Dynamic and Online Fault Tolerant Clustering in Scientific Workflows

Weiwei Chen [a,1], Rafael Ferreira de Silva [a], Ewa Deelman [a], Thomas Fahringer [b] and Rizos Sakellariou [c]

[a] *University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA*

[b] *University of Innsbruck, Institute for Computer Science, Innsbruck, Austria*

[c] *University of Manchester, School of Computer Science, Manchester, U.K.*

Abstract. Large scale scientific workflows can be composed of many fine computational granularity tasks. Task clustering has been proven to be an effective method to reduce execution overhead and increase the computational granularity of workflow tasks executing on distributed resources. However, a job composed of multiple tasks may have a greater risk of suffering from failures than a job composed of a single task. Our theoretic analysis and simulation results demonstrate that transient failures can have a significant impact on the runtime performance of scientific workflows that use existing clustering policies that ignore failures. We aim to optimize the workflow makespan by adjusting the clustering granularity in the presence of failures. We propose a general task failure modeling framework and use a Maximum Likelihood Estimation based parameter estimation process to address these performance issues. We further propose three methods to improve the runtime performance of executing workflows in faulty environments. A trace based simulation is performed and it shows that our methods improve the workflow makespan significantly for five important applications.

Keywords. scientific workflows, fault tolerance, parameter estimation, failure, machine learning, task clustering, job grouping

Introduction

Scientific workflows can be composed of many fine computational granularity tasks and the runtime of these tasks may be even shorter than the system overhead. System overhead is the period time during which miscellaneous work other than the users computation is performed. Task clustering [1,2,3,4,5,6] is a technique that merges several short tasks into a single job such that the job runtime is increased and the overall system overhead is decreased. However, existing clustering strategies ignore or underestimate the impact of the occurrence of failures on system behavior, despite the current and increasing importance of failures in large-scale distributed systems, such as Grids [7,8,9], Clouds [10,11,7] and dedicated clusters. Many researchers [12,13,14,15] have emphasized the importance of fault tolerance design and indicated that the failure rates in modern dis-

[1] Corresponding Author: Weiwei Chen, University of Southern California, Information Sciences Institute, Marina del Rey, CA, USA; E-mail: weiweichen@acm.org.

tributed systems are significant. Among all possible failures, we focus on transient failures because they are expected to be more prevalent than permanent failures [12]. For example, denser integration of semiconductor circuits and lower operating voltage levels may increase the likelihood of bit-flips when circuits are bombarded by cosmic rays and other particles [12].

In task clustering, a clustered job consists of multiple tasks. A task is marked as failed (task failure) if it is terminated by unexpected events during the computation of this task. If a task within a job fails, this job has a job failure, even though other tasks within this job do not necessarily fail. In a faulty environment, there are several options for reducing the influence of workflow failures. First, one can simply retry the entire job when its computation is not successful as in the Pegasus [8], ASKALON [16] and Chemomentum [17]. However, some of the tasks within the job may have completed successfully and it could be a waste of time and resources to retry all of the tasks. Second, the application process can be periodically checkpointed such that when a failure occurs, the amount of work to be retried is limited. However, the overheads of checkpointing can limit its benefits [12]. Third, tasks can be replicated to different nodes to avoid failures that are related to one specific worker node [18]. However, inappropriate clustering (and replication) parameters may cause severe performance degradation if they create long-running clustered jobs. As we will show, a long-running job that consists of many tasks has a higher job failure rate even when the inter-arrival time of failures is long.

In this paper, we assume a task-level monitoring service is available. A task-level monitor tells which tasks in a clustered job fail or succeed, while a job-level monitor only tells whether this job fail or not. The job-level fault tolerant clustering has been discussed in our prior work [3].

We propose two horizontal methods and one vertical methods to improve the existing clustering techniques in a faulty environment. The first horizontal method retries the failed tasks within a job. The second horizontal solution dynamically *adjusts the granularity or clustering size* (number of tasks in a job) according to the estimated inter-arrival time of task failures. The vertical method reduces the clustering size by half in each job retry.

In this paper we view the sequence of failure events as a stochastic process and study the distribution of its inter-arrival times, i.e. the time between failures. Our work is based on an assumption that the the distribution parameter of the inter-arrival time is a function of the *type of task*. Tasks of the same type have the same computational program (executable file). In the five workflows we examine in this paper, tasks at the same horizontal level (defined as the longest distance from the entry task of the workflow) of the workflows has the same type. The characteristics of tasks such as the task runtime, memory peak and disk usage are highly related to the task type [19,20]. Task type related failure is a type of failure that only occurs to some specific types of tasks. Samak [21] et al. have analyzed 1,329 real workflow executions across six distinct applications and concluded that the type of a task is among the most significant factors that impacted failures.

Compared to our prior work in [3], we add a parameter learning process to estimate the distribution of the task runtime, the system overhead and the inter-arrival time of failures. We adopt an approach of prior and posterior knowledge based Maximum Likelihood Estimation (MLE) that has been recently used in machine learning. Prior knowledge about the parameters are modeled as a distribution with known parameters. Posterior knowledge about the execution information are also modeled a distribution with a known *shape parameter* and an unknown *scale parameter*. The shape parameter affects the shape of a distribution and the scale parameter affects the stretching or shrinking of a distribution. Both parameters control the characteristics of a distribution. The distribution of the prior and the posterior are in the same family if the likelihood distribution follows some specific distribution and they are called *conjugate distributions*. For example, if the likelihood is a Weibull distribution and we model prior knowledge as an Inverse-Gamma distribution, then the posterior is also an Inverse-Gamma distribution. This simplifies the estimation of parameters and integrates the prior knowledge and posterior knowledge gracefully. More specifically, we define the parameter learning process with only prior knowledge as the static estimation. The process with both prior knowledge and posterior knowledge is called the dynamic estimation since we update the MLE based on the information collected during the execution.

The two horizontal methods were introduced and evaluated in [3] on two workflows. In this paper, we complement this previous paper by studying (*i*) the performance gain of using two horizontal methods and one vertical method over a baseline execution on a larger set of workflows (five widely used scientific applications); (*ii*) the performance impact of the variance of the distribution of the task runtime, the system overheads and the inter-arrival time of failures; (*iii*) the performance difference of dynamic estimation and static estimation with variation of inter-arrival time of failures.

The rest of this paper is organized as follows. Section 1 gives an overview of the related work. Section 2 presents our workflow and failure models. Section 3 details our fault tolerant clustering methods. Section 4 reports experiments and results, and the paper closes with a discussion and conclusions.

1. RELATED WORK

Failure analysis and modeling [13] presents system characteristics such as error and failure distribution and hazard rates. Schroeder et al. [14] has studied the statistics of the data, including the root cause of failures, the mean time between failures, and the mean time to repair. Sahoo et al. [15] analyzed the empirical and statistical properties of system errors and failures from a network of heterogeneous servers running a diverse workload. Oppenheimer et al. [22] analyzed the causes of failures from three large-scale Internet services and the effectiveness of various techniques for preventing and mitigating service failure. McConnel [23] analyzed the transient errors in computer systems and showed that transient errors follow a Weibull distribution. In [24,25] Weibull distribution is one of the best fit for the workflow traces they used. Based on these work, we measure the inter-arrival time of failures in a workflow and then provide methods to improve task clustering.

More and more workflow management systems are taking fault tolerance into consideration. The Pegasus workflow management system [8] has incorporated a task-level monitoring system and used job retry to address the issue of task failures. They also used provenance data to track the failure records and analyzed the causes of failures [21]. Plankensteiner et. al. [26] have surveyed the fault detection, prevention and recovery techniques in current grid workflow management systems such as ASKALON [16], Chemomentum [17], Escogitare [27] and Triana [28]. Recovery techniques such as replication, checkpointing, task resubmission and task migration etc. have been provided. In our paper, we are specifically joining the work of failure analysis and the optimization in task clustering. To be best of our knowledge, none of existing workflow management systems have provided such features.

Overhead analysis [29,30] is a topic of great interest in the Grid community. Stratan et al. [31] evaluate in a real-world environment Grid workflow engines including DAGMan/Condor and Karajan/Globus. Their methodology focuses on five system characteristics: overhead, raw performance, stability, scalability, and reliability. They pointed out that head node consumption should not be negligible and the main bottleneck in a busy system is often the head node. Prodan et al. [30] offered a complete Grid workflow overhead classification and a systematic measurement of overheads. In Chen et al. [32], we extended [30] by providing a measurement of major overheads imposed by workflow management systems and execution environments and analyzed how existing optimization techniques improve runtime by reducing or overlapping overheads. The prevalent existence of system overheads is an important reason why task clustering provides significant performance improvement for workflow-based applications. In this work, we aim to further improve the performance of task clustering in a faulty environment.

Machine learning has been used to predict execution time [33,34,35,19] and system overheads [32], and develop probability distributions for transient failure characteristics. Duan et.al. [33] used Bayesian network to model and predict workflow task runtimes. The important attributes (such as the external load, arguments etc.) are dynamically selected by the Bayesian network and fed into a radial basis function neural network to make further predictions. Ferreira da Silva et. al. [19] used regression trees to dynamically estimate task behavior including process I/O, runtime, memory peak and disk usage. In our paper, we reuse the knowledge gained from prior work on failure analysis, overhead analysis and task runtime analysis. We then use prior knowledge based Maximum Likelihood Estimation to integrate both the knowledge and runtime feedbacks and adjust the estimation accordingly.

The low performance of *fine-grained* tasks is a common problem in widely distributed platforms where the scheduling overhead and queuing times at resources are high, such as Grid and Cloud systems. Several works have addressed the control of task granularity of bag of tasks. For instance, Muthuvelu et al. [36] proposed a clustering algorithm that groups bag of tasks based on their runtime—tasks are grouped up to the resource capacity. Later, they extended their work [37] to determine task granularity based on task file size, CPU time, and resource constraints. Recently, they proposed an online scheduling algorithm [38] that groups tasks based on resource network utilization, user's budget, and application deadline. Ng et al. [39] and Ang et al. [40] introduced bandwidth in the scheduling framework to enhance the performance of task scheduling. Longer tasks are assigned to resources with better bandwidth. Liu and Liao [41] proposed an adaptive fine-grained job scheduling algorithm to group fine-grained tasks according to processing capacity and bandwidth of the current available resources. Although these techniques significantly reduce the impact of scheduling and queuing time overhead, they did not consider fault tolerance.

Task granularity control has also been addressed in scientific workflows. For instance, Singh et al. [2] proposed a level- and label-based clustering. In level-based clustering, tasks at the same level can be clustered together. The number of clusters or tasks per cluster are specified by the user. In the label-based clustering, the user labels tasks that should be clustered together. Although their work considers data dependencies between workflow levels, it is done manually by the users, which is prone to errors. Recently, Ferreira da Silva et al. [5] proposed task grouping and ungrouping algorithms to control workflow task granularity in a non-clairvoyant and online context, where none or few characteristics about the application or resources are known in advance. Their work significantly reduced scheduling and queuing time overheads, but did not consider fault tolerance.

2. Design and Models

2.1. Workflow Model

A workflow is modeled as a directed acyclic graph (DAG), where each node in the DAG often represents a workflow task (t), and the edges represent dependencies between the tasks that constrain the order in which tasks are executed. *Dependencies* typically represent data-flow dependencies in the application, where the output files produced by one task are used as inputs of another task. Each *task* is a computational program and a set of parameters that need to be executed. This model fits several workflow management systems such as Pegasus [8], Askalon [16], and Taverna [42].

