Incentivizing Energy Efficiency and Carbon Neutrality in Distributed Computing Via Cryptocurrency Mechanism Design

Preprint · January 2022

DOI: 10.13140/RG.2.2.26393.75362

Levi Rybalov

5 PUBLICATIONS **36** CITATIONS

Incentivizing Energy Efficiency and Carbon Neutrality in Distributed Computing Via Cryptocurrency Mechanism Design

Levi Rybalov

January 1, 2022

Abstract

Cryptocurrencies and blockchain technologies have exploded in popularity in recent years. However, a huge amount of energy is consumed by many of these cryptocurrencies, exacerbating climate change, and most of the energy is used for computations which have no value other than securing the blockchain. In light of this fact, a number of cryptocurrencies were created with the intent of being secured by, or rewarding, useful computations. In this paper, we design a reward mechanism for the distributed computing platform BOINC (the Berkeley Open Infrastructure for Network Computing), which is used by many universities and other institutions around the world. The reward mechanism is separate from, but intended to be implemented on top of, a cryptocurrency protocol. The mechanism achieves the three main intended goals of the BOINC credit system via a multi-dimensional generalization of the proportional allocation reward mechanism of Bitcoin, and an anti-cheating mechanism adapted from an existing BOINC cheat-detecting mechanism. Since the generalization must be approximated, we introduce several ways of doing so, and analyze the game theoretic aspects of the approximation. One of these approximations requires a hardware profiling database, which has many use cases well beyond BOINC or cryptocurrencies.

We also explore market mechanisms that balance the preferences of BOINC users with the strengths of their machines using the well-known Top Trading Cycles algorithm and some of its generalizations, with the effect of increasing network output and energy efficiency. Additionally, we explore avenues to make the cryptocurrency carbon-neutral. We conclude with an overview of some existing cryptocurrencies that currently reward distributed computing.

1 Introduction

There are currently thousands of cryptocurrencies in existence, and the technology is under heavy scrutiny by critics who call attention to the environmental impact of many Proof-of-Work (PoW) based coins, most notably Bitcoin. A common criticism of PoW coins like Bitcoin is that the SHA-256 hashing algorithm it uses has no useful output. A number of cryptocurrencies have been designed to deal with this issue. In this paper, we design a reward mechanism for distributed computing using the distributed computing platform BOINC, the Berkeley Open Infrastructure for Network Computing.

BOINC is a middleware that connects *projects* with *crunchers*. Projects are often run by institutions like universities, but can also be run by individuals, companies, or anyone with a server and enough willpower. They send computational workloads to volunteers, called *crunchers*, who use their hardware to *crunch* the computational workloads. All details regarding BOINC necessary for this paper will be explained in Section 1.1. Despite the focus on BOINC, the methods laid out in this paper can also be applied to other distributed computing platforms.

Before jumping into the details of how to design such a cryptocurrency, there are some pertinent questions that must be answered as to why such a cryptocurrency should be created at all. The first question is, what does this currency do that cannot be done with fiat? Indeed, with a number of countries exploring government-approved digital currencies, what is even the purpose of decentralized cryptocurrencies?

One of the primary benefits of a cryptocurrency – and the one that will be most leveraged in this paper – is that its stakeholders have the ability to determine the incentive mechanisms underlying it; certain behaviors can be encouraged, and others discouraged. This is enabled by the numerous checks and balances core to the philosophies underlying cryptocurrencies – the intertwined and overlapping power centers, like core developers, miners (in PoW based coins) or stakers (in Proof-of-Stake (PoS) based coins), merchants and individual users, and payment services – combined with the implicit voting inherent to forking a blockchain in order to update or change the direction of the coin [8], as well as exchanges, which wield significant power in the cryptocurrency world.

Since BOINC is being used as a foundation for this reward mechanism, it is also worth asking what the cryptocurrency that enables these rewards could do that already-existing BOINC credits cannot. Primarily, a cryptocurrency that rewards BOINC computations would enable transferability and fungibility, especially in light of the fact that BOINC projects each have their own credit reward schemes. It is generally acknowledged that mechanism design – the study and design of incentive structures – is much easier with money than without money, and that transferable utility (via the ability to use the cryptocurrency as payment) changes (but mostly expands) the range of possible incentive structures that can be designed.

If we wanted to translate the traditional PoW reward mechanism to a distributed computing setting, the mechanism for the latter would reward crunchers proportionally to their computational contributions – what a cruncher puts into the network, they get out of it in cryptocurrency. However, there is a major difference between traditional PoW mining and computations in distributed computing. In the former, there is only a single function – in the case of Bitcoin, it is the SHA-256 hashing algorithm. All hardware will have a fixed hashrate on the algorithm, all else being equal. However, in distributed computing, there is a wide variety of types of computations, and the performance of different types of hardware on these computations does not vary linearly with the properties of the hardware.

1.1 BOINC Preliminaries

What follows is a brief overview of BOINC, particularly the elements of it that are necessary for understanding this paper; see Anderson's paper [3] for a comprehensive overview.

BOINC is an open-source middleware that connects two main groups: the first group consists of *projects*, which are individuals or institutions that need a large amount of computation that can be divided into smaller, discrete workloads called *jobs*; the second consists of individuals called *crunchers* who perform those computations with their hardware and return the results to the project servers.

Projects often subdivide the types of computations they need completed into *applications*. Applications are usually a program, with each job of that application having different inputs to that program. A project could have some applications that only run on CPUs, and others that are GPU-based and requires very little CPU time. This paper will base the reward mechanism on applications, rather than projects, for reasons that will become clear in later sections.

Since crunchers can either intentionally or unintentionally return bad results, projects will often send out multiple copies of the same job, which are called *instances* of that job; each instance is also referred to as a Work Unit (WU). If a project determines that the results of a WU are valid, which is often based on finding a *quorum* of results that agree with each other within some bounds determined by the project, then that cruncher is awarded *credits* for completing that WU. The bounds are necessary because the multitude of different hardware vendors and operating systems can produce different results despite running the same programs with the same inputs, even when the results should be deterministic. The credits are non-transferable and non-fungible, and are used as an incentive to retain crunchers. The goals of the credit system are reproduced from [3] below:

- be device neutral: similar jobs (in particular the instances of a replicated job) should get about the same credit regardless of the host and computing resource on which they are processed;
- be project neutral: a given device should get about the same credit per time regardless of which project it computes for;
- resist credit cheating: i.e. efforts to get credit for jobs without actually processing them.

In practice, project administrators choose their own credit-granting schemes, and reward different amounts of credits for the completion of different WUs based on a variety of factors. Oftentimes, the credit systems implemented by projects fail to achieve the first goal of the credit system. Since the actual number of credits awarded is also decided arbitrarily by the projects, the second goal has not at all been met, with some projects awarding orders of magnitude more credits for approximately the same amount of computation. Likewise, the third goal has not been achieved – the most recent credit design [1] is known to have vulnerabilities that allow crunchers to receive much more credit than they should.

Every cruncher has an identifier called the *cross-project identifier* (CPID), which is an identity that represents a cruncher and their registered machines across all BOINC projects. Under the current credit scheme, newly awarded credits for every project are factored into both the total credit, and a number called the Recent Average Credit (RAC), which is a measure of the most recent contribution of a CPID to a given project. The maximum RAC that a machine can achieve is equal to the maximum number of credits that that machine can achieve in a day. The RAC function has a half-life build-up/decay time of one week, meaning that after one week of consistent crunching, the machine would achieve 50% of its maximum possible RAC, after two weeks, 75%, etc. Likewise, if a machine stops crunching at any moment in time, then after one week, its RAC would be 50% of what it was at the time that the machine stopped crunching, after two weeks, 25%, etc. See Appendix A for a code snippet simulating RAC build-up.

1.2 The Bitcoin Allocation Rule and Its Properties

In [6], Chen, Papadimitriou, and Roughgarden establish an axiomatic approach to block rewards in the Bitcoin blockchain, demonstrating that the current proportional allocation rule used by Bitcoin is the unique allocation rule that satisfies a number of properties: non-negativity, symmetry, (strong) budget balance, sybil-proofness, and collusion-proofness. The definition of proportional allocation is simple: for a miner i, the expected rewards p_i from a single block is their hash rate h_i divided by the hashrate of all miners:

$$p_i = \frac{h_i}{\sum_{j \in [n]} h_j}$$

where $[n] = 0, 1, \dots n$, where n is the number of miners. We will re-use this latter notation in the rest of this paper. Note that the expected rewards for a miner are linear in that miner's hash rate.

The non-formal definitions of the aforementioned desired properties are reproduced below:

- *Non-negativity.* Expected rewards (the p_i's) should be nonnegative. That is, the protocol cannot require payments from miners.
- *Budget-balance.* The protocol cannot be "in the red," meaning the sum of expected block rewards cannot exceed the unit of block reward available. *Strong* budget-balance insists that the entire unit of block reward is allocated, while *weak* budget-balance allows the protocol to withhold some of the block reward from miners.
- *Symmetry.* The allocation rule should not depend on the names of the miners (i.e., their public keys), only on their contributed hash rates.
- *Sybil-proofness.* No miner can possibly benefit by creating many accounts and splitting its mining power among them.
- *Collusion-proofness.* Two or more miners cannot benefit by pooling their mining resources and somehow splitting the proceeds. This property has different variants depending on what types of payments between colluding miners are permitted; see Section 2.2.

The authors made the above claims for the case of risk-neutral miners. They also examined the cases for risk-averse (and risk-seeking) miners, and supplied corresponding possibility and impossibility results.

