Simulating Cheated Results Dissemination for Volunteer Computing

Nils Kopal, Olga Kieselmann, Arno Wacker
University of Kassel, Group for Applied Information Security,
Pfannkuchstraße 1, 34121 Kassel, Germany
{nils.kopal|olga.kieselmann|arno.wacker}@uni-kassel.de

Abstract—In this paper, we present three novel distribution algorithms (Epoch Distribution Algorithm, Sliding Window Distribution Algorithm, and Extended Epoch Distribution Algorithm) in unstructured peer-to-peer networks for volunteer computing. In volunteer computing, the overall job is split into several sub jobs. These sub jobs are given to the (untrusted) PCs of volunteers. We show the performance of our algorithms with respect to double computed sub jobs. Furthermore, we present our idea of using cellular automata to simulate, evaluate, and visualize the cheated sub job result dissemination. Finally, we estimate and visualize the cheat detection behaviour of our cheat detection algorithm for volunteer computing.

Keywords: volunteer computing; cheat detection; simulation; cellular automata

I. INTRODUCTION

Nowadays, grid computing or cluster computing can be used to solve big computation problems, which a single computer cannot solve solitary. For instance climate research [1], factorization of big numbers [2], or cryptanalysis (i.e. keysearching or hashsearching) [3] are such problems. Since computer grids or clusters are expensive, scientists and researchers can solve their big computation problems with the help of the volunteer computing (or public resource computing). In volunteer computing, a lot of computers of volunteers (i.e. home PCs) are combined to gain the same power, which a big computer cluster or even a supercomputer would provide. An example for such a volunteer computing system is the server-based framework Berkeley Open Infrastructure for Network Computing (BOINC) [4]. With BOINC, a server manages the distribution of sub jobs to each volunteer's PC. The drawback of such a server-based solution is evident: If the central server fails, no volunteer PC is able to get new sub jobs and the overall computation stops. Additionally, for some small research groups, the maintenance of an own server is too expensive. Alternatives are peer-to-peer (P2P) networks. In P2P networks, a central server is not needed since all the peers manage the computation of the sub jobs by their own. There exist P2P solutions [5], which are for example based on structured P2P networks, like CHORD [6]. Since the maintenance of a structured P2P network, i.e. the maintenance of the overlay, is difficult and can fail, we developed in [7] two novel algorithms, namely the Epoch Distribution Algorithm (EA) and the Sliding Window Distribution Algorithm (SWA), which are based on an unstructured P2P network. In this paper, we briefly present these two algorithms, some measurements dealing with double computations, and an Extended Epoch Distribution Algorithm (EEA). In a P2P network double calculations can happen due the lack of a central coordinating place. Thus, double calculations are a measurement for the quality of distributed algorithms. The less double computations occur by those algorithms, the better the overall computation speedup is. In volunteer computing, everyone can participate and offer his computation power. Since volunteer's PCs are not trusted environments, there may be cheating volunteers. A cheat, for instance, delivers wrong results for a job or he peculates correct results. In a server-based volunteer computing environment, such cheaters are countered by giving the same sub job to not only a single volunteer PC, but handling it to a multitude of volunteer PCs. After all volunteer PCs delivered their results, the majority of results is assumed to be correct. Clearly, in P2P networks this is not possible, since there is no server, which could do so. Therefore, the need to detect wrong or cheated results arises at each single peer. In [8], we developed such a cheat detection mechanism. With this mechanism, every peer can detect, if it received wrong or cheated results from one of its neighbors by doing positive verification and negative verification. The probability of the detection of cheated blocks with our approach can be parameterized. The more precise the detection should be, the more effort the peer has to make. To determine the anti-cheating parameters (i.e. the probability of detection and the effort each peer has to make) for a "real P2P network", we use a cellular automata system, which simulates the distribution of wrong or cheated results within a simulated unstructured P2P network and our novel distribution algorithms. Cellular automata are used in different disciplines of science and research. They are used for doing earthquake simulations [9], for modeling brain tumors [10], for neural networks [11], and for many more applications. A cellular automaton is a mathematical model, which follows a simple set of local rules and changes its state based on the states of its direct neighbors. Single cellular automata are not powerful. But a complete system of connected "interacting" cellular automata can model a large and complex system, while each automaton follows only its simple rule set.

