Robust Network Supercomputing without Centralized Control*

Seda Davtyan[1], Kishori M. Konwar[2], and Alexander A. Shvartsman[1]

[1] Department of Computer Science & Engineering, University of Connecticut, Storrs CT 06269, USA
{seda,aas}@engr.uconn.edu
[2] Department of Immunology and Microbiology, University of British Columbia, Vancouver BC V6T 1Z3, Canada
kishori@interchange.ubc.ca

Abstract. Internet supercomputing provides means for harnessing the power of a vast number of interconnected computers. With this come the challenges of marshaling distributed resources and dealing with failures. Traditional centralized approaches employ a master processor and many worker processors that execute a collection of tasks on behalf of the master. Despite the simplicity and advantages of centralized schemes, the master processor is a performance bottleneck and a single point of failure. Additionally, a phenomenon of increasing concern is that workers may return incorrect results, e.g., due to unintended failures, over-clocked processors, or due to workers claiming to have performed work to obtain a high rank in the system. This paper develops an original approach that eliminates the master and instead uses a *decentralized* algorithm, where workers cooperate in performing tasks. The failure model assumes that the *average* probability of a worker returning a wrong result is inferior to 1/2. We present a randomized synchronous algorithm for n processors and t tasks ($t \geq n$) achieving time complexity $\Theta(\frac{t}{n} \log n)$ and work $\Theta(t \log n)$. It is shown that upon termination the workers know the results of all tasks with high probability, and that these results are correct with high probability. The message complexity of the algorithm is $\Theta(n \log n)$, and the bit complexity is $O(tn \log^3 n)$. Simulations illustrate the behavior of the algorithm under realistic assumptions.

Keywords: Distributed Algorithms, Fault-Tolerance, Internet Supercomputing.

1 Introduction

Internet supercomputing is becoming a popular means for harnessing the computing power of an enormous number of processors around the world. A typical Internet supercomputer consists of a *master* computer and a large number of computers called *workers*. Applications submit the tasks to be performed to

* This work is supported in part by the NSF award 1017232.

A. Fernández Anta, G. Lipari, and M. Roy (Eds.): OPODIS 2011, LNCS 7109, pp. 435–450, 2011.

the master that in turn directs the workers to perform the tasks and then collects the results. Several Internet Supercomputers are in existence today. For instance, Internet PrimeNet Server encompasses about 30,000 computers, achieving throughput of over 1 teraflop [1], and even higher throughput is reported by the SETI@home project [2].

A major concern in network supercomputing is the correctness of the results returned by the workers. While most workers may be reliable, workers have been known to return incorrect results. This may be due to unintended failures caused (e.g., by over-clocked processors), or the workers claiming to have performed assigned work so as to obtain incentives, such as getting higher rank on the SETI@home list of contributed units of work. Prior research developed models and algorithms for network supercomputing, e.g., [5,6,11]. In these models it is assumed that a reliable master and a collection of unreliable workers cooperatively perform a set of tasks. Using a variety of probabilistic failure models, the goal is to design algorithms that correctly perform all tasks with high probability. One drawback of this approach is the assumption of the existence of a reliable master processor. Despite the simplicity and advantages of this approach, the master is a single point of failure. The master is further assumed to be able to keep up with the large number of results returned by the workers, making such systems poorly scalable. In any message passing system, during some short time interval, a network node can maintain only a limited number of connections. Thus scalable distributed (i.e., not centralized) solutions are desirable.

In the current paper, we aim to remove the assumption of an infallible and bandwidth-unlimited master processor and consider a fully decentralized solution using just the cooperating workers.

Contributions. We consider the problem of performing t tasks in a distributed system of n workers. The tasks are independent, they admit at-least-once execution semantics, and each task can be performed by any worker in constant time. The workers either obtain the tasks from some repository or the tasks are initially known to all processors. The workers can return incorrect results and ultimately crash. The fully-connected message-passing system is synchronous, and the workers communicate using authenticated messages (to prevent malicious workers from impersonating other workers). Our system of autonomous processors is fully decentralized in the sense that it does not contain any distinguished participants (e.g., a master). We present an original randomized decentralized algorithm of logarithmic time complexity, where in each iteration of the algorithm each worker sends just one message. The algorithm works under several failure models differing in the assumptions about the fraction of possibly faulty workers and the failure probabilities. In more detail our contributions are as follows.

1. We define a general failure model $\mathcal{F}$, where each worker i ($i \in [n]$) independently returns an incorrect result, each time it performs a task, with probability p_i, such that $\frac{1}{n}\sum_{i\in[n]} p_i < \frac{1}{2} - \varepsilon$ for some $\varepsilon > 0$. (We show later how this model specializes to other intuitive models.)

2. We provide a n-processor, n-task decentralized randomized algorithm for model $\mathcal{F}$ that works in synchronous rounds. The number of rounds performed by the algorithm is an external (compile-time) parameter. Within each round each processor performs a random task (for some number of rounds), and communicates its cumulative knowledge to one randomly chosen processor. The algorithm naturally generalizes for t tasks, where $t \geq n$, by having processors work on groups of $\lceil t/n \rceil$ tasks instead of single tasks.
3. We analyze our algorithm under model $\mathcal{F}$ and show that it is sufficient for it to iterate for $\Theta(\log n)$ rounds in order to perform all tasks with high probability (*whp*). More specifically, we prove that after $\Theta(\log n)$ rounds every processor holds the array of computed results that are all correct *whp*, and that the arrays of results are consistent among all processors *whp*. With t tasks ($t \geq n$), the algorithm has time complexity $\Theta(\frac{t}{n} \log n)$, message complexity $\Theta(n \log n)$, bit communication complexity $O(t\, n \log^3 n)$, space complexity is $\Theta(t\, n \log^2 n)$, and work $\Theta(t \log n)$.
4. We show that failure model $\mathcal{F}$ can be extended to incorporate processor crashes in the way that does not require any changes to our algorithm. We also present three additional failure models that specialize model $\mathcal{F}$ and that are more intuitive. Since each of these models is a specialization of model $\mathcal{F}$, the same algorithm works under all these models and has the same (or better) complexity.
5. We present selected simulation results that illustrate and provide insights into the behavior of the algorithm.