1.3 Results

In this paper, we introduce an analogue of the proportional allocation rule used in Bitcoin for a new BOINC credit system, addressing the first two goals of the credit system described above, as well as introduce anti-cheating mechanisms for risk-neutral crunchers (with results that can be extended for risk-seeking crunchers), addressing the third goal of the credit system. We demonstrate how this new proportional allocation rule and the anti-cheating mechanisms incentivize energy efficiency, and describe a path towards carbon-neutrality.

In Section 2, a generalization of the proportional allocation rule used in Bitcoin to the multi-dimensional case is constructed via an *Equivalence Ratio* between applications, which can be seen as an exchange rate between the WUs of different applications. This mechanism is also shown to incentivize energy efficiency. However, calculating the parameters necessary for the generalization is difficult in practice, so approximations must be made. The necessity of an approximation makes it so that sybil-proofness and collusion-proofness are no longer automatically guaranteed by the construction of the rule, and so these properties must be recovered.

In Section 3, a linear regression approach is used to approximate the Equivalence Ratio, and some issues and game-theoretic aspects of the approximation are examined.

In Section 4, another approach to approximating the Equivalence Ratio based on benchmarks of every machine on the network is introduced. This approach offers a wide variety of benefits extending far beyond cryptocurrencies and BOINC, but suffers from the inherent unreliability and unverifiability of benchmarks returned by crunchers.

In Section 5, an anti-cheating mechanism is introduced, which addresses the third intended goal of the BOINC credit system. However, a number of open problems remain, especially regarding the possibility of projects cheating by recognizing fake results as legitimate.

In Section 6, the diverging incentives caused by the difference between applications that reward crunchers the most and the applications that those crunchers most prefer is addressed by a reduction to a market problem and invoking the Top Trading Cycle mechanism and its generalizations, with the most general case remaining an open problem.

In Section 7, opportunities to offset carbon emissions and imbue sustainability into the fabric of the cryptocurrency are explored.

In Section 8, a brief overview of some existing cryptocurrencies which reward distributed computing is provided.

2 Equivalence Ratio

In this section we develop an *allocation rule*, which is a function that maps the set of crunchers' computational contributions to a reward vector. Since each cruncher can control more than one machine, we frame the problem in terms of machines rather than crunchers. This can also be viewed as each cruncher being broken up into multiple crunchers, each of whom owns one machine.

For any given machine under constant utilization, any given application with a stationary probability distribution over parameter inputs, the runtimes of WUs from that application and on that machine follow some probability distribution. By the Law of Large Numbers and the Central Limit Theorem, the sample average of the runtimes converges to the true average of the runtimes. Thus, if rewards are based purely on the number of completed WUs of a particular application, then those rewards would converge to what they would have been if the rewards had been based on the runtimes of the individual WUs. For that reason, the allocation rule rewards the same amount for each WU from a given application, which in expectation accomplishes the first goal of the BOINC credit system. While the remainder of this paper could be easily modified to account for the credit schemes that projects currently use, dealing with WUs removes many of the complications and potential manipulations involved with calculating credit, without losing any necessary properties. However, an allocation rule based on this principle is still susceptible to manipulations via *cherry-picking* attacks, which is discussed in Section 5.

2.1 Setting

We can now formalize the setting: stripping down the environment of BOINC to the essential components, the setting consists of a set of machines $M = \{m_1, m_2, \ldots, m_{|M|}\}$, and a set of applications $A = \{a_1, a_2, \ldots, a_{|A|}\}$. Every machine $m \in M$ is assumed to have the necessary components required for crunching WUs (a CPU, RAM, main memory etc.), and may have co-processors like GPUs.

In order to construct the allocation rule, we need to formally define the computational contribution of a machine. Since each machine can crunch different applications, the concept of a single hash rate like in Bitcoin – or more generally, a single function – no longer applies. Rather, the computational contribution of a machine m_i can be represented as a tuple $w_i = (w_{i1}, w_{i2}, \dots w_{i|A|})$, where w_{ij} is the WU contribution of machine m_i to application a_j measured in some arbitrary time unit. For simplicity, we can assume that this time unit is equivalent to the length of the period for which crunchers are rewarded (this is analogous to the time between blocks in Bitcoin, which is designed to be about 10 minutes), which we will call the *reward period*. Then, the contribution of every machine to every application can be summarized in an $|M| \times |A|$ matrix W, which is the input data to the allocation rule.

The output of the allocation rule is a reward vector $\mathbf{r} = (r_1, r_2, \dots, r_{|M|})$, where r_i is the reward allocated to machine m_i. In the literature, the reward vector is usually called the payoff vector. Note that in contrast to the randomized reward allocation rule used in Bitcoin, rewards in this setting will be deterministic. For now, the reward mechanism should be thought of as being an independent layer on top of some underlying blockchain protocol, whether PoW, PoS, or some other protocol – this topic will be explored more in Section 5.

Throughout the remainder of this paper, when machines are referred to as having agency, it should be assumed that the machine's owner is the one making the strategic decisions.

2.2 Allocation Rule

The approach to constructing the allocation rule and reward vector is straightforward: find an equivalence between the WUs of different applications, and use the equivalence to determine a universal computational contribution for each machine, analogous to the hash rate in Bitcoin.

In order to construct the equivalence, we need to define the *WU completion rate*, which is a measure of how many WUs a machine can crunch on an application.

Definition 2.1. Let ρ^i_j be the WU completion rate for machine m_i on application a_j. The WU completion rate ρ^i_j is the maximum number of WUs that machine m_i can accomplish on application a_j in a reward period.

In practice, BOINC crunchers can set limitations on how much their machines crunch. For the purposes of this paper, machines that are not operating at full capacity can be viewed as less powerful machines operating at full capacity, or virtual machines that have only subsets of their hosts' original computational power available (in fact, a number of BOINC applications run in virtual machines, which negatively impacts performance).

The next step is to define the *Equivalence Ratio* (ER) of a machine. The ER for a machine is a tuple consisting of the machine's WU completion rates on each of the applications. Extending the notation of ρ^i_j, denote by ρ^i the tuple representing the ER for m_i. Then, for every $i \in [|M|]$,

Definition 2.2.

$$\rho^i \coloneqq (\rho^i_1,\ \rho^i_2, \dots,\ \rho^i_{|A|})$$

However, $\rho^{i'}$ obtained by normalization on another $m_{i'} \neq m_i$ would yield a different ER (assuming that m_i and $m_{i'}$ are not identical machines). The next step is to extend the notion of ER beyond a single machine.

Beginning with the base case, the ER for a set of machines $M = \{m_1\}$ is calculated by considering how much this set can accomplish on each application – in this case, the ER is ρ^1. Extending this logic to two machines, the ER of a set of two machines $M = \{m_1, m_2\}$ is calculated by considering how much m_1 and m_2 can accomplish on each application *together* – this would be the element-wise sum of ρ^1 and ρ^2. Generalizing, the network-wide ER can be calculated by considering how much the set of all machines in M can accomplish on each application together. Extending notation again, we denote by ρ_j the j^{th} component of network-wide ER.

Definition 2.3. The j^{th} component of the network-wide Equivalence Ratio is

$$\rho_j = \sum_{i \in [|M|]} \rho_j^i$$

Extending notation one last time, we denote by ρ the network-wide ER.

Definition 2.4. The network-wide Equivalence Ratio ρ can be written as

$$\rho \coloneqq (\rho_1,\ \rho_2, \dots,\ \rho_{|A|})$$

The network-wide ER ρ is the basis for the allocation rule. Let R be the total amount of rewards allocated to machines during a reward period. The allocation rule is described in Algorithm 1. There is a direct analogy to the proportional rule from Bitcoin: the total normalized WU contribution N_i is analogous to h_i, the hash rate, and the total normalized WU contribution $TNWU$ is analogous to $\sum_{j \in [n]} h_j$. In contrast to the hash rate, notice that N_i is a dimensionless number.

Note that most advanced hardware is located in the industrialized world, a large portion of which experiences relatively cold winters. It is very common for BOINC crunchers to use their machines as alternative heat sources during these winters, meaning that there will be more total computational power on the network during winters than at other times of the year. If the rate of cryptocurrency minting remains these same throughout the year, then it will be more profitable to crunch during summer in the northern hemisphere (where the vast majority of the industrialized world, the world's population, and hardware is located), which would lead to a lot of wasted potential.

However, if the rewards are scaled to reflect the amount of computational power on the network, a cruncher could receive the same rewards with the same hardware whether or not they crunch in the summer or the winter. Since crunching during the winter offsets expenditures on heating, that would make it more economical and energy efficient to crunch in the winter. Making a further analogy to the generalized proportional allocation rules discussed in [6], R takes the place of $c(\sum_{j \in [n]} h_j)$, which is some function of the sum of all the hash rates on the network. In particular, if we set $R = TNWU$, then each cruncher is rewarded the same fixed amount for every WU of some application that they complete. Unlike in Bitcoin, this would mean that the total rewards would be proportional to the computational power on the network.

A natural question to ask about this mechanism is what kind of behavior it incentivizes. It turns out that using the ER to derive a universal BOINC credit incentivizes crunchers to direct their hardware towards applications on which they are strongest relative to the network-wide ER, if they can only crunch one application at a time. A generalization of this idea also applies to the case where a machine can crunch more than one application at a time. This is an energy-efficient mechanism in the sense that if the incentives are followed, WUs are going to be crunched by machines which will accomplish them with lower energy consumption relative to the other machines on the network. Note that crunchers may have project or application preferences of their own, which would lead them to not pursue this financial incentive. This contradiction, and attempts to resolve it, is explored more deeply in Section 6.

Theorem 1. The Reward Assignment Algorithm incentivizes directing machines towards applications on which they are strongest relative to the network-wide ER, if the machines can only crunch one application at a time.