In this paper, we assume there is only one execution site with multiple compute resources, such as virtual machines on the clouds. Figure 1 shows a typical workflow execution environment that targets a homogeneous computer cluster, for example, a dedicated cluster or a virtual cluster on Clouds. The submit host prepares a workflow for execution (clustering, mapping, etc.), and worker nodes, at an execution site, execute jobs individually. The main components are introduced below:

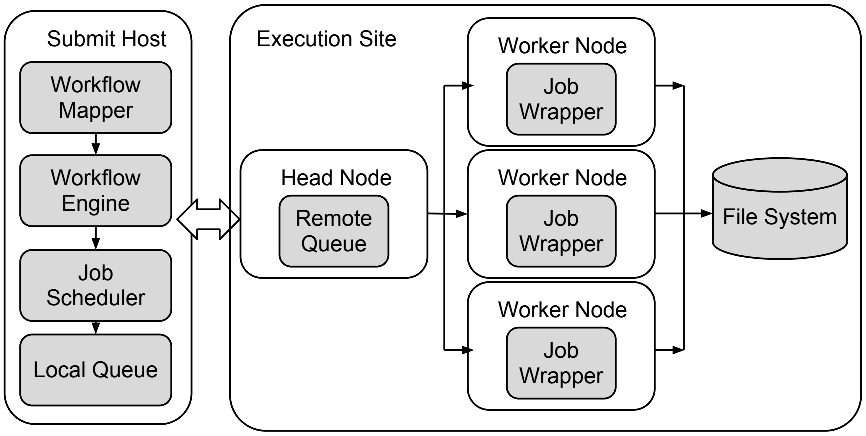

Figure 1. A workflow system model.

Workflow Mapper Generates an executable workflow based on an abstract workflow [8] provided by the user or a workflow composition system. It also restructures the workflow to optimize performance and adds tasks for data management and provenance information generation. In this work, the workflow mapper is used to merge small tasks together into a job such that system overheads are reduced (**task clustering**). A job is a single execution unit in the workflow execution systems and is composed of one or more tasks.

Workflow Engine Executes jobs defined by the workflow in order of their dependencies. Only jobs that have all their parent jobs completed are submitted to the Job Scheduler. The Workflow Engine relies on the resources (compute, storage, and network) defined in the executable workflow to perform computations. The elapsed time from when a job is released (all of its parents have completed successfully) to when it is submitted to the job scheduler is denoted as the workflow engine delay.

Job Scheduler and Local Queue Manage individual workflow jobs and supervise their execution on local and remote resources. The elapsed time from when a task is submitted to the job scheduler to when it starts its execution in a worker node is denoted as the queue delay. It reflects both the efficiency of the job scheduler and the resource availability.

Job Wrapper Extracts tasks from clustered jobs and executes them at the worker nodes. The clustering delay is the elapsed time of the extraction process.

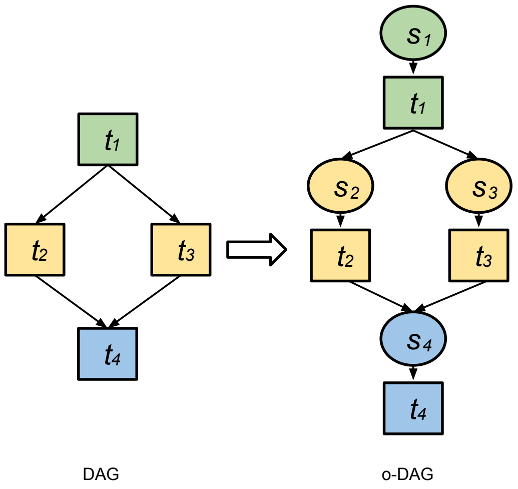

Figure 2. Extending DAG to o-DAG. ‘s’ denotes a system overhead.

We extend the DAG model to be overhead aware (o-DAG). System overheads play an important role in workflow execution and constitute a major part of the overall runtime when tasks are poorly clustered [32]. Figure 2 shows how we augment a DAG to be an o-DAG with the capability to represent system overheads (s) such as workflow engine and queue delays. In addition, system overheads also include data transfer delay caused by staging-in and staging-out data. This classification of system overheads is based on our prior study on workflow analysis [32].

With an o-DAG model, we can explicitly express the process of task clustering. In this paper, we address task clustering horizontally and vertically. **Horizontal Clustering**

(HC) merges multiple tasks that are at the same horizontal level of the workflow, in which the horizontal level of a task is defined as the longest distance from the entry task of the DAG to this task. **Vertical Clustering** (VC) merges tasks within a pipeline of the workflow. Tasks at the same pipeline share a single-parent-single-child relationship, which means a task t_a is the unique parent of a task t_b, which is the unique child of t_a.

Figure 3 shows a simple example of how to perform HC, in which two tasks t_2 and t_3, without a data dependency between them, are merged into a clustered job j_1. A job j is a single execution unit composed by one or multiple task(s). Job wrappers are commonly used to execute clustered jobs, but they add an overhead denoted by the clustering delay c. The clustering delay measures the difference between the sum of the actual task runtimes and the job runtime seen by the job scheduler. After horizontal clustering, t_2 and t_3 in j_1 can be executed in sequence or in parallel, if parallelism in one compute node is supported. In this work, we consider sequential executions only. Given a single resource, the overall runtime for the workflow in Figure 3 (left) is $runtime_l = \sum_{i=1}^{4}(s_i + t_i)$, and the overall runtime for the clustered workflow in Figure 3 (right) is $runtime_r = s_1 + t_1 + s_2 + c_1 + t_2 + t_3 + s_4 + t_4$. $runtime_l > runtime_r$ as long as $c_1 < s_3$, which is the case in many distributed systems since the clustering delay within a single execution node is usually shorter than the scheduling overhead across different execution nodes.

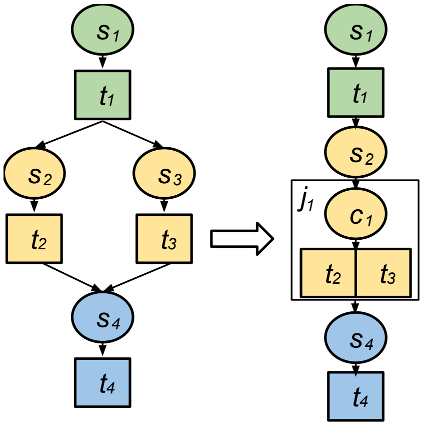

Figure 3. An example of horizontal clustering (color indicates the horizontal level of a task).

Figure 4 illustrates an example of vertical clustering, in which tasks t_2, t_4, and t_6 are merged into j_1, while tasks t_3, t_5, and t_7 are merged into j_2. Similarly, clustering delays c_2 and c_3 are added to j_1 and j_2 respectively, but system overheads s_4, s_5, s_6, and s_7 are removed.

In this paper, the *goal* is to reduce the workflow makespan in a faulty environment by adjusting the clustering size (k). In task clustering, the clustering size (k) is an important parameter to influence the performance. We define it as the number of tasks in a clustered job. For example, the k in Figure 3 is 2. The reason why task clustering can help improve the performance is that it can reduce the scheduling cycles that workflow tasks go through since the number of jobs has decreased. The result is a reduction in the scheduling overhead and possibly other overheads as well [32]. Additionally, in the ideal case without any failures, the clustering size is usually equal to the number of all the

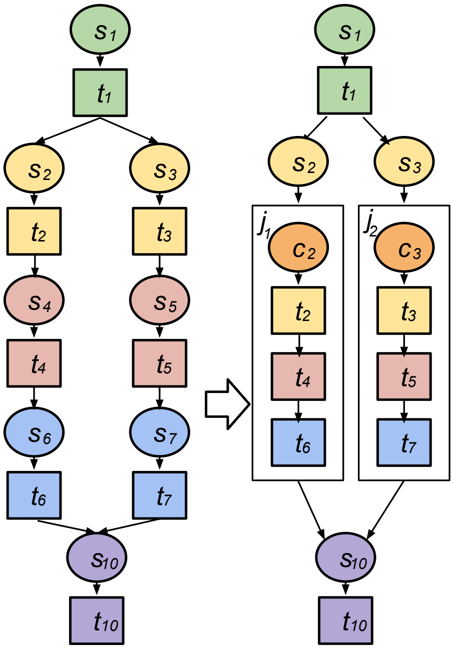

Figure 4. An example of vertical clustering.

parallel tasks divided by the number of available resources. Such a naive setting assures that the number of jobs is equal to the number of resources and the workflow can utilize the resources as much as possible. However, when transient failures exist, we claim that the clustering size should be set based on the failure rates especially the task failure rate. Intuitively speaking, if the task failure rate is high, the clustered jobs may need to be re-executed more often compared to the case without clustering. Such performance degradation will counteract the benefits of reducing scheduling overheads. In the rest of this paper, we will show how to adjust k based on the estimated parameters of the task runtime t, the system overhead s and the inter-arrival time of task failures γ.

2.2. Task Failure Model

In our prior work [32], we have verified that system overheads s fits Gamma or Weibull distribution better than the other two distributions (Exponential and Normal). Schroeder et. al. [14] have verified the inter-arrival time of task failures fits a Weibull distribution with a shape parameter of 0.78 or a gamma distribution better than lognormal and exponential distribution. We will reuse this shape parameter of 0.78 in our failure model. In [24,25] Weibull, Gamma and Lognormal distribution are among the best fit for the workflow traces they used. Without loss of generality, we choose Gamma distribution to model the task runtime ($\boldsymbol{t}$) and the system overhead ($\boldsymbol{s}$) and use Weibull distribution to model the inter-arrival times of failures ($\boldsymbol{\gamma}$). $\boldsymbol{s}$, $\boldsymbol{t}$ and $\boldsymbol{\gamma}$ are all random variables of all tasks instead of one specific task.

Probability distributions such as Weibull and Gamma are usually described with two parameters: the *shape parameter* (ϕ) and the *scale parameter* (θ). The shape parameter affects the shape of a distribution and the scale parameter affects the stretching or shrinking of a distribution. Both of them control the characteristics of a distribution. For example, the mean of a Gamma distribution is $\phi\theta$ and the Maximum Likelihood Estimation or MLE is $(\phi - 1)\theta$.

Assume a, b are the parameters of the prior knowledge, D is the observed dataset and θ is the parameter we aim to estimate. In Bayesian probability theory, if the posterior distribution $p(\theta|D,a,b)$ are in the same family as the prior distribution $p(\theta|a,b)$, the prior and the posterior are then called conjugate distributions, and the prior is called a conjugate prior for the likelihood function. For example, the Inverse-Gamma family is conjugate to itself (or self-conjugate) with respect to a Weibull likelihood function: if the likelihood function is Weibull, choosing a Inverse-Gamma prior over the mean will ensure that the posterior distribution is also Inverse-Gamma. This simplifies the estimation of parameters since we can reuse the prior work from other researchers [14, 25,24,32] on the failure analysis and performance analysis.

After we observe data D, we compute the posterior distribution of θ:

$$\begin{aligned} p(\theta|D,a,b) &= \frac{p(\theta|a,b) \times p(D|\theta)}{p(D|a,b)} \\ &\propto p(\theta|a,b) \times p(D|\theta) \end{aligned}$$

In our paper, D can be the observed inter-arrival time of failures X, the observed task runtime RT or the observed system overheads S. $X = \{x_1, x_2, ..., x_n\}$ is the observed data of $\boldsymbol{\gamma}$ during the runtime. Similarly, we define $RT = \{t_1, t_2, ..., t_n\}$ and $S = \{s_1, s_2, ..., s_n\}$ are the observed data of $\boldsymbol{t}$ and $\boldsymbol{s}$ respectively. $p(\theta|X,a,b)$ is the posterior that we aim to compute. $p(\theta|a,b)$ is the prior, which we have already known from previous work. $p(X|\theta)$ is the likelihood. We only need to know its shape parameter instead of its scale parameter, which reduces the work to estimate its scale parameter.

