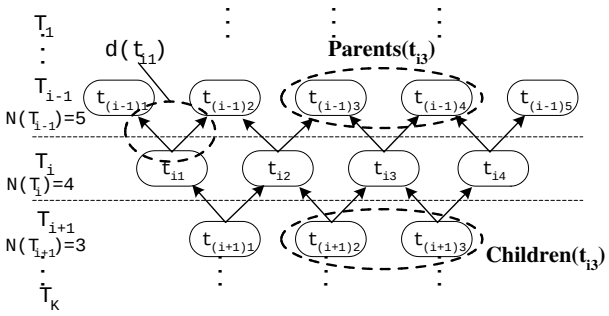

Fig. 2. Description of parameter-sweep task graphs.

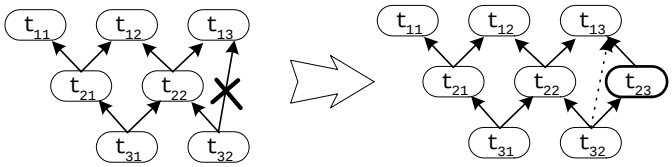

Fig. 3. Using virtual subtask to be the intermediate of task dependency.

II. Optimization Problem

In previous work [12], a workflow language xWFL together with a Workflow Enactment Engine (WFEE) was proposed. The system differs from other similar works due to the event-driven model it follows. The work proposed in this paper is also derived and extended on top of the WFEE engine. However, parameter-sweep tasks in WFEE are simply regarded as normal workflow tasks, while no optimization has been carried out to deal with the internal optimization inside the parameter-sweep tasks. This paper adopts a more fine-grained scheme and extends it to support optimal scheduling of workflow application with parameter-sweep tasks.

A. Terminology

Several concepts should be clarified as they will be used in the following discussion:

1) **Workflow application**: Refers to an application containing several tasks with dependencies represented by a task graph.
2) **Task**: Refers to a task within a workflow which may be parameter-sweep task containing several subtask with no dependencies.
3) **Subtask**: Refers to the sub-task in a parameter-sweep task, holding a specific portion of parameters. Each subtask has its own *length*, representing the time required for executing it on a unit-capability resource.
4) **Resource**: Refers to the abstract resource that provides execution environment for subtasks. Each resource has its own *throughput*. Throughput defines the number of capability units it can provide at the same time.

B. Problem statement

This sub-section gives the formal description of the optimization problem.

As it is shown in Figure 2, let $\Gamma = \bigcup_{i=1}^{K} T_i$ be the task space, including K sequential parameter-sweep tasks and $N(T_i)$ refers to the number of subtasks in T_i. Hence, for each T_i, there are $T_i = \{t_{ij} | j = 1..N(T_i)\}$. The length of subtask t_{ij} is referred by $l(t_{ij})$, while $L(T_i) = \{l(t_{ij}) | j = 1..N(T_i)\}$. For each i where $1 < i \leq K$, we define the task dependency of T_i: $D(T_i) = \{d(t_{ij}) | j = 1..N(T_i)\}$, where $d(t_{ij}) = \{k | t_{(i-1)k} \prec t_{ij}\}$. The symbol $\prec$ is a newly defined binary relation that $\prec (t_a, t_b)$ (also marked as $t_a \prec t_b$) means task t_b must be put into execution strictly after the accomplishment of task t_a. Based on the task dependency, we can define a subtask's parents and children. The function $Parents(t) : \Gamma \rightarrow \bigcup_{i=1}^{K} T_i^*$ (where T_i^* is T_i's closure) refers to the set of all the subtasks that subtask t directly depends on, formally:

$$Parents(t_{ij}) = \begin{cases} \phi, & if\ i = 1 \\ \{t' | t' \in T_{i-1}, t' \prec t_{ij}\}, & if\ i \neq 1. \end{cases}$$

Also we have $Children(t) : \Gamma \rightarrow \bigcup_{i=1}^{K} T_i^*$ that

$$Children(t_{ij}) = \begin{cases} \phi, & if\ i = K \\ \{t' | t' \in T_{i+1}, t_{ij} \prec t'\}, & if\ i \neq K. \end{cases}$$

From the definition we only allow the dependency between the two adjacent tasks. For the dependencies between the non-adjacent tasks, we can also define them by creating virtual subtasks in the intermediate tasks. For example in Figure 3, we use the subtask t_{23} to be the intermediate between subtask t_{32} and t_{13}.

The resources are defined corresponding to the tasks. Assume $\Omega = \bigcup_{i=1}^{K} R_i$ to be the resource space, in which $R_i = \{r_{ij} | j = 1..N(R_i)\}$ refers to the array of resources specially for running all the subtasks in T_i, where $N(R_i)$ refers to the number of individual resources in the resource array. Note the resource here refers to *virtual* resource instead of physical resources like clusters or supercomputers with one or more nodes. For physical resources, we could have $\mho = \{\Re_j | j = 1..M\}$ representing the resource space available. Each virtual resource presents a share of a physical resource that is available for executing a particular parameter-sweep task. A physical resource could be shared by multiple virtual resources, providing different services to different tasks in the

workflow. In the following sections, the term "resources" is used to represent virtual resources.

We define the throughput of a single resource r_{ij} to be marked as $p(r_{ij})$, so that for a subtask t_{ix} and one of its corresponding resources r_{iy}, the time for r_{iy} to finish t_{ix} will be

$$\omega(t_{ix}, r_{iy}) = \frac{l(t_{ix})}{p(r_{iy})}$$

Let $P(R_i) = \{p(r_{ij})|j = 1..N(R_i)\}$.

The optimization lies on the problem of resource queueing. In most cases, the length of resource array will be much less than the number of subtasks in a parameter-sweep task. That means, some subtasks are required to be executed on the same resource. We assume all the resources follow the rule of exclusive. That is, resources are allocated using space-shared scheduling policy (e.g., using queueing system such as PBS for managing resources):

Axiom 1 (Rule of Exclusive): If subtask t_{im} starts on time ξ and consuming resource r_{in}, then no other subtask could be started on r_{in} during the time segment

$$[\xi, \xi + \frac{l(t_{im})}{p(r_{in})})$$

Explanation of the axiom: If there are q subtasks of the same task (marked as $t_1..t_q$) being queued sequentially on the same resource r, there will be temporary dependencies between them as: $t_1 \preceq t_2 \preceq ... \preceq t_q$, where the binary relation $\preceq (a, b)$ is similar to $\prec (a, b)$.

That means, all resources are non-preemptive and exclusive in execution. The users can change the sequence of the queued subtasks, but they cannot put multiple subtasks into parallel execution on the same resource since they are allocated using space-shared policy.