Note that our problem is related to the Do-All problem [4,9] of using n processors to perform a collection of t independent tasks in the presence of adversity. However the two problems are not identical. In Do-All, the problem is solved when *some* correct processor knows that all tasks have been performed. In our problem, with the removal of the infallible master, a client application should be able to obtain the results from any worker. Thus the current problem is solved when *all* correct processors know that all tasks have been performed and are in the possession of the results of all tasks (*whp* in this work).

Consequently, an algorithm solving our problem is also an algorithm for Do-All, but not necessarily vice versa. Additionally a lower bound for Do-All is also a lower bound for the current problem. In [3] Chlebus and Kowalski give a lower bound $\Omega(t + n\frac{\log n}{\log \log n})$ on work of any algorithm solving Do-All, including randomized, against an adaptive linearly bounded adversary. This bound applies also to the current problem, and the work of our algorithm is close to this bound.

Prior work. Several approaches have been explored to improve the quality of the results obtained from untrusted workers. Fernandez, Georgiou, Lopez, and Santos [5,6] and Konwar, Rajasekaran, and Shvartsman [11] consider a distributed system consisting of a reliable master and a collection of workers that execute tasks on behalf of the master, where the workers may act maliciously by deliberately returning wrong results. Works [5,6,11] focus on designing algorithms that help the master determine the correct result with high probability, and at the least possible cost in terms of the total number of tasks executed.

The failure models assume that some fraction of processors can exhibit faulty behavior.

Gao and Malewicz [8] consider the problem of maximizing the expected number of correct results when the tasks have dependencies. Their distributed system is composed of a reliable server that coordinates unreliable workers that compute correctly with some probability, and where any incorrectly performed task corrupts all dependent tasks. The goal is to produce a schedule for task execution by the participants that maximizes the expected number of correct results under a constraint on the computation time.

Paquette and Pelc [13] consider a general model of a fault-prone system in which a decision has to be made on the basis of unreliable information. They assume that a Boolean value is conveyed to the deciding agent by several processors. An *a priori* probability distribution of this value is known to the agent and can be any arbitrary distribution. Relaying processors are assumed to fail independently with a known probability distribution. Fault-free processors relay the correct value, but faulty ones may behave arbitrarily. The deciding agent receives the vector of relayed values and must make a decision concerning the original value. The authors design a deterministic decision strategy with a high probability of correctness, and it is shown that a locally optimal decision strategy need not have the highest probability of correctness globally.

We have already mentioned the related problem of distributed cooperation called Do-All. Many algorithms, both deterministic and randomized, have been developed for Do-All in various models of computation, including message-passing and shared-memory models [10,9]. A related problem is the Omni-Do problem of performing a collection of tasks with the help of group communication services in partitionable networks [9].

Document structure. In Section 2 we give models of computation and failure, and measures of efficiency. Section 3 presents our algorithm. In Section 4 we carry out the analysis of the algorithm and derive complexity bounds. In Section 5 we deal with processor crashes. In Section 6 we present the simulation of the algorithm. We conclude in Section 7 with a discussion.

2 Model of Computation and Definitions

System model. We consider a set of n processors, or workers, each with a unique identifier (id) from set $\mathcal{P} = [n]$. We refer to the processor with id i as processor i. The system is synchronous and the processors communicate by exchanging reliable authenticated messages. Computation is structured in terms of synchronous *steps*, where in each step a processor can send or receive messages, and perform some local computation. The duration of each step depends on the algorithm and need not be constant (e.g., it may depend on n), but it is fixed at compile-time. Messages received by a processor in a given step include all messages sent to it in the previous step.

Tasks. There are t tasks to be performed, each with a unique id from set $\mathcal{T} = [t]$. We refer to the task with id i as $Task[i]$. Workers obtain tasks from some

repository or workers initially know all tasks. The tasks are (a) similar, meaning that any task can be done in constant time by any processor, (b) independent, meaning that each task can be performed independently of other tasks, and (c) idempotent, meaning that the tasks admit at-least-once semantics and can be performed concurrently. For simplicity, we assume that the outcome of each task is a binary value. The problem is most interesting when there are at least as many tasks as there are processors, thus we only consider $t \geq n$.

Models of Failure. Some processors may exhibit faulty behavior by (maliciously) returning an incorrect result for a task. We assume that the result of each task is signed by the performing processor and that the signatures are unforgeable. The main failure model is defined as follows.

Model $\mathcal{F}$: *Each worker, independently of other workers, returns faulty results for a performed task with probability p_i, for $i \in [n]$, such that, $\frac{1}{n}\sum_i p_i < \frac{1}{2} - \varepsilon$ for some $\varepsilon > 0$.*

We use the constant ε to ensure that the average probability of worker misbehavior does not become arbitrarily close to $1/2$ as n tends to infinity.

In algorithm simulations we also use three related specialized models that were introduced in [5,11] in the context of the centralized master-worker setting.

