Uncheatable Grid Computing*

Wenliang Du, Jing Jia, Manish Mangal, and Mummoorthy Murugesan
Department of Electrical Engineering and Computer Science
Syracuse University, Syracuse, NY 13244-1240, USA
Tel: +1 315 443-9180   Fax: +1 315 443-1122
Email: {wedu,jijia,mkmangal,mmuruges}@syr.edu

Abstract

Grid computing is a type of distributed computing that has shown promising applications in many fields. A great concern in grid computing is the cheating problem described in the following: a participant is given $D = \{x_1, \dots, x_n\}$, it needs to compute $f(x)$ for all $x \in D$ and return the results of interest to the supervisor. How does the supervisor efficiently ensure that the participant has computed $f(x)$ for all the inputs in D, rather than a subset of it? If participants get paid for conducting the task, there are incentives for cheating. In this paper, we propose a novel scheme to achieve the uncheatable grid computing. Our scheme uses a sampling technique and the Merkle-tree based commitment technique to achieve efficient and viable uncheatable grid computing.

1. Introduction

The increasing complexity of computations, better processing power of the personal computers and the ever increasing reach and speed of the Internet have laid down the path for grid computing. *Computational grid* is a novel, evolving infrastructure that provides unified, coordinated access to computing resources such as processor cycles, storage, etc. Wide variety of systems, from small workstations to supercomputers can be linked to a grid to form a powerful virtual computer. All the complexities involved in managing resources of a grid are hidden from the clients, providing a seamless access to computing resources. As a great advancement towards cost reduction, computational grids can be used as a replacement for supercomputers that are presently used in many computationally intensive scientific problems [1, 7].

* This work was supported in part by Grant IIS-0219560 and IIS-0312366 from the United States National Science Foundation and by the CASE Center at Syracuse University.

The class of problems dealt by grid computing are those which involve tremendous computations and can be broken down into independent tasks. A general grid computing environment includes a *supervisor* and a group of *participants* who allow the idle cycles of their processors to be used for the computations. The participants are totally ignorant of each other and after completing their tasks report back the results to the supervisor.

Past few years have seen a tremendous growth in grid computing with its effect being felt in the biotechnology industry, entertainment industry, financial industry, etc. The success of the projects like SETI@home [2], IBM smallpox research [3], GIMPS [4] has made the potential of grid computing visible.

For instance, IBM's smallpox research [3] uses grid computing to find potential drugs to counter the smallpox virus. Its main task is to screen hundreds of thousands of molecules, a task that can take years even with supercomputers. By downloading and running the software, participants can add their CPUs to the global grid. Every time their computers are idle, the computing resources can be contributed to the grid, accelerating the screening process while dramatically reducing the cost of the project. The result is that rather than spending years, it will be possible to screen hundreds of millions of molecules in just months. Another highly-profiled grid computing project is SETI@home [2], which is a scientific experiment that uses Internet-connected computers in the Search for Extraterrestrial Intelligence (SETI). SETI@home has more than 4.5 million users contributing their computers' unused processing power, to form a 15 Teraflops grid, faster than IBM's most powerful supercomputer *ASCI White* (12 Teraflops). Also the cost of the SETI grid is only 500K dollars whereas ASCI White costs 110 million dollars [2].

However the untrusted environments in which the computations are performed tend to cast suspicion on the veracity of the results returned by the participants. The participant may not have performed the necessary computations but claims to have done so. This cheating behavior, if undetected, may render the results useless. Project managers from SETI@home have reportedly uncovered attempts by some users "to forge the amount of time they have donated in order to move up on the Web listings of top contributors" [5]. Yet SETI participants are volunteers who do not get paid for the cycles they contribute. When participants are paid for their contribution, they have strong incentives to cheat for maximizing their gain.

Therefore, we need methods to detect the cheating behaviors in grid computing. We formulate the problem of *uncheatable grid computing* in the following:

Problem 1 *(*Uncheatable Grid Computing*)* A participant is assigned a task consisting of computing $f(x)$ for all the inputs $x \in D = \{x_1, \ldots, x_n\}$, where $n = |D|$; the participant needs to return the results of interest to the supervisor. A dishonest participant might compute $f(x)$ for only $x \in D'$, where D' is a subset of D, but claims to have computed f for all the inputs. *How does the supervisor efficiently detect whether the participant is telling a truth or a lie?*

A straightforward solution is to *double-check* every result. The supervisor can assign the same task to more than one participant and compare their results. This simple scheme leads to the wastage of processor cycles that are precious resources in grid computing. Moreover, it introduces $O(n)$ communication cost for each participant. Note that in grid computing, the supervisor only needs the participant to return the results of interest, which is usually a very small number compared to n. Therefore $O(n)$ overhead is substantial.