More specifically, we model the inter-arrival time of failures ($\boldsymbol{\gamma}$) with a Weibull distribution as [14] that has a known shape parameter of ϕ_γ and an unknown scale parameter θ_γ: $\boldsymbol{\gamma} \sim W(\theta_\gamma, \phi_\gamma)$.

The conjugate pair of a Weibull distribution with a known shape parameter ϕ_γ is an Inverse-Gamma distribution, which means if the prior follows an Inverse-Gamma distribution $\Gamma^{-1}(a_\gamma, b_\gamma)$ with the shape parameter as a_γ and the scale parameter as b_γ, then the posterior follows an Inverse-Gamma distribution:

$$\theta_\gamma \sim \Gamma^{-1}(a_\gamma + n, b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}) \tag{1}$$

The MLE (Maximum Likelihood Estimation) of the scale parameter θ_γ is:

$$MLE(\theta_\gamma) = \frac{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}{a_\gamma + n + 1} \tag{2}$$

The understanding of the MLE has two folds: initially we do not have any data and thus the MLE is $\dfrac{b_\gamma}{a_\gamma + 1}$, which means it is determined by the prior knowledge; when $n \to \infty$, the MLE $\frac{\sum_{i=1}^{n} x_i^{\phi_\gamma}}{n+1} \to \overline{x^{\phi_\gamma}}$, which means it is determined by the observed data and it is close to the regularized average of the observed data. The static estimation process only utilizes the prior knowledge and the dynamic estimation process uses both the prior and the posterior knowledge.

We model the task runtime ($\boldsymbol{t}$) with a Gamma distribution as [24,25] with a known shape parameter ϕ_t and an unknown scale parameter θ_t. The conjugate pair of Gamma distribution with a known shape parameter is also a Gamma distribution. If the prior follows $\Gamma(a_t, b_t)$, while a_t is the shape parameter and b_t is the rate parameter (or $\frac{1}{b_t}$ is the scale parameter), the posterior follows $\Gamma(a_t + n\phi_t, b_t + \sum_{i=1}^{n} t_i)$ with $a_t + n\phi_t$ as the shape parameter and $b_t + \sum_{i=1}^{n} t_i$ as the rate parameter. The MLE of θ_t is $\frac{b_t + \sum_{i=1}^{n} t_i}{a_t + n\phi_t - 1}$.

Similarly, if we model the system overhead $\boldsymbol{s}$ with a Gamma distribution with a known shape parameter ϕ_s and an unknown scale parameter θ_s, and the prior is $\Gamma(a_s, b_s)$, the MLE of θ_s is $\frac{b_s + \sum_{i=1}^{n} s_i}{a_s + n\phi_s - 1}$.

We have already assumed the task runtime, system overhead and inter-arrival time between failures are a function of task types. Since in scientific workflows, tasks at the different level (the deepest depth from the entry task to this task) are usually of different type, we model the runtime level by level. Given n independent tasks at the same level and the distribution of the task runtime, the system overhead, and the inter-arrival time of failures, we aim to reduce the cumulative runtime $\boldsymbol{M}$ of completing these tasks by adjusting the clustering size k (the number of tasks in a job). $\boldsymbol{M}$ is also a random variable and it includes the system overheads and the runtime of the clustered job and its subsequent retry jobs if the first try fails. We also assume the task failures are independent in each worker node (but with the same distribution) without considering the failures that brings the whole system down such as a failure in the shared file system.

The runtime of a job is a random variable indicated by $\boldsymbol{d}$. A clustered job succeeds only if all of its tasks succeed. The job runtime is the sum of the cumulative task runtime of k tasks and a system overhead. We assume the task runtime of each task is independent of each other, therefore the cumulative task runtime of k tasks is also a Gamma distribution since the sum of Gamma distributions with the same shape parameter is still a Gamma distribution. We also assume the system overhead is also independent of all the task runtimes and it has the same shape parameter ($\phi_{ts} = \phi_t = \phi_s$) with the task runtime. The job runtime irrespective of whether it succeeds of fails is:

$$\boldsymbol{d} \sim \Gamma(\phi_{ts}, k\theta_t + \theta_s) \tag{3}$$

$$MLE(\boldsymbol{d}) = (k\theta_t + \theta_s)(\phi_{ts} - 1) \tag{4}$$

Let the retry time of clustered jobs to be N. The process to run and retry a job is a Bernoulli trial with only two results: success or failure. Once a job fails, it will be re-executed until it is eventually completed successfully since we assume the failures are transient. For a given job runtime d_i, by definition:

$$N_i = \frac{1}{1-F(d_i)} = \frac{1}{e^{-(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}}} = e^{(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}} \tag{5}$$

$F(x)$ is the CDF of $\boldsymbol{\gamma}$. The time to complete d_i successfully in a faulty environment is

$$M_i = d_i N_i = d_i e^{(\frac{d_i}{\theta_\gamma})^{\phi_\gamma}} \tag{6}$$

Equation 6 has involved two distributions $\boldsymbol{d}$ and θ_γ (ϕ_γ is known). From Equation 1, we have:

$$\frac{1}{\theta_\gamma} \sim \Gamma(a_\gamma + n, \frac{1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}}) \tag{7}$$

$$MLE(\frac{1}{\theta_\gamma}) = \frac{a_\gamma + n - 1}{b_\gamma + \sum_{i=1}^{n} x_i^{\phi_\gamma}} \tag{8}$$

M_i is a monotonic increasing function of both d_i and $\frac{1}{\theta_\gamma}$, and the two random variables are independent of each other, therefore:

$$MLE(M_i) = MLE(d_i) e^{(MLE(d_i) MLE(\frac{1}{\theta_\gamma}))^{\phi_\gamma}} \tag{9}$$

Equation 9 means to achieve the $MLE(M_i)$, we just need to achieve $MLE(d_i)$ and $MLE(\frac{1}{\theta_\gamma})$ at the same time. In both dimensions (d_i and $\frac{1}{\theta_\gamma}$), M_i is a Gamma distribution and each M_i has the same distribution parameters, therefore:

$$\boldsymbol{M} = \frac{1}{r}\sum_{i=1}^{\frac{n}{k}} M_i \sim \Gamma$$

$$MLE(\boldsymbol{M}) = \frac{n}{rk}MLE(M_i) \tag{10}$$

$$= \frac{n}{rk}MLE(d_i)e^{(MLE(d_i)MLE(\frac{1}{\theta_\gamma}))^{\phi_\gamma}} \tag{11}$$

r is the number of resources. In our paper, we consider a compute cluster as a homogeneous cluster, which is usually true in dedicated clusters and Clouds.

Let k^* be the optimal clustering size that minimizes Equation 10. $\arg min$ [2] stands for the argument (k) of the minimum, that is to say, the k such that $MLE(\boldsymbol{M})$ attains its minimum value.

$$k^* = \arg min\{MLE(\boldsymbol{M})\} \tag{12}$$

It is difficult to find a analytical solution of k^*. However, there are a few constraints that can simplify the estimation of k^*: (*i*) k can only be an integer in practice; (*ii*) $MLE(\boldsymbol{M})$ is continuous and has one minimal. Methods such as Newton's method can be used to find the minimal $MLE(\boldsymbol{M})$ and the corresponding k. Figure 5 shows an example of $MLE(\boldsymbol{M})$ using static estimation with a low task failure rate ($\theta_\gamma = 40$s), a medium task failure rate ($\theta_\gamma = 30$s) and a high task failure rate ($\theta_\gamma = 20$s). Other parameters are $n = 50$, $\theta_t = 5$ sec, and $\theta_s = 50$ sec and all the shape parameters are 2 for simplicity. These parameters are close to the level of mProjectPP in the Montage workflow that we simulate in Section 4. Figure 6 shows the relationship between the optimal clustering size (k^*) and θ_γ, which is a non-decreasing function. The optimal clustering size (marked with red dots in Figure 6) when $\theta_\gamma = 20, 30, 40$ is 4,5 and 6 respectively. It is consistent with our expectation since the longer the inter-arrival time of failures is, the lower the task failure rate is. With a lower task failure rate, a larger k assures that we reduce system overheads without retrying many times.

From this theoretic analysis, we conclude that (*i*) the longer the inter-arrival time of failures is, the better runtime performance the task clustering has; (*ii*), adjusting the clustering size according to the detected inter-arrival time can improve the runtime performance.

[2] Wiki: http://en.wikipedia.org/wiki/Arg_max

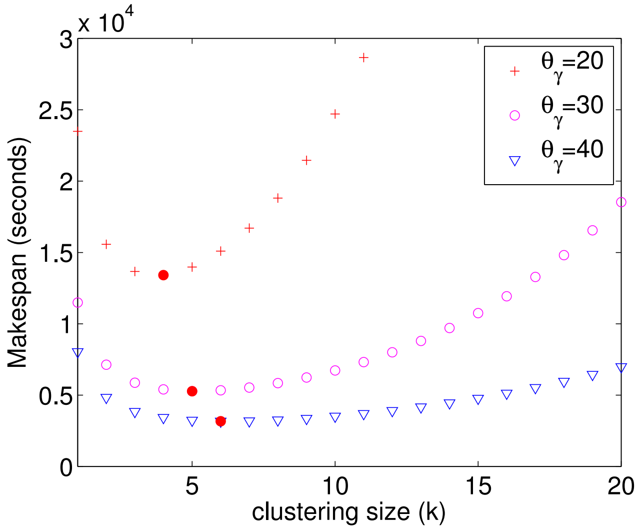

Figure 5. Makespan with different clustering size and θ_γ. ($n = 1000$, $r = 20$, $\theta_t = 5$ sec, $\theta_s = 50$ sec). The red dots are the minimums.

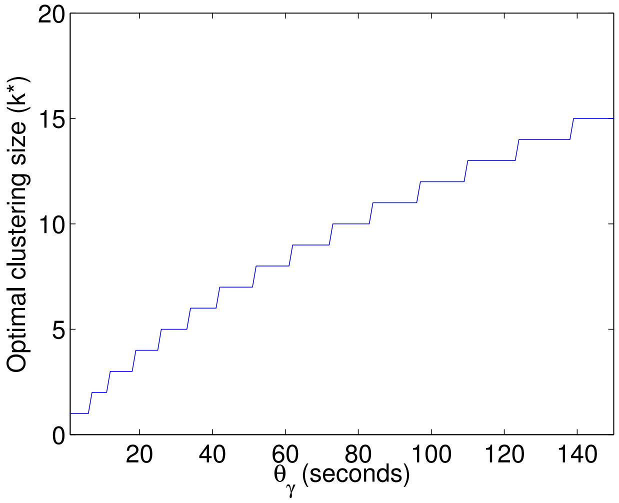

Figure 6. Optimal clustering size (k*) with different θ_γ ($n = 1000$, $r = 20$, $\theta_t = 5$ sec, $\theta_s = 50$ sec)

3. Fault Tolerant Clustering

As indicated in Section 2.1, inappropriate clustering may increase the makespan of executing a workflow. To improve the fault tolerance from the point of view of clustering, we propose three methods: Dynamic Reclustering (*DR*), Selective Reclustering (*SR*) and Vertical Reclustering (*VR*). In the experiments, we compare the performance of our fault tolerant clustering methods to an existing version of Horizontal Clustering (HC) [2] technique. In this subsection, we first briefly describe this algorithm.

Horizontal Clustering (*HC*). Horizontal Clustering (HC) merges multiple tasks that are at the same horizontal level of the workflow. The clustering granularity (number of tasks within a cluster) of a clustered job is controlled by the user, who defines either the number of tasks per clustered job (*clusters.size*), or the number of clustered jobs per horizontal level of the workflow (*clusters.num*). This algorithm has been implemented and used in Pegasus [2]. For simplicity, we define *clusters.num* as the number of available resources. In our prior work [1,43], we have compared the runtime performance with different clustering granularity. The pseudocode of the HC technique is shown in Algorithm 1. The Clustering and Merge Procedure are called in the initial task clustering process while the Reclustering Procedure is called when there is a failed job returned.