Model $\mathcal{F}_a$: *Each worker, independently of other workers, returns faulty results for a task with probability $p < \frac{1}{2}$.*

Model $\mathcal{F}_b$: *A fixed fraction f of workers can return faulty results for any task with probability p, with $fp < \frac{1}{2}$.*

Model $\mathcal{F}_c$: *A fixed fraction f of workers can return faulty results for any task, with $f < \frac{1}{2}$.*

Observe that model $\mathcal{F}$ generalizes these specialized models since in all three cases the average probability of worker returning a wrong result is inferior to $1/2$. Thus any algorithm that solves our problem in model $\mathcal{F}$ also solves it in models $\mathcal{F}_a$, $\mathcal{F}_b$, and $\mathcal{F}_c$. Because the last three models are simpler to implement we use them in simulations.

Measures of efficiency. We use the conventional worst-case measures of *time complexity*, *work complexity*, and *space complexity*. *Message complexity* is the worst-case number of point-to-point messages sent in an execution, and *bit complexity* is the total number of bits sent in all messages.

Lastly, we use the common definition of *an event $\mathcal{E}$ occurring with high probability* (*whp*) to mean that $\mathbf{Pr}[\mathcal{E}] = 1 - O(n^{-\alpha})$ for some constant $\alpha > 0$.

3 Algorithm Description

In this section we present our decentralized algorithm A that employs no master and instead uses a gossip-based approach. We present the algorithm for n

```
    procedure for processor i;
        external n /* the number of processors and tasks */
        external L /* 2L is the the number of rounds */
        Task[1..n] /* set of tasks */
        R_i[1..n] init ∅^n /* set of collected results */
        Results_i[1..n] /* array of results */

        Compute:
1:       Randomly select j ∈ T /* choose task id */
2:       Compute the result v_j for Task[j]
3:       R_i[j] ← {⟨v_j, i, 0⟩} /* Record result for round 0 */

        for r = 1 to 2L do
            Send:
4:              Randomly select a processor q ∈ P
5:              Send the array R_i[ ] to processor q
            Receive:
6:              Let M be the set of received messages
7:              for all j ∈ T
8:                  R_i[j] ← R_i[j] ∪ {R[j] : R[ ] ∈ M}
            Compute:
9:              if r < L then
10:             Randomly select j ∈ T /* choose task id */
11:                 Compute the result v_j for Task[j]
12:                 R_i[j] ← R_i[j] ∪ {⟨v_j, i, r⟩}

13:     for each j ∈ T
14:         Results_i[j] ← u such that triples ⟨u, _, _⟩ form a plurality in R_i[j]
    end
```

Fig. 1. Algorithm A at processor i for $i \in \mathcal{P}$, and $t = n$

processors and $t = n$ tasks. The algorithm naturally generalizes for t tasks, where $t \geq n$, by having processors perform work on fixed groups of $\lceil t/n \rceil$ tasks instead of single tasks (we discuss this in more detail at the conclusion of the analysis). Each processor (worker) maintains two arrays of size linear in n, one used to accumulate knowledge gathered from different processors, and another to store the results. The algorithm works in synchronous rounds. The number of rounds performed by the algorithm is an external (compile-time) parameter. Within each round a processor communicates its cumulative knowledge to one randomly chosen processor and performs a random task (for some determined number of rounds). The pseudocode for the algorithm is given in Figure 1, and we now detail it.

Local knowledge and state variables. Every processor i maintains the following:

- L, the external parameter that is used to control the number of iterations, i.e., $2L$, of the main loop; r is the current round (iteration) number.
- The array of results $R_i[1..n]$, where the element $R_i[j]$, for $j \in \mathcal{T}$, is the set of results for $Task[j]$. Each $R_i[j]$ is a set of triples $\langle v_j, i, r\rangle$ representing the result v_j computed for $Task[j]$ by processor i during round r. The use of such triples eliminates repeated inclusions of the results for the same task, in the same round, by the same processor.
- The array $Results_i[1..n]$ stores the final results.

Control flow. The algorithm contains the main for-loop, and we use the term *round* to refer to a single iteration of the loop. The loop contains three stages (or steps), viz., *Send*, *Receive*, and *Compute*. The algorithm starts by performing a single *Compute* stage, after which it enters the main loop. The algorithm uses an external parameter L (whose value is established in the analysis of the algorithm). The main loop iterates $2L$ times, where in the first L iterations all three stages are executed, and the final L iterations only the *Send* and *Receive* stages are executed. (We will prove that L needs to be $\Theta(\log n)$ to yield our high probability guarantee.)

We now describe the stages in more detail, starting with *Compute*. In *Compute* stage in round r processor i randomly selects a task j, computes the result v_j, and adds the triple $\langle v_j, i, r\rangle$ to the results set $R_i[j]$. This is done in the first L rounds.

In each *Send* stage, a processor choses a target processor q at random from the set of processors $\mathcal{P}$. The array of results $R[\,]$ is sent to processor q.

During the *Receive* stage processor i receives messages (if any) sent to it during the *Send* stage by other processors (including itself). Upon receiving the messages the processor updates its $R_i[j]$ (for each $j \in \mathcal{T}$) by taking a union with the triples for task j received in all messages.

When the main loop terminates after $2L$ rounds, each processor goes over the result set for every task and computes the result that corresponds to the plurality of the results (in the analysis we prove that in fact a majority exists). The results of the tasks are available locally in array $Results_i[1..n]$.

4 Algorithm Analysis

We now analyze algorithm A for $t = n$, then extend the analysis to $t \geq n$. We start by stating the Chernoff bound result that we use in several places.

