# Collusion Detection for Grid Computing

Eugen Staab and Thomas Engel
Faculty of Science, Technology and Communication
University of Luxembourg
L-1359 Luxembourg
{eugen.staab,thomas.engel}@uni.lu

## Abstract

*A common technique for result verification in grid computing is to delegate a computation redundantly to different workers and apply majority voting to the returned results. However, the technique is sensitive to "collusion" where a majority of malicious workers collectively returns the same incorrect result. In this paper, we propose a mechanism that identifies groups of colluding workers. The mechanism is based on the fact that colluders can succeed in a vote only when they hold the majority. This information allows us to build clusters of workers that voted similarly in the past, and so detect collusion. We find that the more strongly workers collude, the better they can be identified.*

## 1 Introduction

**Problem Context and Motivation** In grid computing, a master assigns computational tasks to resources identified as workers, which are expected to execute the tasks and return their results. Since the master has no control over the workers, it cannot be sure whether a returned result is actually correct or not. This is especially relevant in the scenario of *desktop grids* [12]. Workers may return incorrect results because they fail due to overclocking or software errors [13, 21], because they want to harm the master, or because they want to save resources (e.g., by returning random results). Conventional security mechanisms can ensure *data authenticity* and *integrity* [20]. However, these mechanisms cannot ensure the correctness of received results. For specific computations, simple *checkers* can be used to verify results in an efficient way (e.g., see [24]). Unfortunately, no such mechanism is known for general computations [10].

A common principle to tackle this issue in the general case is to rely on *redundancy* (e.g., see [1]): a computation is redundantly outsourced to several randomly selected workers; majority voting is applied to the set of returned results to decide in favor of the result that appears most often. This approach tolerates a certain number of incorrect results in a vote. However, it does not resist a majority of colluding workers that collectively return the same incorrect result. Even though workers are randomly selected for each vote, with the possibility of *massive attacks* (e.g., [4]), the probability for a majority of colluders becomes significant. Therefore, mechanisms are required that detect colluding behavior of malicious workers.

**Approach and Contribution** We present a mechanism for collusion detection that exploits the information of how often pairs of workers are together in the majority/minority of votes, and how often they are in opposite groups. In cases, where colluding workers win a vote, they are always together in the majority, whereas honest workers together form the minority. We first show theoretically that this fact allows a line to be drawn between honest and colluding workers. Secondly, we propose an algorithm that uses graph clustering to discover this division. Finally, we evaluate the algorithm in terms of accuracy and running time, using two different graph clustering algorithms from the literature. We find that, given a certain number of observations, our mechanism can successfully detect sophisticated colluders.

**Organization** The remainder of this paper is organized as follows. In Sect. 2, we detail the model and assumptions used in our work. In Sect. 3, we conduct a theoretical analysis, which forms the basis for the collusion detection algorithm proposed in Sect. 4. We describe the implementation of the algorithm in Sect. 5 and evaluate its accuracy and performance in Sect. 6. We outline related work in Sect. 7 and draw conclusions in Sect. 8.

## 2 Model and Assumptions

In this section, we first detail how the redundancy and majority voting principles are used in our work. Then, we specify the attacker models which are used in the theoretical analysis and on which our mechanism will be evaluated.

978-0-7695-3622-4/09 $25.00 © 2009 IEEE
DOI 10.1109/CCGRID.2009.12

## 2.1 Redundancy and Majority Voting

We assume a model where a computation is divided into *work units*. Each work unit is redundantly assigned to a set of $2m-1$ workers ($m \in \mathbb{N}, m > 1$). We call $m$ the *level of redundancy*. For each vote, the set of workers is randomly selected from the overall population of potential workers. For simplicity, we assume that this population is static, i.e., workers do not enter or leave the system. This assumption is reasonable since every evolving population can be fragmented into a sequence of static snapshots. Furthermore, we allow any fraction smaller than $0.5$ of malicious workers in the population. This assumption is needed later on for deciding which clusters contain the malicious and which the honest workers.

In our model, majority voting is applied in order to determine which result is accepted as correct result: a result is accepted iff there is a majority of workers that returned this result. As it will play an important role in our work, we need to precisely define what majority and its counterpart minority means:

**Definition 1 (Majority and Minority)** *The* majority *of a vote is the strictly largest group of workers that returned identical results. If there is no such group, then there is no majority. All workers that do not belong to the majority of the vote, and only those workers, belong to the* minority.

Majority voting makes redundant result verification more robust because honest workers sometimes fail and return wrong results [11]. But also, majority voting tolerates any number of independent or colluding malicious workers as long as the correct result is returned by the majority. An odd number of redundant requests ($2m-1$) is used, in order to reduce the probability of ties in votes. At this point, we make the reasonable assumption that if two honest workers make both an error in their computation, then their computed results differ.