Algorithm 1 Horizontal Clustering algorithm.

```
Require: $W$: workflow; $C$: max number of tasks per job defined by clusters.size or clusters.num
procedure CLUSTERING($W,C$)
    for $level < depth(W)$ do
        $TL \leftarrow$ GETTASKSATLEVEL($W, level$)        ▷ Partition $W$ based on depth
        $CL \leftarrow$ MERGE($TL, C$)                     ▷ Returns a list of clustered jobs
        $W \leftarrow W - TL + CL$                         ▷ Merge dependencies as well
    end for
end procedure
procedure MERGE($TL, C$)
    $J \leftarrow \{\}$                                    ▷ An empty job
    $CL \leftarrow \{\}$                                   ▷ An empty list of clustered jobs
    while $TL$ is not empty do
        $J$.add ($TL$.pop($C$)                             ▷ Pops $C$ tasks that are not merged
        $CL$.add( $J$)
    end while
    return $CL$
end procedure
procedure RECLUSTERING($J$)                                ▷ $J$ is a failed job
    $J_{new} \leftarrow$ COPYOF($J$)                       ▷ Copy Job $J$
    $W \leftarrow W + J_{new}$                             ▷ Reexecute it
end procedure
```

Figure 7 shows an example where the initial clustering size is 4 and thereby there are four tasks in a clustered job at the beginning. During execution, three out of these tasks (t_1, t_2, t_3) fail. HC will keep retrying all of the four tasks in next try until all of them succeed. Such a retry mechanism has been implemented and used in Pegasus [2].

Selective Reclustering (*SR*). In HC, the clustering engine does not adjust the clustering size even when it continuously sees many failures. We further improve the performance with Selective Reclustering that selects the failed tasks in a clustered job and merges them into a new clustered job. SR is different to the HC in that the latter method retries all tasks of a failed job even though some of the tasks have succeeded. SR requires task level monitoring in workflow management systems.

Figure 8 shows an example of SR. At the first try, there are four tasks and three of them (t_1, t_2, t_3) have failed. One task (t_4) succeeds and exits. Only the three failed tasks are merged again into a new clustered job j_2 and the job is retried. This approach does not intend to adjust the clustering size, although the clustering size will be smaller and smaller spontaneously after each retry since there are less and less tasks in a clustered job. In this case, the clustering size has decreased from 4 to 3. However, the optimal

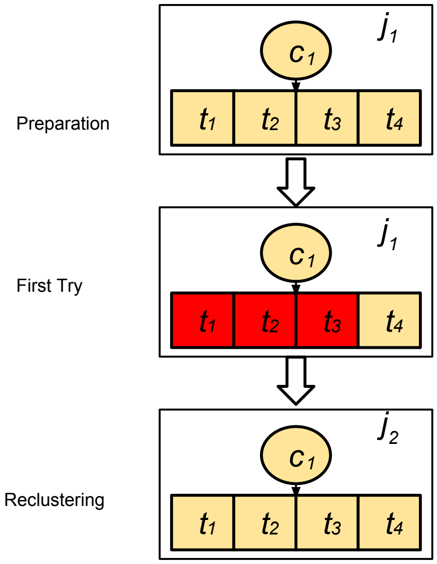

Figure 7. Horizontal Clustering (red boxes are failed tasks)

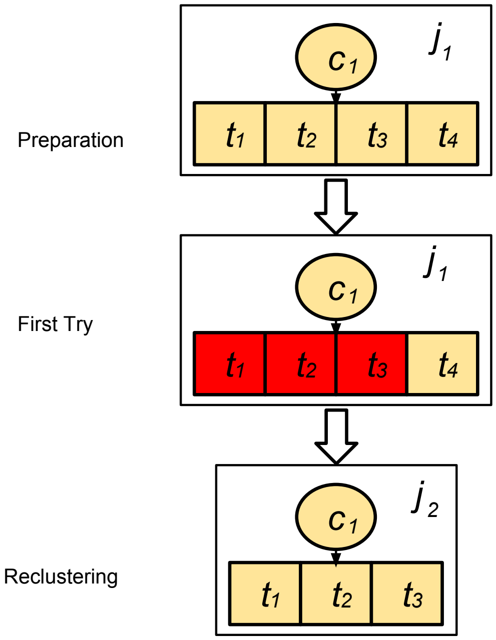

Figure 8. Selective Reclustering (red boxes are failed tasks)

clustering size may not be 3, which limits its performance if the θ_γ is small and k should be decreased as much as possible. The advantage of SR is that it is simple to implement and be incorporated into existing workflow management systems without loss of much efficiency as shown in Section 4. It also serves as a comparison with the Dynamic Reclustering approach that we propose below. Algorithm 2 shows the pseudocode of SR. The Clustering and Merge procedures are the same as those in HC.

Algorithm 2 Selective Reclustering algorithm.

```
Require: $W$: workflow; $C$: max number of tasks per job defined by clusters.size or clusters.num
procedure RECLUSTERING($J$)                                  ▷ $J$ is a failed job
    $TL \leftarrow$ GETTASKS($J$)
    $J_{new} \leftarrow \{\}$                                 ▷ An empty job
    for all Task $t$ in $TL$ do
        if $t$ is failed then
            $J_{new}$.add ($t$)
        end if
    end for
    $W \leftarrow W + J_{new}$                                ▷ Reexecute it
end procedure
```

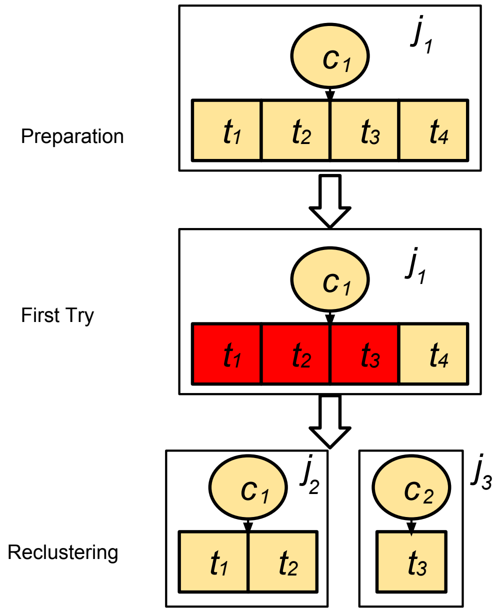

Figure 9. Dynamic Reclustering (red boxes are failed tasks)

Dynamic Reclustering (*DR*). Selective Reclustering does not analyze the clustering size rather, it uses a self-adjusted approach to reduce the clustering size if the failure rate is too high. However, it is blind about the optimal clustering size and the actual clustering size may be larger or smaller than the optimal clustering size. We then propose the second method, Dynamic Reclustering. In DR, only failed tasks are merged into new clustered jobs and the clustering size is set to be k^* according to Equation 12. DR requires the ability to detect which tasks have failed in a job.

Figure 9 shows an example where the initial clustering size is 4 and thereby there are four tasks in a clustered job at the beginning. At the first try, three tasks within a clustered job have failed. Therefore we have only three tasks to retry and further we need to decrease the clustering size (in this case it is 2) accordingly. We end up with two new jobs j_2 (that has t_1 and t_2) and j_3 that has t_3. Algorithm 3 shows the pseudocode of DR. The Clustering and Merge procedures are the same as those in HC.

Algorithm 3 Dynamic Reclustering algorithm.

```
Require: $W$: workflow; $C$: max number of tasks per job defined by clusters.size or clusters.num
procedure RECLUSTERING($J$)                                   ▷ $J$ is a failed job
    $TL \leftarrow$ GETTASKS($J$)
    $J_{new} \leftarrow \{\}$
    for all Task $t$ in $TL$ do
        if $t$ is failed then
            $J_{new}$.add ($t$)
        end if
        if $J_{new}.size() > k^*$ then
            $W \leftarrow W + J_{new}$
            $J_{new} \leftarrow \{\}$
        end if
    end for
    $W \leftarrow W + J_{new}$                                ▷ Reexecute it
end procedure
```

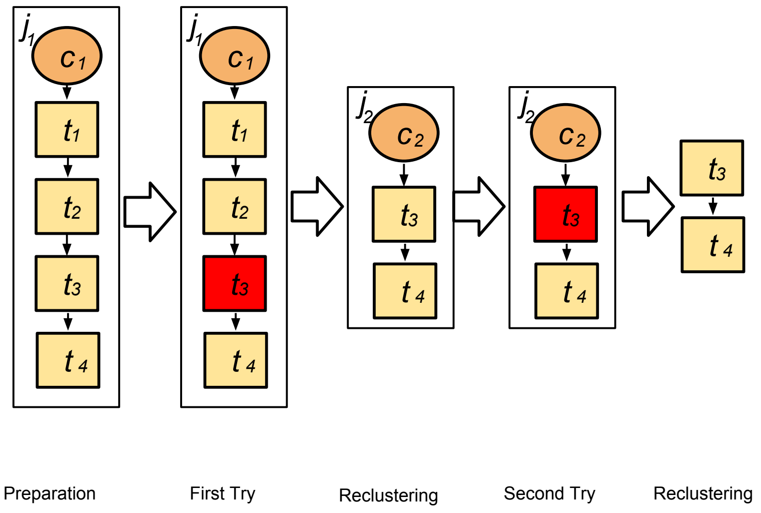

Figure 10. Vertical Reclustering (red boxes are failed tasks)

Vertical Reclustering (*VR*). VR is an extension of Vertical Clustering. Similar to Selective Reclustering, Vertical Reclustering only retries failed tasks. If there is a failure detected, we decrease k by half and recluster them. In Figure 10, if there is no assumption of failures initially, we put all the tasks (t_1, t_2, t_3, t_4) from the same pipeline into a clustered job. t_3 fails at the first try assuming it is a failure-prone task (its θ_γ is short). VR retries only the failed tasks (t_3) and tasks that are not completed (t_4) and merges them again into a new job j_2. In the second try, j_2 unfortunately fails and we divide it into two tasks (t_3 and t_4). Since the clustering size is already 1, VR performs no vertical clustering anymore and would continue retrying t_3 and t_4 (but still following their data dependency) until they succeed. Algorithm 4 shows the pseudocode of VR.

Algorithm 4 Vertical Reclustering algorithm.

```
Require: $W$: workflow;
procedure CLUSTERING($W$)
    for $level < depth(W)$ do
        $TL \leftarrow$ GETTASKSATLEVEL($W, level$)          ▷ Partition $W$ based on depth
        $CL, TL_{merged} \leftarrow$ MERGE($TL$)              ▷ Returns a list of clustered jobs
        $W \leftarrow W - TL_{merged} + CL$                   ▷ Merge dependencies as well
    end for
end procedure
procedure MERGE($TL$)
    $TL_{merged} \leftarrow TL$                               ▷ All the tasks that have been merged
    $CL \leftarrow \{\}$                                      ▷ An empty list of clustered jobs
    for all $t$ in $TL$ do
        $J \leftarrow \{t\}$
        while $t$ has only one child $t_{child}$ do
            $J$.add ($t_{child}$)
            $TL_{merged} \leftarrow TL_{merged} + t_{child}$
            $t \leftarrow t_{child}$
        end while
        $CL$.add( $J$)
    end for
    return $CL$, $TL_{merged}$
end procedure
procedure RECLUSTERING($J$)                                   ▷ $J$ is a failed job
    $TL \leftarrow$ GETTASKS($J$)
    $k^* \leftarrow J.size()/2$                               ▷ Reduce the clustering size by half
    $J_{new} \leftarrow \{\}$
    for all Task $t$ in $TL$ do
        if $t$ is failed or not completed then
            $J_{new}$.add ($t$)
        end if
        if $J_{new}.size() > k^*$ then
            $W \leftarrow W + J_{new}$
            $J_{new} \leftarrow \{\}$
        end if
    end for
    $W \leftarrow W + J_{new}$                                ▷ Reexecute it
end procedure
```

4. Experiments and Discussions

In this section, we evaluate our methods with five workflows, whose runtime information is gathered from real execution traces. The simulation-based approach allows us to control system parameters such as the inter-arrival time of task failures in order to clearly demonstrate the reliability of the algorithms. Our methods can also be applied to real workflow management systems as long as they support task-level failure monitoring.