Next, we define the execution time of the tasks. Suppose we have a collection of subtasks C and a set of dependencies to make C a task graph. $Root(C)$ refers to the collection of subtasks who have no subsequent:

$$Root(C) = \{c|c \in C \wedge (\forall c' \in C, \nexists c \prec c')\}$$

$Tree(c, C)$ represents to all the direct and indirect precedents to node c in the task graph. We give it a recursive definition:

$$Tree(c, C) = \bigcup_{c' \in C, c' \prec c} \{Tree(c', C)\} \cup \{c'|c' \in C, c' \prec c\}$$

The execution time is calculated as the time cost for the longest execution path in the given task graph. It is also defined in a recursive way:

$$Time(C, R) = \begin{cases} 0, & if\ C = \phi \\ \max_{c \in Root(C), r = f(c)}\{\omega(c, r) & if\ C \neq \phi \\ +Time(Tree(c, C), R)\}, & \end{cases}$$

where $f(c) : C \rightarrow R$ is the assignment of the subtasks to resources.

Having defined all the related concepts, the problem statement is defined as follows:

Problem: Given $\{(T_i, R_i, D(T_i), L(T_i), P(R_i))|i = 1..K\}$. Select the mapping $f(t) : \Gamma \rightarrow \Omega$ to minimize $Time(\Gamma, \Omega)$ under the rule-of-exclusive.

III. Mechanisms and Algorithms

There exist two types of scheduling mechanisms, namely, static scheduling and dynamic scheduling. All decisions in the static scheduling are made before starting the execution of an application, which makes the static scheduling to be well-adopted to those systems with highly-predictable environment. In the contrary, dynamic scheduling making decisions during the runtime, according to either the redefined policy or the current environment parameters reflected from the system. Both of them could be equally applied to workflows with parameter-sweep tasks. In some dedicated environments, people may prefer static scheduling to search for the best allocation of resources, while the runtime scheduling is more useful in the shared systems with unpredictable resource usage and network weather. In this section, we discuss the scheduling and optimization under both the two scenarios.

A. Static scheduling

The simplest solution of a static schedule is to allocate all the subtasks in a shuffle way, namely assign task t_{ij} to $r_{i(j\%N(R_i))}$. However, due to the irregularity of both the subtasks and the resources, shuffle scheduling might lead to the inefficient result as less powerful resources may be assigned heavy subtasks leading to imbalance in completion time of various tasks.

Numerous works [13][14][15][16][17] have addressed the topic of static scheduling of task graphs. However, task graphs/workflows with parameter-sweep tasks have some special features compared to normal task graphs:

1) Subtasks and resources are grouped in layers (parameter-sweep tasks). A subtask t_{ij} is allowed to be scheduled to R_i.
2) There is no dependency between subtasks in the same layer.
3) Every subtask depends (if there exist dependency) on only the subtasks in its parent layer, as it is specified in section II.

Based on these criteria, we modified the DCP algorithm, so that it could be adopted well for scheduling workflows with parameter-sweep tasks. For convenience, we denote the proposed algorithm by *xDCP* (extended *DCP*).

The DCP algorithm based on the principle of continuously shortening the longest path (also called *critical path (CP)*) in the task graph, by scheduling tasks in the current CP to an earlier start time. The algorithm was designed for scheduling all tasks to the same set of homogeneous resources. However, the workflow scenario we defined in section II is about scheduling different sets of tasks onto different sets of irregular/heterogeneous resources.

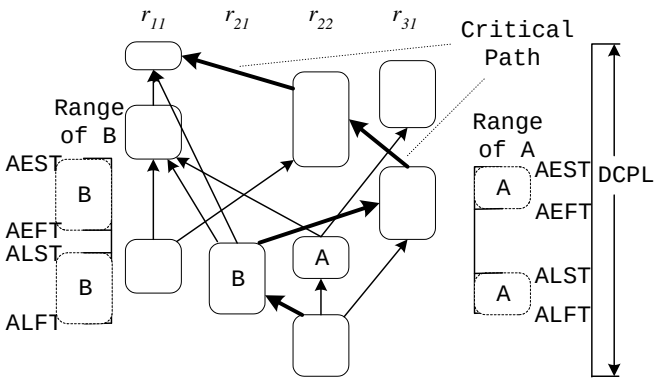

Fig. 4. Description of AEST/AEFT, ALST/ALFT and DCPL.

To tackle with the above conflict, basically we import the following extensions into the original DCP algorithm:

1) The initialization of DCP algorithm is to queue all tasks sequentially in one resource, while leaving other resources empty. In xDCP, we first initialize the tasks in a shuffle way. Experiments show that this adjustment can improve the effectiveness of DCP by 30% in workflows with parameter-sweep tasks.
2) The DCP algorithm uses the term *absolute earliest/latest start time (AEST/ALST)*, which means the possible earliest/latest start time of a subtask on its current resource, as shown in Figure 4. In particular, if a certain task can have a smaller AEST on resource A than resource B, then assignment to A will be regarded as a better schedule for this task. However, in a scenario with heterogenous resources, the executing time of the same task on each resource is different. Therefore we have to use another term *absolute earliest/latest finish time (AEFT/ALFT)* (means the possible earliest/latest finish time of a subtasks on its current resource) to evaluate the schedule.
3) The DCP algorithm ends if all the tasks have been scheduled once. However, we found that under workflow with parameter-sweep tasks it will deliver an extra 10%-20% effectiveness if we further run DCP again on the scheduled result. However, the time used in scheduling should also be considered since DCP has a time complexity of $O(n^3)$. It seems worthless if we keep looping the DCP while the effect increasing is less than a certain speedup-threshold (in our work, we set the speedup-threshold to 5%) .
4) DCP considers the communication overhead incurred in implementing the task dependency. However, there is no inter-process communication between the subtasks of a parameter-sweep task. Therefore we simply remove the terms in DCP that related to the communication cost.
5) After a subtask has been scheduled by DCP, the algorithm checks whether the subtask was scheduled to the same resource queue with any other subtask that the scheduled subtask directly or indirectly depends on, and the scheduled subtask is planned to be executed prior to its ancestor task. Also the subtask must not be executed after any of its offspring subtasks on the same queue. This check is for preventing deadlock generated by the child tasks trying to be executed before parent tasks. However in xDCP, this deadlock will never happen because the subtasks with dependencies among them will never be scheduled to the same set of resources.

