Adaptive Reputation-Based Scheduling on Unreliable Distributed Infrastructures

Jason Sonnek†, Abhishek Chandra and Jon Weissman

†Sandia National Laboratories — Albuquerque, NM 87185

Department of Computer Sc. and Engg. — University of Minnesota, Minneapolis, MN 55455

Email: {sonnek, chandra, jon}@cs.umn.edu

Abstract

This paper addresses the inherent unreliability and instability of worker nodes in large-scale donation-based distributed infrastructures such as P2P and Grid systems. We present adaptive scheduling techniques that can mitigate this uncertainty and significantly outperform current approaches. In this work, we consider nodes that execute tasks via donated computational resources and may behave erratically or maliciously. We present a model in which reliability is not a binary property but a statistical one based on a node's prior performance and behavior. We use this model to construct several reputation-based scheduling algorithms that employ estimated reliability ratings of worker nodes for efficient task allocation. Our scheduling algorithms are designed to adapt to changing system conditions as well as non-stationary node reliability. Through simulation we demonstrate that our algorithms can significantly improve throughput, while maintaining a very high success rate of task completion. Our results suggest that reputation-based scheduling can handle wide variety of worker populations, including non-stationary behavior, with overhead that scales well with system size. We also show that our adaptation mechanism allows the application designer fine-grain control over desired performance metrics.

Index Terms

Distributed Scheduling, Reputation, Reliability, Adaptive, Grids

I. Introduction

Recently, several distributed infrastructures, including peer-to-peer networks and donation Grids, have been proposed to host large-scale wide-area applications ranging from file sharing/file storage to high performance scientific computing [1]–[6]. Despite the attractive features of these platforms (scalability, low cost, reduced cost of ownership, and resilience to local failures), widespread deployment of such systems and applications has been elusive. A key problem is the inherent unreliability of these systems: nodes may leave and join unexpectedly, perform unpredictably due to resource sharing at the node and network level, and behave erratically or maliciously. This paper presents a design and analysis of techniques to cope with the inherent unreliability of nodes that execute tasks via donated computational resources.

We present a model in which reliability is not a binary property but a statistical one based on a node's prior performance and behavior. Such a statistical model is important for two main reasons. First, a node's behavior could change with time and hence nodes cannot be classified as being purely reliable or unreliable always. Second, representing reliability as a statistical property allows us to incorporate the uncertainty inherent in the system's knowledge of individual nodes' reliability. We adopt a reliability model based on the accumulation of the direct observation of node behavior over prior task executions. An example of such an environment is BOINC [4], or its forerunner SETI@home [6], in which a server distributes tasks to worker nodes and collects results. Since nodes are not reliable, the server generally cannot be certain that the results returned by any given worker are valid unless application-specific verifiers are provided. Many factors may contribute to the unreliability of a node. It has been shown [7] that cheating has been a considerable problem in the SETI@home project. However, it is also possible that nodes have incorrectly configured software, are hacked, have poor connections to the server, or are highly loaded and cannot return timely results.

We speculate that when excess resources become a visible standard commodity or utility [1], [8] , cheating or hacking nodes will become even more prevalent due to economic incentives. In addition, it seems likely that as distributed systems become larger and more widely dispersed, reliability will also decrease due to more failure-prone components and increased exposure to malicious agents and viruses. To deal with uncertainty in the absence of inexpensive verifiers, outsourced computations can be redundantly scheduled to a number of nodes. If we assume that

the space of feasible (but not necessarily verifiable) results is sufficiently large, it is very likely that a result returned by a majority of workers will be valid if node collusion has not occurred. Such a majority result could then be treated as the "correct" result of the computation.

A major drawback of using redundancy is that it may reduce the amount of useful work performed. The degree of redundancy is an important parameter: a small degree of replication could decrease the likelihood that the server will receive a verifiable result. On the other hand, a large degree of replication could result in unnecessary duplication of work by multiple resources. Systems like BOINC rely on the application writer to specify this value for each task. Since the reliability of workers in a distributed environment may be uncertain, it is likely that any statically-chosen redundancy value will reduce the effectiveness of the system.

To overcome this problem, we propose techniques to determine the degree of redundancy based on the estimated reliability of the workers. Intuitively, a smaller degree of replication should be possible if the allocated nodes are collectively more reliable. Using a simple reputation system [9], it is possible to determine the likelihood that a given worker will return a correct and timely result with fairly high accuracy. Unlike other systems which have studied the concepts separately, we incorporate metrics of correctness as well as timeliness to generalize the notion of trust to that of *reliability*.

Using individual worker reliability estimates, we introduce an efficient technique for computing a lower-bound on the likelihood that a group of workers will return a majority of correct and timely results. These group reliability ratings can be used by the system to intelligently schedule tasks to workers, such that the throughput of the system is improved, while still maintaining the server's ability to distinguish fraudulent results from valid ones.

Applying these techniques in practice introduces a number of challenges. First, the system must be able to learn the reliability of individual workers. A number of different reputation systems have been proposed for this purpose [10]–[15], although selecting the right one is dependent on the characteristics of the environment in which it will be deployed. Second, given these reliability ratings, the system needs an algorithm or heuristic to determine how to match groups of workers to tasks. Since it is likely that the best scheduling technique will be dependent on the environment, we propose a set of algorithms that are tuned to the characteristics of typical environments. Finally, the environment may be extremely dynamic, and the underlying scheduling mechanisms must be highly adaptive.

We consider several different algorithms which can be used to guide scheduling decisions on the basis of statistical reliability ratings associated with groups of workers. We also present an adaptive algorithm which adjusts scheduling parameters to match conditions in the system. This algorithm provides a "knob" for tuning scheduling decisions in terms of metrics such as success rate and throughput which are familiar to application designers.

Finally, we compare the throughput and computational overhead of each of these techniques through simulation of a BOINC-like distributed computing infrastructure. Our results indicate that reputation-based scheduling can significantly improve the throughput of the system for worker populations modeling several real-world scenarios, including non-stationary behavior, with overhead that scales well with system size.

II. Background and Related Work

A. Distributed Computing Infrastructures

Numerous computing infrastructures have been designed to utilize idle distributed resources. These systems can be loosely categorized into two groups: those that utilize resources under administrative control, such as Globus [16] and Condor [3], and those that rely on unsupervised donated resources such as SETI@Home [6] and Folding@Home [17]. In this paper, we mainly focus on the latter, as these environments are much more susceptible to unreliability.

The @Home applications [6], [17] and their generalization, BOINC [4], are instances of a growing number of systems which utilize donated computing cycles to solve massive scientific problems. BOINC provides application designers with a middleware that can be used to design and deploy systems in which a master task server assigns computational tasks to a pool of donated computing resources. In contrast to BOINC, several unstructured cycle-sharing platforms have been proposed [5], [18], [19] in which nodes can act as both a client and a server. These platforms facilitate the formation of ad hoc communities for solving large-scale computing problems.

B. Dealing with Unreliability

Dealing with unreliability is a core design challenge in any distributed system and many techniques have been proposed in the literature. Redundant task allocation combined with voting, as used in Byzantine fault-tolerant (BFT) systems [20], is popular due to its general applicability. This approach is also used by most BOINC [4] applications to verify the results of outsourced

computations: if a majority of the workers assigned a task return the same result, then the result is deemed valid.

Since task replication could result in lower resource utilization, some techniques have been proposed to verify results for tasks allocated to a single resource. Golle and Mirnov [21] present a verification technique that inserts pre-computed images of special spot-checks called "ringers" into distributed tasks to verify results returned by a worker and identify cheaters. This technique can be used only for verifying computations that exhibit a *one-way* property, and thus is not applicable for general computations. Another verification technique [22], [23] employs pre-computed tasks called 'quizzes' that are embedded into a batch of (otherwise indistinguishable) tasks allocated to a worker. When the task server receives a batch of results from a worker, it assumes the results for the real tasks to be correct if the results for all of the quiz tasks are valid. While not dependent on one-way functions, this technique still requires pre-computation of certain tasks, which may be non-trivial or infeasible in many scenarios.

C. Reputation-Based Scheduling

Reputation systems [24] are commonly applied in peer-to-peer networks to gauge the reliability of nodes [11], [12], [15], [25]. Trust or reputation systems are a general technique for predicting the behavior of distributed entities based on past interactions with these entities.