The rest of the paper is organized as follows: In Section II, we present related work in the field of volunteer computing (VC), peer-to-peer (P2P) computing, security, and cheating
in these areas. After that, in Section III, we present our definition of distributed computing, cellular automata, which are the basic ideas of our distribution algorithms, and the cheat detection in the system model. Then, in Section IV, we present novel distribution algorithms and our cellular automaton-based P2P network simulation model for unstructured P2P networks with cheating peers. Finally, we conclude our paper in the last section and give a short insight in future work.

II. RELATED WORK

In the research domain of VC the best known computing project is SETI@Home [12]. Most of such projects are based on BOINC [4]. In a current list of the English Wikipedia 13, which contains 85 distributed computing projects, 66 projects are BOINC-based (≈ 78%). In the research domain of P2P computing there are three different generations of P2P systems. The first type of P2P systems are centralized P2P networks like the Napster network [14]. The second type of P2P systems are unstructured flooding-based P2P networks like Gnutella [15]. The third generation of P2P systems are structured overlays like CHORD [6] or the Kademlia network [16]. With P2P computing, we already explored the usage of structured P2P overlay networks for cryptanalysis in [3] and unstructured P2P networks in [7]. There exist some preliminary work on combining these research domains. In [17], Cunsolo et al. describe their Cloud@Home system. Their idea is to combine different clouds using a general middleware. In [18], Marosi et al. describe an idea of combining a BOINC system with P2P techniques like BitTorrent [19]. In [20], Marosi et al. present their solution for extending service grids with volunteer (global) and institutional (local) desktop grids. Costa and Dahlin present a BOINC based ’Map and Reduce’ prototype in [21]. In [22], Babaoglu et al. created a P2P Cloud with gossip-based protocols [23]. In the research domain of security and cheating, there are attempts to secure VC. In [24], Golle and Mironov describe their idea of uncheatable distributed computations. Moca et al. present in [25] a method for distributed results checking [26]. Zhao and Lo show their scheme ’Quiz’ in [27], which inserts indistinguishable quiz tasks in distributed computing jobs. Sarmenta presents in [28] his sabotage-tolerant mechanisms for VC.

III. SYSTEM MODEL

In this section, we first present distributed computing in unstructured P2P networks. After that, we show a general definition of a cellular automaton. Finally, we describe the cheat detection mechanism for detecting wrong or cheated results within VC.

A. Distributed Computing in Unstructured P2P Networks

The basis for our algorithms is an unstructured P2P network, which consists of p peers, that are each connected with n random neighbor peers. Peers only communicate with their direct neighbors by flooding the results of their sub job computation and the state of the overall job. Our algorithms compute embarrassingly parallel problems or jobs J with a result R using a computation function \( f(J) = R \). The overall job can be parallelized by dividing the job into a dedicated amount \( m \) of sub jobs, where each sub job \( J_i \) can be computed by the same computation function, i.e. \( f(J_i) = R_i \) with \( 0 \leq i \leq m \). To get the overall result \( R \) of the job \( J \), all sub jobs can be combined using a combination function \( \circ \), i.e. \( R = f(J) = f(J_1 \circ f(J_2) \circ \ldots \circ f(J_m)) \). Furthermore, we assume that the combination function \( \circ \) is associative i.e. \( (R_A \circ (R_B \circ R_C)) = (R_A \circ R_B) \circ R_C \), commutative i.e. \( (R_A \circ R_B) = R_B \circ R_A \), and idempotent i.e. \( (R_B \circ R_B = R_B) \). We assume, that the amount of resources to store the combination \( R_{comb} \) of two results, \( R_i \) and \( R_j \), is smaller (or equal) than the amount of storing both results independently: \( \#(R_{comb}) = \#(R_i \circ R_j) < \#(R_i) + \#(R_j) \). Since the jobs, which our P2P network computes, consist of several thousands or millions (or more) of sub jobs, it is not possible to flood the state of each sub job. Therefore, we invented our distribution algorithms, which separate the total computation space in sub-spaces. These spaces are chosen small enough, thus they can be flooded within an unstructured P2P network.