## 2.2 Attacker models

We assume that malicious workers can efficiently communicate with each other and can rapidly reach collective decisions. This assumption is backed by the *Sybil Attack* scenario [4] where an attacker creates several identities and lets them appear as distinct individuals. As a consequence, malicious workers can collude by deciding which wrong result to return collectively. We call this kind of attacker *colluder* and consider the following two collusion strategies:

UC *Unconditional colluders* try always to collude, i.e., each time a colluder is involved in a vote, he returns the same incorrect result as other colluders in this vote.

CC *Conditional colluders* collude *only* if they know that at least $m$ workers in the vote are colluders. Otherwise they return the correct result.

Conditional colluders cheat in all votes where they know they will succeed. There are only few unlikely cases where they would win but do not collude (e.g., a vote with $m = 3$, two colluders, and two failing honest workers). So, conditional colluders are basically as effective as unconditional colluders. But at the same time, they are harder to detect, because in all votes where they most probably would not win, they behave as if they were honest. However, colluders only know to which other colluders a certain work unit has been assigned, but do in general not know how many honest workers are involved in a vote. Therefore, the CC strategy requires that colluders know the level of redundancy $m$ in order to make their decision[1]. Thus, UC and CC are sophisticated strategies for the scenarios where attackers know $m$ (CC) or not (UC). Although we focus in this paper on UC and CC, the idea on which our mechanism bases is of general nature and applies also to other attacker models.

## 3 Theoretical Analysis

In this section, we describe the main idea of our approach and back it up by means of a probabilistic analysis.

## 3.1 Approach

Our aim is to identify malicious workers within a given set of workers. To this end, we introduce a distinctive feature that reveals the borderline between the set(s) of malicious and the set of honest workers. This distinctive feature is based on the following observation.

For malicious workers, the only effective way to attack the redundancy principle is to collude, i.e., to return collectively the same incorrect result. Colluders will succeed in votes where they hold the majority. But in such votes, all the honest workers will form the minority. So, we count for each pair of workers how often they are together in the same group (majority/minority), and how often they are in opposite groups. As we will see later, the counts for a pair of workers provide statistical evidence about the relation between the two workers, i.e., whether they are both malicious, both honest or one is malicious and one is honest. We use the counts to estimate the distinctive feature, which we call "correlation". This feature has the nice property that, no matter by what collusion strategy, attackers cannot influence the counts for pairs consisting of two honest workers. Moreover, a colluder can put himself in the same group as a honest worker only by withdrawing from collusion.

[1]Note that this advocates a variation in the level of redundancy $m$, because it would make the CC strategy less feasible in practice.

**Table 1. Parameters for computing correlation $p_c$**

| | |
|---|---|
| $m$ | a work unit is requested $2m-1$ times |
| $p_{mal}$ | fraction of colluders in the population |
| $p_f$ | failure probability of a honest worker |
| $p_h$ | probability of picking a honest worker that returns a *correct* result, i.e., $p_h = (1-p_{mal})(1-p_f)$ |
| $p_i$ | probability of picking a honest worker that returns an *incorrect* result, i.e., $p_i = (1-p_{mal})p_f$ |

In the following paragraphs, we analyze the feature and find that the more malicious workers collude, the more the feature helps distinguishing them from honest workers.

## 3.2 Correlation

Given that two specific workers are together in a vote, we define *correlation* $p_c$ as the probability that the two workers are together in the same group (majority/minority) of that vote[2]. That is, if $p_c$ is high for a specific pair of workers, then these two workers are often together in the same group, and not so often in opposite groups. It follows that for a given series of $n$ independent votes where both workers participated in, the number of votes, in which they were in the same group, is binomially distributed with parameters $n$ and $p_c$.

### 3.2.1 Computing correlation

Let $\triangle$ denote the set of workers in the minority of a specific vote, and $\triangledown$ be the set of workers in the majority. Since the events that any two workers $a$ and $b$ are together in the majority/minority of a vote are mutually exclusive, we have:

$$\begin{aligned} p_c :=& p(\{a,b\} \subseteq \triangle \vee \{a,b\} \subseteq \triangledown) && (1) \\ =& p(\{a,b\} \subseteq \triangle) + p(\{a,b\} \subseteq \triangledown) \, . && (2) \end{aligned}$$

This way, we can use enumerative combinatorics to compute $p_c$ for each constellation of workers, i.e., for pairs consisting of two malicious, two honest or one malicious and one honest workers. The computation of $p_c$ depends on certain parameters, which are listed in Table 1. In the following, we will demonstrate how $p_c$ can be computed for a pair of *two honest workers* (we will write for such a pair "honest&honest" or short "h&h"). We will only address the case of unconditional collusion (UC) here. The derivations of the formulas for the pairings malicious&malicious (m&m) and malicious&honest (m&h), as well as for other collusion strategies such as CC can be found in [19].