The concept of trust-aware resource management for the Grid was proposed in [14], where a technique is presented for computing trust ratings in a Grid using a weighted combination of past experience and reputation. GridEigenTrust [13] combines this trust-computation technique with the EigenTrust reputation system [12] to provide a mechanism for rating resources in a Grid. This work presents an architecture for managing reputation ratings in a Grid, and proposes using these ratings to perform reputation-based resource selection. However, it does not provide any specific algorithms for reputation-based scheduling. Zhao and Lo [23] propose augmenting peer-to-peer cycle sharing systems with a reputation system to reduce the degree of replication required to verify results. However, their work makes several assumptions: nodes are either strictly trustworthy or untrustworthy, the number of nodes is large relative to the workload which allows nodes to be discarded if untrustworthy, and node behavior is fixed (for the results presented). These assumptions may not often hold in practical scenarios. The scheduling algorithms proposed by Zhao [23] and Sarmenta [22] are explicitly designed to deal

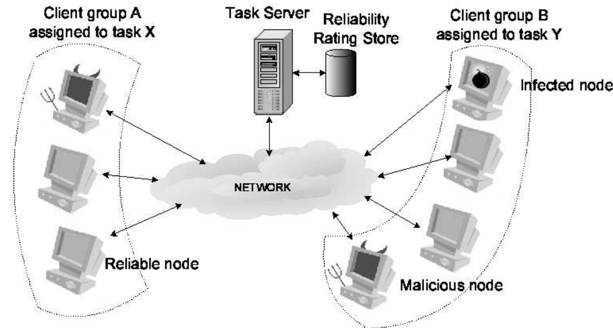


Fig. 1. The system model: a server maintains a reliability rating store and uses the ratings to assign tasks to groups of workers.

with node collusion. In contrast, we assume that each worker acts independently. Song [26] recently proposed trust-based scheduling algorithms designed to avoid compromised resources, as opposed to handling explicitly malicious resources.

Overall, most existing reputation-based scheduling schemes have focused on correctness as the primary metric, and have dealt mainly with binary trust values. The unique elements of our approach include a more general statistical representation of reliability that includes timeliness as well as correctness, and the use of this metric to improve application and system performance.

III. System Model

A. Computational Model

Our distributed computing model consists of a central server that assigns computational tasks to a set of worker nodes as illustrated in Figure 1. The worker nodes in this computation model are not centrally-controlled, and could be participating for various reasons. For instance, they may be donating their idle resources voluntarily (e.g.: PlanetLab [2]), or they may be providing their resources in return for some incentive, such as monetary remuneration [1], [27], credit [4], [6], or use of other nodes' resources in return [28], [29]. Our system model does not make any assumptions about the incentive scheme for worker participation or the workload generation methodology: the computation tasks could either be pre-generated on the server by the application, or they may be submitted by users accessing a common service. We assume that the set of tasks that need to be computed by the available set of worker nodes is large enough to keep all workers busy for the duration of the application.

B. Reliability Model

Since the participation of worker nodes is voluntary and outside the server's control, workers may not return correct results in a timely manner for several reasons. First, a node may be overloaded or behind a slow connection, resulting in slow response. Another reason may be that a node is misconfigured, hacked, or infected by a virus, resulting in incorrect computation. Finally, a node may be malicious (deliberately trying to disrupt a computation) or cheating (to gain an advantage in a remuneration scheme, such as gaining extra credit [7]), thus returning wrong results. We model such unreliable behavior by assigning to each worker a probability of returning a correct response within a "reasonable" time frame. This probability need not be fixed, and could change with time. For instance, nodes may go offline and come back up again, or some malicious nodes may change their behavior with time—returning correct results for a while to improve their reputation and then deliberately injecting bad results into the system. When modeling these unreliable workers, we assume that each worker acts independently, and that there is no collusion between them. This assumption is consistent with behavior observed in popular outsourced computing systems, in which individual cheating has been observed [7], but collusion has not [1].

C. Redundant Computation and Result Verification

A key consideration in our model is that the server may not have an efficient way of independently verifying each worker response for correctness. While several techniques [21], [30], [31] have been proposed to verify the correctness of results, these techniques are application-specific and are not applicable to general computational scenarios. Results of several computational problems may not even be verifiable by the server without performing the computation itself.

In our system model, we employ a verification technique based on *redundant computation* coupled with *voting*. This technique is adopted by several general computing systems such as BOINC [4]. Under this verification technique, each task is redundantly assigned to a set of worker nodes. Once the workers respond, the server conducts a "vote" among the returned results. If a quorum of workers agrees on a result, the server treats that result to be correct.

[1]Based on private communication with Dr. David Anderson, creator of SETI@Home, there has been no observed evidence of collusion in SETI@Home.

In the absence of a quorum after voting, the task is rescheduled. While the quorum size could be application-dependent, *majority* is typically used to determine the correct answer. Note that such a voting-based verification scheme does not require any application-specific support or knowledge.

D. Definitions and Assumptions

Definition 1: **Task (τ_j):** A task is defined as a self-contained computational activity that can be carried out by a worker node. Upon completion, each task generates a well-defined result that is returned to the server.
A task would typically correspond to an independent unit of a larger computation. For example, a task may correspond to computing the determinant of a submatrix, and the result of the task would be the value of the determinant. Another example of a task could be to match a DNA sequence against a subset of gene sequences from a genetic database. In this case, the result could be the best matching gene and the similarity score.

Definition 2: **Solution Space (Σ):** The solution space of a task is the set of potential result values that can be returned for the task.
For instance, a task whose answer is Boolean has a two-element solution space, $\Sigma = \{true, false\}$. On the other hand, a task whose answer is drawn from the set of integers has an infinite solution space, $\Sigma = I$. We assume that the solution space for the tasks in our model is of sufficiently large cardinality, so that it is unlikely that two workers will independently return the same wrong result.

Definition 3: **Reliability (r_i):** Reliability of a worker i is defined as the probability that the worker returns a correct result within a (system-defined) time period.
Note that reliability is not a binary property—a node could return the correct result some of the time, and a wrong result at other times. Moreover, the reliability property of a worker could also change with time (e.g.: due to outages, fluctuating load, malicious node behavior, etc.).

Definition 4: **Redundancy Group** [1] **(G_j):** Redundancy group for a task τ_j is defined as the group of worker nodes assigned to compute the task.
In most existing systems, the size of each redundancy group is typically set to a fixed static value selected by the application designer or the system administrator. This value may be determined

[1] In the rest of the paper, we would refer to a redundancy group simply as a *group* unless required to avoid confusion.

empirically, although often it is simply based on a rule of thumb. In our system model, the redundancy factor for each group can be different and dynamically determined, and is dependent on the reliability of the group's constituent worker nodes.

Definition 5: **Quorum:** We say that a group G_j has reached quorum if some number of worker nodes, which may be fixed or dependent on the group size, return the same result.
In our system model, we say a group has reached quorum if a majority of the workers return the same result. In general, the quorum size could be dependent on the cardinality of the solution space, for instance, a binary solution space would likely require a larger quorum.

Definition 6: **Likelihood-of-Correctness (λ_j):** Likelihood-of-correctness for a group G_j is defined as the probability that the group would return a correct result based on majority voting. The likelihood-of-correctness λ_j for a group represents the collective reliability of the group. This value is dependent on the individual reliability values of the constituent nodes of the group. We will see in the next section how this value can be computed for groups using the reliability of individual workers.

IV. Reputation-based Scheduling

We now present a reputation-based scheduling algorithm for distributing the server workload among the worker nodes. This algorithm employs reliability ratings of individual worker nodes for task assignment in order to improve the overall throughput and success rate of task completions. This reputation-based task scheduling algorithm consists of the following steps:

- Estimating reliability ratings of individual worker nodes.
- Using the estimated worker reliability ratings to compute the likelihood-of-correctness (LOC) of possible groups.
- Grouping workers for task assignment based on LOC estimates to maximize the throughput and success rate of task completions.

We will now describe each of these steps in more detail, discussing the various techniques and algorithms employed in each case.

A. Estimating Reliability Ratings

We use a *reputation system* to estimate the reliability ratings of individual worker nodes. These reliability ratings are learned over time based on the results returned by the workers to

the server. All workers report their results to a centralized server, so a local reputation system can be employed. We estimate a worker's reliability $r_i(t)$, at a given time t, as follows:

$$r_i(t) = \frac{n_i(t) + 1}{N_i(t) + 2},$$

where $n_i(t)$ and $N_i(t)$ are respectively the number of correct responses generated and the total number of tasks attempted by the worker by time t. By this formula, the reliability rating of a worker is initialized to $\frac{1}{2}$, corresponding to having no knowledge about its actual reliability. The rating of each worker is updated each time it is assigned a task, based on the response it returns (a missing or late response is treated as incorrect). While we assume that the server learns the reliability ratings using its own observations, it is possible to use a peer-to-peer reputation reporting system [11], [12] to improve the accuracy of these ratings or reduce the time to learn them, if the system has multiple servers interacting independently with the workers.