B. Cellular Automata

A cellular automaton \( CA(C, N, Q, \delta; Q^N \rightarrow Q) \) is a discrete mathematical model, which consists of a set of cells \( C \), a set of neighbors \( N \subseteq C \), a set of states \( Q \), and a local transition function \( \delta \). In each simulation step, each cell \( c \in C \) changes its state according to the transition function \( \delta \), based on its local state, and the states of all of its neighbors in parallel. An example for a system of such cellular automata is Conway’s game of life [29]. Here, the cells \( C \) are arranged as a grid of “pixels”, each pixel representing a cell. The neighbors of a cell \( r \in R \) in this example are the cells (pixels) directly connected to the cell. So each cell has 8 neighbors. The states \( Q \) are ‘Alive’ or ‘Dead’, meaning \( Q = \{Alive, Dead\} \). The transition function \( \delta \) is defined by the following four rules:

1) If a cell \( c \in C \) is in the state ‘Alive’ and it has less than two neighbor cells, which are in state ‘Alive’, it “dies” and changes to state ‘Dead’

2) If a cell \( c \in C \) is in the state ‘Alive’ and it has two or three neighbor cells in state ‘Alive’, it remains in the state ‘Alive’

3) If a cell \( c \in C \) is in the state ‘Alive’ and it has more than three neighbor cells, it “dies” and changes to state ‘Dead’

4) If a cell \( c \in C \) is in the state ‘Dead’ and it has exactly three neighbor cells in the state ‘Alive’, it “starts living” and changes to state ‘Alive’

With this simple definition of a cellular automaton, Conway managed to simulate rather complex systems, which seem “to be alive”. Simulating the game of life leads to “moving” structures and oscillating behaviour. In Figure 1, we show a small example of a simulation of Conway’s game of life with 2500 cellular automata and 300 random cells in state ‘Alive’. After 350 simulation steps the simulation stabilizes and does not change any more. We made screenshots of...
a visualization of the automata using 50x50 pixels at each 50th iteration (black pixels are 'Dead' and white pixels are 'Alive'). In Section IV, we present our cellular automaton, which simulates the behaviour of our distribution algorithms.

C. Cheat Detection

In [8], we developed an approach for the detection of cheated results within a distributed computing scenario. The detection is based on two different approaches: Positive verification and negative verification. With positive verification, we verify the result of the computation of a block for correctness and with negative verification, we try to find other (better) results, which the delivering peer omitted. If either positive verification fails or negative verification succeeds, we found a cheated result and we decline it. For positive verification, we assume, that we can check the results fast. For that, we use a toplist approach. A sub job $J_i$ can be split into $k$ independent slices $S_j$ and each slice can be computed independently and fast (in a manner of milliseconds or seconds). We use a scoring function $f$ to score the result of each computed slice. Each peer sends as result the toplist, consisting of the result values and the $f$-values. In a distributed job, we try to find the $t$ best slices of each sub job. An example for such a computation problem is generally a search problem, i.e. keysearching a symmetric cipher. With positive verification, we test each result of a slice, which we obtained from a neighbor by recomputing the function $f$ for the toplist entries and compare them with each slice result. If we found a difference, we detected a cheated result. With negative verification, we randomly select slices out of the sub job and compute the function $f$ for these as well. If we find a better value than the values within the toplist, we also found a cheated result. The effectiveness of the verifications relies on the size of the toplist and on the amount of slices tested for each sub job. We showed in [8], that an overall amount of 1% of all slices per sub job need to be tested to gain a cheat detection probability of nearly 100%.

With our distribution algorithms, we combine the results of each block using our combination function $\circ$. Thus, we have a toplist, which is the combination of all so far computed sub jobs. Here, it is possible, that some results, which we already computed, are removed from the combined toplist. This can be a problem for the cheat detection, since it needs the results of blocks for the positive verification. For the negative verification, a verification on each already computed sub job is necessary. Thus, with increasing amount of finished sub jobs, the effort for the cheat detection grows. Therefore, it may not be possible, to test 1% of all slices of all computed sub jobs. Hence, we decrease the amount of tested slices, which also decreases the probability of detecting a cheated result. However, since every peer within our network does the cheat detection, the overall probability to detect a cheated result within the whole network increases.

IV. DISTRIBUTION ALGORITHMS AND CHEAT SIMULATION

In this section we present our three novel distribution algorithms. We compare the algorithms concerning computational overhead, i.e. "double computations". After that, we present our simulation of cheated results using a cellular automaton model.