Lemma 1 (Chernoff Bounds). *Let $X_1, X_2, \cdots, X_n$ be n independent Bernoulli random variables with $\mathbf{Pr}[X_i = 1] = p_i$ and $\mathbf{Pr}[X_i = 0] = 1 - p_i$, then it holds for $X = \sum_{i=1}^{n} X_i$ and $\mu = E[X] = \sum_{i=1}^{n} p_i$ that for all $\delta > 0$, (i) $\mathbf{Pr}[X \geq (1+\delta)\mu] \leq e^{-\frac{\mu\delta^2}{3}}$, and (ii) $\mathbf{Pr}[X \leq (1-\delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}}$.*

The following lemma shows that within $\Theta(\log n)$ rounds of algorithm A every task τ is chosen for execution $\Theta(\log n)$ times *whp*. Weaker variations of Lemma 2 are known in the literature, e.g., see the *Occupancy Problem* [12]. We prove our

lemma for completeness, and more importantly, for acquiring a stronger bound required for our complexity results.

Lemma 2. *In $\Theta(\log n)$ rounds of the algorithm every task is performed $\Theta(\log n)$ times* whp*, possibly by different processors.*

Proof. Let us assume that after $L = k \log n$ rounds of algorithm A, where k is a sufficiently large constant, there exists a task τ that is performed less than $(1-\delta)L$ times among all workers, for some $\delta > 0$. We prove that *whp* such a task does not exist.

According to our assumption at the end of round L for some task τ, we have $|\cup_{j=1}^{n} R_j[\tau]| < (1-\delta)L$. Let X_i be a Bernoulli random variable such that $X_i = 1$ if the task was chosen to be performed in line 10 (only once the task is chosen in line 1) of the algorithm, and $X_i = 0$ otherwise.

Let us next define the random variable $X = X_1 + \cdots + X_{Ln}$ to count the total number of times task τ is performed by the end of L rounds of algorithm A.

Note that according to line 10 any worker picks a task uniformly at random. To be more specific let x be an index of one of Ln executions of line 10. Observe that for any x, $\mathbf{Pr}[X_x = 1] = \frac{1}{n}$ given that the workers choose task τ uniformly at random. Let $\mu = \mathbf{E}[X] = \sum_{x=1}^{Ln} \frac{1}{n} = L$, then by applying Chernoff bound, for the same $\delta > 0$ chosen as above, we have:

$$\mathbf{Pr}[X \leq (1-\delta)L] \leq e^{-\frac{L\delta^2}{2}} \leq e^{-\frac{(k \log n)\delta^2}{2}} \leq \frac{1}{n^{\frac{c\delta^2}{2}}} \leq \frac{1}{n^{\alpha}}$$

where $\alpha > 1$ for some sufficiently large c. Now let us denote by $\mathcal{E}_\tau$ the fact that $|\cup_{i=1}^{n} R_i(\tau)| > (1-\delta)L$ by the round L of the algorithm and we denote by $\bar{\mathcal{E}}_\tau$ the complement of that event. Next by Boole's inequality we have $\mathbf{Pr}[\cup_\tau \bar{\mathcal{E}}_\tau] \leq \sum_\tau \mathbf{Pr}[\bar{\mathcal{E}}_\tau] \leq \frac{1}{n^\beta}$, where $\beta = \alpha - 1 > 0$. Hence each task is performed at least $\Theta(\log n)$ times *whp*, i.e., $\mathbf{Pr}[\cap_\tau \mathcal{E}_\tau] = \mathbf{Pr}[\overline{\cup_\tau \bar{\mathcal{E}}_\tau}] \geq 1 - \frac{1}{n^\beta}$.

The following lemma shows that *whp* after $\Theta(\log n)$ rounds of the algorithm every worker obtains every triple generated in the system by either generating it locally or by means of gossiping.

A somewhat similar result is shown by Fraigniaud and Glakkoupis [7] who study the communication complexity of rumor-spreading in the random phone-call model. They consider n players communicating in parallel rounds, where in each round every player u calls a randomly selected communication partner. Player u is allowed to exchange information with the partner, either by *pulling* or *pushing* information. In order to avoid repetition, we anchor part of our proof to their results related to the *push* part of their algorithm.

The following lemma, proved in [7], shows that every triple $\vartheta = \langle v_j, i, r \rangle$ (in their work a rumor ρ) is disseminated to at least $\frac{3}{4}n$ workers (in their work players) *whp*.

Lemma 3. *With probability $1 - n^{-3+o(1)}$, at least $\frac{3}{4}$ fraction of the players knows ρ at the end of round $\tau = \lg n + 3 \lg \lg n$.*

Our proof also makes use of the *Coupon Collector's* problem [12]:

Definition 1. *The Coupon Collector's Problem (CCP). There are n types of coupons and at each trial a coupon is chosen at random. Each random coupon is equally likely to be of any of the n types, and the random choices of the coupons are mutually independent. Let m be the number of trials. The goal is to study the relationship between m and the probability of having collected at least one copy of each of n types.*

In [12] it is shown that $E[X] = n \ln n + O(n)$ and that *whp* the number of trials for collecting all n coupon types lies in a small interval centered about its expected value. Now we state and prove the needed lemma.

Lemma 4. *If every task is performed $\Theta(\log n)$ times, then* whp *in $\Theta(\log n)$ rounds of the algorithm each worker acquires the results for every task.*

Proof. Let us assume that in some round r task j is performed by worker i; thus a triple $\vartheta \equiv \langle v_j, i, r\rangle$ is generated by worker i, where v_j is the calculated value of task j.

By applying Lemma 3 to our algorithm we infer that in $\Theta(\log n)$ rounds of algorithm A at least $\frac{3}{4}n$ of the workers become aware of triple ϑ *whp*. Next consider any round d such that at least $\frac{3}{4}n$ of the workers are aware of triple ϑ for the first time. Let us denote this subset of workers by S_d ($|S_d| \geq \frac{3}{4}n$.)