Recall from Section III-C that the server employs a majority-based voting scheme to determine the correctness of a task. Thus, if the workers in a group reach a majority on their results, the server accepts the majority answer as the "correct" result. In this case, it would increase the reliability ratings of the workers that are part of the majority, and decrease those of the remaining workers, treating their responses to be incorrect.

However, this still raises the question of how to update the ratings of workers in a group that doesn't reach quorum. In a previous work [32], we considered three different heuristics for updating reliability ratings. For each of the heuristics, we analyzed both the accuracy relative to an optimal heuristic and the impact on system performance. In addition, we considered the effects of using bounded and unbounded history on the accuracy and fluidity of worker reliability ratings.

In this work, we will restrict our attention to the *Optimistic* heuristic, which was the most accurate in simulation. The Optimistic heuristic is defined as follows: In the absence of a quorum, this heuristic increases the reliability ratings of any set of workers that agree on the result value. It penalizes those workers whose answers do not match any other answers from the group. Intuitively, this heuristic is based on the assumption that the probability of two workers returning the same wrong result independently is negligible, thus treating any matching answers to be pseudo-correct.

B. *Computing the Likelihood-of-Correctness*

The likelihood-of-correctness (LOC) of a group represents the probability of getting a correct answer from that group using the majority-based voting criterion of verification. This value can be computed using the individual reliability ratings of the members of the group, as estimated above. Consider a group $G = \{w_1, \ldots, w_{2k+1}\}$ consisting of workers $w_i, i = 1 \ldots 2k+1$ [2]. Let $r_i(t)$ be the reliability rating of a worker $w_i, i = 1 \ldots 2k+1$, at a given point in time t. Then, the LOC $\lambda(t)$ of the group G is given by:

$$\lambda(t) = \sum_{m=k+1}^{2k+1} \sum_{\{\epsilon:||\epsilon||=m\}} \prod_{i=1}^{2k+1} r_i(t)^{\epsilon_i} \cdot (1 - r_i(t))^{1-\epsilon_i} \qquad (1)$$

where $\epsilon = \{\epsilon_1, \ldots, \epsilon_{2k+1}\}$ is a vector of responses from the workers in the group, with 1 representing a correct response, and 0 representing an incorrect response. The criterion for determining correctness is based on achieving a majority, as described above. For simplicity, we will omit the implicit time variable, t, in future discussion of the LOC. For example, for a group G consisting of 5 workers w_1 through w_5, one possible vector could be $\{1, 1, 0, 0, 1\}$, indicating correct responses from workers $w_1, w_2,$ and w_5. Intuitively, Equation 1 considers all possible subsets of the given set of workers in which a majority of workers could respond correctly. It then computes the probability of occurrence of each of these subsets as a function of the reliability rating of the workers. Note that the likelihood of the false-positive case where a majority of workers return the same wrong answer is negligible, and hence ignored in Equation 1.

1) Lower Bound for Likelihood-of-Correctness: As can be seen from Equation 1, calculating the likelihood-of-correctness for a group results in a combinatorial explosion of the possible subsets that need to enumerated. In fact, the complexity of computing the λ value can be shown to be $O(2^{2k})$, which is infeasible for most practical purposes. To reduce the cost of computing λ values for multiple groups, we use a lower bound λ^{lb} for λ that is much simpler and more efficient to compute. This is obtained from Equation 1 using the arithmetic-geometric means inequality which is a special case of Jensen's Inequality [33].

$$\lambda^{lb} \geq \sum_{m=k+1}^{2k+1} \binom{2k+1}{m} \cdot \prod_{i=1}^{2k+1} r_i^{\alpha_m} \cdot (1 - r_i)^{1-\alpha_m}, \qquad (2)$$

[2] We consider odd-sized groups to avoid ambiguity in defining majority for even-sized groups.

where $\alpha_m = \frac{\binom{2k}{m-1}}{\binom{2k+1}{m}}$. It can be shown that the complexity of computing λ^{lb} is $O(k^2)$, and is thus much more efficient to compute than the actual value of λ. The grouping algorithms, which we will describe shortly, use the lower bound function to compute λ. In Section V-E, we compare the effect of the lower bound function on our simulation results.

2) The Role of LOC in Task Scheduling: To determine the size and composition of the groups, the system relies on a parameter indicating whether or not the LOC for a proposed group is acceptable. That is, we require some value λ_{target} such that if $\lambda \geq \lambda_{target}$, then we conclude that G is an acceptable group. We refer to λ_{target} as the *target LOC*.

Choosing an appropriate value for λ_{target} is critical to maximizing the benefit derived from the system. If λ_{target} is too small, many groups may return incorrect results, causing the tasks to be rescheduled. If it is set too high, the scheduler will be unable to form groups which meet the target, and the scheduler will degenerate to forming large fixed-size groups, adversely affecting the system throughput. Thus, the target LOC must be carefully selected to fit the reliability distribution of the workers. In Section IV-D, we will present an algorithm to adaptively determine the target LOC value. But first, we will describe how to group workers into redundancy groups *given* a target LOC.

C. Forming Redundancy Groups

So far, we have described heuristics for estimating individual worker ratings, and provided a mechanism for combining these ratings to determine the reliability of groups. We now present algorithms to assign workers into groups for task allocation, using these heuristics and mechanisms. The goal of forming these groups is to maximize both the throughput of successful task completions (those that result in correct results) and the rate of successful task completion (success rate) given a set of individual worker ratings (We will show in the next section how to incorporate both these metrics into the group formation decisions).

Formally, given a set of workers $W = \{w_1, \ldots, w_n\}$, a *group formation* algorithm would produce a partitioning $G = \{G_j\}$, where a task τ_j is assigned to each group G_j. These groups would be formed in such a way that λ_j of each G_j exceeds λ_{target}, thus achieving two goals: (a) increasing the likelihood of obtaining a correct result from the worker group working on the assigned task (in turn decreasing the likelihood of re-scheduling a task) and (b) increasing resource utilization by forming worker groups whose size varies based on the reliability rating

of its members. The algorithm for selecting an appropriate λ_{target} is deferred to the next section. Here, we present group formation algorithms given a λ_{target} value. The only property of the workers used by these algorithms is their reliability ratings.

1) Fixed-Size: This is the baseline algorithm for our system model as it represents "standard-best-practice" exhibited in systems such as BOINC. The Fixed-size algorithm randomly assigns workers to groups of size R_{max}, where R_{max} is a statically-defined constant. Every worker of a given group G_j is assigned the same task. This algorithm does not use the reliability ratings r_i of workers to size groups in an intelligent way. For a given set of workers, this algorithm will form a fixed number of groups, irrespective of r_i values.

2) First-Fit: In the First-fit algorithm, the available workers are sorted by decreasing reliability rating. Starting with the most reliable, workers are assigned to group G_j until either $\lambda_j \geq \lambda_{target}$ or until the maximum group size R_{max} is reached. This process is repeated until all the available workers are assigned to a group. Intuitively, First-fit attempts to form the first group that satisfies λ_{target} from the available workers in a greedy fashion. By bounding the size of G_j with R_{max}, we ensure that First-fit forms bounded groups and degenerates to the Fixed-size heuristic in the absence of a sufficient number of reliable workers.

Algorithm 1 First-Fit (w worker-list, τ task-list, λ_{target} target LOC, R_{min} min-group-size, R_{max} max-group-size)

1: Sort the list w of all available workers on the basis of the reliability ratings r_i
2: **while** $|w| \geq R_{min}$ **do**
3:     Select task τ_j from τ
4:     **repeat**
5:         Assign the *most reliable* worker w_r from w to G_i
6:         $w \leftarrow w$ - w_r
7:         **if** $|G_j| \geq R_{min}$ **then**
8:             Update λ_j
9:         **end if**
10:     **until** $(\lambda_j \geq \lambda_{target} \wedge |G_j| \geq R_{min}) \vee |G_j| = R_{max}$
11: **end while**

3) Tight-Fit: The Tight-fit algorithm attempts to form a group G_j such that λ_j is as close as possible to λ_{target}. The Tight-fit algorithm searches the space of available workers to find the smallest G_j that exceeds λ_{target} by the minimum possible margin. If no group of size R_{max} or smaller meets λ_{target}, the algorithm forms a group that falls short of the target by the smallest amount. As with First-fit, this process is repeated until the worker pool is exhausted. Intuitively,

this algorithm attempts to form the best-fit of worker nodes for a given λ_{target}. As a result, tasks are not overprovisioned with more reliable resources than necessary, and well-balanced groups are more likely to be formed.