Algorithm 1 Reward Assignment Algorithm

```
Input:
    Network-Wide ER $\rho$
    WU completion matrix $W$
    Total network-wide rewards $R$
Output:
    $|M| \times |A|$ *normalized* WU matrix $N$
    $TNWU$, the Total Normalized Work Units (network-wide)
    $r_i$, the rewards that $m_i$ receives in the time interval

Calculate $N$ and $TNWU$
Initialize $TNWU = 0$
for i = 1 ... $|M|$ do
  for j = 1 ... $|A|$ do
    $N_{ij} = W_{ij}/\rho_j$
    $TNWU = TNWU + N_{ij}$
  end for
end for
Calculate reward vector
for i = 1 ... $|M|$ do
    $N_i = \sum_{j \in [|A|]} N_{ij}$
    $r_i = \frac{N_i}{TNWU} \cdot R$
end for
```

Proof. By definition, maximizing rewards for m_i is maximizing r_i. From the definition of r_i, $\sum_{j\in[|A|]} N_{ij}$, and by extension $TNWU$, are the factors that can be directly affected by a cruncher.

As in the algorithm, let $N_i = \sum_{j\in[|A|]} N_{ij}$, and let $TNWU_{-i} = TNWU - N_i$ be the $TNWU$ without the contribution of m_i. Then, $TNWU = TNWU_{-i} + N_i$, and

$$r_i = \frac{N_i}{TNWU_{-i} + N_i}$$

r_i is an increasing function in N_i, meaning that maximizing N_i also maximizes r_i. Since $N_{ij} = W_{ij}/\rho_j$, and the ER is given, this in turn means that W_{ij} is the only factor that can be modified by the owner of m_i in order to maximize their reward.

The maximum possible W_{ij} for machine m_i and all applications $a \in A$ is in fact ρ^i by definition. So maximizing N_i becomes finding the application a' that maximizes $\rho^i_{a'}/\rho_{a'}$, i.e.

$$\underset{a'\in A}{\operatorname{argmax}} \;\; \rho^i_{a'}/\rho_{a'}$$

Thus, given some ER, the maximum possible reward for any machine that can only crunch one application at a time is realized by dedicating all of that machine's computational power to the application for which the machine's ER has the largest relative difference compared to the network-wide ER. □

Corollary 1. If a machine cannot maximize its rewards by crunching only a single application, then it can maximize its reward by crunching two or more applications which together yield a higher normalized WU contribution than the application on which it is strongest relative to the ER.

Remark. For a single machine that has both a CPU and GPU, the ER equally values CPU and GPU calculations. Since GPUs with strong FP64 capabilities tend to be more expensive, and increases in FP64 performance do not always scale at the same rate as increases in FP32 performance, FP64 can in a sense be considered a different type of computation. Then, a single machine actually also equally values CPU, GPU FP32, and GPU FP64 calculations. There is a similar phenomenon on a network-wide level, but with the added twist that all machines will have CPUs, but not all machines will have GPUs.

The "value" of CPU calculations relative to the "value" of GPU calculations depends on a number of factors and is ultimately subjective. For that reason, it may be preferable to consider ERs for CPU and GPU applications separately, and then manually choose an equivalence between the two classes – for example, by considering the average power consumption of recent CPUs and recent GPUs. This would have the effect of roughly valuing the computational output per unit of energy equally between CPUs and GPUs, thus directing focus on the most energy-efficient way of programming an application.

Remark. The ER can change based on the machines that are on the network and the applications available to be crunched. An interesting implication of this fact is that there is no intrinsic measure of computational power like FLOPs or runtime that describes the equivalence of computational requirements between applications; rather, the equivalence is relative to the machines that are on the network at any given moment in time. This novel approach diverges substantially from prior attempts to solve this problem, such as in the current credit systems used by many projects, the most recent iteration of the recommended BOINC credit system [1], or in [4], where Awan and Jarvis proposed a new type of fixed credit based on a diverse set of benchmarks.

However, determining ρ is impractical for a number of technical reasons, the most obvious and disqualifying of which is the difficulty or even impossibility of getting trustworthy benchmarks of every machine crunching every application. Even if getting trustworthy benchmarks was possible, another major issue is the difficulty of running enough WUs of each application in order to get an accurate WU completion rate. For some machines, a single WU from some applications can take days or even weeks. For that reason, the rest of this paper will deal with acquiring approximations of these values. We propose two main methods for achieving this. The first method can be implemented using only minor modifications to the data currently openly provided by BOINC projects; in particular, the data needed would be a breakdown of a CPID's RAC to more granular data. The second method requires the construction of a hardware profiling database, which could be implemented using lightweight benchmarks of each machine on the network.

Due to the fact the ER must be approximated, there must be assurances on how the actions of one machine or a group of machines can be used to alter the ER for the benefit of the machine(s). Two of the main desirable properties discussed in Section 1.2, sybil-proofness and collusion-proofness, can now be framed in the following way:

Sybil-proofness: no machine can benefit by creating more than one CPID and splitting its computational resources among those CPIDs.

Collusion-proofness: two or more machines cannot benefit by pooling their resources and splitting the rewards in some manner.

The other desirable properties – non-negativity, symmetry, and budget balance – are satisfied by construction.

3 The Linear Regression Approach

In order to build an intuition for the linear regression approach to approximating the ER, we will use an example involving a sample universe of two applications: $A = \{a_1, a_2\}$. After introducing some assumptions, we will go through the process of approximating the ER, and afterwards address each of the assumptions and their consequences in a real-world implementation of this approach.

In this universe, machine m_i can crunch either one application exclusively, the other application exclusively, or a combination of both applications. All three of these settings can be chosen by assigning different priorities (called "Resource share" in the BOINC Manager) to each of the applications.

Consider a priority vector $(u_i, 1-u_i),\ u_i \in [0,1]$ for m_i, which denotes the fraction of resources that goes to each application. One of the simplifying assumptions we will make is that a linear decrease in the WU output for one application implies a linear increase in the output of another application; that is,

$$w_i = (\rho_1^i \cdot u_i,\ \rho_2^i \cdot (1-u_i))$$

An example of a line defined by w_i ranging between 0 and 1 is shown in Figure 1.

Using this model, we begin with a set of machines crunching these two applications, each with its own computational contribution w, which is visualized as a point in two-dimensional space. Next, we run a linear regression on the points. Finally, we take the resulting best-fit line, place a line of equal slope through each of the points, and use these lines to make an approximation $\widehat{\rho^i}$ of ρ^i for every m_i. See Figure 2 for a visualization of this process.

Calculating the points of intersection with the axes is straightforward. Suppose that for each m_i, the standard-form equations describing ρ^i are $c_1a_1 + c_2a_2 + e_i = 0$, with c_1, c_2 determined by

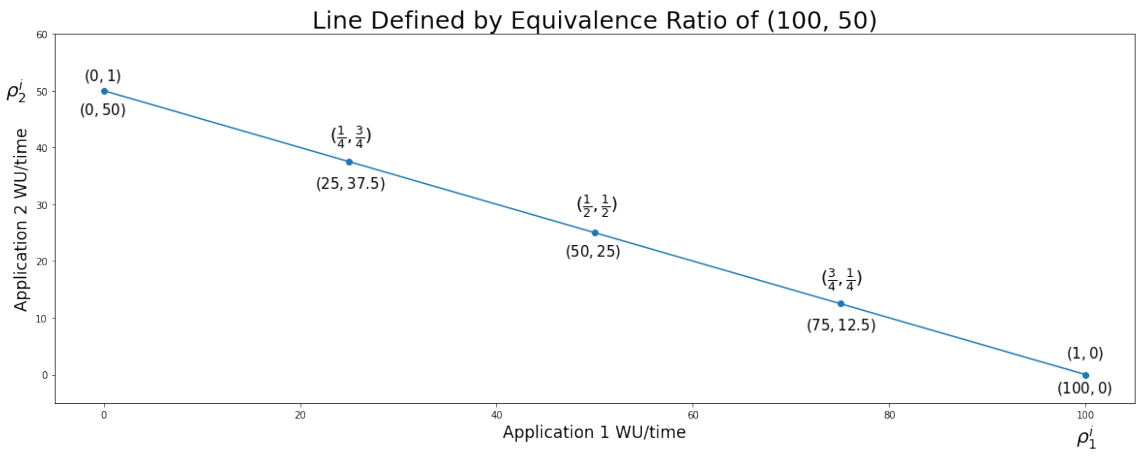

Figure 1: Example of a line defined by an ER of $(100, 50)$ for machine m_i, with u_i above, and w_i below, their corresponding points.