Now we provide a formal description of the xDCP algorithm. First, we will discuss the definition of the previous and next subtask of a certain subtask in its resource queue:

In a given resource mapping $f(\sigma) : \Gamma \rightarrow \Omega$, for any subtask t, we have its previous and next subtasks mapped in the same resource queue. Formally:

$$Prev(t) = \begin{cases} \psi, & if \ \ \forall t' \in \Gamma, \nexists f(t') = f(t) \wedge t' \preceq t \\ t', & if \ \ f(t') = f(t) \wedge t' \preceq t \\ & \wedge (\forall t'' \in \Gamma, \nexists t' \preceq t'' \preceq t) \end{cases}$$

$$Next(t) = \begin{cases} \psi, & if \ \ \forall t' \in \Gamma, \nexists f(t') = f(t) \wedge t \preceq t' \\ t', & if \ \ f(t') = f(t) \wedge t \preceq t' \\ & \wedge (\forall t'' \in \Gamma, \nexists t \preceq t'' \preceq t') \end{cases}$$

Then comes the definition of *AEFT, DCPL (dynamic critical path length) and ALFT (absolute latest finish time)*:

A subtask can be started only after all its parent subtasks are finished, and all the previous tasks in the same resource queue are also finished. The earliest finish time can be calculated by adding the execution time on the current resource onto the earliest start time. In a given resource mapping $f(\sigma) : \Gamma \rightarrow \Omega$, the absolute earliest finish time of any subtask t, denoted by $AEFT(t)$ is recursively defined as follows:

$$AEFT(t) =$$

$$\begin{cases} 0, & if \ \ t = \psi \vee (Parents(t) = \phi \wedge Prev(t) = \psi) \\ \max\{\max_{\forall \tau \in Parents(t)}\{AEFT(\tau) + \omega(t, f(t))\}, & \\ AEFT(Prev(t)) + \omega(t, f(t))\}, & otherwise \end{cases}$$

The dynamic critical path length of the task graph is another form for defining the term $Time(C, R)$ in section II by using the AEFT:

$$DCPL(f, \Gamma, \Omega) = \max_{\tau \in \Gamma}\{AEFT(\tau)\}$$

Having the DCPL, we can define the latest finish time of a subtask. The DCPL is the summation of the execution time of all the subtasks on the critical path of the task graph. Therefore, if we want to finish the whole execution by the time of DCPL, all the subtasks on the critical path should be finished at exactly their AEFT. The absolute latest finish time of subtask t should be no later than the latest start time of all its children subtasks and its next subtask in the same resource queue. We define it as:

$$ALFT(t) =$$

$$\begin{cases} DCPL(f,\Gamma,\Omega), \\ \qquad if\ \ t=\psi \vee (Childrens(t)=\phi \wedge Next(t)=\psi) \\ \\ \max\{\max_{\forall \tau \in Children(t)}\{ALFT(\tau)-\omega(\tau,f(\tau))\}, \\ ALFT(Next(t))-\omega(Next(t),f(Next(t)))\}, \\ \qquad\qquad otherwise \end{cases}$$

The xDCP algorithm is listed below:

1) Shuffle all the subtasks in Γ onto the resources in Ω. Let $DCPL = 0$;
2) $\forall t \in \Gamma$, set t to be unallocated;
3) $\forall t \in \Gamma$, compute $AEFT(t)$ and $ALFT(t)$;
4) Let t be the subtask t_{ij} selected from Γ which, orderly, follow the three criterions below:
   a) Minimize the value of $ALFT(t_{ij}) - AEFT(t_{ij})$;
   b) Minimize the value of i;
   c) Minimize the value of $AEFT(t_{ij})$;
5) $\forall r \in \Omega$, select r and the slot in r's queue that, assuming t is allocated onto this slot, orderly satisfying:
   a) $ALFT(t) - AEFT(Prev(t)) \geq \omega(t, f(t))$;
   b) Minimize the value of $AEFT(t)$;

   If there exists such a resource and slot, move t onto the selected slot of r, or else not move anything;
6) Set t_{ij} allocated. If $\exists t \in \Gamma$ unallocated, goto 3);
7) If $DCLP(f,\Gamma,\Omega)/DCLP * 100\% < 95\%$, goto 2); or else the algorithm ends.

B. Dynamic scheduling

A most-commonly used dynamic scheduling mechanism is the *Master-Slave (M-S) model*. The M-S model is proven to be quite efficient under most scenarios. It specifies the scheduler to be the master, and all resources as slaves. Initially all the tasks are queued on the master side, and the master will do an initialization by dispatching the first n tasks from the queue head to n resources. After that, the master will wait until some slave reports that his task has been finished. Then it will dispatch the task located in the queue head to the newly spared slave.

For the workflow applications with parameter-sweep tasks, we employ K masters corresponding to K parameter-sweep tasks. For each master there are two queues instead of only one queue in the original M-S model. Initially, all the subtasks are stored in the *Unscheduled* queue. There is another queue called the *Ready* queue. Only subtasks in the Ready queue can be directly dispatched to slave nodes. At the start of scheduling, all the subtasks t with $Parents(t) = \phi$ can be put into the Ready queue, and then some of them (on the head of the Ready queue) will be dispatched. On the accomplishment of subtask t, the master will check all the subtasks in the $Children(t)$. We put all the subtasks in the set

$$UtoR(t) = \begin{array}{l}\{t' | t' \in Children(t) \wedge (\forall t'' \in Parents(t'), \\ \nexists Queue(t'') = Unscheduled)\}\end{array}$$

into their Ready queue respectively. There are two threads for a master. One of them is responsible for dispatching subtasks in the Ready queue onto slaves, while the other is to stage subtasks from the Unscheduled queue to the Ready queue.

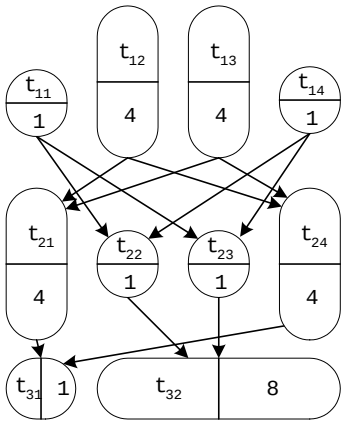

Fig. 5. An example task graph.

However, the M-S model will also fail to achieve effectiveness under some cases. Consider the task graph shown in Figure 5 (with the length $l(t)$ of each subtask indicated). Suppose in $R_1 = \{r_{11}\}, R_2 = \{r_{21}, r_{22}\}, R_3 = \{r_{31}\}$, where $\forall r \in \Omega, p(r) = 1$. From Figure 7 we can see that there should be possibility of a parallelization between the two subtasks in T_1 with the length of 4 and the longest subtask in T_3, namely t_{32}. However, by following the M-S model, the master have to execute the subtasks in T_1 sequentially (because there is only one resource in R_1), which holds t_{32} from being dispatched earlier (due to its indirect dependency to t_{14}).

A priority based mechanism is proposed in this paper to deal with the above problem. In this mechanism, we assume that it would be effective to bring forward the subtasks whose children's ancestors (refer to Figure 6) have been partially finished or already being put into execution. Base on this assumption, we define the priority of each subtask according to how much its children's ancestors have been finished.