An improved solution is to use *sampling* techniques. The supervisor randomly selects a small number of inputs from D (we call these randomly selected inputs samples or sample inputs); it only double-checks the results of these sample inputs. If the dishonest participant computes only one half of the inputs, the probability that it can successfully cheat the supervisor is one out of 2^m, where m is the number of samples. If we make m large enough, e.g. $m = 50$, the cheating is almost impossible. This solution has a very small computational overhead ($O(m)$), because $m \ll n$. However, this scheme still suffers from the $O(n)$ communication cost because it requires the participant to send all the results back to the supervisor, including those that are of no interest to the supervisor. To improve this situation, we have developed a Commitment-Based Sampling (CBS) scheme. Our scheme reduces the communication overhead to $O(m \log n)$. Because n is usually large (e.g., $n = 2^{40}$), this result is a substantial improvement.

1.1. Related Work

To defeat cheating in grid computing, Golle and Mironov proposed a ringer scheme [8]. In the ringer scheme, the supervisor sends to the participant some pre-computed results, without disclosing the corresponding inputs. The participant must find out those secret inputs. Golle and Mironov have shown that by selecting the secret inputs in proper ways, the chance for a participant to cheat successfully is slim. This scheme is generally referred to as the *ringer* scheme. The ringer scheme assumes that finding the secret inputs from the pre-computed results is no easier than using the brute-force approach to try all the inputs. Therefore the function f must have the one-way property, i.e., it is difficult to find x from $f(x)$. The ringer scheme is thus restricted to computations that have such a one-way property and it cannot be applied to generic computations.

Szada, Lawson, and Owen extend the ringer scheme to deal with other general classes of computations, including optimization and Monte Carlo simulations [10]. They propose effective ways to choose ringers for those computations. It is still unknown whether the schemes proposed in [10] can be extended further to generic computations.

2. Problem Definition

2.1. Model of Grid Computing

We consider a grid computing in which *untrusted participants* are taking part. The computation is organized by a *supervisor*. Formally, such computations are defined in our model by the following elements:

- **A function $f : X \mapsto T$ defined on a finite domain X.** The goal of the computation is to evaluate f on all $x \in X$. For the purpose of distributing the computation, the supervisor partition X into subsets. The evaluation of f on subset X_i is assigned to participant i.
- **A screener S.** The screener is a program that takes as input a pair of the form $(x; f(x))$ for $x \in X$, and returns a string $s = S(x; f(x))$. S is intended to screen for "valuable" outputs of f that are reported to the supervisor by means of the string s. We assume that the run-time of S is of negligible cost compared to the evaluation of f.

2.2. Models of Cheating

A participant can choose to cheat for a variety of reasons. We categorize the cheating using the following two models. We assume that the participant is given a domain $D \subset X$, and its task is to compute $f(x)$ for all $x \in D$. From now on, we use D as the domain of f for the participant.

- *Semi-Honest Cheating Model:* In this model, the cheating participant follows the supervisor's computations with one exception: for $x \in \check{D} \subset D$, the participant uses $\check{f}(x)$ as the result of $f(x)$. Function $\check{f}$ is usually much less expensive than function f; for instance, $\check{f}$ can be a random guess. The goal of the cheating participant in this model is to reduce the amount of computations, such that it can maximize its gain by "performing" more tasks during the same period of time.

- *Malicious Cheating Model:* In this model, the behavior of the participant can be arbitrary. For example, a malicious participant might have calculated function f on all $x \in D$, but when it computes the screener function S, instead of computing $S(x; f(x))$, it might compute $S(x; z)$, where z is random number. In other words, the participant intentionally returns wrong results to the supervisor, for the purpose of disrupting the computations.

To maximize their gains, rational cheaters tend to use minimal cost to falsify the contributions they have never made. Their behaviors fall into the semi-honest cheating model. Therefore, in this paper, we focus on the *semi-honest* cheating model.

2.3. Definition of Uncheatable Grid Computing

Assume that a participant is assigned a task that consists of computing $f(x)$ for all $x \in D$, where $D = \{x_1, \ldots, x_n\}$. If a participant computes the function f only on $x \in D'$, where $D' \subseteq D$, we define the *honesty ratio* r as the value of $\frac{|D'|}{|D|}$. When the participant is fully honest, the honesty ratio is $r = 1$; otherwise $r < 1$.

Definition 2.1 (Uncheatable Grid Computing) Let $Pr(r)$ be the probability that a participant with honesty ratio r can cheat without being detected by the supervisor. Let $C_{cheating}$ be the expected cost of successful cheating, and C_{task} be the overall computation cost of the required task. We say a grid computing is *uncheatable* if one of the following or both inequalities are true:

$$\begin{array}{ll} & Pr(r) < \varepsilon, \text{for a given } \varepsilon (0 < \varepsilon \leq 1) \\ \text{or} & C_{cheating} > C_{task}. \end{array}$$

3. The Commitment-Based Sampling Scheme

The naive sampling scheme can solve the uncheatable grid computing problem with efficient computation cost, but it requires expensive $O(n)$ communication cost. To each participant, this cost might not be too high, but to the supervisor, the cost might be overwhelming. For example, if the task of grid computing is to break a 64-bit password using the brute force method, the total communication cost at the supervisor side is $O(2^{64})$, which is about 16 million terabytes. Very few networks can handle such a heavy network load.