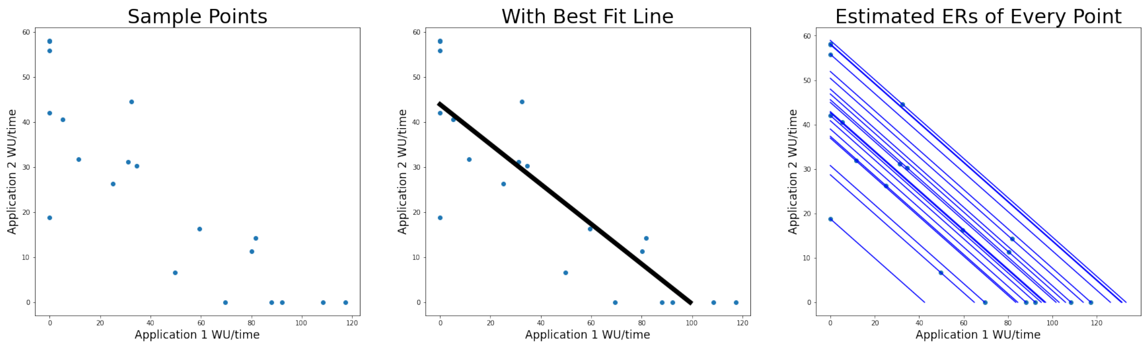

Figure 2: The Linear Regression approach for approximating the ER. On the left is a set of points w_i for each machine $m_i,\ i = 1, \dots,\ 20$, which plot the computational contributions of each machine. In the center is the regression line. On the right are lines with the same slopes as the regression line, passing through each of the points. The intersection of each line with the axes is an approximation $\widehat{\rho^i}$ of ρ^i, from which an approximation $\widehat{\rho}$ of ρ can be calculated.

the linear regression, and an intercept e_i for each machine m_i. Then, the network-wide ER would become

$$(-\frac{1}{c_1} \sum_{i \in [|M|]} e_i, -\frac{1}{c_2} \sum_{i \in [|M|]} e_i) \to (\frac{1}{c_1}, \frac{1}{c_2})$$

In three dimensions, the logic generalizes: the regression will define a plane, and the three points where the plane intersects the axes are the corresponding values in the ER. Generalizing further to $|A| > 3$ applications, every w_i can be mapped to its own $(|A| - 1)$-dimensional hyperplane, with the intersection of that hyperplane with the $|A|$ axes constituting the $\widehat{\rho^i}$. By Definitions 2.3 and 2.4, adding up these points of intersection application-wise would yield an approximation of the network-wide ER. Since the $\widehat{\rho^i}$ are all constant multiples of the points of intersection of the $(|A|-1)$-dimensional hyperplane with the $|A|$ axes, the sum would also be a constant multiple of the points of intersection, and therefore, only the output of the linear regression need be considered, just as in the preceding 2-dimensional example.

When running linear regression in practice, one application is written as a linear function of the others:

$$a_1 = c_1 + c_2 a_2 + c_3 a_3 + \dots c_{|A|} a_{|A|}$$

Then $\widehat{\rho_1} = c_1$, and

$$\widehat{\rho_i} = -\frac{c_1}{c_i} \ \forall i \neq 1$$

Factoring out c_1,

$$\widehat{\rho} = (1, -\frac{1}{c_1}, -\frac{1}{c_2}, \dots, -\frac{1}{c_{|A|}})$$

In Figure 3 are the results of the linear regression approach applied to data obtained from the Gridcoin network (a cryptocurrency that rewards crunchers for their BOINC contributions) for a subset of BOINC projects. However, the available data was limited to a list of CPIDs by their RACs on every project, and had to be whittled down. In order to implement the linear regression approach properly, this data would need to be even more fine-grained by machine (rather than by CPID), by application (rather than by project), and by number of WUs completed (rather than by RAC); these data can only be acquired via changes to the BOINC source code. Since that data is not currently available, only a subset of the data that likely corresponds to individual machines crunching one application was used. See Appendix B and the corresponding Jupyter notebook for a detailed description of how the data was handled and why the granular data is needed.

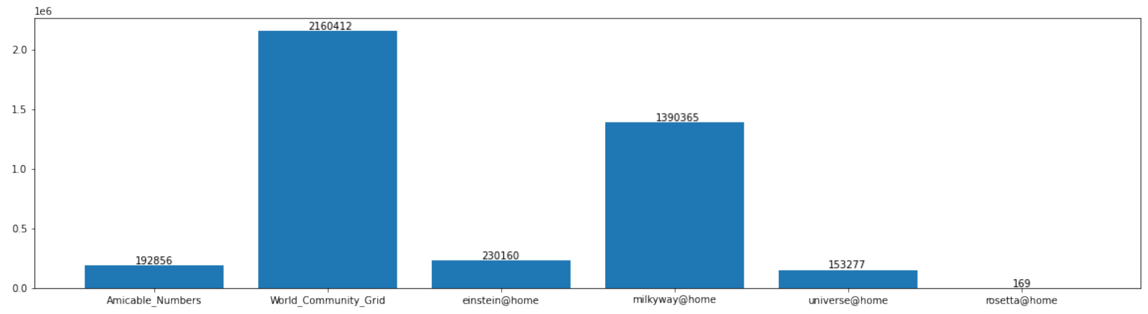

Figure 3: Actual ER Calculation

3.1 Shortcomings of the Linear Regression Approximation

3.1.1 Inaccuracies in Estimation

The assumption that under constant hardware utilization a linear decrease in output for one application implies a linear increase in output for another application is not necessarily correct. For example, by varying u, it is possible that some threshold of cache, RAM, or other resource is reached which results in a non-linear change or discontinuity in the amounts of each application that a machine could crunch. Furthermore, the underlying relationship could even be a (piecewise) curve. Another assumption made in the linear regression approach was that, in addition to the relationships being linear, the hyperplanes defining the ER for each machine had the same coefficients, which also is not necessarily correct. The following theorem bounds the error between ρ and $\widehat{\rho}$ as determined by the linear regression.

Theorem 2. The application-wise absolute difference between the true ER and the estimated ER defined by the linear regression is bounded above by the number of machines times the maximum application-wise absolute difference for any machine m_i between ρ^i and ρ:

$$|\rho_a - \widehat{\rho_a}| \leq |M| \cdot \max_{i \in [|M|]} |\rho_a^i - \widehat{\rho_a^i}|$$

$\forall a \in A$.

Proof. By Definition 2.3,

$$\rho_a = \sum_{i \in [|M|]} \rho_a^i$$

We also have that

$$\widehat{\rho_a} = \sum_{i \in [|M|]} \widehat{\rho_a^i}$$

Then

$$\begin{aligned} \rho_a - \widehat{\rho_a} &= \sum_{i \in [|M|]} \rho_a^i - \sum_{i \in [|M|]} \widehat{\rho_a^i} \\ \implies |\rho_a - \widehat{\rho_a}| &= |\sum_{i \in [|M|]} \rho_a^i - \sum_{i \in [|M|]} \widehat{\rho_a^i}| \\ &= |\sum_{i \in [|M|]} \rho_a^i - \widehat{\rho_a^i}| \\ &\leq \sum_{i \in [|M|]} |\rho_a^i - \widehat{\rho_a^i}| \\ &\leq |M| \cdot \max_{i \in [|M|]} |\rho_a^i - \widehat{\rho_a^i}| \end{aligned} \tag{1}$$

□

In practice, the estimated ER will be much more accurate, because it will not be the case that the difference between $\widehat{\rho_a^i}$ and ρ_a^i will have the same (maximum) magnitude and direction for all $i \in [|M|]$.

Another key issue is the fact that if one application uses only a CPU, while the other uses a bit of the CPU and mostly the GPU, then the linear relationship described earlier will not hold even approximately. For example, in the two-application universe, while it will be true that the $\widehat{\rho^i}$ for a particular machine on those applications will intersect the axes, the relationship will not be linear – it will look more like a rectangle with a rounded upper right corner. If there are enough such machines, then the linear regression over all the machines on the network will not even intersect the two axes on their positive rays.

As was discussed in Section 2, this problem can be addressed by separating the CPU applications and GPU applications, determining the ERs within these classes, and then choosing an equivalence between the two classes of computations. This process can also be done at an even more granular level – for example, by splitting the GPU class into FP32 and FP64 applications.

A further potential shortcoming of the linear regression approach is that converging on an accurate estimate of the ER in the limit of a large number of machines requires that the distribution of the computational powers of the machines crunching each of the applications is identical – i.e. that for each application, each machine crunching it is i.i.d. from the same distribution as all the other applications. However, there is no way to know whether this is in fact the case, or to create such a situation.

Setting aside things like driver versions, if we assume that there are no WUs that are too simple to run on any hardware, then we can conclude that WUs can only not run on some hardware if those WUs are too demanding. For example, consider again a two-application universe, and suppose that WUs from one application require more RAM or GPU VRAM than many machines have, and the other application supports all machines. A linear regression would roughly lead to an approximated ER that equates the averages of the outputs on the two applications. However, since only more powerful machines can crunch the more restrictive application, then the more powerful machines that crunch the more restrictive application will be undervalued. This would incentivize application developers to develop applications that can be run on as many different types of machines as possible.

Overall, while this may not affect strategic aspects regarding sybil-proofness and collusion-proofness, it does add another error to the way that the ER is being calculated.

3.1.2 Game-theoretic Analysis

It is possible that sybil-attacks or collusion-attacks could also be used to alter the ER. These types of attacks can be partially mitigated by using weighted linear regression, where the weights could be, for example, the computational contribution of the points. An initial ER could be calculated by standard linear regression, which is the same as every point having the same weight. Then, each machine could be weighted by its N_i, and the weighted linear regression could be iterated until convergence. It is not clear whether this method of calculating the ER is sybil-proof and/or collusion-proof.

In recent game-theoretic literature, there has been a focus on determining the computational tractability of computing moves that would benefit agents. It is possible that the weighted linear regression approach, or even the linear regression approach without weights, would have so many agents and so many unpredictable factors that computing moves that maximized rewards even in expectation could be in some complexity class that in practice renders attempts at manipulation unfeasible. Furthermore, by Theorem 1 and by construction of the weighted linear regression, any such manipulations intended to award the manipulator(s) higher rewards would still ultimately incentivize the manipulator(s) to be energy efficient. Whether or not the linear regression approach is sybil-proof or collusion-proof either by the definitions given earlier, or because finding beneficial strategies is computationally intractable, remains an open problem.

The remainder of this section is a game-theoretic analysis of what would happen if every cruncher chose to crunch applications which maximized their rewards based on the most recent ER. The case where crunchers do not act solely in their financial interests is addressed in Section 6. While it is not necessary for understanding the rest of the paper (since the underlying assumption that crunchers will only act in their financial interest almost certainly will not hold), some readers may find it interesting.