Before we list the new algorithm, a term $Ancestors(t)$ should be defined, which refers to all the subtasks that t directly or indirectly depends on:

$$Ancestors(t) =$$

$$\begin{cases} \phi, & if\ Parents(t) = \phi \\ (\bigcup_{t' \in Parents(t)} Ancestors(t')) \cup Parents(t), & otherwise \end{cases}$$

Also the term of the priority and two types of queues: Let $Pri(t)$ denotes the priority of subtask t. Let the Ready queue to be denoted by $Q_R(T_i, \preccurlyeq)$, the Unscheduled queue is denoted by $Q_U(T_i, \preccurlyeq)$, where $\preccurlyeq$ (x, y) is a binary relation that $\forall x, y = 1..N(T_i)$,

$$Pri(t_{ix}) * N(T_i) + x < Pri(t_{iy}) * N(T_i) + y \Leftrightarrow t_{ix} \preccurlyeq t_{iy}$$

means lower the value of $Pri(t)$, higher the priority. For both the queues, the $head$ subtask is defined as $Head(Q(T, \preccurlyeq)) = t$, where $t \in T \wedge (\forall t' \in T, \nexists t' \preccurlyeq t)$. The head refers to the subtask with the highest priority, and will be scheduled first.

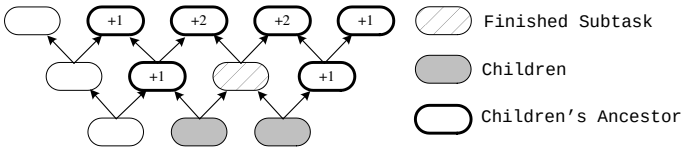

Fig. 6. An example of priority calculation.

The priority of subtasks will be updated on the event of any subtask being finished. As it is shown in Figure 6, once a subtask has been finished, the master first collects all its children into a set. Then it parses all the elements of the set. For each children subtask, all its ancestors will receive an unit-increment on their priority (namely $Pri(t) = Pri(t) - 1$). After the priority adjustment, the master will retrieve a certain number of tasks, according to the allocation status, from the head of the Ready queue and then dispatch them into execution.

The new algorithm, denoted by *pM-S*, is listed below:

1) $\forall t \in \Gamma, Pri(t) = 0$. $\forall T_i \in \Gamma$, $Q_R(T_i, \preccurlyeq) = \{t | t \in T_i \wedge Parents(t) = \phi\}$, $Q_U(T_i, \preccurlyeq) = \{t | t \in T_i \wedge Parents(t) \neq \phi\}$.
2) For $i = 1$ to K, do
   a) For $j = 1$ to $N(R_i)$, if $Q_R(T_i, \preccurlyeq) = \phi$ then break the current for-j-loop; if r_{ij} is not empty, continue the current for-j-loop; or else dispatch the subtask $Head(Q_R(T_i, \preccurlyeq))$ to the resource r_{ij}.
3) If all subtasks in Γ have been scheduled, then algorithm ends;
4) Wait for the event of any subtask t's execution being finished;
5) For each subtask $\tau \in \bigcup_{\forall t' \in Children(t)} Ancestors(t')$, $Pri(\tau) = Pri(\tau) - 1$;
6) Stage subtasks in $UtoR(t)$ to their Ready queues respectively, goto 2).

The right part of Figure 7 shows the schedule result produced by pM-S algorithm. It only costs 15 time units while the M-S schedule costs 22.

IV. Performance evaluation

In this section, we present the comparative evaluation of static and dynamic algorithms (shuffle, xDCP, M-S and pM-S). Also we adjust some parameters in the task graph configuration to investigate how the scheduling results vary. Finally, we compare the algorithms for execution on several sample task graphs of workflow applications. All the experimental results are based on simulation. The workflows and resources in our experiments are implemented following the definition in section II.

A. Metrics and important factors

Since we use randomly generated task graphs in the experiments with factors discussed in sub-section IV-B. Therefore a statistical analysis will be adopted in this paper by calculating

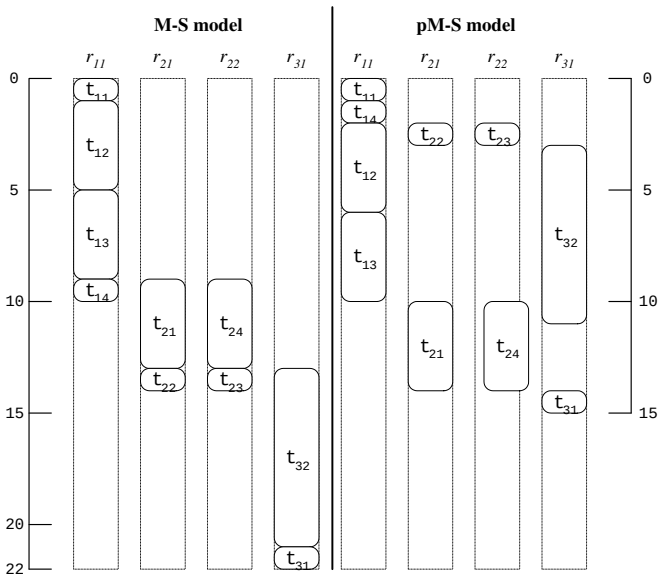

Fig. 7. Comparison of the finish time of subtasks in Figure 5 under M-S model and pM-S model.

and presenting the mean value for a large number of scenarios. Basically we have two metrics for this simulation.

Average Scheduling Effectiveness (ASE): The ASE refers to a ratio (in the format of percentage) between the to-be-measured algorithm and the shuffle algorithm in terms of the simulated total execution time of the whole application after the schedule. Lower the ASE value, higher the effectiveness resulted. The total execution time equals to the value of $DCLP(f, \Gamma, \Omega)$ in xDCP algorithm. Therefore in static algorithm, we simply calculate DCLP value after the schedule has been finished. In M-S and pM-S scheduling, we give each subtask a *Time-to-Live (TTL)* value. All the subtasks that are unscheduled or yet to complete their execution will be given a negative TTL value. For a newly scheduled subtask, we set its TTL value to its execution time on its target resource. Then we look for the subtask t in Γ with the lowest non-negative (≥ 0) TTL value – the accomplishment of t will be the next event. At this point, we subtract all subtasks' TTL value by $TTL(t)$. And add $TTL(t)$ to another value representing the *Totally Lapsed Time (TLT)* which was set to 0 at the very beginning of the schedule. At the end of the schedule, the TLT value will be the total execution time of the application.

Average Beat-down Time (BDT): A *beat-down* of an algorithm means the algorithm has produced the best scheduling result out of all the to-be-measured algorithms under certain task graph. For each experiment we try 30 randomly generated task graphs. The algorithm which wins the lowest ASE in the competition is supposed to have the highest BDT.

Since the complexity of algorithm DCP has been proven to be $O(n^3)$ (n refers to the number of subtasks) and the complexity of xDCP is on the same magnitude to DCP whereas M-S and pM-S algorithms have the complexity of $O(n^2/K)$ and $O(n^2)$ respectively (where K is the number of

parameter-sweep type tasks).