[2]We do not use the common *correlation coefficient* $\rho$ [14, p. 103], because it measures the linear relationship between quantitative variables.

**Both workers in $\triangle$ (h&h, UC)** If honest workers $a$ and $b$ are together in a vote, they are both in the minority in either of the following cases (numbers refer to equations below):

- both return incorrect results (3) (see Sect. 2.1 for the assumption that the two results differ),
- one of them (4) or both (5) return the correct result, but there are at least as many colluders as correct results in the vote,
- exactly one of them returns the correct result and apart of them there are only failing honest workers (6).

Since these cases are mutual exclusive, we can add them up:

$$\begin{aligned} p(\{a,b\} \subseteq \triangle) =& {p_f}^2 && (3) \\ &+ 2(p_f(1-p_f))P(1) && (4) \\ &+ (1-p_f)^2 P(2) && (5) \\ &+ 2(p_f(1-p_f)){p_i}^{2m-3} \, , && (6) \end{aligned}$$

where $P(k)$ denotes the probability that at least as many colluders are in the vote as honest workers returning the correct result; the $k$ ($k \in \{1,2\}$) specifies how many of the two honest workers $a$ and $b$ return the correct result. To compute $P(k)$, we need to look at the $2m-3$ remaining workers in the vote and sum up the probabilities of all cases where the number of correct results plus $k$ is not higher than the number of colluders. Let index $i$ denote the number of colluders, and $j$ denote the number of additional honest workers in the vote returning the correct result. Then $2m-3-i-j$ is the number of failing honest workers. We find:

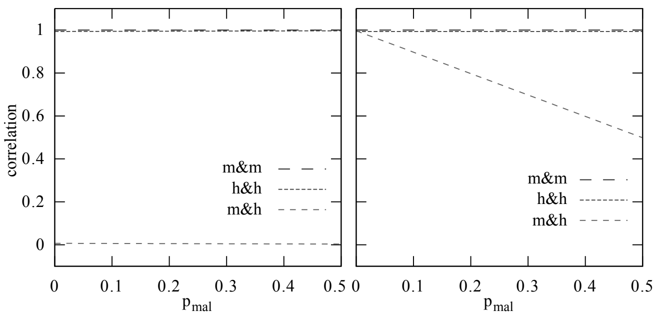
$$P(k) := \sum_{i=k}^{2m-3} \binom{2m-3}{i} {p_{mal}}^i \cdot \sum_{j=0}^{\substack{\min(2m-3-i,\\ i-k)}} \binom{2m-3-i}{j} {p_h}^j {p_i}^{2m-3-i-j} \, . \qquad (7)$$

**Both workers in $\triangledown$ (h&h, UC)** If honest workers $a$ and $b$ are together in a vote, they are both in the majority, iff:

- they both return the correct result ($(1-p_f)^2$), and there are in total more honest workers that return correct results than colluders.

Analogously to the $\triangle$-case, we can add the probabilities for all possible voting outcomes (as above, $i$ is the number of colluders, and $j$ is the number of additional honest workers returning the correct result):

$$p(\{a,b\} \subseteq \triangledown) = (1-p_f)^2 \sum_{i=0}^{m-1} \binom{2m-3}{i} {p_{mal}}^i \cdot \sum_{j=max(i-1,0)}^{2m-3-i} \binom{2m-3-i}{j} {p_h}^j {p_i}^{2m-3-i-j} \, . \qquad (8)$$



(a) UC case (b) CC case

**Figure 1. Correlation ($m = 2$, $p_f = 0.0022$).**

### 3.2.2 Correlation as Distinctive Feature

In this work, we will detail results of computations for a subset of possible parameter settings (see [19] for a more comprehensive analysis). We set the level of redundancy $m$ to the value of 2 (see [1]), i.e., a vote consists of three workers. Furthermore, we use $p_f = 0.0022$ as failure rate of honest workers with reference to [11]. Figure 1 shows correlations computed for the UC and the CC settings. Each figure depicts, for a varying proportion of malicious workers $p_{mal}$, the correlation for m&m, h&h and m&h pairings. Since we want to use the correlation to differentiate malicious and honest workers, we need to check whether an m&h pair of workers can be distinguished from m&m and h&h pairs. In the UC case, the correlation of m&h pairs differs strongly from the correlation of m&m and h&h pairs, almost independently of $p_{mal}$. In the CC case, the correlation of m&h differs increasingly with increasing $p_{mal}$. Note that in the CC case, the probability that malicious workers win a vote decreases also with decreasing $p_{mal}$ – because the probability of having $m$ or more malicious workers in a vote decreases, too. Thus, the more malicious workers collude, the better they are identifiable.