Algorithm 2 Tight-Fit (w worker-list, τ task-list, λ_{target} target LOC, R_{min} min-group-size, R_{max} max-group-size)

1: Sort the list w of all available workers on the basis of the reliability ratings r_i
2: **while** $|w| \geq R_{min}$ **do**
3: Select task τ_j from τ
4: Use binary search to identify the smallest set s of n workers w_n from w such that λ_s exceeds λ_{target} minimally
5: **if** such a set s is found **then**
6: Assign the w_n workers to G_j
7: **else**
8: Select the set of n workers s for which $\lambda_{target} - \lambda_s$ is minimized
9: Assign the w_n workers to G_j
10: **end if**
11: $w \leftarrow w$ - w_n
12: **end while**

4) Random-Fit: The Random-fit algorithm uses reliability ratings to form groups by randomly adding workers to a group G_j until either λ_j meets λ_{target} or the group has R_{max} workers. It differs from First-fit in that workers are added to groups randomly, rather than in sorted order.

Given a set of workers and a λ_{target}, each algorithm is likely to produce different groups. A simple example for $\lambda_{target} = 0.5$ is illustrated in Figure 2. In this example, First-fit is only able to form a single group which meets λ_{target} because it uses all of the highly reliable workers in the first group. Similarly. Random-fit also produces only one group that is able to meet the target as it assigns workers randomly to groups. In contrast, Tight-fit is able to form two groups which meet λ_{target} because it searches for groupings whose λ values deviate from the target by the smallest amount.

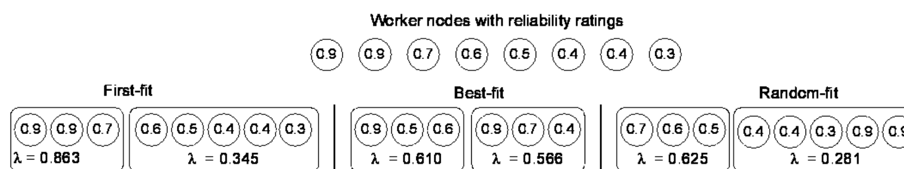


Fig. 2. Example node groupings produced by different algorithms for $\lambda_{target} = 0.5$, $R_{min} = 3$ and $R_{max} = 5$.

The time spent computing the LOC for a given group is the primary component of the overhead

incurred by the reputation-based schedulers, so we can compare the algorithms in a system-independent manner. The First-fit and Random-fit algorithms form groups in a sequential fashion. Each grouping considers at most a constant number of worker pairs, and the number of groups is linear in the number of workers, so the number of calls to the LOC function scales linearly with the size of the network. The Tight-fit algorithm is more expensive because it tries to form the best possible groups by using a binary search of the available workers. Thus, the number of calls to the LOC function is $O(n \log n)$, where n is the size of the network.

D. Adaptive Determination of Target LOC

As we saw in the previous section, the target LOC λ_{target} is a critical parameter in forming redundancy groups. There are several drawbacks associated with using a static value for this parameter. First, it is difficult to select an appropriate value without prior knowledge of the expected worker population. For instance, while a large value of λ_{target} can be satisfied efficiently by small groups in a highly reliable population, it may lead to extremely low resource utilization and may even be unachievable for a population of largely unreliable workers. Second, the reliability of workers may vary with time due to node churn, changes in node behavior, and other events which affect a node's reliability. Such dynamic changes could make even a carefully chosen λ_{target} value undesirable. To summarize, λ_{target} is influenced by the underlying system characteristics, which are not easy for a user or system administrator to determine statically.

To maximize the benefit derived from the system, it would be desirable if the system was capable of selecting an appropriate target LOC value automatically. Furthermore, we would like the system to be able to dynamically adapt λ_{target} to meet current system conditions. The task server is ideally suited to select an appropriate λ_{target}, since it constantly updates the reliability ratings of the workers and monitors the performance of the system.

Besides the system characteristics, the choice of λ_{target} also depends on the metric (throughput or success rate) being optimized by the application designer. There is a natural trade-off between the throughput of successful task completion and the success rate. By forming larger groups, we generally increase the likelihood that an individual group will return a correct answer, but we decrease the number of tasks attempted, which may in turn decrease the throughput of successful tasks. Conversely, decreasing the average group size will make each group less likely to return correct results, but may increase the number of successful tasks completed due to the increase

in the number of tasks attempted. One can imagine scenarios in which either metric would be preferred over the other. Thus, neither throughput nor success rate alone is a sufficient metric for determining an optimal value of λ_{target}. In particular, if we wish to bound the latency experienced by individual tasks, success rate is a more important metric than throughput (as high success rate reduces the need to re-execute the tasks). On the other hand, if we simply wish to maximize the number of tasks completed, throughput is more important.

Thus, determining an optimal value for λ_{target} requires us to consider both throughput and success rate simultaneously. Such an optimization is an instance of a *multi-objective optimization (MO)* problem. A common approach to solving an MO problem is to use techniques such as Goal Programming [34], [35] or Multilevel Programming [35] that reduce the multiple objectives to a single objective, and then employ standard Linear Programming techniques to obtain a solution.

Depending on the specific application, we can tailor our objective to favor either throughput or success rate by using a weighted combination of these two objectives, which we refer to as the *gain*, G:

$$G(\rho, s) = \alpha \cdot \rho + (1 - \alpha) \cdot s,$$

where ρ and s represent the normalized throughput and success rate, respectively. α is a tunable parameter that can be set by a user or administrator to express their relative bias towards one of the metrics: $\alpha = 1$ would correspond to a throughput-optimal system, while $\alpha = 0$ would correspond to a success rate-optimal system.

We use an adaptive algorithm to update the target LOC λ_{target} based on measurements of the current value of the gain G. The detailed algorithm is given in the Appendix due to space constraints: here, we present the intuition behind it. The adaptive algorithm employs a custom hill-climbing algorithm to converge to an initial λ_{target} value reflecting the underlying reliability distribution of the system. The algorithm then constantly monitors the current gain values (using the observed throughput and success rate) and compares them to an exponentially smoothed average over time. A significant change in the current value of gain serves as an indicator that the underlying worker distribution has changed, and results in the selection of a new λ_{target} based on the current measures of gain.

The description of the adaptive algorithm may lead one to believe that we are removing one user-specified parameter (λ_{target}) at the expense of adding several new parameters (such

as α, and hill-climbing algorithm parameters such as its period, significance thresholds, etc.). However, most of these parameters can be configured empirically or determined automatically using feedback, without any input from the user. Effectively, the user is only responsible for specifying the value of α, which is a much more intuitive value than λ_{target}, as α only depends on the relative importance of the metrics to the user. The adaptive algorithm is then able to incorporate this fixed user preference in determining the choice of λ_{target}, which is highly system-dependent and dynamic in nature.

V. Evaluation

In this section, we evaluate the performance of the rating techniques and grouping algorithms described in the previous section through simulation of a donation-based distributed computing platform. In our simulations, we model a large number of real-world scenarios using different distributions for worker reliability values.

A. Evaluation Methodology

Our evaluation is based on a simulator loosely modeled around the BOINC [4] distributed computing infrastructure, which consists of a task server and some number of worker machines. We make two simplifying assumptions to enable fair comparison between different grouping algorithms.

First, the simulator is *round-based*—work assignment and verification is done periodically in fixed-duration time periods called rounds. The task server assigns work to all the workers at the beginning of a round, and then waits for the workers to return their results. At the end of each round, the server collects and verifies the received results, updates the reliability ratings using the Optimistic heuristic described earlier, and re-forms groups for task allocation in the next round. Workers who fail to respond by the end of a round are simulated as having returned incorrect results. In the results shown here, we ran our simulations for a total of 1000 rounds each. In practice, the length of a round would be linked to the expected execution time of the tasks within it.