4.1. Scientific workflow applications

Five real scientific workflow applications are used in the experiments: LIGO Inspiral analysis [44], Montage [45], CyberShake [46], Epigenomics [47], and SIPHT [48]. In this subsection, we describe each workflow application and present their main characteristics and structures.

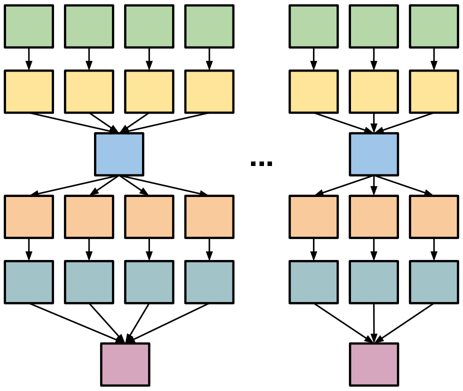

Figure 11. A simplified visualization of the LIGO Inspiral workflow.

LIGO Laser Interferometer Gravitational Wave Observatory (LIGO) [44] workflows are used to search for gravitational wave signatures in data collected by large-scale interferometers. The observatories' mission is to detect and measure gravitational waves predicted by general relativity (Einstein's theory of gravity), in which gravity is described as due to the curvature of the fabric of time and space. The LIGO Inspiral workflow is a data-intensive workflow. Figure 11 shows a simplified version of this workflow. The LIGO Inspiral workflow is separated into multiple groups of interconnected tasks, which we call branches in the rest of our paper. However, each branch may have a different number of pipelines as shown in Figure 11.

Montage Montage [45] is an astronomy application that is used to construct large image mosaics of the sky. Input images are reprojected onto a sphere and overlap is calculated for each input image. The application re-projects input images to the correct orientation while keeping background emission level constant in all images. The images are added by rectifying them to a common flux scale and background level. Finally the reprojected images are co-added into a final mosaic. The resulting mosaic image can provide a much deeper and detailed understanding of the portion of the sky in question. Figure 12 illustrates a small Montage workflow. The size of the workflow depends on the number of images used in constructing the desired mosaic of the sky.

Cybershake CyberShake [46] is a seismology application that calculates Probabilistic Seismic Hazard curves for geographic sites in the Southern California region. It identifies all ruptures within 200km of the site of interest and converts rupture definition into multiple rupture variations with differing hypocenter locations and slip distributions. It then calculates synthetic seismograms for each rupture variance, and peak intensity measures are then extracted from these synthetics and combined with the original rupture

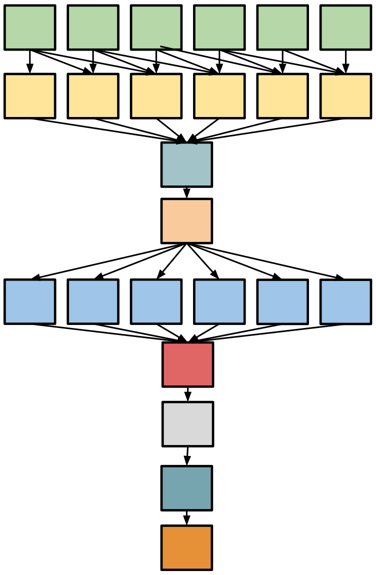

Figure 12. A simplified visualization of the Montage workflow.

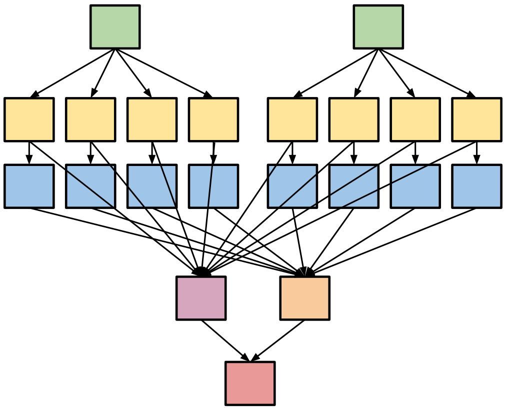

Figure 13. A simplified visualization of the CyberShake workflow.

probabilities to produce probabilistic seismic hazard curves for the site. Figure 13 shows an illustration of the Cybershake workflow.

Epigenomics The Epigenomics workflow [47] is a data-parallel workflow. Initial data are acquired from the Illumina-Solexa Genetic Analyzer in the form of DNA sequence lanes. Each Solexa machine can generate multiple lanes of DNA sequences. These data are converted into a format that can be used by sequence mapping software. The mapping software can do one of two major tasks. It either maps short DNA reads from the sequence data onto a reference genome, or it takes all the short reads, treats them as small pieces in a puzzle and then tries to assemble an entire genome. In our experiments,

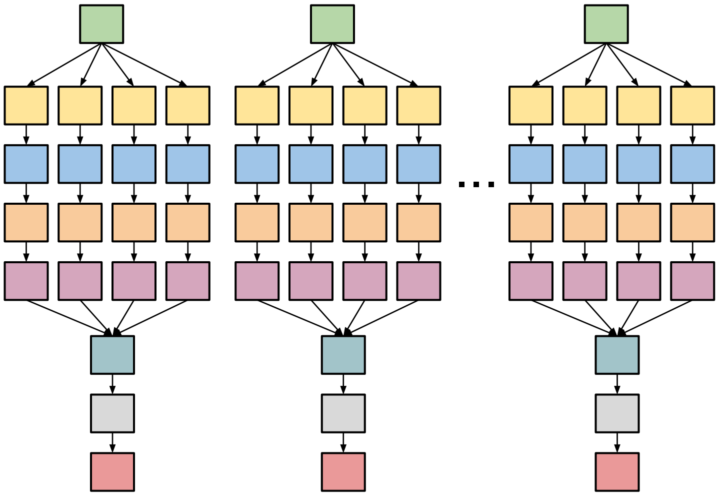

Figure 14. A simplified visualization of the Epigenomics workflow with multiple branches.

the workflow maps DNA sequences to the correct locations in a reference Genome. This generates a map that displays the sequence density showing how many times a certain sequence expresses itself on a particular location on the reference genome. Epigenomics is a CPU-intensive application and its simplified structure is shown in Figure 14.

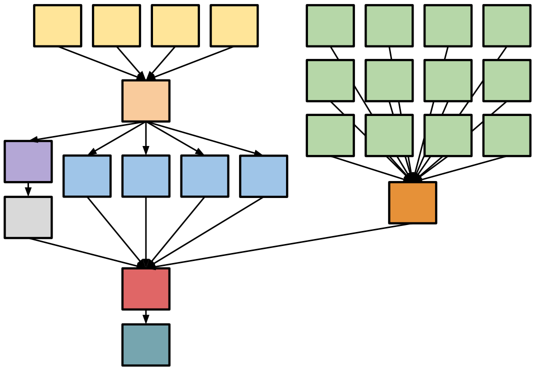

Figure 15. A simplified visualization of the SIPHT workflow.

SIPHT The SIPHT workflow [48] conducts a wide search for small untranslated RNAs (sRNAs) that regulates several processes such as secretion or virulence in bacteria. The kingdom-wide prediction and annotation of sRNA encoding genes involves a variety of individual programs that are executed in the proper order using Pegasus [8]. These involve the prediction of ρ-independent transcriptional terminators, BLAST (Basic Local Alignment Search Tools [49]) comparisons of the inter genetic regions of different replicons and the annotations of any sRNAs that are found. A simplified structure of the SIPHT workflow is shown in Figure 15.

Workflow	Number of Tasks	Average Data Size	Average Task Runtime
LIGO	800	5 MB	228s
Montage	300	3 MB	11s
CyberShake	700	148 MB	23s
Epigenomics	165	355 MB	2952s
SIPHT	1000	360 KB	180s

Table 1. Summary of the scientific workflows characteristics.

Table 1 shows the summary of the main **workflows characteristics**: number of tasks, average data size, and average task runtimes for the five workflows.

4.2. Experiment conditions

We adopt a trace-based simulation approach, where we extended our WorkflowSim [50] simulator with the fault tolerant clustering methods to simulate a controlled distributed environment. WorkflowSim is an open source workflow simulator that extends CloudSim [51] by providing support for task clustering, task scheduling, and resource provisioning at the workflow level. It has been recently used in multiple workflow study areas [50,3,52] and its correctness has been verified in [50].

The simulated computing platform is composed by 20 single homogeneous core virtual machines (worker nodes), which is the quota per user of some typical distributed environments such as Amazon EC2 [53] and FutureGrid [54]. Amazon EC2 is a commercial, public cloud that has been widely used in distributed computing, in particular for scientific workflows [11]. FutureGrid is a distributed, high-performance testbed that provides scientists with a set of computing resources to develop parallel, grid, and cloud applications. Each simulated virtual machine (VM) has 512MB of memory and the capacity to process 1,000 million instructions per second. The default network bandwidth is 15MB according to the real environment in FutureGrid from where our traces were collected. By default, we merge tasks at the same horizontal level into 20 clustered jobs initially, which is a simple selection of granularity control of the strength of task clustering. The study of granularity size has been done in [43], which shows such selection is acceptable.

We collected workflow execution traces [20,32] (including overhead and task runtime information) from real runs (executed on FutureGrid and Amazon EC2) of the scientific workflow applications described in Section 4.1. The traces are used to feed the Workflow Generator toolkit [55] to generate synthetic workflows. This allows us to perform simulations with several different configurations under controlled conditions. The toolkit uses the information gathered from actual scientific workflow executions to generate synthetic workflows resembling those used by real world scientific applications. The number of inputs to be processed, the number of tasks in the workflow, and their composition determine the structure of the generated workflow. Such an approach of traced based simulation allows us to utilize real traces and vary the system setting (i.e., the inter-arrival time of failures) and workflow (i.e., avg. task runtime) to fully explore the performance of our fault tolerant clustering algorithms.

Three sets of experiments are conducted. Experiment 1 evaluates the performance of our fault tolerant clustering methods (DR, VR, and SR) over a baseline execution (HC) that is not fault tolerant for the five workflows. The goal of the experiment is to identify conditions where each method works best and worst. In addition, we also evaluate the performance improvement under different θ_γ (the inter-arrival time of task failures). The range of θ_γ is chosen from 10x to 1x of the average task runtime such that the workflows do not run forever and we can visualize the performance difference better.

Experiment 2 evaluates the performance impact of the variation of the average task runtime per level (defined as the average of all the tasks per level) and the average system overheads per level for one scientific workflow application (CyberShake). In particular, we are interested in the performance of DR based on the results of Experiment 1 and we use $\theta_\gamma = 100$ since it has the maximum difference between the four methods. The original average task runtime of all the tasks of the CyberShake workflow is about 23 seconds as shown in Table 1. In this experiment, we multiply the average task runtime by a multiplier from 0.5 to 1.3. The scale parameter of the system overheads (θ_s) is 50 seconds originally based on our traces and we multiply the system overheads by a multiplier from 0.2 to 1.8.

Experiment 3 evaluates the performance of dynamic estimation and static estimation. In the static estimation process, we only use the prior knowledge to estimate the MLEs of θ_t, θ_s and θ_γ. In the dynamic estimation process, we also leverage the runtime data collected during the execution and update the MLEs respectively.