While taking the results from [19] into account, where we also examined other parameter settings and mixed collusion strategies, we can conclude that correlation is either a good feature for identifying the set of colluders, or it is not, but then the colluders cannot sabotage effectively.

## 4 Algorithm

In the following, we first describe the overall procedure of the algorithm for collusion detection (Alg. 1). Then, we explain each of the called subroutines in more detail.

The algorithm takes as input a population of workers $N$, a set containing past voting outcomes $\mathcal{V}_N$ for these workers (i.e., how often pairs of workers were together in the majority/minority), and a parameter for clustering $P$. The

**Algorithm 1** Collusion Detection

```
1: procedure DETECT($N, \mathcal{V}_N, P$)
2:     $M \leftarrow$ COMPUTE_CORRELATION_MATRIX($N, \mathcal{V}_N$)
3:     $G \leftarrow$ CONSTRUCT_GRAPH($M$)
4:     $\{C_1, \ldots, C_k\} \leftarrow$ CLUSTER($G, P$)
5:     $C_{max} \leftarrow \max(C_1, \ldots, C_k)$  ▷ Select largest cluster
6:     $S \leftarrow N \setminus C_{max}$  ▷ Take all but largest cluster
7:     return $S$  ▷ Return IDs of suspects
8: end procedure
```

algorithm proceeds as follows. First, it estimates the correlation for each pair of workers which results in *correlation matrix* $M$ (line 2). $M$ is a symmetric matrix where rows and columns represent workers; an entry $M_{ij}$ is the estimate of the correlation between the two workers $i$ and $j$. We set $M_{ii}$ for all workers to 1 (perfect correlation). In line 3, an undirected weighted complete graph $G$ is constructed, where the workers constitute the vertices and the entries of the correlation matrix are used as edge weights. In a next step, the graph clustering algorithm partitions the graph $G$ into clusters of workers that correlate much (line 4). The largest cluster is supposed to contain the strongly correlating honest workers, so it is selected (line 5) and subtracted from $N$ (line 6). The remaining part of $N$ is returned as set containing the suspect workers (line 7).

In Sect. 4.1, we describe in detail, how the correlation for a pair of workers is estimated (line 2). In Sect. 4.2, it is detailed how graph clustering is used to partition the set of workers into honest and malicious workers (line 4). The complexity of the algorithm is analyzed in Sect. 6.2.

## 4.1 Estimating the Correlation

For each pair of workers, we count in how many votes they were together in the majority or the minority ($c_1$), and how often they were in opposite groups ($c_2$). These numbers form the set $\mathcal{V}_N$. As mentioned in Sect. 3.2, the number $c_1$ is drawn from the binomial distribution with $c_1 + c_2$ many experiments and the actual correlation $p_c$ as success probability. We know that the beta distribution is the prior distribution for binomial proportions [6, p. 34]. As a consequence, the correlation of two workers can be estimated by the beta distribution with parameters $\alpha = c_1 + 1$ and $\beta = c_2 + 1$. Hence, we choose the expected value of this distribution as the *sample correlation* $\hat{p}_c$, which is given by [6]:

$$\hat{p}_c := \frac{\alpha}{\alpha + \beta} = \frac{c_1 + 1}{c_1 + c_2 + 2} . \tag{9}$$

## 4.2 Graph Clustering

An important part of the algorithm consists in the graph clustering method called in line 4. A graph clustering algo-

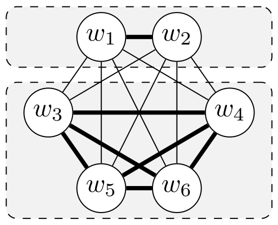


**Figure 2. Clustering of a Graph ($|N| = 6$)**

rithm takes a graph and one or more tuning parameters as input. It removes a set of edges from the graph such that the connected components of the modified graph form the clusters (see also [2]). Graph clustering algorithms for weighted undirected graphs usually try to find disjoint sets of vertices that are strongly connected in terms of the edge weights. In other words, when interpreting edge weights as similarity measures, vertices within the same group should be similar and vertices belonging to different groups should be dissimilar. How this can be achieved depends on the concrete implementation of the algorithm. The parameters passed to a graph clustering algorithm determine the granularity of the clustering, i.e., how much dissimilarity within clusters and similarity in-between clusters is tolerated.