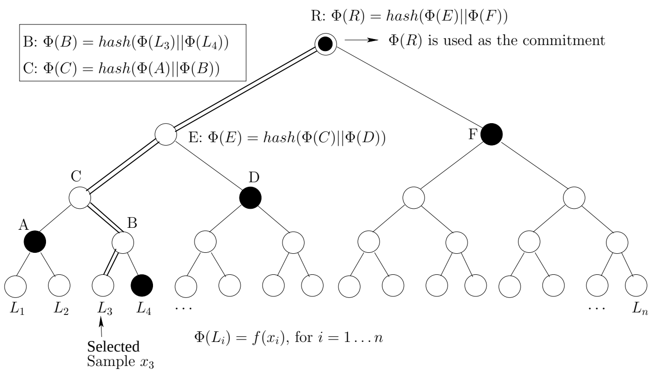
R: $\Phi(R) = hash(\Phi(E)||\Phi(F))$
B: $\Phi(B) = hash(\Phi(L_3)||\Phi(L_4))$
C: $\Phi(C) = hash(\Phi(A)||\Phi(B))$
$\Phi(R)$ is used as the commitment
E: $\Phi(E) = hash(\Phi(C)||\Phi(D))$
F
C
D
A
B
L_1
L_2
L_3
L_4
$\cdots$
$\cdots$
L_n
Selected Sample x_3
$\Phi(L_i) = f(x_i)$, for $i = 1 \ldots n$

Figure 1. CBS Scheme: the Merkle Tree and the Verification

Is it possible not to require each participant to send all the outputs? Or is it possible just to ask the participant to send the results for those sample inputs? The solution is non-trivial because we have to prevent the participant from computing the results for the sample inputs after it learns which inputs are samples. For example, if x_k is selected as a sample, the supervisor needs $f(x_k)$ from the participant to check whether the participant has correctly calculated $f(x_k)$. However, without a proper security measure, the participant, who has not computed $f(x_k)$, can always compute it after learning x_k is a sample. This defeats the purpose of sampling.

One way to solve the above problem is to use *commitment*. Before the participant knows that x_k is a sample, it needs to send the commitment for $f(x_k)$ to the supervisor. Once the participant commits, it cannot change $f(x_k)$ without being caught. The supervisor then tells x_k to the participant, which has to reply with the original value of $f(x_k)$ that was committed. Since any input has equal probability to become a sample, this means the participant has to commit all the results for those n inputs; how can it be done efficiently? Obviously the participant cannot afford to send the commitment for each single input, because the $O(n)$ communication cost makes it no better than the naive sampling scheme. The participant cannot hash all these n results together to form one single commitment either; although this method achieves the commitment for all results, it makes verifying a single result difficult because to do that, the supervisor needs to know all the other $n - 1$ results.

In summary, we need a commitment scheme that (1) allows all the n results to be committed efficiently, and (2) allows the verification of each single result to be performed efficiently. We use the Merkle Tree [9] to achieve these goals.

3.1. The Commitment-Based Sampling Scheme

The Merkle tree (also called hash tree) is a complete binary tree equipped with a function $hash$ and an assignment Φ, which maps a set of nodes to a set of a fixed-size strings. In a Merkle tree, the leaves of the tree contain the data, and the Φ value of an internal tree node is the hash value of the concatenation of the Φ values of its two children.

To build a Merkle tree for our problem, the participant constructs n leaves $L_1, \dots, L_n$. Then it builds a complete binary tree with these leaves. The Φ value of each node is defined as the following (we use V to denote an internal tree node, and V_{left} and V_{right} to denote V's two children):

$$\begin{aligned} \Phi(L_i) &= f(x_i), \text{ for } i = 1, \dots, n \\ \Phi(V) &= hash(\Phi(V_{left}) || \Phi(V_{right})), \end{aligned} \quad (1)$$

where "$||$" represents the concatenation of two strings, and the function $hash$ is a one-way hash function such as MD5 or SHA. To make a commitment on all the data on the leaves, the participant just needs to send $\Phi(R)$ to the supervisor, where R is the root of the Merkle tree. Fig. 1 depicts an example of the Merkle tree built for our purpose.