There are also some important factors which can affect the schedule effectiveness:

Range of subtask size (RSS) and of resource throughput (RRT): The range of subtask size can be presented as the ratio between the size of the longest subtasks and the shortest one in the same parameter-sweep task, formally

$$RSS(T_i) = \frac{\max_{\tau \in T_i} \{l(\tau)\}}{\min_{\tau \in T_i} \{l(\tau)\}}$$

Let $RSS = RSS(T_1) = RSS(T_2) = ... = RSS(T_K)$. Also, we have

$$RRT(R_i) = \frac{\max_{\gamma \in R_i} \{p(\gamma)\}}{\min_{\gamma \in R_i} \{p(\gamma)\}}$$

Let $RRT = RRT(R_1) = RRT(R_2) = ... = RRT(R_K)$. We set the RSS and RRT as the boundary of generating random task size and resource throughput. Then we multiply the random value by a base value (each parameter-sweep task has its own base value $\bar{l}$ for task and $\bar{p}$ resource respectively), to get the final value of individual subtask size and resource throughput, namely

$$SubtaskSize = rand(1, RSS) * \bar{l}$$

$$ResourceThpt = rand(1, RRT) * \bar{p}$$

Ratio #Subtask/#Resource (RSR): This ratio will also be calculated inside the parameter-sweep task. As the value of RSR increases, the size of resource queue will be also increased.

Number of parameter-sweep tasks (#PST): As the workflow has more and more parameter-sweep tasks (namely the task graph owns more and more layers), the dependency structure will get more complicated; a shuffle algorithm will leave more gaps in its output schedule, hence the space for optimization will be enlarged.

Average number of the dependencies per subtask (ADPS: This value can be presented as:

$$ADPS = \frac{\sum_{i=2..K} N(D(T_i))}{\sum_{i=2..K} N(T_i)}$$

The index i starts from 2 because T_1 has no dependency at all, thus it will not be counted in either the numerator or the denominator. As the value of ADPS increases, the optimizeability of the task graph will be decreased, since more task dependencies, more restrictions in task placement.

Task graph topology (TGT): It is evident that the topology of task graph will affect greatly on the output schedule results. However, it is difficult to adjust the TGT parametrically due to the existence of too many factors. In subsection IV-C, we will measure the algorithms under 6 representative sample task graphs.

TABLE I

THE CONFIGURATION OF ALL THE MEASURED FACTORS.

Experiment	#PST	RSS	RRT	RSR	ADPS	$\bar{l}$	$\bar{p}$
#PST	3-12	5	5	5	2	20	10
RSS	5	1-10	5	5	2	20	10
RRT	5	5	1-10	5	2	20	10
RSR	5	5	5	1-10	2	20	10
ADPS	5	5	5	5	1.0-5.5	20	10

B. Measurement for randomly generated task graphs

First we consider about the randomly generated task graphs. For fairness, the total number of subtasks is fixed at 512, although the number of subtasks in each parameter-sweep tasks might be randomly generated. The steps of generating task graph is listed below:

1) Given #PST, generating subtasks by ensuring:
   a) The total number of subtasks equals to 512;
   b) The size of subtasks in the same parameter-sweep task follows the given RSS;
2) Generating resources by ensuring:
   a) The number of subtasks and resources in a parameter-sweep task follows the given RSR;
   b) The throughput of resources in the same parameter-sweep task follows the given RRT;
3) Generating task dependencies, ensuring the given ADPS;
4) Run shuffle scheduling, calculating the DCLP value of the output schedule.

For the random task graphs, we observe how the ASE (Figure 8) and BDT (Figure 9) value varies with the factors listed above, for the algorithm xDCP, M-S and pM-S. When we change the value of a factor, the other factors will remain constant at a default value. The configuration of all the factors are shown in Table I. From the Figure 8, we can conclude:

1) The increment of #PST slightly raises the effectiveness of algorithm M-S and pM-S, while drops the effectiveness of algorithm xDCP.
2) The increment of RSS almost has no effect on the effectiveness of algorithm M-S and pM-S, but slightly raises the effectiveness of algorithm xDCP.
3) The increment of RRT raises the effectiveness of all the three algorithms in a large extent.
4) The increment of RSR dramatically raises the effectiveness of all the three algorithms.
5) The increment of ADPS drops the effectiveness of all the three algorithms, but it affects to the xDCP more than the other two algorithms.

By comparing algorithms in Figure 9, we find that in most measured cases, the pM-S will be the best choice for scheduling. However, under the cases of the #PST and RSR decreases, or RRT increases to a certain value, or the ADPS equals to 1, the xDCP may become more effective. That is, the M-S algorithm is always not the best choice.

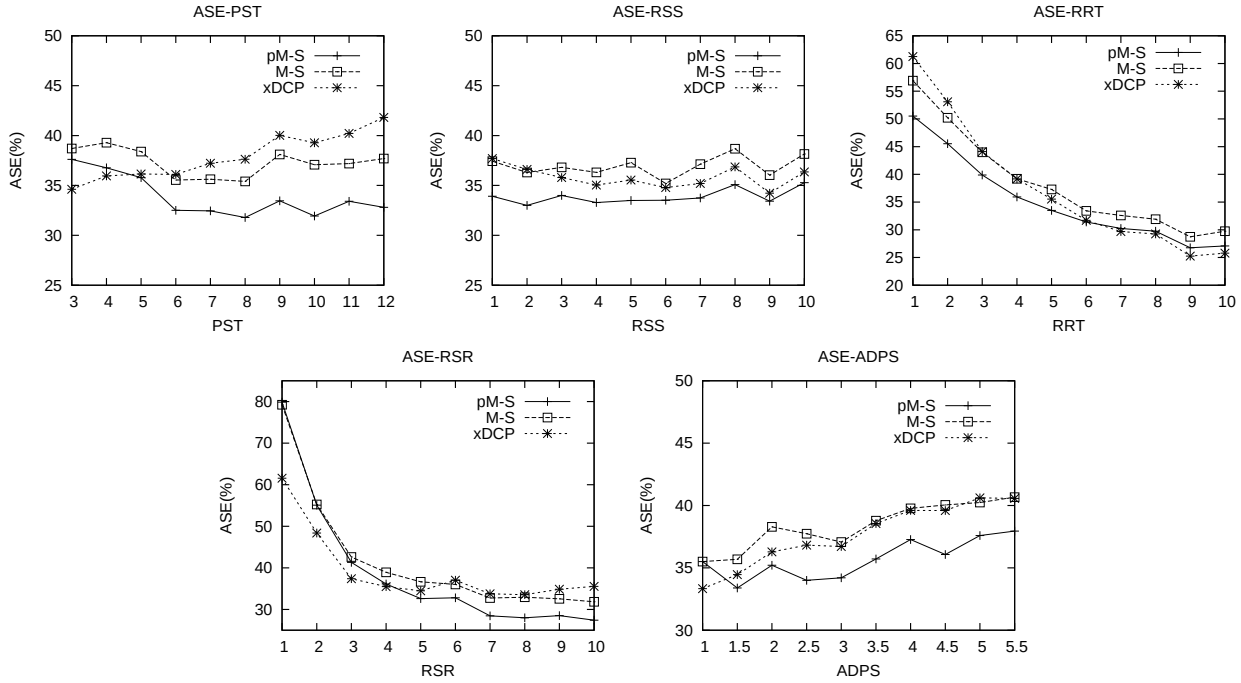

Fig. 8. Average Scheduling Effectiveness (ASE) varies with PST, RSS, RRT, RSR and ADPS.