Second, the task server has an extremely large pool of work relative to the number of workers available. This assumption is consistent with the projects hosted by the BOINC infrastructure, and is likely to be true for future large-scale scientific computing applications as well. As a

Name	Distribution (over [0,1])	Real-world Scenario
Uniform	Uniform	General environment
Heavy-tail-high	1-Pareto($a = 1, b = 0.1$)	Majority of reliable workers; a few unreliable workers
Heavy-tail-low	Pareto($a = 1, b = 0.2$)	Majority of workers unreliable; major virus/outage
Normal-high	Normal: $\mu = 0.9$, $\sigma = 0.05$	Reliable environment; most workers reliable
Bimodal	Bi-Normal: $\mu = 0.2/0.8$, $\sigma = 0.1$	50% reliable workers, 50% unreliable
Normal-low	Normal: $\mu = 0.3$, $\sigma = 0.1$	Hostile environment, e.g., military scenarios

TABLE I

PROBABILITY DISTRIBUTIONS USED IN THE SIMULATIONS TO EMULATE DIFFERENT REAL-WORLD SCENARIOS.

result, the task server will always attempt to utilize all of the available workers, and workers will never have to wait for work.

An individual worker's reliability is modeled by assigning it a probability p of returning a correct result within a round. When a worker is assigned a task, it returns the correct result with probability p. These probabilities are known only to the workers - the task server has no knowledge of these values a priori.

To simulate various real-world reliability scenarios, we generate individual worker probabilities from several different probability distributions. Table I lists some of the distributions used in our simulations and the corresponding scenarios modeled by each of them. For instance, we use a normal distribution with a high mean to emulate a highly-reliable system, where most workers are well-connected and return correct results most of the time. On the other hand, we use a bimodal distribution to represent a system that has a mix of highly-reliable workers and compromised or poorly-connected nodes.

B. Reputation-Based Scheduling

We now evaluate the various reputation-based grouping algorithms described in Section IV-C. We start by evaluating these algorithms for a fixed target LOC value in this section. We first describe the metrics and parameters used in our evaluation.

1) Metrics and Parameters: To evaluate the effectiveness of the grouping algorithms, we use the following metrics:

- *Throughput (ρ)*: The throughput during a round is defined as the number of tasks for which a majority was achieved during that round (i.e., the number of 'successful' tasks).

$$\rho = |T_{success}|,$$

where $T_{success}$ is the set of successfully completed tasks during a round.

- *Mean Group Size (g)*: The mean group size for a round is the mean number of workers assigned to each task during the round.

$$g = \frac{\sum_{i=1}^{N_G} |G_i|}{N_G},$$

where N_G is the total number of groups formed during the round.

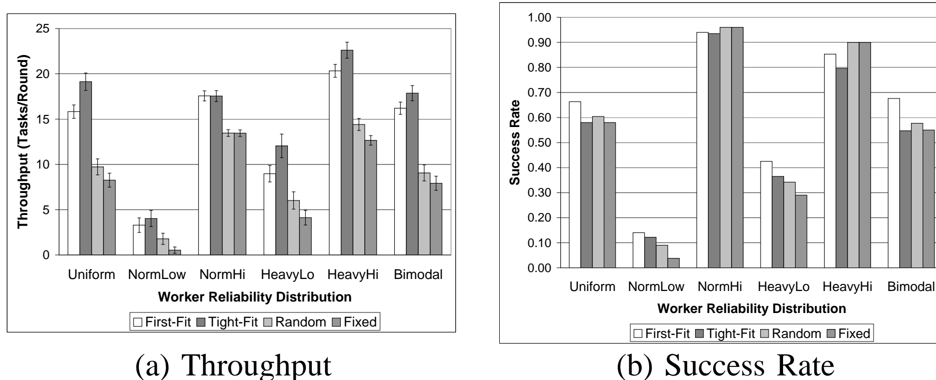
- *Success Rate (s)*: The success rate during a round is defined as the ratio of successfully completed tasks to the number of tasks attempted (equal to the number of groups formed) during that round.

$$s = \frac{\rho}{N_G}.$$

To fully understand the behavior of the reputation-based schedulers, we ran an exhaustive set of simulations covering a large parameter space: the worker reliability distributions described in Table I, a worker pool size of 100 and 1000, minimum group size (R_{min}) of 3, and maximum group sizes (R_{max}) of 3, 5, 7, and 9. For each parameter setting, we compare the four algorithms described in Section IV-C (First-fit, Tight-fit, Random-fit, and Fixed).

For a given distribution and R_{max}, we set λ_{target} equal to the success rate of the Fixed algorithm for the same parameter values. This ensures that the success rate of the various algorithms will be approximately the same, facilitating a comparison between our proposed algorithms and the baseline Fixed algorithm. Due to space constraints, we will present a subset of the results here, including descriptions of the most interesting findings.

2) Comparing Scheduling Algorithms: In our first experiment, we compared the different grouping algorithms using a pool of 100 workers. In Figure 3(a), we present the mean throughput across all rounds for an R_{max} value of 7 workers. For R_{max} = 7, the theoretical success rate of a Fixed strategy under the realistic Heavy-High distribution is 90%, which seems like a reasonable 'target' success rate. This led to selection of R_{max} = 7 as a representative value for the fixed group size. The First-fit and Tight-fit algorithms improve on the throughput of Fixed by 25-



(a) Throughput (b) Success Rate

Fig. 3. Algorithm Comparison

250%, depending on the worker reliability distribution. The Random-fit algorithm, while not performing as well as First-fit and Tight-fit, still outperforms Fixed by about 20-50%.

Figure 3(b) plots the mean success rate across all rounds. Since we set λ_{target} equal to the success rate achieved by the Fixed algorithm, we would expect that the mean success rate for the other algorithms to be similar. The success rate of Random-fit and Tight-fit is equal or greater to that of Fixed—the minor shortfalls in some cases are due to the use of approximate worker reliability measures and maximum group sizes. First-fit deviates significantly for most of the distributions due to its greedy group formation policy— it attempts to form groups starting with the most reliable workers, and working down to the least reliable workers, so that it can form highly reliable groups for distributions with low average reliability. Conversely, it also forms several unreliable groups for high reliability distributions.

Overall, these results indicate that reputation-based scheduling algorithms significantly increase the average throughput for all of the reliability distributions, while maintaining a high success rate.

Figure 4 shows the mean group-size results for the above experiment. Both First-fit and Tight-fit are able to form substantially smaller groups satisfying the target LOC requirement. As a result, these algorithms attempt significantly more tasks in each round, resulting in the substantially higher throughput shown in Figure 3.

In Table II, we present the throughput results for varying maximum group sizes using the Heavy-High distribution. As the R_{max} parameter is reduced, the gap between the Fixed algorithm and the reputation-based algorithms starts to narrow, since it becomes harder to form smaller

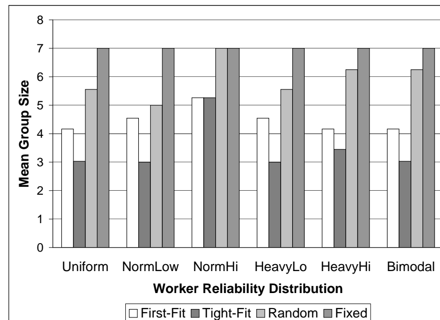


Fig. 4. Mean Group Size

	$R_{max} = 9$	$R_{max} = 7$	$R_{max} = 5$	$R_{max} = 3$
First-fit	18.21 ± 1.29	20.33 ± 1.42	22.37 ± 1.55	26.13 ± 1.86
Tight-fit	20.56 ± 1.65	22.61 ± 1.77	24.03 ± 1.74	26.01 ± 1.86
Random-fit	11.06 ± 1.05	14.41 ± 1.32	18.95 ± 1.60	26.78 ± 2.08
Fixed-size	10.20 ± 0.83	12.65 ± 1.05	17.32 ± 1.48	26.84 ± 2.13

TABLE II

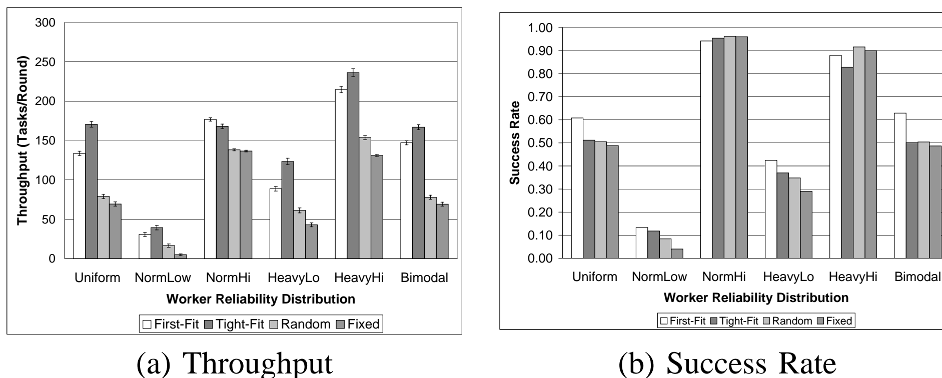
EFFECT OF DECREASING R_{max} ON THROUGHPUT (HEAVY-HIGH DISTRIBUTION)

groups that meet λ_{target}. In particular, if we set $R_{min} = R_{max}$, then all of the scheduling algorithms are essentially the same. In this case, all the algorithms form groups of size 3,causing them to have nearly the same throughput.