We denote by U_d the remaining fraction of the workers that are not aware of ϑ. We are interested in the number of rounds required for every worker in U_d to learn about ϑ *whp* by receiving a message from one of the workers in S_d in some round following d.

We show that, by the analysis very similar to CCP, in $\Theta(\log n)$ rounds triple ϑ is known to all workers *whp*. Every worker in $\mathcal{P}$ has a unique id, hence we can think of those workers as of different types of coupons and we assume that the workers in S_d collectively represent the coupon collector. In this case, however, we do not require that every worker in S_d contacts all workers in U_d *whp*. Instead, we require only that the workers in S_d *collectively* contact all workers in U_d *whp*. According to our algorithm in every round every worker in $\mathcal{P}$ ($S_d \subset \mathcal{P}$), selects a worker uniformly at random and sends all its data to it. Let us denote by m the collective number of trials by workers in S_d to contact workers in U_d. According to CCP if $m = O(n \ln n)$ then *whp* workers in S_d collectively contact every worker in $\mathcal{P}$, including those in U_d. Since there are at least $\frac{3}{4}n$ workers in S_d then in every round the number of trials is at least $\frac{3}{4}n$, hence in $O(\ln n)$ rounds *whp* all workers in U_d learn about ϑ. Therefore, in $\Theta(\log n)$ rounds *whp* all workers in U_d learn about ϑ.

Thus we showed that if a new triple is generated in the system then *whp* it will be known to all workers in $\Theta(\log n)$ rounds. Now by applying Boole's inequality we want to show that *whp* in $\Theta(\log n)$ rounds all generated triples are spread among all workers.

According to our algorithm every worker generates $L = \Theta(\log n)$ triples before it terminates. We have n workers which means that by the end of the algorithm the number of generated triples is $\Theta(n \log n)$. Let us denote the set of all generated triples by $\mathcal{V}$. Let $\overline{\mathcal{E}}_\vartheta$ be the event that some triple ϑ is not spread around

among all workers when the algorithm terminates. In the preceding part of the proof we have shown that $\mathbf{Pr}[\overline{\mathcal{E}}_\vartheta] < \frac{1}{n^\beta}$, where $\beta > 1$. By Boole's inequality, the probability that there exists one triple that did not get spread to all workers, can be bounded as

$$\mathbf{Pr}[\cup_{\vartheta \in \mathcal{V}} \overline{\mathcal{E}}_\vartheta] \leq \Sigma_{\vartheta \in \mathcal{V}} \mathbf{Pr}[\overline{\mathcal{E}}_\vartheta] = \Theta(n \log n) \frac{1}{n^\beta} \leq \frac{1}{n^\gamma}$$

where $\gamma > 0$. This implies that upon termination every worker collects all $\Theta(n \log n)$ triples generated in the system *whp*.

Next theorem shows that at termination the correct result for each task is obtained from the collectively computed results, whether correct or incorrect.

Theorem 1. *In $\Theta(\log n)$ rounds algorithm A produces the results of all n tasks correctly at every processor* whp.

Proof. We first prove that at termination the algorithm computes correctly a majority of the results for any task τ *whp*. Then we argue that *whp* at termination the result computed for each task by any processor is correct.

In order to prove the first step we estimate (with a concentration bound) the number of times the results are computed correctly. Then we estimate the bound on total number of times task τ was computed (whether correctly or incorrectly), and we show that a majority of the results are computed correctly.

Let us consider random variables X_{ir} that denote the success or failure of correctly computing the result of some task τ in round r by worker i. Specifically, $X_{ir} = 1$ if in round r, worker i computes the result of task τ correctly, otherwise $X_{ir} = 0$. According to our algorithm we observe that $\mathbf{Pr}[X_{ir} = 1] = \frac{q_i}{n}$ and $\mathbf{Pr}[X_{ir} = 0] = 1 - \mathbf{Pr}[X_{ir} = 1]$, where $q_i \equiv 1 - p_i$.

Let $X_r \equiv \sum_{i=1}^{n} X_{ir}$ denote the number of correctly computed results for task t among all workers during round r. By linearity of expected values of a sum of random variables we have

$$\mathbf{E}[X_r] = \mathbf{E}[\sum_{i=1}^{n} X_{ir}] = \sum_{i=1}^{n} \mathbf{E}[X_{ir}] = \sum_{i=1}^{n} \frac{q_i}{n}$$

We denote by $X \equiv \sum_{r=1}^{L} X_r$ the number of correctly computed results for some task τ at termination. Again, using the linearity of expected values of a sum of random variables we have

$$\mathbf{E}[X] = \mathbf{E}[\sum_{i=1}^{n} \sum_{r=1}^{L} X_{ir}] = \frac{L}{n} \sum_{i=1}^{n} q_i$$

Note that since $\frac{1}{n} \sum_{i=1}^{n} q_i > \frac{1}{2} + \varepsilon$, for some fixed $\varepsilon > 0$, there exists some $\delta > 0$, such that, $(1 - \delta)\frac{L}{n} \sum_{i=1}^{n} q_i > (1 + \delta)\frac{L}{2}$. Also, observe that the random variables $X_1, X_2, \cdots, X_L$ are mutually independent. Therefore, by applying Chernoff bound on $X_1, X_2, \cdots, X_L$ we have

$$\mathbf{Pr}[X \leq (1-\delta)E[X]] \equiv \mathbf{Pr}[X \leq (1-\delta)\frac{L}{n}\sum_{i=1}^{n} q_i] \leq e^{-\frac{\delta^2 L(1+\delta)}{4(1-\delta)}} \leq \frac{1}{n^{\alpha_1}}$$

where $\alpha_1 > 1$ such that $L = k \log n$ for some sufficiently large constant $k > 0$.