Recall that we begin with the input W, from which the ER is calculated. Suppose then that every machine tries to maximize its reward for the next reward period based on the ER from the current reward period. Recall that the true slopes of the ERs, ρ^i, may not all be constant multiples of $\widehat{\rho}$ or ρ itself – see Figure 4 for an example of the general case of the example ERs from Figure 2. Thus, depending on W, it may be possible that $\widehat{\rho}$ is changed during its next calculation such that machines have an incentive to switch to a different application (note that if in fact the slopes were all constant multiples of $\widehat{\rho}$, then switching from one application to another would not change $\widehat{\rho}$, since w_i would remain the same distance away from $\widehat{\rho}$ for any u_i).

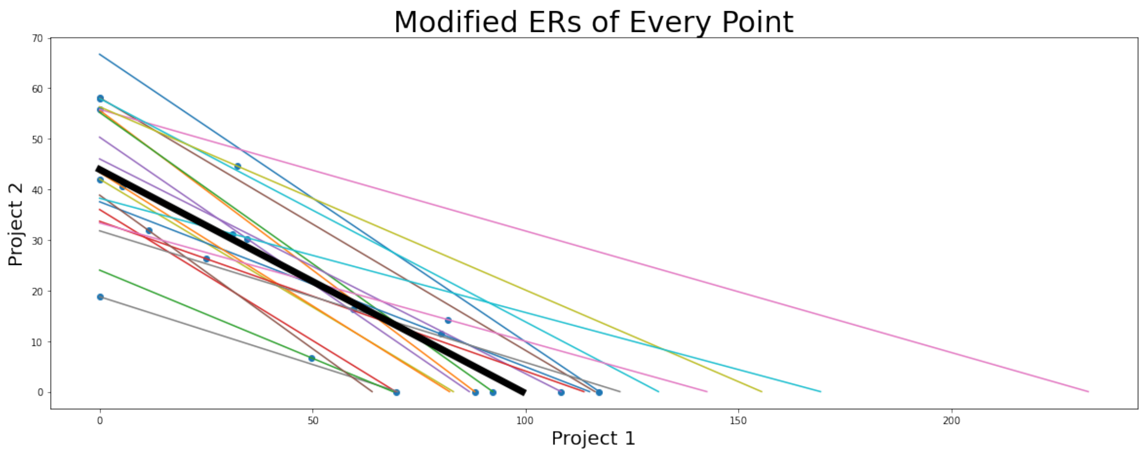

Figure 4: General ERs

A two-dimensional implementation of this game can be found in Appendix C. We were able to find instances which converged to a stable ER, diverged, and cycled. The latter finding indicates that even in this simple scenario, a pure-strategy Nash equilibrium does not exist. For much larger numbers of machines, which would be the case in practice, we did not find an instance that did not converge.

In theory, a mixed-strategy Nash equilibrium should exist. There are a finite number of crunchers, and each each cruncher has a finite (although quite large) set of strategies, which are the various combinations of WUs from different applications that they can crunch in a reward period. The simulations just described were repeated games in which crunchers were choosing the best possible strategies based on the results of the previous game.

In practice, a number of the assumptions necessary for a Nash equilibrium to be reached would not be fulfilled. As already mentioned, not all crunchers will try to maximize their rewards. Even if we kept fixed the contributions of crunchers who chose not to maximize their rewards, and only considered crunchers who did, there are a number of reasons that the game would still not converge to a Nash equilibrium. First, there is no guarantee that each remaining cruncher would execute their strategy flawlessly. Furthermore, it is not the case that a deviation of a cruncher would not cause deviations by other crunchers. Additionally, due to the dynamic nature of the network, with hardware coming online and offline, along with crunchers switching which applications they crunch for other reasons, computing the best possible strategy (or even a profitable one) might be in a complexity class that is not possible to compute.

4 Hardware Profiling Database

Like the linear regression approach introduced in Section 3, the basis of the Hardware Profiling Database (HPD) approach to estimating ρ relies on obtaining predicted ERs for each machine. At the core of the HPD method is a set of benchmarks $B = \{b_1, b_2, \ldots, b_{|B|}\}$, which are a variety of tasks that measure processor speeds, RAM size and bandwidth, bandwidth between CPU and GPU, available disk space, etc.; another attempt to construct a new BOINC credit by Awan and Jarvis also included a larger number of benchmarks than the current BOINC system uses [4]. As before, machines running at a fraction of their capacity can be viewed as either weaker machines or virtual machines. Machines without GPUs would have their GPU benchmarks equal to 0. Then, the HPD is an $|M| \times |B|$ matrix H, where H_{ij} is the output of machine m_i on benchmark $b_j \in B$.

With H in hand, it is possible to map a machine's benchmarks to its expected output on each application. Define a family of functions $F = \{f_1, f_2, \ldots f_{|A|}\}$. Each function $f \in F$ maps the benchmarks b of a machine m to an expected output on an application a. Let the benchmarks for machine m_i be b^i. Then,

$$f_a : b^i \mapsto \widehat{\rho_a^i},\ \forall a \in A,\ m \in M$$

With $\widehat{\rho_a^i}$ in hand for all machines and all applications, and using Definitions 2.3 and 2.4, $\widehat{\rho}$ can be easily constructed.

The family of functions F can be any of a variety of traditional statistical methods or machine learning algorithms. Below we list some costs and benefits of linear regression, non-linear regressors, and neural networks, but these methods are non-exhaustive and should be considered merely as examples.

Linear regression would provide weights (coefficients) for each benchmark, which helps explain which aspects of a machine are important in determining its $\widehat{\rho^i}$. However, it is limited by its linearity, which can be somewhat compensated for by feature engineering. Additionally, linear regression naturally supports multiple outputs. Further improvements can be made on top of linear regression, such as generalized linear models, or in the case that cruncher-supplied data is being used to make the mappings, using linear regression first on the machines that only crunch single applications, and once an equivalence ratio is found, adjusting it based on the machines that crunch multiple applications.

Non-linear regressors, such as random forests or gradient boosting, would also provide the relative importances of benchmarks in predicting outputs. Unlike linear regression however, they can find how non-linear relationships between the benchmarks determine a machine's performance on a given application. One downside of this approach is that these algorithms (at least, the ones mentioned) are not inherently multi-output. Multi-output wrappers do not take into account how the predictions for each component of the output depend on each other, and so only data involving single outputs (applications) could be used.

Neural networks are both multi-output and non-linear; however, they do not provide an easy way of determining relative importances of particular inputs.

4.1 Implementation Challenges

There are two main implementation issues in the HPD approach. The first issue is the difficulty of getting reliable benchmarks from each machine, or a statistically large enough sample of machines. The second issue is determining F.

Suppose that these benchmarks were obtained by sending WUs from a BOINC project. With the assumption that the benchmarks would either be posted to the blockchain or verified in some other way by the network, the verification that the benchmarks are in fact what crunchers submitted them as is trivial. The issue is that WUs are sent as binaries, and should be treated as being completely reverse-engineerable, in the sense that one could recover the original code that constructed the WU. From this point, the WU could be manipulated – a cruncher could run any code, and submit any result, that they wanted to. Unlike deterministic BOINC WUs, there is currently no way to verify whether the results from a benchmarking WU are honest, barring obvious and extreme cases. The most obvious method of ensuring that such benchmarks would be trustworthy – trusted computing – is unpopular because of privacy concerns.

From a computational standpoint, determining F is trivial. The troubles arise when determining which benchmarks and application outputs are used. In a more centralized setting, there could be a set of trusted machines, and their outputs could be used to determine F. These machines and their trustworthiness could be a combined effort of crunchers and projects, or some cryptographically secure approach. An alternative approach is to operate under the assumption that some subset of the benchmarks is unreliable, and try to deal with that problem as best as possible. Additionally, the benchmarks in H could also be combined with W in order to approximate the ER, in a similar way to what was done with linear regression, but with the addition of this new and useful dataset.

4.2 Benefits of the HPD

This database would have a wide variety of uses beyond trying to approximate the ER, and even beyond cryptocurrencies and BOINC.

First, projects would be able to use benchmarks and outputs to design their applications in ways that take advantage of potential optimizations, and plan the structure of future applications based on existing hardware or anticipated new hardware on the market.

Non-BOINC researchers could use the benchmarks for similar purposes as the project administrators – data-mine the database to find interesting patterns that could yield new insights into hardware evolution over time, which has the added advantage of attracting new researchers to the BOINC community. For example, in [9] and [10], Al-Qawasmeh et al. take a database with the same form as H and define three measures on the set of machines: Machine Performance Homogeneity (MPH), Task-Machine Affinity (TMA), and Task Type Difficulty Homogeneity (TDH). These measures are used to characterize heterogeneous computing environments to provide insights on how to optimize the use of the machines.

Crunchers could use the database to inform themselves about future hardware purchases, as well as use the information to fully predict the ERs of their machines, which is useful in addressing the tradeoff between crunching their favorite applications and the most profitable applications, as addressed in Section 6.

Companies or other institutions could use the database for their own hardware-purchasing decisions, and even use the network and benchmarking projects to send out their own benchmarking WUs while paying crunchers for their services with the cryptocurrency, creating a source of demand.

5 Anti-Cheating Mechanism

Many projects have multiple instances of a job sent to different machines in order to detect cheating or bad results due to numerical instabilities across different platforms and types of hardware. Some projects use *adaptive replication* [2] of jobs, in which trust is established between an application and a machine, resulting in that machine being sent jobs with fewer and fewer replicated instances from that application, as long as the WUs are consistently returned with reliable results by that particular machine. If results are found to be invalid, then the host has to start again from zero trust.