We showed in Sect. 3.2.2 that the correlation for m&m and h&h pairs is higher than the correlation for m&h pairs. This qualifies the correlation estimates to be used as edge weights and so act as a similarity measure. A successful graph clustering then outputs groups that contain only malicious workers, and groups that contain only honest workers. Figure 2 illustrates a clustering for a very small graph, where thick edges indicate strong and thin edges weak correlation. Since we assumed that more than $50\%$ of the workers are honest (see Sect. 2.1), the biggest cluster will very likely contain the "heavily connected" honest workers. Our algorithm outputs the union of all smaller clusters as set of suspect workers. Therefore, our algorithm can also deal with several independent groups of colluders. This also implies that a collusion strategy, where m&m pairs keep the correlation low in order to become indistinguishable from m&h pairs, will not work.

## 5 Implementation

The implementation of the main part of Alg. 1 is straightforward and is not detailed here. We rather want to focus on the graph clustering algorithm called in line 4.

Several approaches for clustering undirected weighted graphs have been proposed in literature (e.g., see [2]). For our experiments, we chose two different algorithms, the *Markov Cluster Algorithm (MCL)* [23] and *Mininmum Cut Tree Clustering* [7], which we will call *MinCTC*. In [2], MCL was shown to be more accurate than several other algorithms. The authors did not test MinCTC and so we compare it to MCL in this work. In the following, we describe the choice of clustering parameters for the two algorithms.

### 5.1 MCL

MCL is based on the simulation of stochastic flows in graphs. The only parameter for MCL is the *inflation parameter* $I \in [2, 30]$ that has an impact on the level of granularity of obtained clusterings. The evaluation in Sect. 6.1 shows that the accuracy of the algorithm does not depend much on an exact choice of this parameter.

To improve the accuracy, we propose to preprocess the correlation values $\hat{p}_c$ with an additional parameter $\theta$. Since MCL uses a probabilistic interpretation of the edge weights, the relative differences between the correlation values can be amplified by raising them to the power of some $\theta > 1$.

### 5.2 MinCTC

MinCTC is based on the construction of *minimum-cut trees* in graphs. In [9], the authors experimentally compared different approaches for constructing mincut trees. In our work, we use the *GHs* implementation (a variant of the Gomory-Hu algorithm [22] using the source selection heuristic) because it turned out to be more robust than other codes [9].

Through experimentation we found that depending on the setting, there is a particular range of suitable parameters. Below this range, the algorithm returns only one single cluster containing all workers (underfitting). Above this range, it returns as many clusters as workers, i.e., each worker makes up its own cluster (overfitting). So, only for parameters within the range, the algorithm may find suitable clusterings – if it is assumed that there are colluders among the workers. Therefore, we implemented *binary search* to find a parameter that lies in this range and produces a specified number of clusters $k$ (we used $k = 2$). If the search is not successful, all workers are classified as honest (to avoid false positives). As for MCL, we preprocessed the correlation values in the experiments with different exponents $\theta$.

## 6 Evaluation

The algorithm is evaluated in a setting as described in Sect. 3.2.2 for a set of $|N| = 100$ workers and $p_{mal} = 0.1$. It should be reemphasized here, that the feasibility of the algorithm was shown theoretically for the other settings as well [19]. Each run was averaged over $10^3$ randomly generated sets of voting outcomes $\mathcal{V}_N$. Each $\mathcal{V}_N$ is based on a certain number of observed votes – for the simulation

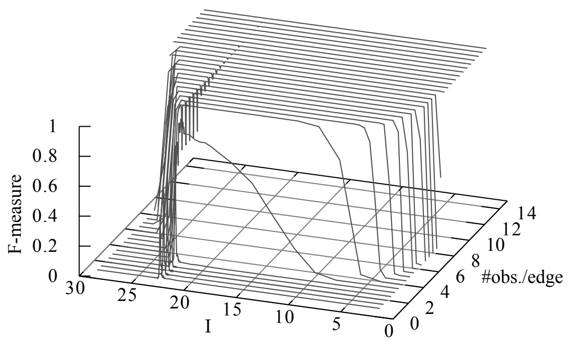


**Figure 3. Accuracy with MCL ($\mathrm{UC}$, $\theta = 1$)**

results we indicate the average number of observations per edge in the graph (#obs./edge). In the following, we first evaluate the algorithm's accuracy. Secondly, we analyze the complexity of the algorithm and compare the running times of the two graph clustering algorithms MCL and MinCTC.