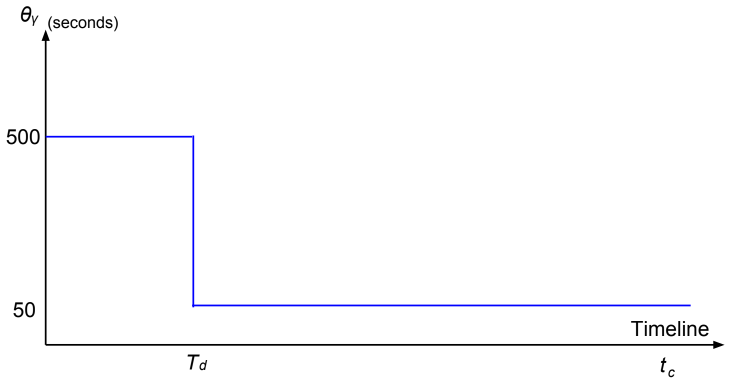

Figure 16. A Step Function of θ_γ. t_c is the current time and T_d is the moment θ_γ changes from 500 to 50 seconds

$$\theta_\gamma(t_c) = \begin{cases} 500 & \text{if } t_c \geq T_d \\ 50 & \text{if } 0 < t_c < T_d \end{cases} \tag{13}$$

In this experiment, we use two sets of θ_γ function. The first one is a step function, in which we decrease θ_γ from 500 seconds to 50 seconds at time T_d to simulate the scenario where there are more failures coming than expected. We evaluate the performance difference of dynamic estimation and static estimation while $1000 \leq T_d \leq 5000$ based on the estimation of the workflow makespan. The function is shown in Figure 16 and Equation 13 and t_c is the current time. Theoretically speaking, the later we change θ_γ, the less the reclustering is influenced by the estimation error and thus the less the makespan is. There is one special case when $T_d \to 0$, which means the prior knowledge is wrong at the very beginning. The second one is a pulse wave function, which the amplitude alternates at a steady frequency between fixed minimum (50 seconds) to maximum (500 seconds) values. The function is shown in Figure 17 and Equation 14. T_c is the period and τ is the duty cycle of the oscillator. It simulates a scenario where the failures follow a periodic pattern [56] that has been found in many failure traces obtained from production distributed systems. In our paper, we vary T_c from 1000 seconds to 10000 seconds based on the estimation of workflow makespan and τ from $0.1T_c$ to $0.5T_c$.

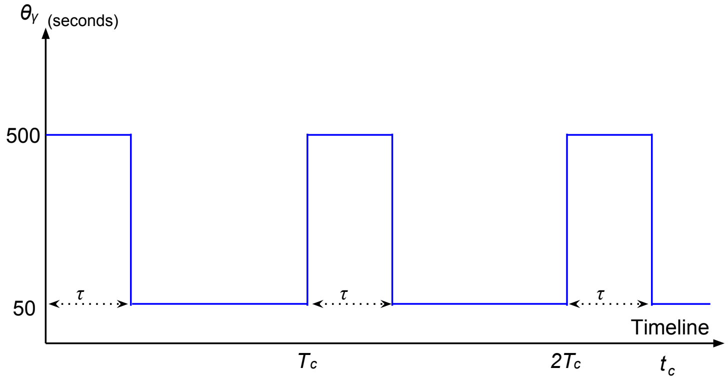

Figure 17. A Pulse Function of θ_γ. t_c is the current time and T_c is the period of the wave. τ is the width of the pulse.

$$\theta_\gamma(t_c) = \begin{cases} 500 & \text{if } 0 < t_c \le \tau \\ 50 & \text{if } \tau < t_c < T_c \end{cases} \tag{14}$$

Table 2 summarizes the clustering methods to be evaluated in our experiments. In our experiments, our algorithms take less than 10ms to do the reclustering for each job.

Abbreviation	Method
DR	Dynamic Reclustering
SR	Selective Reclustering
VR	Vertical Reclustering
HC	Horizontal Clustering

Table 2. Methods to Evaluate in Experiements

4.3. Results and discussion

Experiment 1 Figure 18, 19, 20, 21 and 22 show the performance of the four reclustering methods (DR, SR, VR and HC) with five workflows respectively. We draw conclusions:

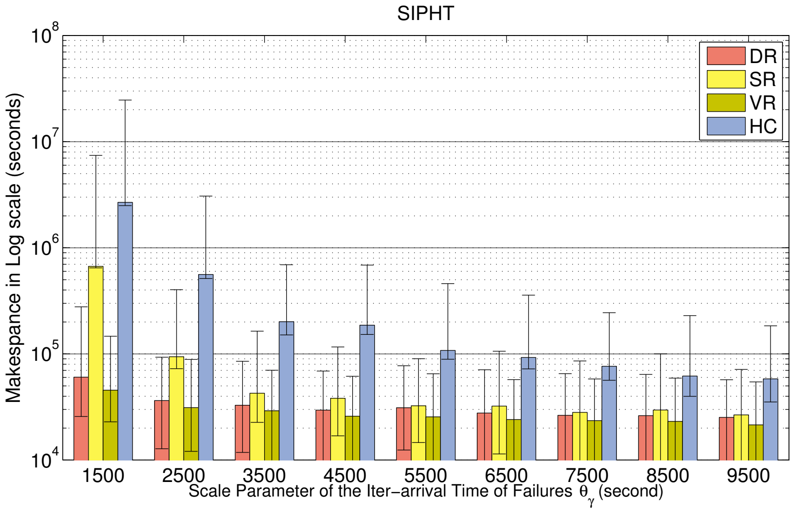

Figure 18. Experiment 1: SIPHT Workflow

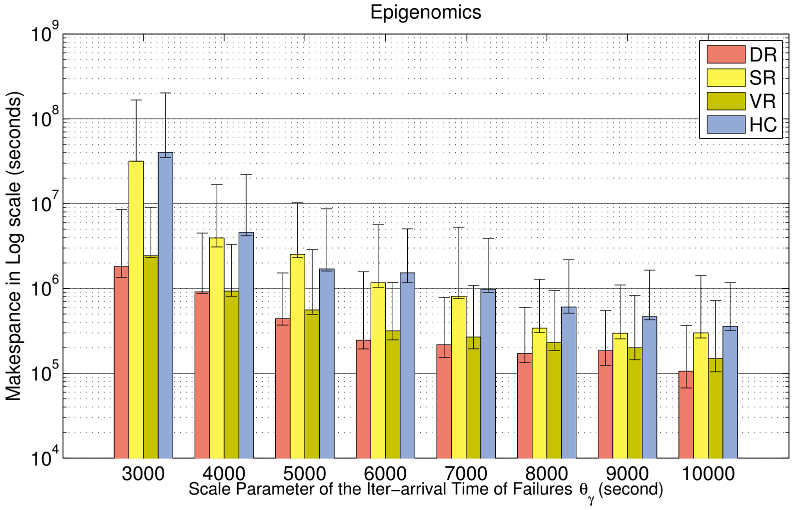

Figure 19. Experiment 1: Epigenomics Workflow

1). DR, SR and VR have significantly improved the makespan compared to HC in a large scale. By decreasing of the inter-arrival time (θ_γ) and consequently more failures are generated, the performance difference becomes more significant.

2). Among the three methods, DR and VR perform consistently better than SR, which fail to improve the makespan when θ_γ is small. The reason is the SR does not adjust k according to the occurrence of failures.

3). The performance of VR is highly related to the workflow structure and the average task runtime. For example, according to Figure 14 and Table 1, we know that the Epigenomics workflow has a long task runtime (around 50 minutes) and the pipeline

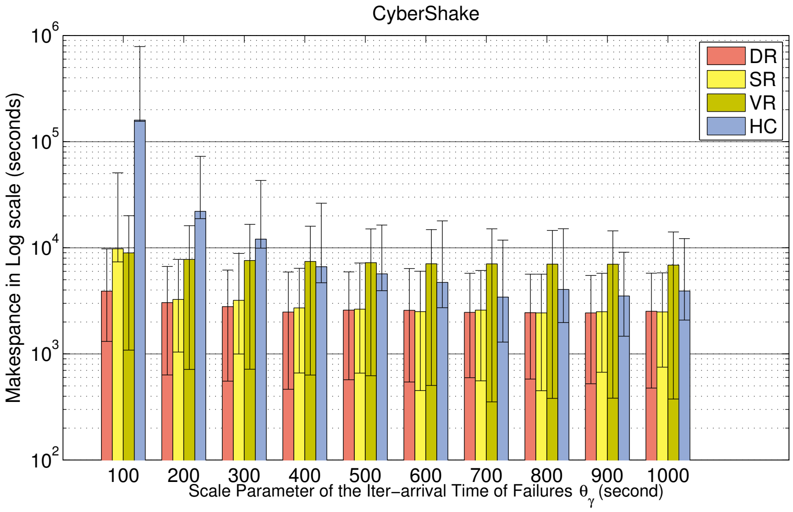

Figure 20. Experiment 1: CyberShake Workflow

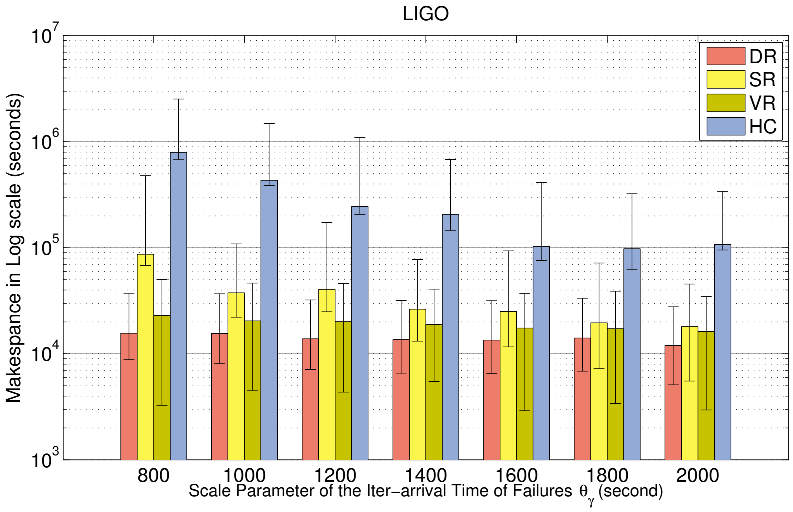

Figure 21. Experiment 1: LIGO Workflow

length is 4. It means vertical clustering creates really long jobs ($50 \times 4 = 200$ minutes, theoretically speaking) and thereby VR is more sensitive to the decrease of γ. As indicated in Figure 19, the makespan increases more significantly with the decrease of θ_γ than other workflows. In comparison, the CyberShake workflow does not leave much space for vertical clustering methods to improve since it does not have many pipelines as shown in Figure 13. In addition, the average task runtime of the CyberShake workflow is relatively short (around 23 seconds). Compared to horizontal methods such as HC, SR and DR, vertical clustering does not generate long jobs and thus the performance of VR is much less sensitive to θ_γ.