Let us now count the total number of times task τ is chosen to be performed during the execution of the algorithm in the course of the first L rounds. We represent the choice of task τ by worker i during round r by the random variable Y_{ir}. We assume $Y_{ir} = 1$ if τ is chosen by worker i in round r, otherwise $Y_{ir} = 0$. Since Y_{ir}'s are mutually independent we have $\mathbf{E}[Y_{ir}] = \frac{1}{n}$. We denote by $Y \equiv \sum_{i=1}^{n}\sum_{r=1}^{L} Y_{ir}$ the number of times task t is computed at termination. By linearity of expected values we have $E[Y] = L$. Then by applying Chernoff bound for the same $\delta > 0$ chosen as above we have

$$\mathbf{Pr}[Y \geq (1+\delta)\mathbf{E}[Y]] \equiv \mathbf{Pr}[Y \geq (1+\delta)L] \leq e^{-\frac{\delta^2 L}{3}} \leq \frac{1}{n^{\alpha_2}}$$

for some $\alpha_2 > 1$. Hence, applying Boole's inequality to the bounds on the above two events

$$\mathbf{Pr}[\{X \leq (1-\delta)\frac{L}{n}\sum_{i=1}^{n} q_i\} \cup \{Y \geq (1+\delta)L\}] \leq \frac{2}{n^{\alpha}}$$

where $\alpha = \min\{\alpha_1, \alpha_2\} > 1$

Therefore, from above and by using $(1-\delta)\frac{L}{n}\sum_{i=1}^{n} q_i > (1+\delta)\frac{L}{2}$ we have

$$\begin{aligned}
\mathbf{Pr}[Y/2 < X] &\geq \mathbf{Pr}[\{Y < (1+\delta)L\} \cap \{X > (1-\delta)\frac{L}{n}\sum_{i=1}^{n} q_i\}] \\
&= 1 - \mathbf{Pr}[\{Y \geq (1+\delta)L\} \cup \{X \leq (1-\delta)\frac{L}{n}\sum_{i=1}^{n} q_i\}] \\
&\geq 1 - \frac{1}{n^{\beta}}
\end{aligned}$$

for some $\beta > 1$. Hence, at termination of the algorithm *whp* the majority of calculated results for task τ are correct. Let us denote this event by $\mathcal{E}_t$.

From above we have $\mathbf{Pr}[\overline{\mathcal{E}}_\tau] \leq \frac{1}{n^{\beta}}$. Now, by Boole's inequality we obtain

$$\mathbf{Pr}[\bigcup_{t \in \mathcal{T}} \overline{\mathcal{E}}_\tau] \leq \sum_{\tau \in \mathcal{T}} \mathbf{Pr}[\overline{\mathcal{E}}_\tau] \leq \frac{1}{n^{\beta-1}} \leq \frac{1}{n^{\gamma}}$$

where $\mathcal{T}$ is the set of all n tasks, and $\gamma > 0$.

By Lemma 4 *whp* all calculated results of every task are disseminated across all workers. Thus, the majority of the results computed for any task at any worker is the same among all workers, and moreover it is correct *whp*. Recall that according to our algorithm (line 14) every processor computes the result of every task by taking the plurality of calculated results, and hence the claim of the theorem.

Algorithm A terminates after $\Theta(\log n)$ rounds and thus every processor generates $\Theta(\log n)$ triples. This implies that at termination $\Theta(n \log n)$ triples are generated. To obtain consistent and correct results among all processors *whp* we want all processors to hold the same set of triples. Each triple consists of the calculated result of a task, the id of the processor that performed the task, and the round number. Thus $\Theta(\log n)$ bits are required to represent each triple. Next we assess work, message, bit, and space complexities.

Theorem 2. *Algorithm A has work complexity $\Theta(n \log n)$, message complexity $\Theta(n \log n)$, bit complexity $O(n^2 \log^3 n)$, and space complexity $\Theta(n^2 \log^2 n)$.*

Proof. Algorithm A terminates in $\Theta(\log n)$ rounds, thus its work is $\Theta(n \log n)$. In every round every worker sends one message to a randomly chosen worker (including itself). Hence, the message complexity is $\Theta(n \log n)$.

Now let us estimate bit complexity. For every performed task algorithm adds a triple to the result set, where $\Theta(\log n)$ bits are required to store a triple. According to our algorithm every processor sends $O(n \log^2 n)$ bits in every round, where the additional multiplicative $\log n$ factor represents the number of different triples per task. On the other hand, the algorithm terminates in $\Theta(\log n)$ rounds, hence every processor communicates $O(n \log^3 n)$ bits of information to other processors. Therefore, the bit complexity of the algorithm is $O(n^2 \log^3 n)$.

Finally, it is easy to see that space complexity of the algorithm is $\Theta(n^2 \log^2 n)$. Indeed, by termination of the algorithm every processor i holds an array of sets R_i and the result vector $Results_i$, for $i \in [n]$. The result vector consists of just n bits. On the other hand, according to Lemmas 2 and 4, after algorithm terminates each $R_i[j]$ contains $\Theta(\log n)$ triples *whp*, hence the number of bits required for each $R_i[j]$ is $\Theta(\log^2 n)$, where $i, j \in [n]$. Considering that the number of tasks and processors is n, the total bit complexity is $\Theta(n^2 \log^2 n)$.

Finally, we extend the algorithm to handle the number of tasks larger than the number of processors as follows. Let $\mathcal{T}' = [t]$ be the set of unique task identifiers, where $t \geq n$. We segment the t tasks into groups of $\lceil t/n \rceil$ tasks, and construct a new array of super-tasks with identifiers $\mathcal{T} = [n]$, where each super-task takes $\Theta(t/n)$ time to perform by any processor. For a super-task τ, the result v_τ is now a sequence of $\lceil t/n \rceil$ bits, instead of a single bit. We now use algorithm A, where the only difference is that each *Compute* stage takes $\Theta(t/n)$ time, and the data structures are larger to accommodate the results consisting of $\lceil t/n \rceil$ bits. We call the resulting algorithm A' and we show the following.