## 6.1 Accuracy

We measure the accuracy of Alg. 1 for MCL and MinCTC and the two cases UC and CC. As is customary, we label the output of the algorithm with "true positive" (*tp*), "true negative" (*tn*), "false positive" (*fp*) or "false negative" (*fn*). As accuracy measure, we use the f1-Measure [15], which is computed as follows:

$$\mathrm{f1} := \frac{2 \cdot \#tp}{2 \cdot \#tp + \#fp + \#fn} \tag{10}$$

An f1-Measure of 1 occurs iff our algorithm returns a set of suspects that contains all colluders but no honest workers.

### 6.1.1 Using MCL

Figure 3 shows the accuracy of the algorithm using MCL for the UC case. No amplification $\theta$ has been applied. An average of $\mathrm{f1} = 1$ first occurs for an average number of 4 observations per edge and inflation parameter $I$ between ca. 13 and 22.8. For more than 10 obs./edge, the algorithm provides perfect accuracy for all possible parameters. MCL showed problems for a small range of parameters $22.8 < I < 23.3$. For this set of parameters, MCL did not always converge before 5 seconds, and returned a particularly high ratio of false positives, which points to overfitting.

Figure 4 shows the accuracy of the algorithm using MCL for the CC scenario. An almost perfect precision of $\mathrm{f1} = 1$ on average is not achieved before around 60 obs./edge. The same high ratio of *fp* as for the UC case is observed here for $22.5 < I < 23.3$. For smaller $\theta$ the accuracy became worse, and for $\theta = 1$ the algorithm was not able to detect colluders.

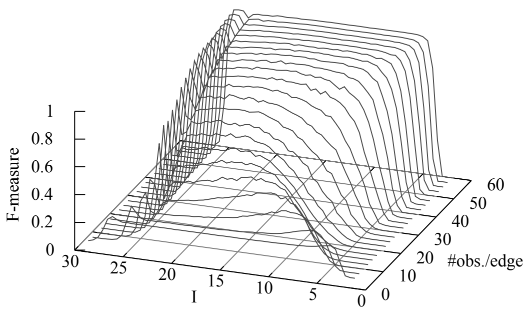


**Figure 4. Accuracy with MCL ($\mathrm{CC}$, $\theta = 20$)**

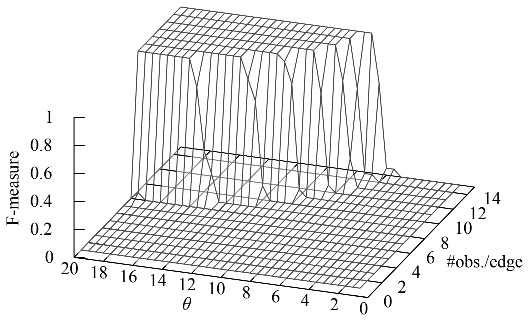


**Figure 5. Accuracy with MinCTC ($\mathrm{UC}$)**

To summarize, when using MCL, our algorithm can cope both with the UC and the CC case. It works for broad sets of parameter $I$ and improves for an increasing number of observations. For the CC case, a very high number of observations is needed until reasonable accuracy is reached.

### 6.1.2 Using MinCTC

Figure 5 shows the accuracy of the MinCTC variant of Alg. 1 for varying levels of amplification $\theta$. The algorithm does not succeed for less than 8 obs./edge. Also for a higher number of observations, still a preprocessing of the correlation values with a rather high exponentiation $\theta$ is needed until the algorithm succeeds. As an advantage of this tentative classification we found that in our experiments the algorithm did not produce false positives.

The accuracy of the algorithm using MinCTC for the CC scenario is not shown, because for up to 200 obs./edge and varying $\theta$ it throughout performed with an average of $\mathrm{f1} = 0.0$. Apparently, MinCTC could not cope with cases where m&h pairs differ only slightly from other types of pairs.

To summarize, when using MinCTC, the algorithm's accuracy was poor in the UC case for small numbers of observations or suboptimal choices of the amplification parameter. In the CC case, no colluders could be detected. However, the performance of MinCTC could possibly be improved by a better choice of the clustering parameter (which seems to be very tricky).

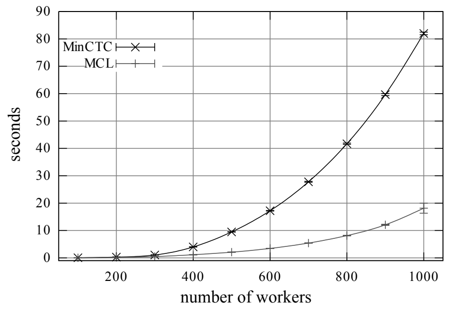


**Figure 6. Running times of MCL and MinCTC**

## 6.2 Complexity and Running Time

Lines 2 to 3 of Alg. 1 comprise computations for each pair of workers, and thus lie in $\mathcal{O}(|N|^2)$, where $N$ is the population of workers. Line 4 depends on the concrete implementation of the used graph clustering algorithm. MCL's complexity is essentially dominated by matrix multiplication [2] and is in $\mathcal{O}(|N|^3)$, but can implementation wise be brought to worst case $\mathcal{O}(|N|k^2)$ [23], where $k$ is a pruning constant smaller than $|N|$. MinCTC relies on maximum flow algorithms which are also polynomial in complexity [7]. Lines 5 to 7 of Alg. 1 can be computed in $\mathcal{O}(|N|)$.