After receiving the commitment, the supervisor randomly selects a number of samples, and sends them to the participant. The participant needs to provide the evidence to show that, before making the commitment, it has already computed f for those samples. Let x be a sample, and L be x's corresponding leaf node in the tree. Let λ denote the path from L to the root (not including the root), and let H represent the length of the path. In order to prove its honesty regarding $f(x)$, the participant sends $f(x)$ to the supervisor; in addition, for each node $v \in \lambda$, the participant also sends $\Phi(v\text{'s sibling})$ to the supervisor. We use $\lambda_1, \dots, \lambda_H$ to represent these Φ values.

To verify the participant's honesty on sample x, the supervisor first verifies the correctness of $f(x)$. If $f(x)$ sent by the participant is incorrect, the participant is caught cheating immediately. Even if $f(x)$ from the participant is correct, it cannot prove the participant's honesty because the participant, who did not compute $f(x)$, can compute the correct $f(x)$ after knowing x is the sample. The supervisor uses the commitment $\Phi(R)$ made by the participant to ensure that the correct $f(x)$ is used at the time of building the Merkle tree. To achieve this, the supervisor uses $f(x)$ (correct) and $\lambda_1, \dots, \lambda_H$ to reconstruct the root of the Merkle tree R', thus getting $\Phi(R')$. Only if $\Phi(R') = \Phi(R)$, will the supervisor trust that the participant has correctly computed $f(x)$ before building the Merkle tree. The communication cost of this process is proportional to the height of the tree. Because the Merkle tree is a complete binary tree with n leaves, its height is $O(\log n)$, where $n = |D|$.

We demonstrate how the verification works using an example depicted in Fig. 1. Assume that x_3 is selected as an sample, whose corresponding leaf node in the tree is L_3. The participant finds the path from L_3 to the root (depicted by the double lines). Then the participant sends to the supervisor $f(x_3)$ and all the Φ values of the sibling nodes (L_4, A, D, and F) along the path. The sibling nodes are depicted by the black nodes in the figure. To verify whether, before committing $\Phi(R)$, the participant has computed $f(x_3)$ or not, the supervisor first makes sure $f(x_3)$ is correct. Then the supervisor reconstructs the root R' from $f(x_3)$, $\Phi(L_4)$, $\Phi(A)$, $\Phi(D)$, and $\Phi(F)$.[1] If $\Phi(R') = \Phi(R)$, we can say that the participant knows $f(x_3)$ before building the Merkle tree.

We call the scheme described above the Commitment-Based Sampling (CBS) scheme. Its steps are described in the following:

Step 1: Building Merkle Tree. Using Eq. (1), the participant builds a Merkle tree with leaf nodes $L_1, \dots, L_n$, and $\Phi(L_i) = f(x_i)$, for $i = 1, \dots, n$. The participant then sends $\Phi(R)$ to the supervisor.

Step 2: Sample Selection. The supervisor randomly generates m numbers (i_1, ..., i_m) in domain $[1, n]$, and sends these m numbers to the participant. These numbers are the sample inputs.

Step 3: Participant's Proof of Honesty. For each $i \in \{i_1, \dots, i_m\}$, the participant finds the path λ from the leaf node L_i to the root R; then for each node $v \in \lambda$, the participant sends to the supervisor $\Phi(v\text{'s sibling})$. These Φ values are denoted by $\lambda_1, \dots, \lambda_H$. The participant also sends $f(x_i)$ to the supervisor.

Step 4: Supervisor's Verification. For each $i \in \{i_1, \dots, i_m\}$, the supervisor verifies whether $f(x_i)$ from the participant is correct.

1. If $f(x_i)$ is incorrect, the verification stops and the participant is caught cheating.
2. If $f(x_i)$ is correct, using the recursive procedure defined in Eq. (1), the supervisor reconstructs the root $\Phi(R')$ of the hash tree from $f(x_i)$ and $\lambda_1, \dots, \lambda_H$. If $\Phi(R) \neq \Phi(R')$, the verification stops and the participant is caught cheating. If $\Phi(R) = \Phi(R')$, the verification succeeds for the sample i.

If the above verification succeeds for all $i \in \{i_1, \dots, i_m\}$, the supervisor is convinced that, with high probability, the participant has not cheated.

1 The reconstruction of R' can be conducted using the following procedure: with $f(x_3)$ and $\Phi(L_4)$, we can compute $\Phi(B)$; then with $\Phi(A)$, we can compute $\Phi(C)$; then with $\Phi(D)$, we can compute $\Phi(E)$; finally we compute $\Phi(R')$ from $\Phi(E)$ and $\Phi(F)$.

To verify whether $f(x_i)$ is correct does not necessarily mean that the supervisor has to re-compute $f(x_i)$. There are many computations whose verification is much less expensive than the computations themselves. For example, factoring large numbers is an expensive computation, but verifying the factoring results is trivial.

Regarding the communication cost, for each sample, the participant needs to send $O(\log n)$ data to the supervisor. Therefore, the total communication overhead for m samples is $O(m \log n)$.