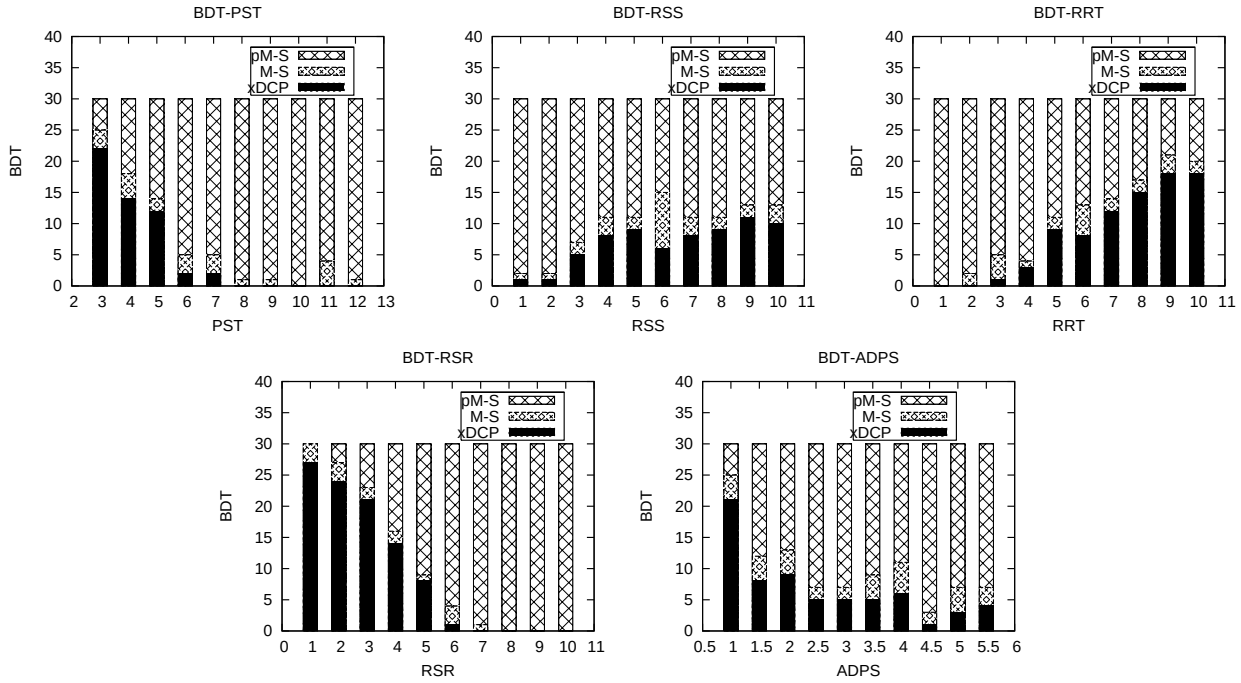

Fig. 9. Average Beat-down Time (BDT) varies with PST, RSS, RRT, RSR and ADPS.

C. Measurement for sample task graphs

In Figure 10, there are several sample task graphs listed. Table II shows the configuration of these task graphs in our experiments.

Parallel task graph (PTG). In parallel task graph, the number of subtasks in each parameter-sweep task is the same constant N, hence the total number of subtasks is $N * K$. For task dependency, we have

$$Parents(t_{ij}) = \{t_{(i-1)j}\}.$$

Of course, $Parents(t_{1j}) = \phi$.

Out-tree task graph (OTG). In OTG, the number of subtasks in each parameter-sweep task is $N(T_i) = \lambda^{(i-1)}$, where λ is called the *branch number* and we suppose that λ is constant in each task graph. The total subtask number of an OTG is $\lambda^K - 1$. In Figure 10(b), $\lambda = 2$. For task dependency, we have

$$Parents(t_{ij}) = \{t_{(i-1)((j-1)/\lambda+1)}\}.$$

In-tree task graph (ITG). In ITG, the number of subtasks in each parameter-sweep task is $N(T_i) = \lambda^{(K-i)}$. The total subtask number of an ITG is the same to an OTG with the same K and λ, that is, $\lambda^K - 1$. For task dependency, we have

$$Parents(t_{ij}) = \{t_{(i-1)x} | \lambda(j-1) < x \leq \lambda j\}.$$

Densified out-tree task graph (DOTG). Densified tree is actually not a tree structure, it is more like a trapezia. In DOTG, the number of subtasks in each parameter-sweep task can be defined in a recursive way: Given $N(T_1) = N_1$, $N(T_i) = \delta(N(T_{i-1}) - 1) + \lambda$, where δ is called the *step value*. If t_{ij}'s parents start at $t_{(i-1)k}$, then $t_{i(j+1)}$'s parents start at $t_{(i-1)(k+\delta)}$. Here we define the task dependency in DOTG:

$$Parents(t_{ij}) = \{t_{(i-1)(x+1)} | \lceil \frac{j-\lambda}{\delta} \rceil \leq x \leq \lfloor \frac{j-1}{\delta} \rfloor\}.$$

Densified in-tree task graph (DITG). In DITG, the number of subtasks in each parameter-sweep task can be defined in a recursive way: Given $N(T_1) = N_1$,

$$N(T_i) = \frac{N(T_{i-1}) - \lambda}{\delta} + 1.$$

The task dependency in DITG:

$$Parents(t_{ij}) = \{t_{(i-1)x} | \delta(j-1) + 1 \leq x \leq \delta(j-1) + \lambda\}.$$

Composite task graph (CTG). Composite task graph is generated from two or more sets of the above sample frameworks. In the example shown in Figure 10(f), the layer $T_1 \rightarrow T_2$ is an OTG with $\lambda = 2$, $T_2 \rightarrow T_3$ is an OTG with $\lambda = 3$, $T_3 \rightarrow T_4$ is an ITG with $\lambda = 3$, $T_4 \rightarrow T_5$ is an ITG with $\lambda = 2$.

Figure 11 shows the comparison the ASE value of xDCP, M-S and pM-S under the six sample task graph. Figure 12 shows the BDT value of the three algorithms. The BDT varies in a larger extent than ASE.