We experimentally compared the running times of MCL and MinCTC for different sizes of $N$. For the tests, both algorithms took their inputs from a file but did not write any output. The input mainly consists of the sample correlation for each pair of workers, and so the file size $f$ grows nearly quadratic with the number of vertices in a graph. However, the two algorithms worked on files with the same size and thus the difference in their running time was left unaffected. Figure 6 shows the running times of MCL and MinCTC in seconds for the same setting (UC, 10 obs./edge); experiments were run on a 2.33 GHz Linux machine with an Intel Xeon CPU and 3GB RAM. Although MCL was shown in [2] to be very slow, it still clearly outperforms MinCTC and its complexity seems to grow considerably slower.

## 7 Related Work

Silaghi et al. [18] proposed an algorithm for detecting *collusion*. They consider three different types of malicious workers: first, naive workers that randomly return incorrect results with a certain probability; secondly, workers that only return incorrect results if they know that they hold the majority in a vote; and third, a mixture of the first two types. Their core mechanism counts for each worker the number of votes it got against in past votes. Then, as the probability of getting votes against is different for naive workers, this type and partly the third type can be identified and removed from the votes. In order to spot the second type of colluding workers, they again apply the redundancy principle to test workers in the remaining "conflicting" votes. At this point, we face again the problem of collusion. Opposed to that, our collusion detection mechanism fully does without making use of redundancy checks.

Sarmenta [17] proposed to combine redundancy with *spot-checking*. In spot-checking, workers are regularly assigned work units for which the correct result is already known. Workers that get caught returning incorrect results are blacklisted and their results are ignored (if identities can be checked). In their model, attackers are described by a Bernoulli process and return correct and incorrect results with a specific probability. Assuming this attacker model, the author attains probabilistically guaranteed levels of correctness. However, their model does not consider attackers that follow more sophisticated strategies such as collusion.

Zhao et al. [26] proposed the scheme "Quiz" for result verification in peer-to-peer grids. In this scheme, a worker delegates a whole package of work units to a single worker. In such a package, quizzes are interspersed for which the correct result has been precomputed (as in spot-checking). If one of the quizzes is answered incorrectly, the whole package is discarded, the trustworthiness of the respective worker is reduced and it is possibly blacklisted. Under collusion, Quiz outperforms redundancy; in the absence of collusion however, Quiz performs worse. This argues for a combination of redundancy with collusion detection, in cases where collusion cannot be precluded.

Roch and Varrette [16] proposed a spot-checking mechanism for dependent tasks in the scenario of divide&conquer computations. Independently of the attacker model, they reach a user-defined error probability bound for detecting massive attacks, where the bound depends on the minimum ratio of malicious workers in the population. Again, correct results for the spot-checks are assumed to be recomputed reliably. Du et al. [5] proposed an effective spot-checking mechanism for scenarios where many computations have to be performed but only few of the results have to be returned. In their approach, a worker constructs a Merkle-tree that combines the hashes of all results belonging to a specific task. The worker commits the root of the tree to the master, who can afterwards spot-check single results.

*Game-theory* and *mechanism design* were proposed to prevent malicious behavior of workers by making it unprofitable for them to act maliciously (e.g., [10, 25]). These approaches assume that workers want to maximize some (financial) profit and act rational on this score. However, in systems where workers participate voluntarily (e.g., [8, 1, 3]), "rational behavior" is if any hard to define. But also in commercial settings, workers may be maliciously destructive or act with an unknown rationality behind. For these cases, game-theoretic analysis and mechanism design are inadequate.

## 8 Conclusion & Future Work

We presented an easy-to-implement collusion detection algorithm for grid computing that identifies colluding workers on the basis of correlated outcomes in votings. It was shown for different collusion strategies that correlation is a good feature for distinguishing honest workers from malicious ones. We found that the more malicious workers collude, the more they become identifiable. In an experimental study, we evaluated our algorithm and showed that accuracy strongly depends on the number of observed voting outcomes and the used graph clustering algorithm. The latter also dominates the runtime complexity. Using the MCL cluster algorithm, our algorithm performs well for the UC case. However, a large number of observed votes is needed for getting reasonable results in the CC case, which can be precluded though by varying the level of redundancy.