3.2. Security Analysis

In the following theorem, we use L to denote the input x's corresponding leaf node. We use T to denote the Merkle tree built by the participant, and we use R to denote the root of the tree.

Let λ be the path from the leaf L to the root R, and let $\lambda_1, \ldots, \lambda_H$ represent the Φ values of the sibling nodes along the path λ. According to the property of the Merkle tree, $\Phi(R)$ can be computed using $\Phi(L)$ and $\lambda_1, \ldots, \lambda_H$. We use $\Lambda(\Phi(L), \lambda_1, \ldots, \lambda_H) = \Phi(R)$ to represent this calculation, where $\Phi(R)$ is already committed to the supervisor by the participant.

Theorem 1 (Soundness). *If the participant indeed has computed $f(x)$ at the time of building the Merkle tree, it should succeed in proving its honesty on x.*

Proof. If the participant is indeed honest, according to the CBS scheme, when building the Merkle tree, we have $\Phi(L) = f(x)$. Therefore, during the verification, the supervisor gets

$$\begin{aligned} \Phi(R') &= \Lambda(f(x), \lambda_1, \ldots, \lambda_H) \\ &= \Lambda(\Phi(L), \lambda_1, \ldots, \lambda_H) \\ &= \Phi(R). \end{aligned}$$

Therefore, according to the CBS scheme, the participant succeeds in proving its honesty on x. ∎

Theorem 2 (Uncheatability). *If the participant is dishonest on $f(x)$, i.e., when building the Merkle tree, $\Phi(L) \neq f(x)$. Using the CBS scheme, it is computationally infeasible for the participant to convince the supervisor that it knows $f(x)$ when building the Merkle tree.*

Proof. According to the CBS scheme, the participant sends $f(x)$ and $\lambda'_1, \ldots, \lambda'_H$ to the supervisor. After verifying the correctness of $f(x)$, the supervisor uses $\Phi(R') = \Lambda(f(x), \lambda'_1, \ldots, \lambda'_H)$ to reconstruct the root (denoted by R') of the tree. The supervisor believes that the participant is honest on $f(x)$ only if $\Phi(R') = \Phi(R)$.

If the participant is dishonest and $\Phi(L) \neq f(x)$, to cheat successfully, the participant must find $\lambda'_1, \ldots, \lambda'_H$, such that

$$\begin{aligned} & \Lambda(f(x), \lambda'_1, \ldots, \lambda'_H) \\ = \; & \Lambda(\Phi(L), \lambda_1, \ldots, \lambda_H) = \Phi(R). \end{aligned}$$

Because Λ consists of a series of one-way hash functions, given $\Phi(R)$, when $\Phi(L) \neq f(x)$, it is computationally infeasible to find $\lambda'_1, \ldots, \lambda'_H$ to satisfy the above equation. This proves that it is computationally infeasible for the dishonest participant to convince the supervisor that it knows $f(x)$ at the time of building the Merkle tree. ∎

In the following theorem, let q be the probability that the participant can guess the correct result of $f(x)$, i.e., $Pr_{guess}(\Phi(L) = f(x)) = q$. Let D' be the set of inputs that are computed honestly by the participant, so honesty ratio is $r = \frac{|D'|}{|D|}$.

Theorem 3 *When m samples are used in the CBS scheme, The probability that a participant with honesty ratio r can cheat successfully is*

$$Pr(cheating\ succeeds) = (r + (1-r)q)^m. \tag{2}$$

Proof. Since each sample is uniformly-randomly selected, the probability that a sample x belongs to D' is r. When $x \in D'$, i.e., the participant has indeed computed $f(x)$, according to Theorem 1, the participant should be able to convince the supervisor of its honesty on sample x. When $x \in D - D'$, i.e., the participant did not compute $f(x)$ when building the tree, according to Theorem 2, it is computationally infeasible for the participant to cheat unless $\Phi(L)$ happens to equal $f(x)$. Since $Pr_{guess}(\Phi(L) = f(x)) = q$, when $x \in D - D'$, the probability to cheat successfully is q.

Combining both cases of $x \in D'$ and $x \in D - D'$, for one sample x, the probability that the participant can prove its honesty on sample x is $(r + (1-r)q)$. Therefore, the probability that the participant can prove its honesty on all m samples is $(r + (1-r)q)^m$. ∎

To keep the probability of successful cheating below a small threshold ε, the sample size m should be

$$m \geq \frac{\log \varepsilon}{\log (r + (1-r)q)}. \tag{3}$$

Fig. 2 shows how large m should be for different honesty ratios r, given $\varepsilon = 0.0001$. For example, let us consider a situation where the participant has conducted only one half of the task, which means only one half of the leaf nodes in the Merkle tree contain the actually computed results, and the other half contain guessed results. When the probability of guessing the correct results is 0.5 (i.e., $q = 0.5$), we need at least 33 samples to ensure the probability of successful cheating to be below $\varepsilon = 0.0001$. When $q \approx 0$ (i.e., it is almost impossible to make a correct guess on $f(x)$ without computing it), we only need 14 samples.