3) Effect of Scale: In our second experiment, we use the same parameter settings as the previous experiment, but increase the network size from 100 to 1000 workers. Figure 5 shows the throughput and success-rate results for this experiment. Scaling the size of the network up to 1000 workers causes a proportional increase in the throughput, without affecting success rate much. Clearly, scaling up the network will have little to no impact on the relative throughput or success rate in simulation. We will consider the impact of scale on the overhead associated with the different scheduling algorithms in Section V-D.

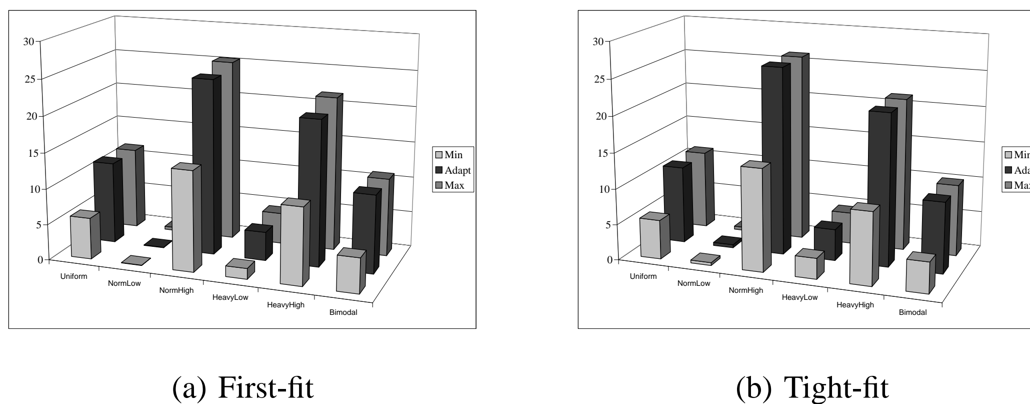
C. Adaptive Algorithm for Determining Target LOC

In the previous section, we compared the various group formation algorithms using a fixed value of the target LOC λ_{target}, which was selected based on the observed success rate for the Fixed algorithm. In this section, we evaluate the adaptive algorithm presented in Section IV-D



(a) Throughput (b) Success Rate

Fig. 5. Mean Throughput and Success Rate, Network Size=1000



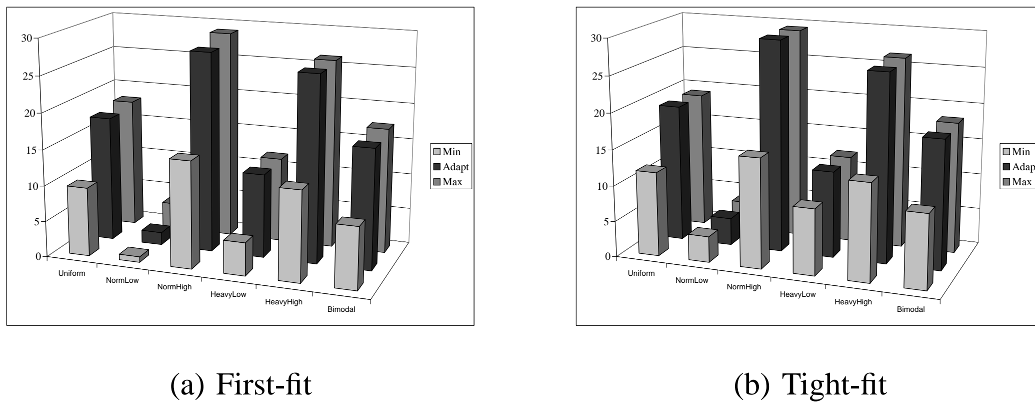
(a) First-fit (b) Tight-fit

Fig. 6. Comparison of min/max gain achieved using static LOC to gain achieved using adaptive algorithm ($\alpha = 0.5$)

for its ability to determine a desirable λ_{target} value based on current system conditions and the relative importance of throughput and success rate metrics. These experiments use the same values for R_{min} and R_{max} as the previous experiments.

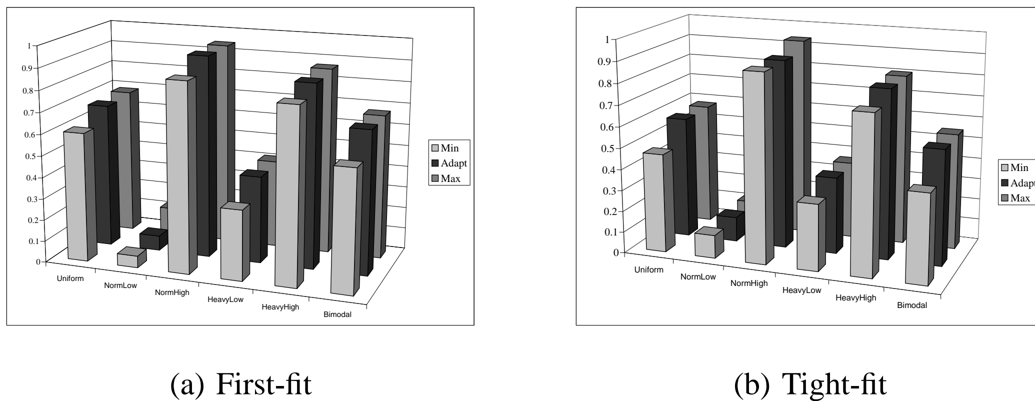
The default values for the parameters specific to the hill-climbing algorithm (the period p, significance thresholds, etc.) were empirically determined based on a comprehensive evaluation of the parameter space. The selected values were chosen to minimize noise in the periodic gain measurements, and to improve the stability and speed of convergence.

The period p was set to 10, the significance thresholds δ_{sig}, δ_{mod} and δ_{in} were set to 1.15, 1.05, and 1.01, respectively. $maxrounds$ was set to 5, $weight_{curr}$ was set to 0.3 and $weight_{hist}$ was set to 0.7.



(a) First-fit (b) Tight-fit

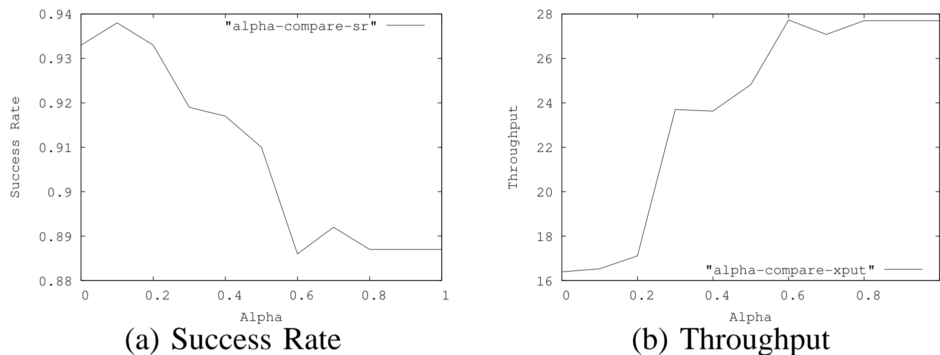
Fig. 7. Comparison of min/max throughput achieved using static LOC to throughput achieved using adaptive algorithm ($\alpha = 1$)



(a) First-fit (b) Tight-fit

Fig. 8. Comparison of min/max success rate achieved using static LOC to that achieved using adaptive algorithm ($\alpha = 0$)

1) Convergence to a Desirable Value: Our first experiment illustrates the adaptive algorithm's ability to converge to an appropriate λ_{target} despite starting with no knowledge of the underlying worker population. In this experiment, we measured the average gain achieved over 10,000 rounds using the First-fit and Tight-fit scheduling algorithms coupled with the adaptive λ_{target}-determination algorithm (referred to as adaptive First-fit and adaptive Tight-fit, respectively). Then, we measured the average gain achieved using the non-adaptive First-fit and Tight-fit algorithms for every possible value of λ_{target} from 0 to 1 (with a granularity of 0.01), to determine the best and worst achievable values. Figures 6, 7 and 8 compare the minimum and maximum gains achieved using a static value of λ_{target} to that achieved by the corresponding