Theorem 3. *For $t \geq n$ algorithm A' has time complexity $\Theta(\frac{t}{n} \log n)$, work complexity $\Theta(t \log n)$, message complexity $\Theta(n \log n)$, bit complexity $O(t\ n \log^3 n)$, and space complexity $\Theta(t\ n \log^2 n)$.*

Proof sketch. As with algorithm A, algorithm A' takes $\Theta(\log n)$ iterations to produce the results *whp*, except that each iteration now takes $\Theta(t/n)$ time. This yields time complexity $\Theta(\frac{t}{n} \log n)$. Work complexity is then $n \cdot \Theta(\frac{t}{n} \log n) = \Theta(t \log n)$.

The message complexity remains the same at $\Theta(n \log n)$ as the number of messages does not change. The messages are larger, however, by a factor of t/n relative to the result of Theorem 2, thus the bit complexity is $O(t\ n \log^3 n)$. Lastly, the storage requirements are increased by the same factor, resulting in space complexity $\Theta(tn \log^2 n)$. □

In closing this section we note that the same results hold for models $\mathcal{F}_a$, $\mathcal{F}_b$, and $\mathcal{F}_c$, since they are direct specializations of model $\mathcal{F}$.

5 Tolerating Crash Failures

We now show that algorithm A correctly performs n tasks *whp* even if up to fn processors crash for a constant f, where $0 < f < 1$, under failure model $\mathcal{F}$. We prove that the asymptotics of the algorithm are unchanged if crashes do not invalidate the definition of model $\mathcal{F}$, meaning that the *average* probability of a non-crashed worker returning an incorrect result remains inferior to 1/2. Specifically, we show that Lemmas 2 and 4, and Theorem 1 remain valid under this model.

In any execution of Algorithm A we denote the set of processors that do not crash by $\mathcal{P}'$, and we let $n' = |\mathcal{P}'|$. As before, we start with $t = n$.

Lemma 5. *In $\Theta(\log n)$ rounds of the algorithm every task is performed $\Theta(\log n)$ times* whp*, possibly by different processors when at most fn processors can crash.*

Proof sketch. In the worst case all failure prone processors will crash in the first round of the algorithm. Thus, it is sufficient to prove that *whp* every task is performed $\Theta(\log n)$ times among the processors in $\mathcal{P}'$. In order for every task to be performed $\Theta(\log n)$ times *whp* by processors in $\mathcal{P}'$ it is sufficient to increase the value of L by a factor $\lambda = \frac{1}{1-f}$ (compared to the case without crashes). Since all processors pick a new task to be performed from the set of n tasks uniformly at random (line 10 of algorithm A) we can prove the results by carrying out the computation using Chernoff bound as in the proof of Lemma 2. □

Now we prove that *whp* after $\Theta(\log n)$ rounds of the algorithm every worker in $\mathcal{P}'$ holds the same set of triples for every task.

Lemma 6. *If processors in $\mathcal{P}'$ collectively hold $\Theta(\log n)$ calculated results for every task, then* whp *in $\Theta(\log n)$ rounds of the algorithm each processor $i \in \mathcal{P}'$ obtains all $\Theta(\log n)$ triples for every task j, when at most fn processors crash.*

Proof sketch. Consider a triple ϑ that is generated (or obtained by gossiping) by some processor in $\mathcal{P}'$. The proof of Lemma 4 uses the results from Lemma 3 and CCP. Both of these results rely on the fact that there are $\Theta(n)$ participating processors, and since there are at most fn processors that crash we have $\Theta(n)$ processors left in $\mathcal{P}'$. Therefore, following a similar line of analysis we can claim the lemma with respect to the processors that do not crash until the end of algorithm A and the triples possessed by them. □

The final theorem shows that *whp* the correct results for each task are computed in $\Theta(\log n)$ rounds by the processors in $\mathcal{P}'$.

Theorem 4. *Algorithm A computes all n tasks correctly at every live processor in* $\Theta(\log n)$ *rounds* whp *and has work* $\Theta(n \log n)$ *in the presence of at most* fn *crashes.*

Proof sketch. To prove this we need to show that, at termination, for any task t the majority of the results are computed correctly *whp*. Note that if we consider only the results (triples) that are generated by the processors in $\mathcal{P}'$ then our high probability correctness results can be shown similarly to the proof of Theorem 1. Suppose that we also consider the triples that are generated by the processors that are not in $\mathcal{P}'$. Note that according to our assumption the *average* probability of a worker returning an incorrect result remains inferior to 1/2 in spite of crashes. Hence, the probability of correctly choosing the result for a task is not affected. Since algorithm A terminates in $\Theta(\log n)$ rounds its work cannot exceed $\Theta(n \log n)$. □

Clearly in the presence of up to fn crashes the message and bit complexities, as well as the space complexity of the algorithm A remains unchanged. Although the complexity results do not change in the presence of crashes, it is important to note that the overall number of rounds may increase by a constant factor of $\lambda = \frac{1}{1-f}$.

Finally, the algorithm is extended as discussed in the previous section to deal with t tasks when $t \geq n$. Given Theorem 4, the complexity bounds established in Theorem 3 remain valid in the crash-extended failure model.