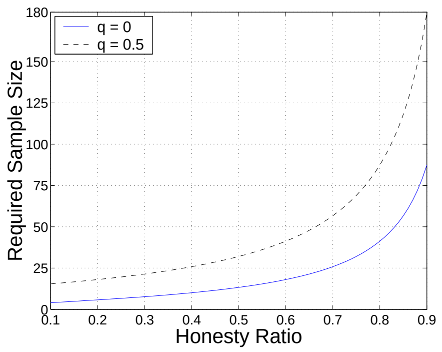

Figure 2. Required sample size vs. cheating efforts ($\varepsilon = 0.0001$)

3.3. Storage Usage Improvement

It should be noted that the CBS scheme requires the participant to store the entire Merkle tree in its memory or hard-disk, and the amount of space required is $O(|D|)$. Today's hard-disk technologies make it possible for a participant to accept tasks with $|D|$ as large as 2^{30} (by using gig-bytes of storage); however, storage becomes a problem when $|D|$ is much larger than 2^{30}.

We noticed that if a task is as large as 2^{40}, then computing $f(x)$ must be very fast; otherwise it might take the participant unreasonablely long time to finish the task. So we can make a tradeoff between time and storage in the following way: Assume the height of the entire Merkle tree is $H = \log |D|$, and the root is at level 0. Instead of storing the entire Merkle tree, the participant only stores the tree up to level $H - \ell$, where $0 < \ell < H$. Fig. 3(a) depicts the part of the tree that needs to be stored. The total amount of storage required is $O(\frac{|D|}{2^\ell})$, a decrease of 2^ℓ folds.

To prove that it has computed $f(x)$ (in Step 3 of the CBS scheme), the participant must find the path from the sample x's corresponding leaf node to the root, and then send to the supervisor the Φ values of the sibling nodes along this path. Unfortunately the sibling nodes in the lower part of the tree cannot be obtained from the storage. The shading area in Fig. 3(a) represents the subtree that contains the sample x but not saved in the storage. Fig. 3(b) depicts an example of the unsaved subtree. From the figure, we can see that nodes V_1, V_2, and V_3 are also the sibling nodes along the path, but their Φ values are not saved, but need to be recomputed. From Fig. 3(b), it is not difficult to see that recom-

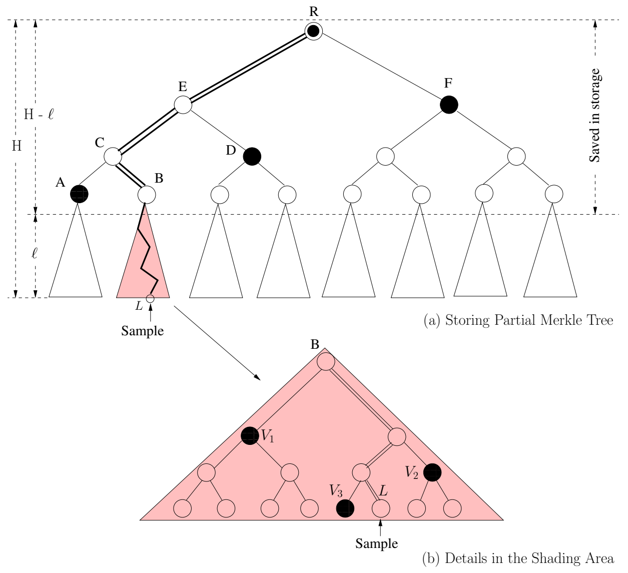

Figure 3. Storage Usage Improvement

puting those Φ values requires the rebuilding of the whole subtree depicted in the shading area. The cost of the rebuilding is $O(2^\ell)$, an increase of 2^ℓ folds compared to the CBS scheme.

We use the *relative computation overhead (rco)* to indicate how the $O(2^\ell)$ computation overhead impacts the entire task. The rco is defined as the ratio of the total computation overhead for m samples to the cost of computing $f(x)$ for all inputs in D. Let f_c represent the cost of computing $f(x)$ for one input. Let $S = 2^{H-\ell+1}$ represent the amount of space for storing the partial tree. To rebuild one subtree of height ℓ, we need to compute f functions for 2^ℓ inputs. If we ignore the cost of hash function, the cost of rebuilding a subtree equals computing f for 2^ℓ times. Hence we have the following formula:

$$\begin{aligned} rco &= \frac{m \cdot 2^\ell \cdot f_c}{|D| \cdot f_c} = \frac{m \cdot 2^\ell}{2^H} \\ &= \frac{m}{2^{(H-\ell)}} = \frac{2m}{S}. \end{aligned}$$

The above equation indicates that rco is only affected by m and S, not by the amount of inputs in D. The more storage a participant uses for storing the tree, the lower is the relative computation overhead. For example, when $m = 64$, if we use $4G$ (2^{32}) hard disk space to store the partial Merkle tree, we have $rco = 2^{-25}$. This means that, regardless of how large a task is, compared to the cost of the task, the computation overhead at the participant side is negligible when we use $4G$ disk space. Therefore, even for a task of size 2^{40}, using $4G$ disk space provides a feasible solution both storage-wise and computation-wise.