(a) Success Rate (b) Throughput

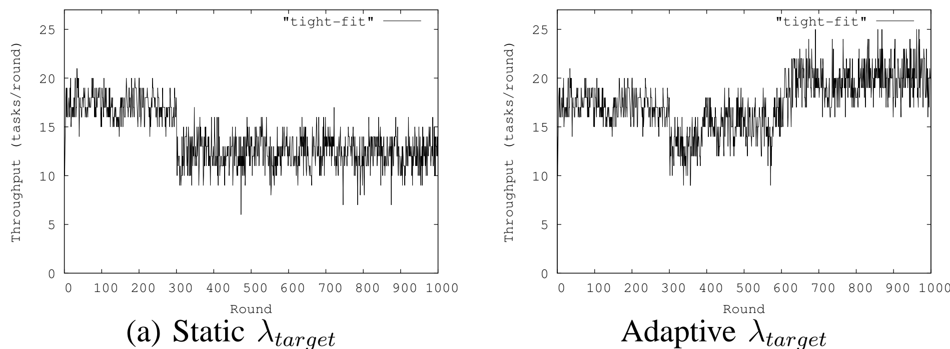
Fig. 9. Comparison of throughput/success rate achieved using adaptive algorithm with varying α

adaptive algorithm. These figures show the gain computed using α values of 0.5, 1, and 0 respectively where Figure 6 gives equal weight to both throughput and success rate, Figures 7 and 8 correspond to throughput-optimal and success rate-optimal algorithms respectively.

As seen from the figures, the average gain achieved by the adaptive algorithm is very close to the maximum gain possible using a static λ_{target} value in all instances. This observation holds for both First-fit and Tight-fit algorithms. Overall, we tested 36 algorithm/worker-distribution configurations, out of which the adaptive algorithm deviated from the maximum achievable gain by less than 2% in 25 cases, and by less than 5% in 32 cases. The only meaningful deviation experienced was for the Normal-Low worker distribution using the First-fit algorithm, where the algorithm has a higher likelihood of getting stuck in a local minimum.

2) Effectiveness of α: In our next experiment, we evaluated the effectiveness of the gain metric and the α parameter to represent the relative importance of the throughput and success rate metrics. Figures 9(a) and (b) show the values of success rate and throughput using adaptive First-fit, as we vary the value of α from 0 to 1. Recall that $\alpha = 0$ corresponds to a pure success rate-oriented system, while $\alpha = 1$ corresponds to a throughput-oriented system. As can be seen from the figures, an increase in α results in decreasing success rate and increasing throughput. This result implies that the gain G is an effective metric for incorporating user preferences.

3) Dealing with Non-Stationary Workers: We next illustrate the effectiveness of the adaptive algorithm to deal with non-stationary behavior of workers, i.e., when their reliability varies with time. We consider a large-scale worker blackout scenario that corresponds to a real-world event such as a network partitioning, a large organization crash, or a major virus, which may



(a) Static λ_{target} Adaptive λ_{target}

Fig. 10. Large-Scale Blackout: Effect of Adaptive λ_{target} on throughput

suddenly compromise the reliability of a large number of workers. To emulate such an event, we modified the simulation so that 30% of the workers transitioned from a highly-trusted normal-high distribution to an unreliable heavy-low distribution after round 300. These experiments use the Tight-fit algorithm with $R_{max} = 7$.

Figures 10(a) and (b) show the effect of the large-scale blackout on the system throughput using static and adaptive λ_{target} values, respectively. Figure 10(a) shows a considerable dip in throughput after the blackout. This is because Tight-fit continues to operate with a λ_{target} value that was tailored to the higher reliability environment. Since the system has fewer trusted workers at its disposal after the blackout, it ends up forming very large (and thus fewer) groups in an attempt to satisfy this high λ_{target}. This failure to adapt to the new reliability distribution results in the observed dip in throughput.

Figure 10(b) shows that although the throughput drops drastically in round 300 (immediately after the blackout), the system immediately starts compensating for the drop in reliability by reducing λ_{target}. The average throughput returns to near pre-blackout levels approximately 100 rounds later. There is a slight drop in the throughput between rounds 550 and 600, because the system is probing for higher λ_{target} in an attempt to improve the success rate. However, the system automatically corrects for this drop in performance and stabilizes near round 700 at a higher throughput (but considerably lower success-rate - 80% vs. 96%) than it was achieving before the blackout. This experiment clearly demonstrates the value of dynamically updating λ_{target} based on current conditions in the system.

	uniform	normal-high	heavy-low	heavy-high	bimodal
First-fit	340	130	210	80	90
Tight-fit	350	130	230	210	200

TABLE III

NUMBER OF ROUNDS REQUIRED TO ACHIEVE STEADY-STATE λ_{target} FOR SEVERAL CLIENT DISTRIBUTIONS

4) Convergence Time: The time to converge for the adaptive algorithm is dependent on several different variables - variance in the underlying client distribution, the wait time between target LOC adjustments, the granularity of adjustments and the number of stationary rounds before steady-state.

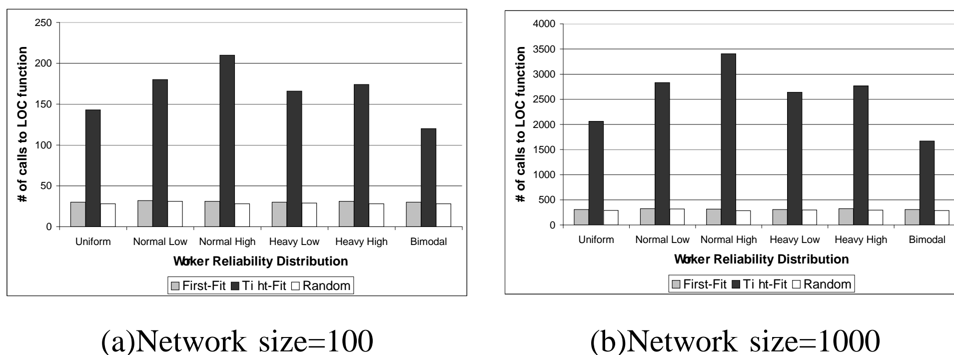
In Table III, we list the number of rounds required to converge starting with zero knowledge (initial target LOC of 0.5) using one particular set of parameters. The wait time between adjustments was set to 10 rounds and the number of stationary rounds required was set to 5, thus the minimum number of rounds for convergence is 50. These values are fairly conservative, but they yielded excellent average gain measurements for our data sets. Selecting a smaller wait time will cause the average time to convergence to decrease, but may result in a loss of average gain due to noisy feedback measures. Moreover, the time to converge due to a change in the underlying client distribution will depend on the magnitude of the change.

D. Overhead

In this section, we compare the overhead of the grouping algorithms in terms of the number of invocations of the LOC function. In Section IV-C, we determined that the theoretical overhead is $O(n)$ for First-fit and Random-fit, and $O(n \log n)$ for Tight-fit, where n is the size of the network. Figure 11 shows the average number of calls to the LOC function during a single round for network sizes of 100 and 1000. As expected, the number of calls to the LOC function grows significantly faster for Tight-fit.

E. Accuracy of λ^{lb}

In Section IV-B, we presented a function (Equation 2) to compute λ^{lb}, a lower bound for LOC of a group. We now analyze the impact of using this approximation function on the effectiveness



(a)Network size=100 (b)Network size=1000

Fig. 11. Overhead (Number of calls to LOC function)

	uniform	normal	heavy-tail	bimodal
First-fit	0.0010 ± 0.0016	0.0000 ± 0.0002	0.0014 ± 0.0052	0.0012 ± 0.0052
Tight-fit	0.0784 ± 0.0587	0.0015 ± 0.0023	0.0210 ± 0.0123	0.0910 ± 0.0755
Random-fit	0.3532 ± 0.2325	0.0025 ± 0.0033	0.0561 ± 0.0442	0.3328 ± 0.2252

TABLE IV

λ VS. λ^{lb}: AVERAGE ERROR IN PRACTICE

of the system. The lower bound function corresponds to the GM in the AM-GM inequality, a special case of Jensen's inequality. For functions that comply to this inequality, the difference between the AM (the actual LOC function in our case), and the GM increases as the spread of the values increases. Therefore, as the disparity between the low and high reliability ratings within a group increases, the lower bound diverges more and more from the actual λ value.

We empirically computed the difference between λ and λ^{lb} for each of the reputation-based algorithms (shown in Table IV). As expected, we found that the error for Random-fit, which may end up grouping workers with very dissimilar ratings, was quite high (0.05-0.3). In contrast, the First-fit algorithm, which groups similarly rated workers, had negligible error ($\leq$ 0.001). The error for Tight-fit fell in the middle but is still relatively small (0.002-0.09).