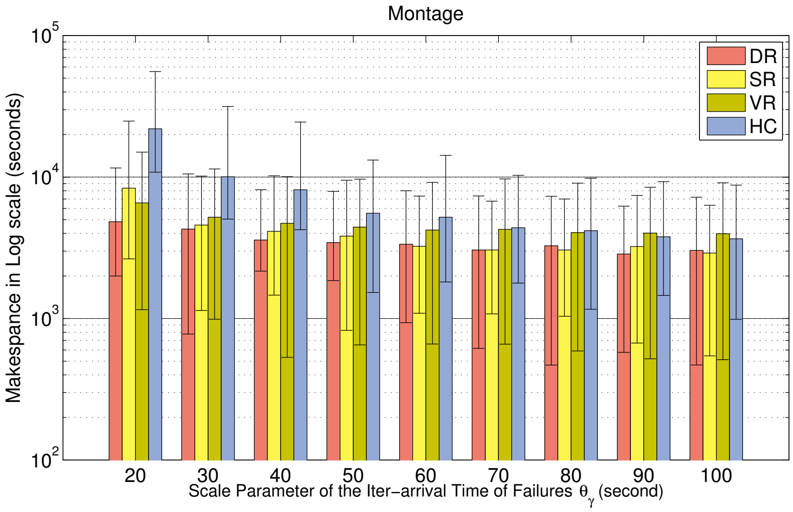

Figure 22. Experiment 1: Montage Workflow

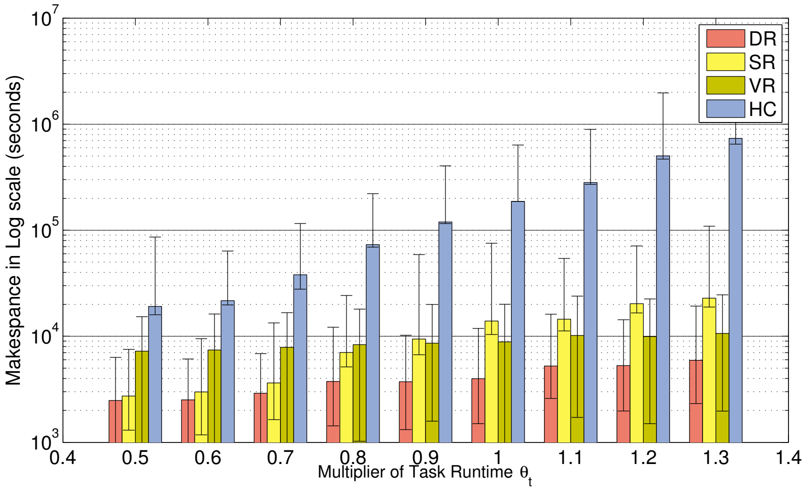

Figure 23. Experiment 2: Influence of Varying Task Runtime on Makespan (CyberShake)

Experiment 2 Figure 23 shows the performance of our methods with different multiplier of θ_t for the CyberShake workflow. We can see that with the increase of the multiplier, the makespan increases significantly (increase from a scale of 10^4 to $\sim 10^6$), particularly for HC. The reason is HC is not fault tolerant and it is sensitive to the increase of task runtime. While for DR, the reclustering process dynamically adjusts the clustering size based on the estimation of task runtime and thus the performance of DR is more stable.

Figure 24 shows the results with different multiplier of θ_s for the CyberShake workflow. Similarly, we can see that with the increase of the multiplier, the makespan in-

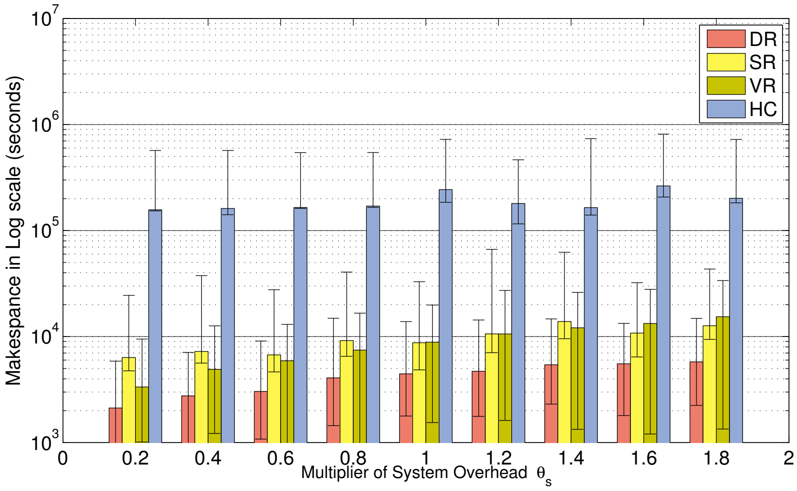

Figure 24. Experiment 2: Influence of Varying System Overhead on Makespan (CyberShake)

creases for all the methods and DR performs best. However, the increase is less significant than that in Figure 23. The reason is we may have multiple tasks in a clustered job but only one system overhead per job.

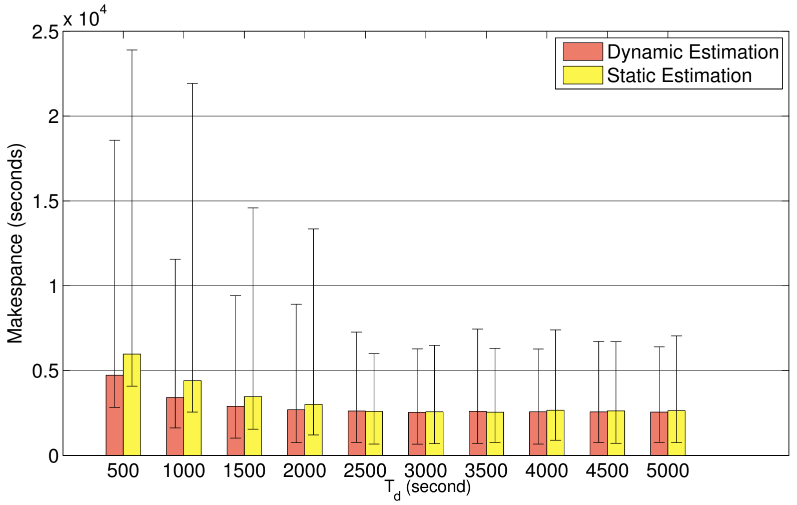

Figure 25. Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Step Function)

Experiment 3 Figure 25 further evaluates the performance of the dynamic estimation and static estimation for the CyberShake workflow. The reclustering method used in this experiment is DR since it performs best in the last two experiments. In the experiment, we use a step signal and change the inter-arrival time of failures (θ_γ) from 500 seconds to 50 seconds at T_d. We can see that: 1). with the increase of T_d, both makespan decrease since the change of θ_γ has less influence on the makespan and there is a lower failure rate on average; 2). Dynamic estimation improves the makespan by up to 22.4% compared to the static estimation. The reason is the dynamic estimation is able to update the MLEs of the θ_γ and decrease the clustering size while the static estimation is not.

For the pulse signal, we use $\tau = 0.1T_c, 0.3T_c, 0.5T_c$. Figure 26, 27 and 28 show the performance difference of dynamic estimation and static estimation respectively. When $\tau = 0.1T_c$, DR with dynamic estimation improves the makespan by up to 25.7% compared to case with static estimation. When $\tau = 0.3T_c$, the performance difference between dynamic estimation and static estimation is up to 27.3%. We can also see when T_c is small (i.e., $T_c = 1000$), the performance difference is not significant since the inter-arrival time of failures changes frequently and the dynamic estimation process is not able to update swiftly. While when T_c is 10000, the performance difference is not significant either since the period is too long and the workflow has completed successfully. When $\tau = 0.5T_c$, the performance different between dynamic estimation and static estimation is up to 9.1%, since the high θ_γ and the low θ_γ have equal influence on the failure occurrence.

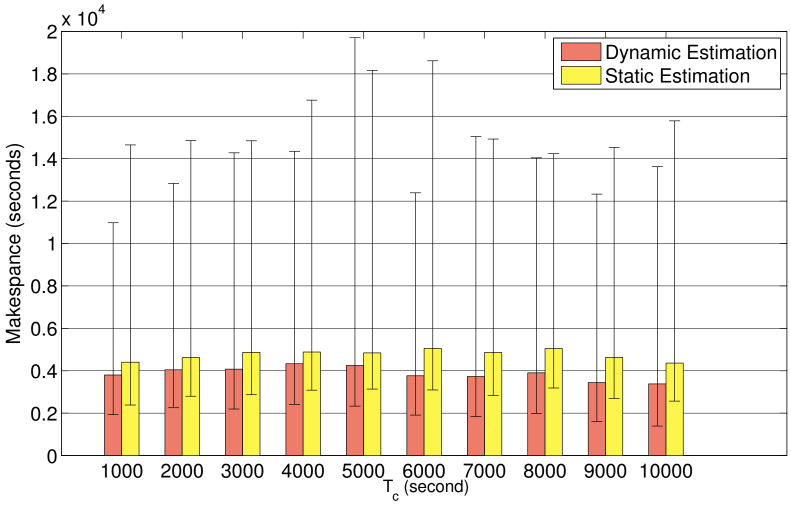

Figure 26. Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Pulse Function ($\tau = 0.1T_c$))

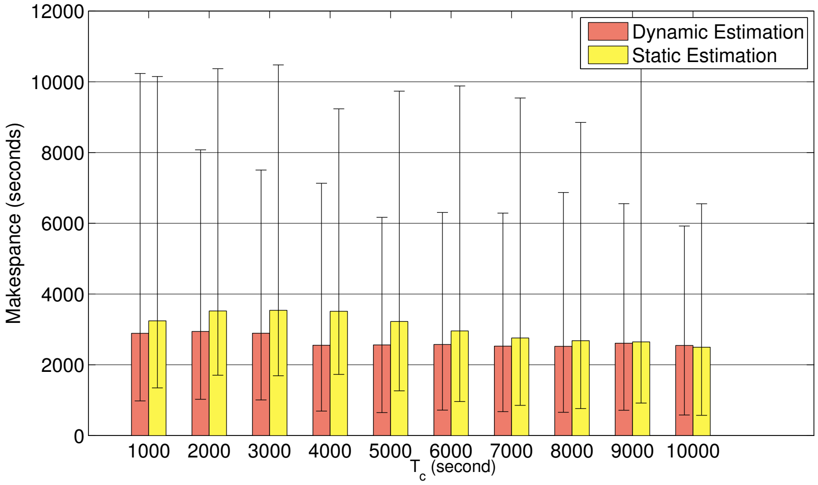

Figure 27. Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Pulse Function ($\tau = 0.3T_c$))

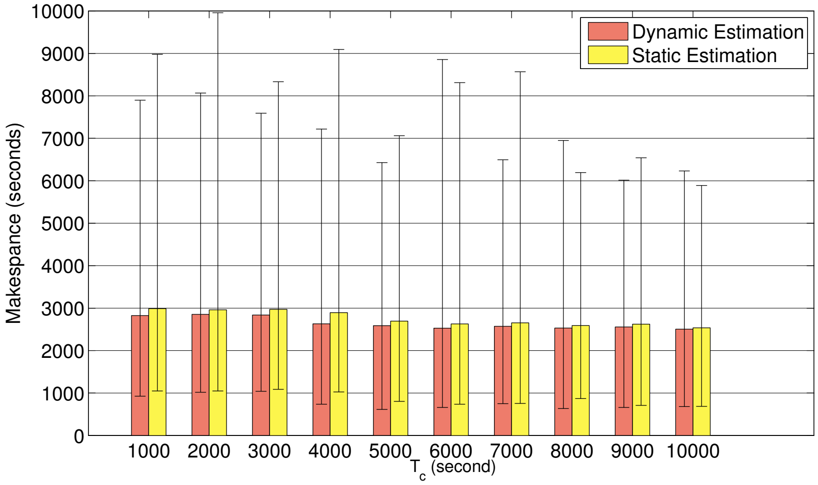

Figure 28. Experiment 3: Static Estimation vs. Dynamic Estimation (CyberShake, Pulse Function ($\tau = 0.5T_c$))

5. Conclusion and Future Work

In this paper, we model transient failures in a distributed environment and propose three dynamic clustering methods to improve the fault tolerance of task clustering. We apply our methods to five widely used scientific workflows. From our experiments, we conclude that the three proposed methods improve the makespan significantly compared to an existing algorithm widely used in workflow management systems. In particular, our Dynamic Reclustering method performs best among the three methods since it can adjust the clustering size based on the Maximum Likelihood Estimation of task runtime, system overheads and the inter-arrival time of failures. Our Vertical Reclustering method improves the performance significantly for workflows that have a short task runtime. Our dynamic estimation using on-going data collected from the workflow execution can further improve the fault tolerance in a dynamic environment where the inter-arrival time of failures is fluctuant.