Both the adaptive replication policy and the reward mechanism introduced in Section 2 can be modified to reward a machine more for being trusted and consequently crunching jobs with fewer replicated instances. The modified reward mechanism would discourage cheating by making it as least as profitable (more profitable) to be honest as it is to cheat, making honesty a weakly (strongly) dominant strategy, and consequently lead to gains in energy efficiency.

This anti-cheating mechanism can be implemented on the project-level and/or the network-level. In the former case, the incentive for crunchers would be to minimize the number of replicated instances that they crunch, but otherwise there would be no inter-project effects. In the latter case, the energy efficiency of the network can be improved even further. Machines have an incentive to crunch jobs that have the fewest number of replications, and projects have an incentive to have their workloads computed quickly, want to attract the aforementioned machines, and so likewise have an incentive to reduce the number of replications. This would create a system where projects attempt to attract crunchers and compete with each other by reducing replication. Whether or not this is a good dynamic to introduce to the network is beyond the scope of this paper, but the possibility should be seen as one of a number of parameters that can be changed to incentivize different types of behavior.

Such a mechanism could also instigate new research into cross-hardware, cross-platform, cross-operating system numerical stability. Interestingly, it may also whittle down the machines that crunch particular applications to only those that consistently return reliable results (which are the machines that are numerically stable on those application/platform pairs), in a sense sorting the machines by their ability to return correct results, if the cause of incorrect results is because of some numerical instability. Note that this latter effect might undermine one of the underlying assumptions of the linear regression approximation of the ER by making the distribution of machines across applications different.

5.1 Past

Assume that a project can temporarily rescind credit for, and verify through additional replication, the results of previously approved WUs. The case where only future WUs can be additionally replicated is addressed in the subsequent section.

Consider a machine m_i that has a stream of incoming WUs from some application a. Let w_j be the j^{th} WU sent to m_i after some initial starting time. Let t_j be the number of instances of the job corresponding to w_j that were sent to different crunchers. Let $r_j = r$ be the constant reward given to the cruncher for completing a WU from application a.

In order to disincentivize cheating, the expected reward from being honest must be greater than or equal to the expected reward from cheating. One natural mechanism to try would be to scale the rewards for the completion of a given WU by the inverse of the number of instances of that job, so that $r_j = r/t_j$ rather than $r_j = r$. This can be interpreted as granting a total reward r for every job, with every machine that crunched one of the instances of that job getting its equal share of the reward. If $t_j = 1$, then m_i is the only one that crunched w_j, and so $r_j = r$, and m_i gets all of the reward; if $t_j = 2$, then m_i gets half of the reward, etc.

Consider the simple case where either there is one copy or there are two copies of w_j, i.e. $t_j = 1$ or $t_j = 2$ – the following results generalize easily. Towards the aim of discouraging cheating, project administrators replicate w_j with some probability p. We assume that the event that a job is replicated is independent of the event that m_i cheats, and that m_i chooses to cheat using some strategy of when to cheat and when to return legitimate results.

Since every job has probability p of being replicated independently of other jobs being replicated, it is only necessary to consider a subset of the WUs that m_i crunches – in particular, those for which m_i cheats. Since these WUs had probability p of being verified when considering all the WUs, they still have probability p when considering only the subset. Thus, the strategy by which m_i cheats is not relevant. Since the probability of a WU being replicated is Ber(p), the number of instances sent until one is replicated is Geom(p), and so in expectation m_i will be caught after $\frac{1}{p}$ WUs, implying that there will be $\frac{1}{p} - 1$ occurrences of m_i cheating and not getting caught.

The goal is to have the loss in rewards be greater than or equal to the expected rewards from cheating. Let n be the number of verified WUs. Now, the loss

$$L = (\text{rewards if past } n \text{ WUs were not verified}) \text{ - } (\text{rewards if past } n \text{ WUs are verified})$$

and the rewards from cheating

$$C = \text{rewards if past WUs were not verified while machine was cheating and was not caught}$$

and we want $L \geq E[C]$. When considering the results of WUs which are being retroactively replicated, it will be assumed that they are all verified as being legitimate results – otherwise, this entire process of replicating already accepted WUs can be applied recursively.

Now,

$$\begin{aligned} & L \\ &= \sum_{i=1}^{n} r_i - \sum_{i=1}^{n} r'_i \\ &= \sum_{i=1}^{n} \frac{r}{1} - \sum_{i=1}^{n} \frac{r}{2} \\ &= \sum_{i=1}^{n} (r - \frac{1}{2} r) \\ &= \frac{n}{2} r \end{aligned} \tag{2}$$

Invoking the memorylessness property of the geometric distribution, a sequence of cheat results – the aforementioned subset – will be caught in expectation $\frac{1}{p}$ WUs into the sequence. Thus,

$$E[C] = (\frac{1}{p} - 1)\frac{r}{1}$$

Finally,

$$L \geq E[C] \implies \frac{n}{2} r \geq (\frac{1}{p} - 1) r \implies n \geq 2(\frac{1}{p} - 1)$$

It is important to note that this mechanism will not catch all illegitimate results submitted by a cruncher; rather, it will financially disincentivize cheating in expectation, since crunchers might still be able to profit by cheating in some cases, but not on average. Increasing n beyond the lower bound found above can be used to make cheating an arbitrarily bad strategy.

5.2 Future

When only future WUs can be additionally replicated, the goal is still similar to the one in the past case: to have the expected rewards under an honest strategy be greater than (or equal to) the expected rewards under a cheating strategy.

Since a machine could get to the point where it is sent the fewest number of replicated instances, cheat for as long as it can, and then stop crunching the application once it is caught, it could profit from cheating as long as it is not possible to retroactively verify WUs. For the future setting, we introduce a deposit d, which can either be in the form of non-replicated WUs which are treated as replicated WUs (meaning that the cruncher receives a lower reward than they should have for crunching those WUs), or some amount of the cryptocurrency itself. If the deposit is withdrawn or withheld, it must be rebuilt if the machine/cruncher wants to continue running that application. The idea is that a machine can gain at least as much by taking its deposit and discontinuing crunching that application as it can by cheating.

Let n and r be as they were in the prior case, and as before, consider only the WUs for which the machine is cheating. Consider some starting WU w_1, during which m_i is already under the WU verification scheme with minimum replication, and assume that m_i is cheating starting when they received w_1. Let w_j be the WU for which m_i was caught returning bad results. Since by construction we are only considering WUs for which m_i is cheating, $w_1 \dots w_{j-1}$ have no other instances – that is, m_i is the only one crunching them.

Up until w_j, m_i has gained $(j-1)r$ from cheating, so the loss for m_i must be greater than (or equal to) $(j-1)r$; setting $d \geq (j-1)r$ addresses this problem. Since we do not know j ahead of time, we can set $d \geq E[(j-1)r] = (\frac{1}{p} - 1)r$. Once m_i is caught, the deposit is lost, and it must be rebuilt in the same manner it was constructed.

5.3 Cherry-picking Attacks

Under the most recent iteration of the BOINC credit system [1], there is an attack vector by which a cruncher only crunches shorter WUs and abandons longer WUs, called a *cherry-picking* attack. This would still be an attack vector in the reward mechanism described in this paper. There are two main sources of information that a potential attacker might have: 1) the estimated WU runtime as given by projects, and 2) empirical estimates of the mean, percentiles, variance, and potentially other statistics of the runtimes on the attacker's machine.

The problem caused by the first source of information can be solved by simply not providing crunchers with such detailed information; perhaps an average runtime and variance would suffice. The second problem is trickier. For a given machine, job runtimes from a particular application will come from some probability distribution X. In order to eliminate the cherry-picking attack, X would have to come from some family of distributions where, given the elapsed time t, X is such that $E[X - t \mid t] \leq E[X]$ for all t.

For example, if the runtime distribution is bimodal, then the time difference between the arguments of the modes would need to be larger than the time difference between zero and the minimum of the arguments of the modes. If the latter condition does not hold, then several WUs could be wrapped together into a single WU such that the distribution of runtimes becomes unimodal, although this would still not necessarily satisfy $E[X - t \mid t] \leq E[X]$ for all t. For example, a unimodal distribution skewed strongly to the left might still not meet the necessary criterion.

Alternatively, some form of punishment could be used to dissuade crunchers from using this attack. For example, the number of WUs sent to a machine could be reduced if too many WUs are returned as abandoned. Another form of punishment could be higher levels of replication for hosts that have returned many abandoned tasks. Additionally, it is worth noting that use of the cherry-picking attack could actually damage the cruncher because of its effect on the ER (in the linear regression approach), but it is likely not to matter on the scale of a single machine.

5.4 Project Collusion

While the crunchers can be incentivized to not cheat, the projects must also be disincentivized from cheating. Consider a project that wants to confirm that some particular machine(s) completed more WUs than they actually did. Split the machines on the network into two groups: machines knowingly colluding with the project, and all other machines. If the WU distribution is randomized in a trustworthy manner, then it would be impossible to favor any machines in the second group. For example, the WUs could be randomly distributed by relying on randomized data from the blockchain to determine which WUs go to which machines, with that process carried out and verified on the blockchain. To know whether or not a job should be replicated, projects could be told to which machine the WU was sent after it was already sent, after determining whether the result should be accepted at all, or at some other point. The randomization process just described might interfere with the methods that BOINC projects use to schedule jobs (in particular, projects need to know which platform a machine uses, since there are different application versions for different platforms); see [3]. However, the fact that machines are incentivized to crunch the applications on which they are most efficient somewhat counterbalances this interference. It may also be possible to reconcile these problems with further research – the purpose here is merely to demonstrate feasibility.