To quantify the effect of these errors on the effectiveness of the reputation-based algorithms, we repeated the experiment from Section V-B.2 using the actual LOC function instead of the lower-bound. The throughput results are shown in Figure 12. As expected, these results show a correlation between the error experienced by an algorithm and the benefit associated with using λ instead of λ^{lb}. In addition, the benefit is the highest for the distributions where the error was

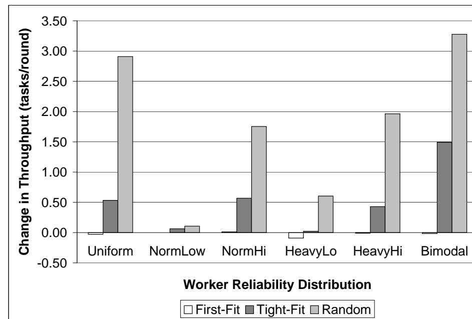


Fig. 12. Change in throughput when using λ vs. λ^{lb}

the worst (uniform, bimodal). Based on these measurements, we conclude that Tight-fit could improve its throughput by up to 10% by using a more accurate value for λ, and Random-fit could gain up to 35% over the values presented earlier for λ^{lb}.

VI. Conclusions and Future Work

In this paper, we have presented a design and analysis of techniques to handle the inherent unreliability of nodes in large-scale donation-based distributed infrastructures, such as P2P and Grid systems. We proposed a reputation-based scheduling model to achieve efficient task allocation in such an unreliable environment. Our reputation system represents the underlying reliability of system nodes as a statistical quantity that is estimated based on the prior performance and behavior of the nodes. Our scheduling algorithms use the estimated reliability ratings to form redundancy groups that achieve higher throughput while maintaining desired success rates of task completion. In addition, we present a technique for adaptively adjusting scheduling parameters to match the underlying reliability distribution, which can be used to control the system's response to non-stationary node reliability. We evaluate our algorithms using a simulator based on the BOINC distributed computing infrastructure. In our simulation, we varied the underlying reliability distribution of the worker reliability values to emulate several real-world scenarios. Our simulation results indicate that reputation-based scheduling can significantly improve the throughput of the system (by as much as 25-250%) for worker populations modeling several real-world scenarios, including non-stationary behavior, with overhead that scales well with system size. As part of future work, we intend to implement our techniques in a real testbed (e.g., one

using BOINC) and to use real workload traces to evaluate the efficacy and overhead of our algorithms under real-world deployment.

ACKNOWLEDGMENT

We would like to thank Arindam Banerjee for providing helpful pointers on the lower-bound function and its analysis.

APPENDIX
ADAPTIVE ALGORITHM FOR TARGET LOC

The appendix has been provided as a supplemental document to this article.

REFERENCES

[1] "Sun grid," http://www.sun.com/service/sungrid/.

[2] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, July 2003.

[3] "Condor project," http://www.cs.wisc.edu/condor/.

[4] D. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th ACM/IEEE International Workshop on Grid Computing*, 2004.

[5] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova, "Entropia: Architecture and performance of an enterprise desktop Grid system," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 597–610, 2003.

[6] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: An Experiment in Public-Resource Computing," *Communications of the ACM*, vol. 45, no. 11, 2002.

[7] D. Molnar, "The SETI@Home problem," *ACM Crossroads*, Sept. 2000.

[8] P. Shread, "Gateway offers Computing on Demand," 2002, http://www.gridcomputingplanet.com/news/article.php/3281_1555061.

[9] J. Sonnek and J. Weissman, "A Quantitative Comparison of Reputation Systems in the Grid," in *Proceedings of the Sixth ACM/IEEE International Workshop on Grid Computing*, 2005.

[10] S. Lee, R. Sherwood, and B. Bhattacharjee, "Cooperative peer groups in nice," in *Proceedings of IEEE Infocomm*, 2003.

[11] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, P. Samarati, and F. Violante, "A Reputation-Based Approach for Choosing Reliable Resources in Peer-to-Peer Networks," in *Proceedings of Ninth ACM Conf. on Computer and Communications Security*, Nov. 2002, pp. 207–216.

[12] S. Kamvar, M. Schlosser, and H. Garcia-Molina, "The EigenTrust Algorithm for Reputation Management in P2P Networks," in *Proceedings of the Twelfth International World Wide Web Conference*, 2003.

[13] B. Alunkal, I. Veljkovic, G. von Laszewski, and K. Amin, "Reputation-Based Grid Resource Selection," in *Workshop on Adaptive Grid Middleware*, 2003.

[14] F. Azzedin and M. Maheswaran, “Integrating Trust into Grid Resource Management Systems,” in *Proceedings of the International Conference of Parallel Processing*, 2002.

[15] R. Zhao and K. Hwang, “PowerTrust: A Robust and Scalable Reputation System for Trusted Peer-to-Peer Computing,” *IEEE Transcations on Parallel and Distributed Systems*, vol. 18, no. 5, 2007.

[16] I. Foster and C. Kesselman, Eds., *Grid2: Blueprint for a New Computing Infrastructure*. Morgan Kauffman, CA, USA, 2004.

[17] “Folding@home distributing computing project,” http://folding.stanford.edu.

[18] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao, “Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet,” in *Proceedings of the IEEE Fourth International Conference on Peer-to-Peer Systems*, 2004.

[19] A. Awan, R. Ferreira, S. Jagannathan, and A. Grama, “Unstructured Peer-to-Peer Networks for Sharing Processor Cycles,” *Journal of Parallel Computing*, 2005.

[20] M. Castro and B. Liskov, “Practical Byzantine Fault Tolerance,” in *Proceedings of Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[21] P. Golle and I. Mironov, “Uncheatable Distributed Computations,” in *Proceedings of CT-RSA*, Apr. 2001.

[22] L. F. G. Sarmenta, “Sabotage-Tolerance Mechanisms for Volunteer Computing Systems,” in *Proceedings of ACM/IEEE International Symposium on Cluster Computing and the Grid*, 2001.

[23] S. Zhao and V. Lo, “Result Verification and Trust-based Scheduling in Open Peer-to-Peer Cycle Sharing Systems,” in *IEEE Fifth International Conference on Peer-to-Peer Systems*, Sept. 2005.

[24] P. Resnick, R. Zeckhauser, E. Friedman, and K. Kuwabara, “Reputation Systems,” *Communications of the ACM*, vol. 43, no. 12, pp. 45–48, 2000.

[25] K. Aberer and Z. Despotovic, “Managing Trust in a Peer-2-Peer Information System,” in *Proceedings of the Ninth International Conference on Information and Knowledge Management*, 2001.

[26] S. Song, K. Hwang, and Y. Kwok, “Risk-Resilient Heuristics and Genetic Algorithms for Security-Assured Grid Job Scheduling,” *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 703–719, 2006.

[27] R. Gupta and A. Somani, “CompuP2P: An Architecture for Sharing of Compute Power in Peer-to-Peer Networks with Selfish Nodes,” in *Proceedings of Second Workshop on Economics of Peer-to-Peer Systems*, 2004.

[28] K. Anagnostakis and M. Greenwald, “Exchange-Based Incentive Mechanisms for Peer-to-Peer File Sharing,” in *Proceedings of the 24th International Conference on Distributed Computing Systems*, 2004.

[29] B. Chun, Y. Fu, and A. Vahdat, “Bootstrapping a Distributed Computational Economy with Peer-to-Peer Bartering,” in *Proceedings of First Workshop on Economics of Peer-to-Peer Systems*, 2003.

[30] W. Du, J. Jia, M. Mangal, and M. Murugesan, “Uncheatable Grid Computing,” in *24th IEEE International Conference on Distributed Computing Systems)*, Mar. 2004, pp. 4–11.

[31] P. Golle and S. G. Stubblebine, “Secure Distributed Computing in a Commercial Environment,” in *Proceedings of the 5th International Conference on Financial Cryptography*, Feb. 2002, pp. 289–304.

[32] J. Sonnek, M. Nathan, A. Chandra, and J. Weissman, “Reputation-Based Scheduling on Unreliable Distributed Infrastructures,” Dept. of CSE, Univ. of Minnesota, Tech. Rep. 05-036, Nov. 2005.

[33] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. New York: Wiley, 1991.

[34] A. Charnes and W. Cooper, “Goal programming and multiple objective optimization - part I,” *European Journal of Operational Research*, vol. 1, pp. 39–54, 1977.

[35] R. E. Steuer, *Multiple Criteria Optimization: Theory, Computation and Application*. New York: Wiley, 1986.