We find that for all the tree-like task graphs (ITG, OTG, DITG and DOTG), the effectiveness of algorithms on out-trees are greater than the in-trees, especially for the xDCP

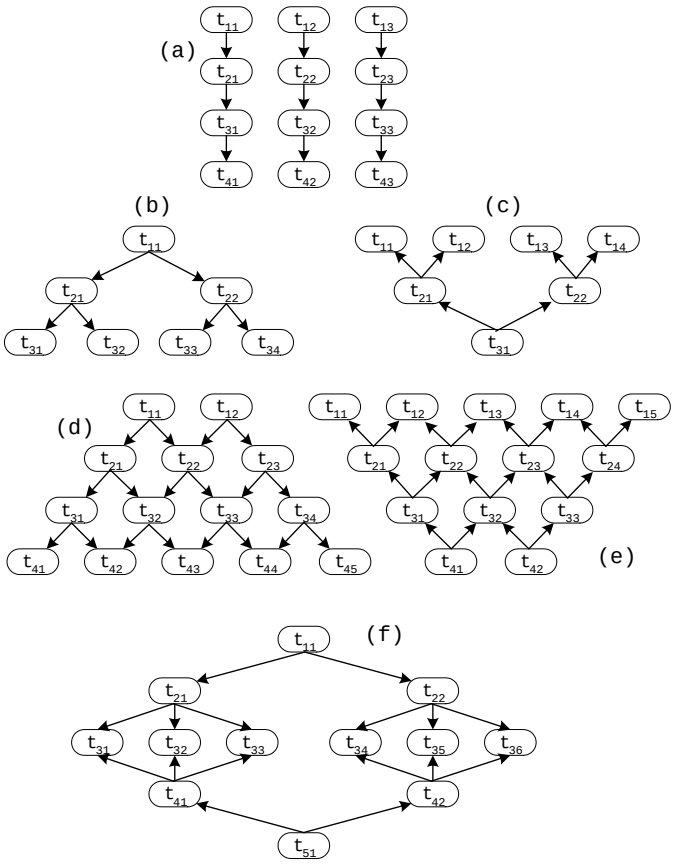

Fig. 10. Six sample task graphs: (a) a parallel task graph (PTG); (b) an out-tree task graph (OTG); (c) an in-tree task graph (ITG); (d) a densified out-tree task graph (DOTG); (e) a densified in-tree task graph (DITG); (f) a composite task graph (CTG).

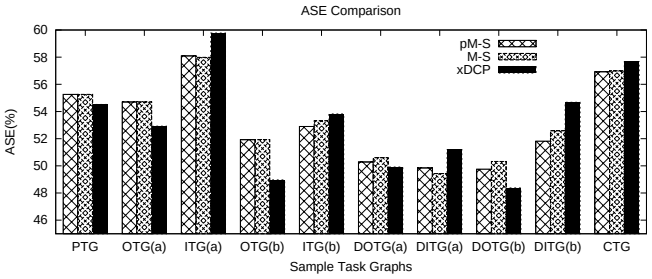

Fig. 11. Comparing ASE value under all the sample task graphs.

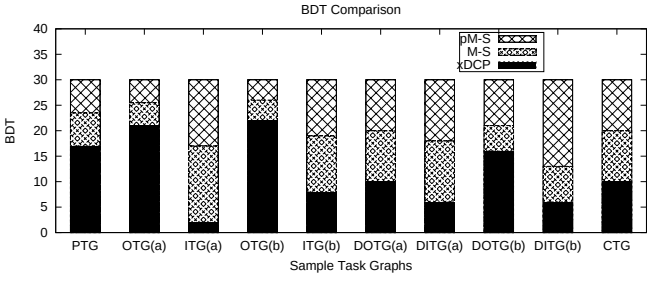

Fig. 12. Comparing BDT value under all the sample task graphs.

TABLE II

CONFIGURATION OF ALL TASK GRAPHS INVOLVED IN THE EXPERIMENT.

TGT	#PST	RSS	RRT	RSR	ADPS	λ	δ
PTG	12	5	5	6	1	—	—
OTG(a)	10	5	5	10	1	2	—
ITG(a)	10	5	5	10	2	2	—
OTG(b)	7	5	5	10	1	3	—
ITG(b)	7	5	5	10	3	3	—
DOTG(a)	5	5	5	10	3	3	1
DITG(a)	5	5	5	10	3	3	1
DOTG(b)	5	5	5	10	2	4	2
DITG(b)	5	5	5	10	4	4	2
CTG	13	5	5	10	1-3-2	3-2	—

algorithm. This is because the ADPS value for OTGs and DOTGs is less than it for ITGs and DITGs. For all OTGs, the ADPS equals to 1 exactly, while for ITGs, the ADPS equals to λ. For all DOTGs, the ADPS equals to

$$\lfloor \frac{j-1}{\delta} \rfloor - \lceil \frac{j-\lambda}{\delta} \rceil + 1,$$

while for DITGs, the ADPS equals to

$$\delta(j-1) + \lambda - \delta(j-1) - 1 + 1 = \lambda.$$

For scenario (a) $(\lambda = 3, \delta = 1)$, the ADPS of DOTG almost equals to the ADPS of DITG (ADPS=3), therefore the ASE has no distinct difference. But for scenario (b) $(\lambda = 4, \delta = 2)$, the ADPS of DITG (ADPS=4) is greater than the average ADPS of DOTG (ADPS=2), therefore the ASE of DITG is remarkably higher than it of DOTG, means the effectiveness of all the algorithms drops.

For tree task graphs (ITG and OTG), comparing the configuration of (a) and (b), we found that the value of #PST has been decreased, and the value of ADPS of DITGs has been increased. According to Figure 8, the ASE value of all the algorithms (at least, for M-S and pM-S) should slightly decrease, which is consistent with the Figure 11.

For parallel graph PTG and out-tree graph OTGs, the resulting ASE value of algorithm M-S and pM-S are completely equal, means they have drawn the same schedule result. This is easy to justify because in these task graphs, every subtask has only one parent. Therefore the priority in pM-S can only be given to the subtasks that are already finished their execution, hence will not do any help in the scheduling.

V. RELATED WORK

In [18], a dynamic, adaptive algorithm is proposed for adjusting the scheduling queue in Grid based task farming applications. The algorithm can keep the queue size to fit the computational capability of current environment. This is a type of implementation of the M-S algorithm, and can certainly be also adopted in our pM-S algorithm.

A pipeline model has been proposed in [19] for task farming in Grid. However, the mechanism it discusses is actually conventional Master-Slave based, with no priority involved. According to the experiment results in section IV, there is still a gap between the effectiveness of M-S and our pM-S algorithm. Also, [19] has not mentioned about optimizing the scenario of multiple parameter-sweep tasks linked together.