A. Distribution Algorithms

A naive idea of distributing a job in an unstructured P2P network is to distribute (flood) the complete state of the job to the network. Since there is no central unit, which assigns sub jobs, all peers select sub jobs randomly out of the complete computation space. After finishing the computation of the sub job, the peer has to distribute (flood) the result to his neighbors. Since we assume, that new peers are coming at every possible time, those peers have to get the whole state of the complete job. Thus, every peer floods the complete state after the computation of a sub job. But even if we only use one bit per sub job to indicate whether the sub job has already been computed or not, this approach is not feasible, since we cope with very large computation spaces. For instance, a common search problem is the brute-force search of a DES (Data Encryption Standard) cipher [30]. Here, we search for the one correct key to decrypt a given cipher text. The computation space of the overall job is $2^{256}$ (keyspace of DES). Even if we divide this space into sub jobs each of the size of $2^{25}$, we need $2^{31}$ bits to model every sub job with one bit. This is 256 megabyte. Thus, every flood message would be of that size. This is not practical. Therefore, we have to minimize the size of the flood messages. For small jobs ($n < 2^{19}$), the naive approach is realizable. Since we assume, that $\#(R_{comb}) = \#(R_t \circ R_j) < \#(R_t) + \#(R_j)$, all peers combine all local and received results before they flood them as well. Thus, the size of the results of the sub jobs is negligible small. The combination of the results is possible, since our requirements state, that the combination function is associative, commutative, and idempotent. In the next section we present our three novel distribution algorithms for VC in unstructured P2P networks.

1) Epoch Distribution Algorithm: The EA divides the computation space into sub-spaces, so called epochs. An epoch consists of a dedicated amount of sub jobs, which is the same
for each epoch. A peer starting to work on a job in the P2P network starts working in epoch 0. He randomly selects a sub job out of the epoch from which he knows that it has not been computed yet. For storing the state of an epoch, an epoch mask, as exemplarily shown in Figure 2, is used. After finishing the computation of a sub job, a peer combines the result of the sub job with all the results he already computed and with all results, which he received formerly from his neighbors. For this combination, he uses the combination function $\circ$. Additionally, he sets the corresponding bit in his bitmask to 1, indicating this sub job is completed. To store the number of the current epoch, we use an epoch index.

If all sub jobs of an epoch have been computed, the peer increments this index and sets his epoch mask to zero. If the last epoch is completed, the overall job is completed and the epoch algorithm terminates. After each sub job computation, a peer floods his current epoch mask, the results, and the epoch offset to his neighbors.

2) Sliding Window Distribution Algorithm: The SWA uses a sliding window, which slides over the computation space as depicted in Figure 3. The window consists of a dedicated amount of sub jobs. The position of the window in the computation space is defined by an window offset. After finishing the computation of a sub job, a peer combines the result of the sub job with all the results, he already computed, and with all results, which he received formerly from his neighbors. For this combination, he uses the combination function $\circ$. Additionally, he sets the corresponding bit in his window to 1, indicating this sub job is completed. Each time the least significant job, i.e. with the lowest job-id in the current window, has been computed, the window is left-shifted by one. If the window can not be shifted any more, i.e. it reached the "end" of the computation space, and every sub job inside the window is computed, the overall job is completed and the window algorithm terminates. After each sub job computation, a peer floods his current window, the results and the window offset to his neighbors.

3) Extended Epoch Distribution Algorithm: The EEA uses double epochs, with different probabilities to be selected by peers, as depicted in Figure 4. It is similar to the EA, but instead of sending a single bitmask between the peers, it uses two bit masks. The peers start to select bits out of the first of the two masks. If this mask is “filled” by 80%, the peers start also filling the second mask. With a probability of 20% the peers select unset bits of the first mask, with a probability of 80% they select bits out of the second bit mask. If the first bitmask is filled completely, the second bitmask becomes the first bitmask and a new second bitmask is created. The algorithm terminates, when the “last” bitmask is filled completely, yielding all sub jobs have been computed by the peers.

B. Double Computations

Our above introduced algorithms all work on randomized selections of sub jobs to be computed. Due to the randomness in each of the algorithms, peers by chance compute the same sub job. This results in a worse speedup of computation. We evaluated the behaviour of each of our algorithms with respect to such double computations. In Figure 5 we depicted a simulation graph of the Epoch Algorithm for double computations and average free bits in our data structure. Here, we simulated a job with 5 epochs, each with an amount of 80 000 bits. Thus, the overall amount of simulated sub jobs was $5 \cdot 80 000 = 400 000$. Furthermore, we simulated jobs of the same size for the SWA in Figure 6 and for the EEA in Figure 7. We evaluated, that the Double Epoch Algorithm performs best, since it only introduces $\approx 16 000$ double computed sub jobs in our simulation. The Sliding Window Distribution Algorithm leads to $\approx 45 000$ double computed sub jobs. And the EA leads to $\approx 20 000$ double computed sub jobs. In the graphs, one can see, why this behaviour occurs. The probability (over time) for two peers to select the same sub job is minimal with the EEA. This probability is anti-proportional to the amount of “free” bits in the data structure to select from. In the EEA the minimal size to select from is 20% of the bitmask width all the time, with the only exception being in the last epoch.