In this paper, we only discuss the fault tolerant clustering and apply it to a homogeneous environment. In the future, we aim to combine our work with fault tolerant scheduling in heterogeneous environments, i.e, a scheduling algorithm that avoids mapping clustered jobs to failure-prone nodes. We are also interested to combine vertical clustering methods with horizontal clustering methods. For example, we can perform vertical clustering either before or after horizontal clustering, which we believe would bring different performance improvement. As shown in our experiments, our dynamic estimation works well under some constraints, which encourages us to improve its performance further. For example, we may weight data based on the time it was collected, the closer the more weight on it.

6. ACKNOWLEDGMENT

This work is supported by NFS under grant number IIS-0905032. We thank Gideon Juve, Karan Vahi, Mats Rynge and Rajiv Mayani for their help.

References

[1] W. Chen, E. Deelman, R. Sakellariou, Imbalance optimization in scientific workflows, in: Proceedings of the 27th international ACM conference on International conference on supercomputing, ICS ’13, 2013, pp. 461–462.

[2] G. Singh, M. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, G. Mehta, Workflow task clustering for best effort systems with pegasus, in: 15th ACM Mardi Gras Conference, 2008.

[3] W. Chen, E. Deelman, Fault tolerant clustering in scientific workflows, in: IEEE Eighth World Congress on Services (SERVICES), 2012, pp. 9–16.

[4] K. Maheshwari, A. Espinosa, D. S. Katz, M. Wilde, Z. Zhang, I. Foster, S. Callaghan, P. Maechling, Job and data clustering for aggregate use of multiple production cyberinfrastructures, in: Proceedings of the fifth international workshop on Data-Intensive Distributed Computing Date, DIDC ’12, ACM, New York, NY, USA, 2012, pp. 3–12.

[5] R. Ferreira da Silva, T. Glatard, F. Desprez, On-line, non-clairvoyant optimization of workflow activity granularity on grids, in: Euro-Par 2013 Parallel Processing, Vol. 8097 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 255–266.

[6] W. Chen, E. Deelman, Integration of workflow partitioning and resource provisioning, in: The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid ’12), 2012.

[7] J. Bresnahan, T. Freeman, D. LaBissoniere, K. Keahey, Managing appliance launches in infrastructure clouds, in: Teragrid Conference, 2011.

[8] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, M. Livny, Pegasus: Mapping scientific workflows onto the grid, in: Across Grid Conference, 2004.

[9] R. Duan, R. Prodan, T. Fahringer, Run-time optimisation of grid workflow applications, in: 7th IEEE/ACM International Conference on Grid Computing, 2006, pp. 33–40.

[10] E. Deelman, G. Singh, M. Livny, B. Berriman, J. Good, The cost of doing science on the cloud: The montage example, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008.

[11] G. B. Berriman, G. Juve, E. Deelman, M. Regelson, P. Plavchan, The application of cloud computing to astronomy: A study of cost and performance, in: Workshop on e-Science challenges in Astronomy and Astrophysics, 2010.

[12] Y. Zhang, M. S. Squillante, Performance implications of failures in large-scale cluster scheduling, in: The 10th Workshop on Job Scheduling Strategies for Parallel Processing, 2004.

[13] D. Tang, R. K. Iyer, S. S. Subramani, Failure analysis and modeling of a vaxcluster system, in: Proceedings of the International Symposium on Fault-tolerant computing, 1990.

[14] B. Schroeder, G. A. Gibson, A large-scale study of failures in high-performance computing systems, in: Proceedings of the International Conference on Dependable Systems and Networks, 2006.

[15] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, Y. Zhang, Failure data analysis of a large-scale heterogeneous server environment, in: Proceedings of the International Conference on Dependable Systems and Networks, 2004.

[16] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, et al., Askalon: A development and grid computing environment for scientific workflows, in: Workflows for e-Science, Springer, 2007, pp. 450–471.

[17] B. Schuller, B. Demuth, H. Mix, K. Rasch, M. Romberg, S. Sild, U. Maran, P. Bała, E. Del Grosso, M. Casalegno, et al., Chemomentum-unicore 6 based infrastructure for complex applications in science and technology, in: Euro-Par 2007 Workshops: Parallel Processing, Springer, 2008, pp. 82–93.

[18] K. Plankensteiner, R. Prodan, T. Fahringer, A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact, in: Fifth IEEE International Conference on e-Science (e-Science '09), 2009, pp. 313–320.

[19] R. F. da Silva, G. Juve, E. Deelman, T. Glatard, F. Desprez, D. Thain, B. Tovar, M. Livny, Toward fine-grained online task characteristics estimation in scientific workflows, in: Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science, ACM, 2013, pp. 58–67.

[20] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Vol. 29, 2013, pp. 682 – 692, special Section: Recent Developments in High Performance Computing and Security.

[21] T. Samak, D. Gunter, M. Goode, E. Deelman, G. Mehta, F. Silva, K. Vahi, Failure prediction and localization in large scientific workflows, in: The 6th Workshop on Workflows in Suppporting of Large-Scale Science, 2011.

[22] D. Oppenheimer, A. Ganapathi, D. A. Patterson, Why do internet services fail and what can be done about it?, Computer Science Division, University of California, 2002.

[23] S. R. McConnel, D. P. Siewiorek, M. M. Tsao, The measurement and analysis of transient errors in digital computer systems, in: Proc. 9th Int. Symp. Fault-Tolerant Computing, 1979, pp. 67–70.

[24] X.-H. Sun, M. Wu, Grid harvest service: a system for long-term, application-level task scheduling, in: International Parallel and Distributed Processing Symposium (IPDPS), 2003, p. 8.

[25] A. Iosup, O. Sonmez, S. Anoep, D. Epema, The performance of bags-of-tasks in large-scale distributed systems, in: Proceedings of the 17th International Symposium on High Performance Distributed Computing, HPDC '08, ACM, New York, NY, USA, 2008, pp. 97–108.

[26] K. Plankensteiner, R. Prodan, T. Fahringer, A. Kertész, P. Kacsuk, Fault detection, prevention and recovery in current grid workflow systems, in: Grid and Services Evolution, Springer, 2009, pp. 1–13.

[27] D. Laforenza, R. Lombardo, M. Scarpellini, M. Serrano, F. Silvestri, P. Faccioli, Biological experiments on the grid: A novel workflow management platform, in: 20th IEEE International Symposium on Computer-Based Medical Systems (CBMS'07), IEEE, 2007, pp. 489–494.

[28] I. Taylor, M. Shields, I. Wang, A. Harrison, The triana workflow environment: Architecture and applications, in: Workflows for e-Science, Springer, 2007, pp. 320–339.

[29] P.-O. Ostberg, E. Elmroth, Mediation of service overhead in service-oriented grid architectures, in: 12th IEEE/ACM International Conference on Grid Computing (GRID), 2011, pp. 9–18.

[30] R. Prodan, T. Fabringer, Overhead analysis of scientific workflows in grid environments, in: IEEE Transactions in Parallel and Distributed System, Vol. 19, 2008.

[31] C. Stratan, A. Iosup, D. H. Epema, A performance study of grid workflow engines, in: 9th IEEE/ACM International Conference on Grid Computing, IEEE, 2008, pp. 25–32.

[32] W. Chen, E. Deelman, Workflow overhead analysis and optimizations, in: The 6th Workshop on Workflows in Support of Large-Scale Science, 2011.

[33] R. Duan, F. Nadeem, J. Wang, Y. Zhang, R. Prodan, T. Fahringer, A hybrid intelligent method for performance modeling and prediction of workflow activities in grids, in: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09), 2009, pp. 339–347.

[34] J. Cao, S. Jarvis, D. Spooner, J. Turner, D. Kerbyson, G. Nudd, Performance prediction technology for agent-based resource management in grid environments, in: IEEE International Parallel and Distributed

Processing Symposium (IPDPS 2002), 2002, p. 14.

[35] H. Li, D. Groep, L. Wolters, Efficient response time predictions by exploiting application and resource state similarities, in: The 6th IEEE/ACM International Workshop on Grid Computing, 2005, p. 8.

[36] N. Muthuvelu, J. Liu, N. L. Soe, S. Venugopal, A. Sulistio, R. Buyya, A dynamic job grouping-based scheduling for deploying applications with fine-grained tasks on global grids, in: Proceedings of the 2005 Australasian workshop on Grid computing and e-research, 2005.

[37] N. Muthuvelu, I. Chai, C. Eswaran, An adaptive and parameterized job grouping algorithm for scheduling grid jobs, in: 10th International Conference on Advanced Communication Technology (ICACT 2008), Vol. 2, 2008, pp. 975 –980.

[38] N. Muthuvelu, I. Chai, E. Chikkannan, R. Buyya, On-line task granularity adaptation for dynamic grid applications, in: Algorithms and Architectures for Parallel Processing, Vol. 6081 of Lecture Notes in Computer Science, 2010, pp. 266–277.

[39] W. K. Ng, T. Ang, T. Ling, C. Liew, Scheduling framework for bandwidth-aware job grouping-based scheduling in grid computing, Malaysian Journal of Computer Science 19 (2) (2006) 117–126.

[40] T. L. L. P. T.F. Ang, W.K. Ng, C. Liew, A bandwidth-aware job grouping-based scheduling on grid environment, Information Technology Journal (8) (2009) 372–377.

[41] Q. Liu, Y. Liao, Grouping-based fine-grained job scheduling in grid computing, in: First International Workshop on Education Technology and Computer Science, 2009.

[42] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, P. Li, Taverna: a tool for the composition and enactment of bioinformatics workflows, Bioinformatics 20 (17) (2004) 3045–3054.

[43] W. Chen, R. Ferreira da Silva, E. Deelman, R. Sakellariou, Balanced task clustering in scientific workflows, in: 2013 IEEE 9th International Conference on eScience, 2013, pp. 188–195.

[44] Laser Interferometer Gravitational Wave Observatory (LIGO), `http://www.ligo.caltech.edu`.

[45] G. B. Berriman, E. Deelman, J. C. Good, J. C. Jacob, D. S. Katz, C. Kesselman, A. C. Laity, T. A. Prince, G. Singh, M. Su, Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand, in: SPIE Conference on Astronomical Telescopes and Instrumentation, 2004.

[46] R. Graves, T. Jordan, S. Callaghan, E. Deelman, E. Field, G. Juve, C. Kesselman, P. Maechling, G. Mehta, K. Milner, D. Okaya, P. Small, K. Vahi, CyberShake: A Physics-Based Seismic Hazard Model for Southern California, Pure and Applied Geophysics 168 (3-4) (2010) 367–381.

[47] USC Epigenome Center, `http://epigenome.usc.edu`.

[48] SIPHT, `http://pegasus.isi.edu/applications/sipht`.

[49] Basic local alignment search tools, `http://blast.ncbi.nlm.nih.gov/Blast.cgi`.

[50] W. Chen, E. Deelman, Workflowsim: A toolkit for simulating scientific workflows in distributed environments, in: The 8th IEEE International Conference on eScience, 2012.

[51] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, R. Buyya, CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Software: Practice and Experience 41 (1) (2011) 23–50.

[52] F. Jrad, J. Tao, A. Streit, A broker-based framework for multi-cloud workflows, in: Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds, ACM, 2013, pp. 61–68.

[53] Amazon.com, Inc., Amazon Web Services, `http://aws.amazon.com`.

[54] FutureGrid, `http://futuregrid.org/`.

[55] Workflow Generator, `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`.

[56] N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, D. Epema, Analysis and modeling of time-correlated failures in large-scale distributed systems, in: 2010 11th IEEE/ACM International Conference on Grid Computing (GRID), IEEE, 2010, pp. 65–72.