6 Simulation Results

To illustrate our analytical findings we present selected simulation results of algorithm A (for $t = n$) in model $\mathcal{F}$ and in model $\mathcal{F}_c$. We use model $\mathcal{F}$ as the most general model, and we use model $\mathcal{F}_c$ to show the behavior of the algorithm in one of the specialized settings. (We do not show simulations for all defined models for paucity of space.)

Theorems 1 and 4 show that algorithm A performs all n tasks correctly *whp* at every node in $\Theta(\log n)$ rounds. In simulations we let $L = k \log n$, where $k > 0$ is a constant. We carried out simulations for up to $n = 1000$ tasks and processors, and for modest values of $k \in \{2, 3, 4\}$. For every n paired with every k we ran the simulation for 100 times and graphed the average of the percentage of *incorrectly* calculated results as the function of n and k. In all simulations the calculated results are *always consistent* among all processors in every run of the algorithm as anticipated by Lemmas 4 and 6.

Figures 2 and 3 show results for model $\mathcal{F}_c$ and model $\mathcal{F}$ (without crashes) respectively. For model $\mathcal{F}_c$ we let $f = \frac{1}{4}$ of processors be faulty: these processors return incorrect results with probability $p = 1$. The rest of the processors are correct. For model $\mathcal{F}$ we assume that the average probability of returning incorrect results is inferior to 0.25. The results for models $\mathcal{F}_c$ and $\mathcal{F}$ are similar, showing the percentage of incorrect results is diminishing rapidly even for modest k. Analysis shows that this error can be made as small as necessary by

Fig. 2. Simulation results for model $\mathcal{F}_c$

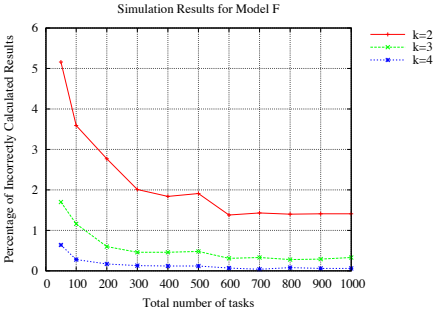

Fig. 3. Simulation results for model $\mathcal{F}$

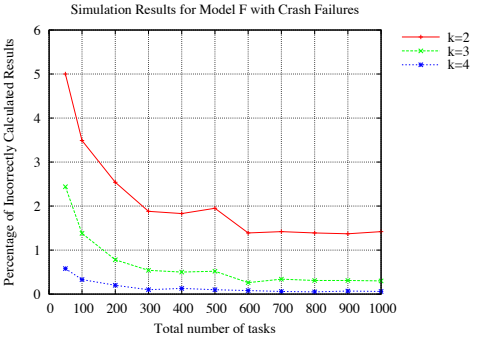

Fig. 4. Simulation for model $\mathcal{F}$ with crashes

increasing k (of course if the average probability of calculating results incorrectly tends to $\frac{1}{2}$, k may need to be substantial to guarantee the results).

Figure 4 shows the percentage of incorrectly calculated results in model $\mathcal{F}$ with crashes. Here we let $f = \frac{3}{5}$ fraction of processors be crash-prone, keeping similar probabilities of returning incorrect results as before. Hence, the average probability of returning an incorrect result is still inferior to 0.25 for all processors that do not crash. The results again show diminishing error as k grows.

7 Conclusion

Abstracting the setting of network supercomputing with untrusted workers, we defined a model of failures for workers that may return incorrect results, and we presented and analyzed a decentralized algorithm that allows correct workers to cooperatively perform a collection of tasks. The new algorithm breaks with tradition and removes the assumption of the central infallible master processor. The algorithm imposes only a logarithmic time overhead, while sharing information about the progress of computation by means of gossip. Noteworthy, each processor sends only one message for each iteration of the algorithm. We showed that the algorithm performs all tasks correctly *whp* and we developed

a simulation of the algorithm to illustrate our analytical findings. Future work includes considering more virulent failure behaviors and task sets with inter-task dependencies.

References

1. Internet primenet server, `http://mersenne.org/ips/stats.html`
2. Seti@home, `http://setiathome.ssl.berkeley.edu/`
3. Chlebus, B., Kowalski, D.: Randomization helps to perform independent tasks reliably. Random Structures and Algorithms 24(1), 11–41 (2004)
4. Dwork, C., Halpern, J.Y., Waarts, O.: Performing work efficiently in the presence of faults. SIAM J. Comput. 27(5), 1457–1491 (1998)
5. Fernandez, A., Georgiou, C., Lopez, L., Santos, A.: Reliably executing tasks in the presence of untrusted entities. In: Proc. of the 25th IEEE Symposium on Reliable Distributed Systems, pp. 39–50 (2006)
6. Fernandez, A., Georgiou, C., Lopez, L., Santos, A.: Algorithmic mechanisms for internet-based master-worker computing with untrusted and selfish workers. Tech. rep., Proc. of the 24th IEEE Int'l Symposium on Parallel and Distributed Processing (2010)
7. Fraigniaud, P., Giakkoupis, G.: On the bit communication complexity of randomized rumor spreading. In: Proc. of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2010, pp. 134–143 (2010)
8. Gao, L., Malewicz, G.: Toward maximizing the quality of results of dependent tasks computed unreliably. Theory of Computing Systems 41(4), 731–752 (2007)
9. Georgiou, C., Shvartsman, A.A.: Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity. Springer, Heidelberg (2008)
10. Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computation. Kluwer Academic Publishers (1997)
11. Konwar, K.M., Rajasekaran, S., Shvartsman, M.M.A.A.: Robust Network Supercomputing with Malicious Processes. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 474–488. Springer, Heidelberg (2006)
12. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (1995)
13. Paquette, M., Pelc, A.: Optimal decision strategies in byzantine environments. Parallel and Distributed Computing 66(3), 419–427 (2006)