However, it would still be possible to favor colluding machines by simply having them return a result from a set of results that the project automatically recognizes as legitimate, but knows is actually being returned from a colluding machine. It may be the case that the only way to deal with this problem is by having some sort of trustworthy oversight over which types of WUs are considered legitimate results – however, this problem is beyond the scope of this paper. (Note that there is nothing wrong with a (non-profit) project using its own machines to crunch WUs.)

5.5 Open Problems

There remain a number of open questions about this anti-cheating mechanism. Does the expected payoff from cheating increase with the number of machines under the control of the attacker? What fraction of the machines on the network (or crunching a particular application) would an attacker have to control in order to have a higher expected payoff from cheating than from being honest? What would happen if some fraction of WUs were sent on to triple verification – could this mitigate large scale attacks?

In another vein, the anti-cheating mechanism as described only works properly for WUs that have deterministic results. It is an open problem on how to disincentivize cheating for WUs with non-deterministic results.

Note another interesting open problem: the project collusion issue is one of the main obstacles in constructing a true Proof-of-Research coin, which is why both Curecoin and Gridcoin have their reward mechanisms built on top of PoS protocols. One possible solution, theoretically but not practically, is to have every project for which a cruncher can receive rewards publish data on the blockchain – in particular, how jobs are created, to which machines they are distributed, the results, and by what rules the results are accepted or rejected. Beyond obvious privacy concerns, this would also be an enormous amount of data to be stored, in addition to the fact that it could potentially make exploits much easier.

6 Top Trading Cycles

Despite financial incentives, crunchers might still choose to crunch less profitable applications. In order to balance the efficiency of a machine and the desires of its owner, we offer two restricted formulations of this problem and describe how they can be solved using the Top Trading Cycles algorithm and its generalizations, and conclude with a general formulation describing a more realistic version of the resource allocation problem.

6.1 Intuition

Suppose that two crunchers each have a machine of their own, with the following ERs:

	Application 1	Application 2
Machine 1	$\rho_1^1 = 100$	$\rho_2^1 = 50$
Machine 2	$\rho_1^2 = 50$	$\rho_2^2 = 100$

A preference profile is a set of binary relations $\succ$ between each application where $a \succ_i b$ means that i prefers a to b. We will assume that preference profiles are not *complete* – that is, it is not necessary that all applications are in the relation. However, they are *transitive*: if $a \succ b$ and $b \succ c$, then $a \succ c$. $\succeq$ denotes a weak preference, while $\succ$ denotes a strong preference.

Now suppose that the two preference profiles for the two crunchers are $\succ_1 = a_2 \succ_1 a_1$ and $\succ_2 = a_1 \succ_2 a_2$. Cruncher 1 can offer to crunch their less preferred application in exchange for cruncher 2 crunching cruncher 1's more preferred application, while keeping the rewards for crunching the less preferred application, and vice versa. This way, both crunchers can have their more preferred applications crunched while still utilizing the full potential of their machines, and realizing an overall better outcome.

Consider another example with three crunchers, each with their own machine:

	Application 1	Application 2	Application 3
Machine 1	$\rho_1^1 = 100$	$\rho_2^1 = 50$	$\rho_3^1 = 25$
Machine 2	$\rho_1^2 = 50$	$\rho_2^2 = 25$	$\rho_3^2 = 100$
Machine 3	$\rho_1^2 = 25$	$\rho_2^2 = 100$	$\rho_3^3 = 50$

with preference profiles $\succ_1 = a_3 \succ_1 a_2 \succ_1 a_1$, $\succ_2 = a_2 \succ_2 a_1 \succ_2 a_3$, and $\succ_3 = a_1 \succ_3 a_3 \succ_3 a_2$. It is easy to see that having each cruncher crunch their machine's most profitable application results in amounts of WUs of each application being crunched that is more preferred by each cruncher than if they each crunch their favorite application.

What follows is an informal explanation of desired properties of mechanisms in the field of resource allocation problems.

In both examples, it is clear that every cruncher, in terms of their application preferences, prefers the outcome where they each crunch their most profitable applications than if they each crunched their favorite applications. We can say that the former assignment *Pareto dominates* the latter assignment, meaning that every cruncher at least weakly prefers the former assignment to the latter, and at least one cruncher strictly prefers it. An assignment is *Pareto efficient* if it is not Pareto dominated by any other assignment.

We also introduce the notion of the *core*. The core is the set of assignments in which no subset of the agents can deviate and arrive at a better assignment for that subset. As explained in [7], an assignment in the core also implies *individual rationality* – that is, no cruncher can possibly do worse by participating in the mechanism than they could on their own – and Pareto efficiency, since if an assignment is in the core, and it is Pareto dominated by some other assignment, then all the agents would have an incentive to deviate to the other assignment, meaning that the original assignment was not in the core.

The final concept is *strategyproofness*, which is a property that means no cruncher can benefit by misreporting their preferences.

We can now define the first part of the problem: each machine m_i has an *endowment*, which in this case is ρ^i. The second part of the problem is traditionally the set of preferences profiles $\succ = (\succ_1, \succ_2, \ldots, \succ_{|M|})$. However, the preference profiles over applications need to be modified to be preference profiles over the machines.

Note that the mechanics of the exchange in the context of BOINC or blockchains/cryptocurrencies are not discussed here; the purpose of this section is to explore the feasibility of the underlying mechanism.

6.2 Reduction to Top Trading Cycles

We begin with the simplest possible reduction: suppose that each cruncher only owns one machine, and each machine can only crunch one application at a time.

In order to transform preferences over applications to preferences over machines, the application preferences just introduced need to be refined. Let the new preference profile over applications $\succ_i$ for machine m_i be an ordering over amounts of WUs from applications in decreasing order of preference. For example, consider the second machine from the second example above. One possible new preference profile for m_2 is $\succ_2 = a_2^{25} \succ a_1^{50} \succ a_3^{100}$. This new preference profile over applications indicates that m_2 would prefer to trade for more than 25 WUs of a_2, which is logical, since it is m_2's most preferred application, and they can only achieve 25 WUs alone. If there are no machines available for trade that can achieve that many WUs, either because no other cruncher is offering that, or because all the machines that could achieve that number of WUs were already traded for, then m_2's subsequent preference is more than 50 WUs of a_1 – which again, m_2 cannot achieve on its own. Finally, m_2 would prefer more than 100 WUs of a_3 – which again, m_2 cannot achieve on its own.

Another possible sequence is $\succ_2 = a_1^{75} \succ a_2^{25} \succ a_1^{50} \succ a_3^{100}$. This is the same as the previous sequence, except with an additional preference at the beginning, indicating that m_2 prefers more than 75 WUs of a_1 to more than 25 WUs of a_2. Note that the assumption so far has been that these minimums are exclusive. If the minimums are not exclusive, then a machine might end up preferring itself to any other machine – this is easily taken care of by TCC and its generalizations.

Now these new preferences over amounts of WUs need be transformed into preferences over machines. Consider the following transformation: for the first element in a preference profile, the subset of machines satisfying the corresponding minimum can be sorted in decreasing order of their performances on that application, appended to the final list of preferences over machines, and removed from the pool of machines; this is the first subset of preferences for the machine. This process would be repeated for every element in the preference profile. Note that a cruncher can have two or more separate values for a single application, as long as those ranges are separated by at least one other application (otherwise they could be reduced to the minimum among them), as just illustrated with the second possible sequence. This would create an ordering over all machines with no repeats, and is formalized in Algorithm 2.

This transformation makes the problem setting the same as the traditional setting for TTC (a housing market), and so the original TTC algorithm can be invoked. TTC is known to be core-selecting and strategyproof.

6.3 Multiple Endowments

A natural extension of this formulation is to the case where crunchers can own more than one machine, which of course is the case in reality. This is a well-researched area with a plethora of useful results, some of which we describe below.

First, Sönmez showed in [11] (Corollary 2) that in indivisible goods economies, if there is at least one agent that owns more than one good, then there is no allocation rule that is both core-selecting and strategyproof (the author originally used Pareto-efficient and individually rational in place of core-selecting, but the statement follows simply when taking into account that the core is both Pareto-efficient and individually rational, as already described).

In [7], Fujita et al. present a generalized version of the housing market problem with conditionally lexicographic preferences over bundles of objects, and introduce an algorithm called *Augmented*

Algorithm 2 Preference Assignment Algorithm for m_i

```
Input:
    $M$, $\rho^j \; \forall j \in [|M|]$, and preference profile over amounts of WUs $\succ_i$
Output:
    Preference profile over machines $\succ_i^{'}$

$\succ_i^{'} = [\,]$
for $k = 1 \ldots \text{len}(\succ_i)$ do
    amount, application = $\succ_i [k]$
    satisfying = $\{i \in [|M|] \mid \rho^i_{\text{application}} > \text{amount}\}$
    $\succ_i^{'}$.append(satisfying machines sorted in decreasing order of performance on application)
    remove satisfying machines from $M$
end for
```

Top Trading Cycles (ATTC), which they showed is also core-selecting, and prove that finding manipulations is NP-complete. These manipulations include lying about preferences, splitting endowments, and hiding endowments.

In [5], Aziz presents the *Fractional Top Trading Cycle* (FTTC) algorithm. The setting is one in which agents can own more than one house or fractions of houses, and the author proceeds by breaking each agent into subagents for each of the houses, where each subagent owns what its superagent owned. We reproduce a theorem from that paper below:

Theorem 6 [Aziz]. For the housing markets with strict preferences, discrete but multi-unit endowments, FTTC is equivalent to the ATTC mechanism.

A corollary is that FTTC is also as hard to manipulate as ATTC.

6.4 General Setting/Open Problem

A further extension of the problem is to the case where crunchers can offer to crunch more than one application. Beyond this, there are also generalizations where crunchers can have non-strict preferences, or even preference classes, over some ranges of WUs of different applications; however, the latter generalizations are beyond the scope of this paper.