C. Simulating Cheated-Results-Dissemination

We model our P2P simulation network using a set of cellular automata. The goal of our simulation is to estimate the behaviour of a network in which we inserted a cheated result of a sub job. We have $\#(C)$ peers and each peer has $N \subset C$ random neighbors. The peers can be in one of the states
$Q = \{ \text{Good}, \text{Bad}, \text{Neutral} \}$. All peers, with the exception of one single peer, start in the state $\text{Neutral}$, meaning that they did not compute the result of our cheated block. One of the peers starts in the state $\text{Bad}$, meaning it tries to insert a cheated block in our P2P network. In each iteration of the model, each peer executes the transition function $\delta$, which is defined by the following set of rules:

1) If a cell $c \in C$ is in the state 'Neutral' and one or more of its neighbor cells are in state 'Good', it changes to state 'Good'

2) If a cell $c \in C$ is in the state 'Neutral' and one or more of its neighbor cells are in state 'Bad', it changes to state 'Bad'

3) If a cell $c \in C$ is in the state 'Bad' and one ore more of its neighbor cells is in state 'Good', it changes to state 'Good'

Our P2P simulation network behaves in such a way, that it tries to simulate the detection and the elimination of the "cheated result". Our cheat detection mechanism works as described in Subsection III-C. We assume, that our cheat detection has a probability of $p_{\text{detect}}$ to detect a cheated result. In each simulation iteration step, a peer "sends" its results to all of its neighbors. When a peer receives a result of a sub job, it starts the cheat detection algorithm. If it successfully detects a cheated sub job, it immediately computes the correct result of the sub job. In this case the state of the cell goes to $\text{Good}$. If the cheat detection fails, the cell goes to state $\text{Bad}$. If an already “Bad” cell receives a “Good” result, it “overwrites” the bad result and goes to state $\text{Good}$. Our simulation terminates either if all cells are in state $\text{Good}$ or all cells are in state $\text{Bad}$.

In Figure 8, we depicted an example simulation run of our solution. We simulated a network of $\#(C) = 1024$ peers (cellular automata). All peers start in state $\text{Neutral}$ and only the peer in the middle of the first image starts in state $\text{Bad}$. Each peer is connected to $\#(N) = 5$ random neighbors. The detection probability $p$ of a cheated block is 1%. A neutral peer is drawn using black color, a good peer is drawn using white color, and a bad peer is drawn using gray color. The bad result is distributed within the network. In the 2nd iteration step an automaton “randomly decided” to detect the cheated result. From Iteration 4 to Iteration 7, the peers distribute either the bad result as well, as the good result throughout the whole network. In the end of the simulation, all peers are in the state $\text{Good}$, meaning that the bad result has been erased within the P2P simulation network. The probability, that a cheated result will flood to every peer in our network, is $p_{\text{overall}} = (1 - p_{\text{detect}})^{\#(C)}$, where $p_{\text{detect}}$ is the detection probability and $\#(C)$ is the overall amount of peers in our network. In this case $p_{\text{overall}} = (1 - 0.01)^{1024} = 0.0000339187 \approx 0.004\%$. 

\[\]
With our simulation, we can simulate and visualize the cheated result dissemination with different amounts of peers, detection probabilities, and different amounts of connected neighbors. We plan to use this to evaluate perfect settings for the real-world usage of our distribution algorithm and cheat detection algorithms.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented three different algorithms for volunteer computing in unstructured peer-to-peer networks, namely the Epoch Distribution Algorithm, the Sliding Window Distribution Algorithm, and the Extended Epoch Distribution Algorithm. We showed the performance of our algorithms in terms of double computations. Here, the Extended Epoch Distribution Algorithm performs best in case of double computed sub jobs. Furthermore, we presented our idea to use cellular automata to simulate, visualize, and evaluate the cheated result dissemination within an unstructured P2P network. In the future, we plan to implement all of our algorithms for real-world usage, i.e. cryptanalysis with volunteer computing. Furthermore, we plan to simulate and evaluate all of our algorithms with respect to speedup, double computations, and cheat dissemination (cheated results acceptance of our networks).

REFERENCES