The work in [20] focuses on how many slave resources needed in a task farming application where the number of subtasks is given, to achieve the certain effectiveness. Several factors are used to construct the model, including "the workload defined as the work percentage done when executing the largest 20% subtasks", as well as "the variance of the size among the largest 20% subtasks". Also it gives a heuristic algorithm for optimizing (1) the number of slaves (2) the task allocation. Similarly, in this paper we use more elaborate factors like RSS (variance of size of subtasks), RRT (variance of throughput of resources) and RSR (how many slave resources need for a parameter-sweep task) to establish our metrics model, in order to give more precise evaluation on the algorithm's effectiveness.

In [21], a model of Resource Efficiency (RE) is given to help evaluating the effectiveness of task farming scheduling algorithms. In our work, we use the average ratio (ASE) between the total waiting time of the to-be-measured algorithm and the shuffle algorithm. Here the performance of shuffle algorithm is regarded as the basic performance under a specific configuration (includes the value of all the factors, and the topology of the task graph). The ASE then represents the reciprocal of the speedup. We adopt ASE value instead of RE because even under the same configuration of factors, the topology of task graph might still result a remarkable influence on the performance of algorithms, hence gives the distribution of waiting time a large deviation and makes the average value meaningless. By using ASE, the influence of task graph topology could be shielded by the performance of the shuffle algorithm, hence will do little interference on the evaluation result.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented two algorithms that address the problem of optimal scheduling of workflow applications with parameter-sweep tasks. The experiment results shown in section IV have indicated the effectiveness of the proposed algorithms. Also, we compared the algorithms under different configuration and sample task graphs, which shows our effort on comprehensively examining the useability of each algorithm.

In the paper, we assume every parameter-sweep task consume its resource array exclusively. This constraint can be removed by letting each task of workflow utilize multiple resources from Grid resource space. Of course, we can assume that the resources are able to allocate fixed throughput to each parameter-sweep task consuming them, which makes the resource itself could be logically regarded as multiple sets of resources with each being allocated to different tasks. However, in real scheduling, resources may not treat different tasks separately. Instead, all the subtasks may be put into the same queue under the scheduling of an unique algorithm. This makes the scheduling optimization more complicated, because the algorithm need to consider not only the transverse subtasks

in the same parameter-sweep task, but also vertical subtasks sharing the same set of resources.

ACKNOWLEDGMENT

We would like to thank Srikumar Venugopal, Shushant Goel, Jia Yu and Prof. Chen-Khong Tham for their critical comments that helped us in improving the quality of paper presentation.

REFERENCES

[1] D. Bhardwaj, J. Cohen, S. McGough, and S. Newhouse. A Componentized Approach to Grid Enabling Seismic Wave Modeling Application. *The International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, Singapore, Dec. 2004.

[2] C F Sanz-Navarro, S D Kenny, S M Pickles, and A R Porter. Real-time Visualization and Computational Steering of Molecular Dynamics simulations of Materials Science. *Proceedings of the UK e-Science All Hands Meeting*, 31st August - 3rd September, 2004.

[3] Meichun Hsu, editor. *Special Issue on Workflow and Extended Transaction Systems, volume 16(2) of Bulletin of the IEEE Technical Committee on Data Engineering*. June 1993.

[4] F. Leymann and W. Altenhuber, Managing business processes as information resources, *IBM Systems Journal 33(2) (1994)* 326 - 348.

[5] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Lamnitchi, C. Kesselman, P. Kunst, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney. Giggle : A Framework for Constructing Scalable Replica Location Services. In *Supercomputing (SC2002)*, Baltimore, USA: IEEE Computer Society, Washington, DC, USA, November 16-22, 2002.

[6] Z. Guan, F. Hernandez, P. Bangalore, J. Gray, A. Skjellum, V. Velusamy, Y. Liu. Grid-Flow: A Grid-Enabled Scientific Workflow System with a Petri Net-based Interface. *Technical Report*, Dec. 2004. http://www.cis.uab.edu/gray/Pubs/grid-flow.pdf

[7] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Workflow Management in GriPhyN. *The Grid Resource Management*, Kluwer, Netherlands, 2003.

[8] J. Cardoso. Stochastic Workflow Reduction Algorithm. *Technical Report*, LSDIS Lab, Department of Computer Science University of Georgia, 2002.

[9] D. Abramson, J. Giddy and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? *IPDPS 2000*, Cancun, Mexico, 2000.

[10] Y.K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: an Effective Technique for Allocating Task Graphs to Multiprocessors", *IEEE Transaction on Parallel and Distributed Systems*, V7, N5, 1996, pp. 506-521.

[11] G. Riccardi, B. Traversat, U. Chandra, A Master-Slaves Parallel Computation Model, *Supercomputer Research Institute Report*, Florida State University, June 1989.

[12] J. Yu and R. Buyya: A Novel Architecture for Realizing Grid Workflow using Tuple Spaces. *GRID 2004*: 119-128.

[13] L. Wang, H. J. Siegel, V. P. Roychowdhury and A. A. Maciejewski. Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach. *Journal of Parallel and Distributed Computing, Special Issue on Parallel Evolutionary Computing*, Vol. 47, No. 1, Nov. 1997, pp. 8-22.

[14] Shanshan Song, Yu-Kwong Kwok, Kai Hwang: Security-Driven Heuristics and A Fast Genetic Algorithm for Trusted Grid Job Scheduling. *IPDPS 2005*, Denver, Colorado, USA, Apr. 2005.

[15] Soumya Sanyal, Sajal K. Das: MaTCH : Mapping Data-Parallel Tasks on a Heterogeneous Computing Platform Using the Cross-Entropy Heuristic. *IPDPS 2005*, Denver, Colorado, USA, Apr. 2005.

[16] D. Nicol and J. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Transaction on Computers*, 37(9):1073-1087, 1988.

[17] D. Brent Weatherly, David K. Lowenthal, Mario Nakazawa, Franklin Lowenthal: Dyn-MPI: Supporting MPI on Non Dedicated Clusters. *SC 2003*, Phoenix, Arizona, USA, Nov. 2003.

[18] H. Casanova and M. Kim and J. S. Plank and J. Dongarra. Adaptive Scheduling for Task Farming with Grid Middleware. *5th International Euro-Par Conference*, Toulouse, Aug 1999.

[19] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor and I. Wang. Programming Scientific and Distributed Workflow with Triana Services. In *Grid Workflow 2004 Special Issue of Concurrency and Computation: Practice and Experience*, 2005.

[20] M. A. Robers, L. P. Kondi, and A. K. Katsaggelos. SNR scalable video coder using progressive transmission of DCT coefficients. *Proc. SPIE*, pp. 201–212, 1998.

[21] G. Shao, R. Wolski, F. Berman. Performance effects of scheduling strategies for master/slave distributed applications. In *Proc. PDPTA'99*, CSREA, Sunnyvale, CA, 1999.