4. A Non-Interactive CBS Scheme

The CBS scheme has an extra round of interaction between the supervisor and the participant. This interaction involves the participant's sending the commitment and the supervisor's sending the samples. The interaction ensures that the supervisor sends the samples only after it receives the commitment. Although the communication cost of this extra round of interaction is not a concern, the interaction is often found less appealing because of the implementation issues involved in grid computing.

In many grid computing architectures, the supervisor might not be able to directly interact with the participants. For example, in the GRACE (Grid Architecture for Computational Economy) architecture [6], which represents a futuristic paradigm of a service oriented computing industry, there exists a Grid Resource Broker (GRB), which acts as a mediator between the supervisor and the participant. The GRB is responsible for finding more resources (participants) and scheduling of tasks among the resources depending on their availability and capability.

In the GRACE architecture, the supervisor assigns a big bulk of tasks to GRB, and relies on GRB to interact with and assign tasks to the participants. The supervisor does not even know which participant is conducting what tasks. If the supervisor wants to verify the participant's honesty on its own using the CBS scheme, it will be difficult because GRB hides the participants from the supervisor.

One way to get rid of this extra round of interaction is to let the participant generate the sample choices. Obviously, if the participant is to select the samples, the sample selection must satisfy the following properties:

1. The samples are selected *after* the Merkle tree is built.
2. The samples must be hard to predict.

When the supervisor selects the samples, the above two requirements are easily enforced because the supervisor does not tell the participant the sample choices until the participant sends the commitment. How can we enforce these requirements when we rely on the participant to generate the sample choices?

4.1. A Non-Interactive CBS Scheme

We modified the CBS scheme, so that the sample choices are generated by the participant. We call this improved scheme the *Non-Interactive CBS (NI-CBS) scheme*. Due to the page limit, we will not repeat the steps that are the same as in the CBS scheme.

Step 1: Building Merkle Tree. This step is exactly like the CBS scheme. At the end, the participant sends $\Phi(R)$ to the supervisor.

Step 2: Sample Selection. Let g be a one-way hash function. Assume $D = \{x_1, \ldots, x_n\}$ is assigned to the participant. The participant uses the following method to generate m numbers $\{i_1, \ldots, i_m\}$ in domain $[1, n]$:

$$i_k = (g^k(\Phi(R)) \ \ mod \ \ n) + 1, \quad for \ \ k = 1 \ldots m \qquad (4)$$

where

$$g^k(\Phi(R)) = \begin{cases} g(\Phi(R)), & \text{for } k = 1 \\ g(g^{k-1}(\Phi(R))), & \text{for } k = 2 \ldots m \end{cases}$$

Inputs x_i, for $i \in \{i_1, \ldots, i_m\}$, are the selected samples. In other words, the kth sample is the result of applying the one-way hash function g on R for k times.

Step 3: Participant's Proof of Honesty. This step is also exactly like the CBS scheme.

Step 4: Supervisor's Verification. Using Eq. (4), the supervisor regenerates the sample choices $\{i_1, \ldots, i_m\}$ from $\Phi(R)$. It then uses the Step 4 of the CBS scheme to verify the participant's results.

4.2. Security Analysis

Assume the participant has conducted the computations only for the inputs in D', where $D' \subset D$, and the honesty ratio is $r = \frac{|D'|}{|D|} < 1$. Also assume that the sample choices generated by the participant are $S_1, \ldots, S_m$. The only way that the participant can cheat successfully is to make sure that all the S_i for $i = 1 \ldots m$ fall into D'.[2]

Assuming the perfect randomness of the one-way hash values, the probability that all these m sample choices are in the set of D' is r^m. Namely when building the Merkle tree, the participant can use whatever values to replace $f(x)$ for $x \in D - D'$, the probability to produce the sample choices that are all in set D' is r^m.

The one-way hash function acts as an unbiased random-bit generator for the sample generation. There is no way for the participant to force the one-way function to produce certain values or to guess which values it will produce. It is also computationally infeasible for the participant to work in the reverse way, i.e., the participant cannot select the samples first, and then build a Merkle tree that generates these selected samples.