To improve accuracy and runtime, one could try to replace the graph clustering component, which is used for analyzing the structure of correlation among workers, for instance by *pattern recognition* methods. Concerning the integration of our algorithm into existing grid architectures, parallelization would be an interesting issue in order to run it in a distributed fashion with the help of trusted workers.

## References

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[2] U. Brandes, M. Gaertler, and D. Wagner. Experiments on graph clustering algorithms. In *Proc. of the 11th Annual European Symposium on Algorithms*, volume 2832 of *LNCS*, pages 568–579. Springer, 2003.

[3] distributed.net projects, February 2009. `http://www.distributed.net/`.

[4] J. R. Douceur. The sybil attack. In *Revised Papers from the 1st Int. Workshop on Peer-to-Peer Systems (IPTPS '01)*, pages 251–260. Springer Verlag, 2002.

[5] W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *Proc. of the 24th Int. Conf. on Distributed Computing Systems (ICDCS'04)*, pages 4–11. IEEE Computer Society, 2004.

[6] M. Evans, N. Hastings, and B. Peacock. *Statistical Distributions*. John Wiley & Sons, Inc., New York, NY, USA, third edition, 2000.

[7] G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4), 2004.

[8] Folding@home distributed computing, February 2009. `http://folding.stanford.edu/`.

[9] A. V. Goldberg and K. Tsioutsiouliklis. Cut tree algorithms: An experimental study. *Journal of Algorithms*, 38(1):51–83, 2001.

[10] P. Golle and S. G. Stubblebine. Secure distributed computing in a commercial environment. In *Proc. of the 5th Int. Conf. on Financial Cryptography (FC '01)*, pages 289–304. Springer Verlag, 2002.

[11] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. M. Silva, G. Fedak, and F. Cappello. Characterizing result errors in internet desktop grids. In *Proc. of the 13th European Conf. on Parallel Processing (EURO-PAR '07)*, volume 4641 of *LNCS*, pages 361–371. Springer Verlag, 2007.

[12] D. Kondo, M. Taufer, C. L. Brooks, H. Casanova, and A. A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *Proc. of the Int. Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 26–35. IEEE Computer Society, 2004.

[13] D. Molnar. On patrol: The SETI@home problem. *ACM Crossroads: E-commerce*, 7(1), September 2000.

[14] D. S. Moore. *The Basic Practice of Statistics*. W. H. Freeman & Co., New York, NY, USA, fourth edition, 2007.

[15] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 1979.

[16] J.-L. Roch and S. Varrette. Probabilistic certification of divide & conquer algorithms on global computing platforms: application to fault-tolerant exact matrix-vector product. In *Proc. of the Int. Wksh. on Parallel Symbolic Computation (PASCO '07)*, pages 88–92. ACM, 2007.

[17] L. F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. In *Proc. of the 1st Int. Symposium on Cluster Computing and the Grid (CCGrid '01)*, pages 337–346. IEEE Computer Society, 2001.

[18] G. C. Silaghi, F. Araujo, L. M. Silva, P. Domingues, and A. Arenas. Defeating colluding nodes in desktop grids computing platforms. In *Proc. of the 2nd Workshop on Desktop Grids and Volunteer Computing (PCGrid '08)*. IEEE Computer Society, 2008.

[19] E. Staab, V. Fusenig, and T. Engel. Using correlation for collusion detection in grid settings. Technical Report 000657499, University of Luxembourg, July 2008.

[20] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, Upper Saddle River, New Jersey, USA, third edition, 2003.

[21] M. Taufer, D. Anderson, P. Cicotti, and C. L. B. III. Homogeneous redundancy: a technique to ensure integrity of molecular simulation results using public computing. In *Proc. of the 19th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS '05) - Workshop 1*, page 119.1, Washington, DC, USA, 2005. IEEE Computer Society.

[22] R. G. und T.C. Hu. Multi-terminal network flows. *Journal of SIAM*, 9:551–570, 1961.

[23] S. van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000.

[24] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, 1997.

[25] M. Yurkewych, B. N. Levine, and A. L. Rosenberg. On the cost-ineffectiveness of redundancy in commercial P2P computing. In *Proc. of the 12th Conf. on Computer and Communications Security (CCS '05)*, pages 280–288. ACM, 2005.

[26] S. Zhao, V. Lo, and C. GauthierDickey. Result verification and trust-based scheduling in peer-to-peer grids. In *Proc. of the 5th IEEE Int. Conf. on Peer-to-Peer Computing (P2P '05)*, pages 31–38. IEEE Computer Society, 2005.