Unfortunately, the non-interactive feature brings up a potential attack. In the CBS scheme, the participant has only one chance to cheat. For $m = 10$ and $r = 0.5$; Pr(successfull cheating) = 1 in 2^{10}. If one cheating attempt fails, the supervisor will not give the participant more chances to cheat. The probability of 1 in 2^{10} tends to be

2 We assume that the probability that the participant can guess the correct computation results without conducting the computation is negligible, i.e., $q \approx 0$.

small enough for the interactive scheme, but it is still too large for the non-interactive scheme. The participant can use the following strategy to cheat:

1. Build the Merkle tree, using random numbers as the results of $f(x)$ for the inputs x that are not in D' (i.e., $x \in D - D'$).
2. Compute the sample selections from the root of the Merkle tree. If they are all within D', cheating is successful; otherwise pick other random numbers as $f(x)$ for $x \in D - D'$.
3. Revise the Merkle tree based on the newly selected values, and repeat step (2) until the cheating becomes successful.

The participant can use the above strategy to repeatedly make many cheating attempts until it finds out that all the m generated samples are in D'. Since the process is non-interactive, the supervisor knows nothing about these attempts.

There are two ways to defeat this strategy: one way is to increase the number of samples. For example, we can use 128 samples, because making 2^{128} attempts is a computationally infeasible task. However, this also increases the computation cost for the verification at the supervisor side, because the supervisor now has to verify 128 computations, much more than it needs to do in the CBS scheme.

Another way to defeat the cheating strategy is to let the participant pay for all those cheating attempts. If the cost of conducting the cheating becomes more expensive than the cost of conducting all the required computations (i.e. computing $f(x)$ for all $x \in D$), the cheating brings no benefit. To achieve this, we increase the cost of sample generation. Let C_g be the cost of the one-way function g, and C_f be the cost of function f. Because the probability that each attempt being successful is r^m, the expected number of attempts the participant needs to make is $\frac{1}{r^m}$. Therefore, to make the expected cost of the total cheating attempts more expensive than the total cost of the task, we need the following inequality:

$$\frac{1}{r^m} \cdot m \cdot C_g \geq n \cdot C_f. \tag{5}$$

To achieve the above inequality without increasing m, we can increase C_g. Most of the one-way functions, such as MD5, are very fast. To find a one-way function g such that C_g satisfies the above inequality, we just need to let $g \equiv (MD5)^k$, namely applying MD5 for k times, where k is a number that makes C_g expensive enough. If we let the left side of the inequality just a slightly greater than the right side, this extra cost of g does not bring significant overhead to the supervisor or the *honest* participant because the ratio between the cost of sample generation ($m \cdot C_g$) and the cost of the entire task ($n \cdot C_f$) is about r^m, which can be significantly small when we choose proper values for m.

5. Conclusion and Future Work

We present a scheme to prevent cheating in Grid computing. Our Scheme uses a Commitment-Based Sampling (CBS) technique to detect whether the participant is cheating or not. Unlike the old schemes [8, 10], CBS handles generic computations gracefully. To prevent the participant from changing the computation results after learning the samples, the CBS scheme uses the Merkle tree for the participant to commit its results before learning the sample selections. The CBS scheme can be used for generic computations in grid computing. It is efficient in communication as well as in computation. Based on the CBS scheme, we have addressed two important issues (1) how to reduce the storage requirement, and (2) how to convert the CBS scheme from an interactive scheme to a non-interactive scheme.

One assumption made in the CBS scheme scheme is that $|D|$ should be significantly large. When each participant is assigned a task with very few inputs, the sampling scheme does not work well. For example, when $|D| = 1$, i.e., each task consists of only one input, the cost of verifying a sample (for the CBS scheme) is as expensive as conducting the task. Therefore, the scheme is no better than the naive double-check-every-result scheme. Developing efficient schemes for a situation when $|D|$ is small is a challenging open problem that we plan to pursue.

References

[1] IBM Grid computing. Available: http://www-1.ibm.com/grid/about_grid/what_is.shtml.

[2] SETI@Home: The Search for Extraterrestrial Intelligence Project. University of California, Berkeley. Available: http://setiathome.berkeley.edu/.

[3] The Smallpox Research Grid. Available: http://www-3.ibm.com/solutions/lifesciences/research/smallpox.

[4] The Great Internet Mersenne Prime Search. Available: http://www.mersenne.org/prime.htm.

[5] D. Bedell. Search for extraterrestrials–or extra cash. In *The Dallas Morning News*, December 2 1999. also available at: http://www.dallasnews.com/technology/1202ptech9pcs.htm.

[6] R. Buyya. *Economic Based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Monash University, Melbourne, Australia, April 2002.

[7] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[8] P. Golle and I. Mironov. Uncheatable distributed computations. *Lecture Notes in Computer Science*, 2020:425–440, 2001.

[9] R. C. Merkle. Protocols for public key cryptography. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[10] D. Szajda, B. Lawson, and J. Owen. Hardening functions for large scale distributed computations. *IEEE Symposium on Security and Privacy*, 2003.