We will re-use the assumption from Section 3 that given a constant level of use for a given machine, a linear decrease in the output of one application implies a linear increase in the output of another application (keeping in mind the separation between CPU and GPU applications). Formalizing this, let the non-negative weights for each application be $g = (g_1, g_2, \ldots g_{|A|})$, and $\sum_{i\in[|A|]} g_i = 1$. Then the endowment of a machine m_i is $g \otimes \rho^i$, where $\otimes$ denotes element-wise multiplication. The preference profiles remain as they were previously. This setting differs from the aforementioned multi-endowment and fractional settings in that the endowment here is not only multi-dimensional, but the endowments and the allocations can be any arbitrary fraction of every application, as long as the fractions sum to 1. Note the critical fact that the linear increase/decrease property does not hold in general, and especially does not hold when taking into account CPU and GPU applications. The purpose here is to present the most basic extension of this problem.

In [12], Yu and Zhang drop entirely the idea of trading cycles, and instead propose describing the trades in terms of parameterized linear equations. This approach may be much better suited for the general setting we have here, since it simplifies the complex problem of deciding who trades what with whom, and the linearity of the equations might make it much easier to handle the linearly balanced endowments.

There may also be entirely different approaches that are better suited to solving this problem. For example, genetic algorithms and other techniques intended to optimize use of distributed computing resources could find allocations that satisfy certain social welfare functions better than generalizations of TTC or other market mechanisms. However, the other approaches may not have properties such as core-selection.

7 Carbon Neutrality

There remains a fundamental issue: what about the carbon emissions caused by running the hardware? Also, who would buy this cryptocurrency, and thus pay for these computations?

There are a number of ways that this cryptocurrency could be made carbon-neutral. First, renewable energy sources create excess energy on the grid during peak sunlight and wind hours, which do not coincide with peak energy usage hours, meaning that the excess energy must be stored for later use – this is the most critical issue facing renewable energy integration today. Distributed computing can convert excess electricity into heat, which can be stored in water heaters or household thermal energy storage systems for later use.

Second, for-profit companies which need distributed computing can buy this cryptocurrency in order to mediate transactions by paying crunchers for use of their hardware, which would boost the price and liquidity of the currency (price is affected by mediation [8]). Since some machines may be used to heat air and/or water for immediate use (not energy storage), if there are no tasks from companies that are willing to pay, but the heat is still needed, then the machines can fall back on the volunteer projects.

Relatedly, vast.ai is a platform connecting hosts owning hardware suitable for deep learning with willing buyers. Until recently, many vast.ai hosts had their machines defaulting to Ethereum mining when their hardware was not in use for deep learning – this mining can easily be replaced with BOINC tasks. There are also companies which construct machines specifically for the recovery of waste heat from computing, like Qarnot, which offers opportunities for future cooperation.

Additionally, there are a number of BOINC projects dedicated to matters of environmental concern – the most notable is climateprediction.net, but there have also been projects in the past that attempted to improve solar panels, and there are many more opportunities in this area.

Finally, crunchers could donate some of their currency – potentially profits, after paying for electricity – to foundations that protect land from deforestation, or responsible reforestation projects. Then, those who are looking to offset their carbon emissions can buy the currency from the foundations/projects, granting the latter the fiat currency necessary to buy and protect the land. Alternatively, crunchers can invest their profits in other forms of carbon reduction, green energy, or sustainable and recyclable hardware, which is a relatively new but growing field.

8 Current State of Distributed Computing Cryptocurrencies

There are a number of cryptocurrencies that reward distributed computing. Curecoin rewards computations for Folding@home, a distributed computing platform for protein folding. Primecoin is a PoW coin where the PoW computations are searches for prime numbers. Obyte rewards computations for World Community Grid, a BOINC project, but does not reward other projects. Gridcoin rewards a subset of BOINC projects suited to its distribution mechanism. Since Gridcoin is the only cryptocurrency currently rewarding more than one BOINC project, we will use it as an example of how blockchains and their economics develop.

Gridcoin was started in 2013 as a fork of Blackcoin, which itself was a fork of Peercoin, which itself was a fork of Bitcoin. It was started as a PoW coin using the scrypt hashing algorithm, with a layer on top of the PoW protocol that rewarded BOINC contributions. Beginning in 2014-2015, the coin was forked, the protocol was eventually shifted to PoS, and the GRC in existence at the time was inflated to $\sim$340 million GRC. The PoS protocol awarded every UTXO that staked a block with a percent yield based on the age of the UTXO, as well as rewards for that person's BOINC contributions. Another fork in 2018 resulted in constant block rewards, rather than a percent return based on age, with $\sim$14 million GRC being minted per year – 25% going to the PoS block rewards, and the remaining 75% going to rewards for BOINC computations. There is strong evidence to suggest that most of the GRC at the time of the 2014-2015 fork remains in the hands of those who were present before the fork.

As of this writing, there is a total of $\sim$455 million GRC in existence; of that, there is $\sim$30 million GRC in the Foundation wallet. The Foundation wallet is a multi-signature wallet entrusted to long-standing members of the community and is intended to be used for enhancement or promotion of Gridcoin. It is still almost entirely derived from the initial fork, since developers have been reluctant to accept payment for their contributions. Many Gridcoin crunchers still send donations to the Foundation, despite the current rate of minting being small relative to the initial $\sim$340 million GRC after the initial fork.

While Gridcoin experienced a rapid rate of growth during the 2017 cryptocurrency boom, it plummeted in activity and price during the subsequent decline. It has experienced a not-nearly-as-large amount of growth in price and users since the 2020 cryptocurrency boom, despite the absolute size of the 2020 boom far eclipsing the size of the 2017 boom. Correspondingly, Gridcoin also has mostly empty blocks and a very low velocity of money. However, it has one of the highest rates of development in the entire cryptocurrency industry, and many non-core developers regularly add new uses or external features to the blockchain, indicating the viability of such cryptocurrency communities.

9 Conclusion

Unfortunately, BOINC has stalled in growth. As Anderson noted in [3],

> The original BOINC participation model was intended to produce a dynamic and growing ecosystem of projects and volunteers. This has not happened: the set of projects has been mostly static, and the volunteer population has gradually declined. The reasons are likely inherent in the model. Creating a BOINC project is risky: it's a significant investment, with no guarantee of any volunteers, and hence of any computing power. Publicizing VC [Volunteer Computing] in general is difficult because each project is a separate brand, presenting a diluted and confusing image to the public. Volunteers tend to stick with the same projects, so it's difficult for new projects to get volunteers.

At the same time that BOINC has stalled, cryptocurrencies have exploded in popularity, with the computational power and resources devoted to the latter far outstripping the former. The stall in growth has come despite a massive increase in the amount and availability of computing power among consumers. There is a good opportunity to direct this ever-increasing amount of computational power towards more useful work; cryptocurrencies and BOINC can play a big role in this. As for the problems just quoted, a minor modification to the reward mechanism could create a floor of computational power for each application/project, guaranteeing that if it receives below a certain amount of normalized WU contributions, then crunchers of that application/project are rewarded more, thus incentivizing crunchers to crunch them. Ideally, this would create a situation where no application/project falls below that minimum. Likewise, it is possible to create a cap on the amount of total rewards for all crunchers that can be awarded for crunching a particular application/project; this upper bound would incentivize crunchers to crunch less popular applications/projects in case a particular application/project became extremely popular.

10 Acknoledgements

The author would like to thank James C. Owens, Cy Rossignol, Itai Feigenbaum, and Vitalii Koshura for useful comments and discussions.

References

[1] *A new system for runtime estimation and credit.* URL: `https://boinc.berkeley.edu/trac/wiki/CreditNew`.

[2] *Adaptive replication.* URL: `https://boinc.berkeley.edu/trac/wiki/AdaptiveReplication`.

[3] D. P. Anderson. "BOINC: A Platform for Volunteer Computing". In: *J Grid Computing* (2020), pp. 99–122.

[4] M. S. K. Awan and S. A. Jarvis. "MalikCredit - A New Credit Unit for P2P Computing". In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems* (2012), pp. 1060–1065.

[5] H. Aziz. "Generalizing Top Trading Cycles for Housing Markets with Fractional Endowments". In: *arXiv preprint arXiv:1509.03915* (2015).

[6] X. Chen, C. Papadimitriou, and T. Roughgarden. "An Axiomatic Approach to Block Rewards". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies.* 2019, pp. 124–131.

[7] E. Fujita et al. "A Complexity Approach for Core-Selecting Exchange under Conditionally Lexicographic Preferences". In: *Journal of Artificial Intelligence Research* (2018).

[8] A. Narayanan et al. *Bitcoin and Cryptocurrency Technologies.* 2016.

[9] A. M. Al-Qawasmeh, A. A. Maciejewski, and H. J. Siegel. "Characterizing Heterogeneous Computing Environments using Singular Value Decomposition". In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW).* IEEE. 2010, pp. 1–9.

[10] A. M. Al-Qawasmeh et al. "Characterizing Task-Machine Affinity in Heterogeneous Computing Environments". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum.* IEEE. 2011, pp. 34–44.

[11] T. Sönmez. “Strategy-Proofness and Essentially Single-Valued Cores”. In: *Econometrica* 67.3 (1999), pp. 677–689.

[12] J. Yu and J. Zhang. “Efficient and fair trading algorithms in market design environments”. In: *arXiv preprint arXiv:2005.06878v3* (2021).

A RAC Code

This code, which can also be found here, has been slightly modified from the original code, which can be found here.

```
#include <iostream>
#include <math.h>
#define SECONDS_PER_DAY 86400

using namespace std;

void update_average (
    double work_start_time,         // when new work was started // (or zero if no new work)
    double work,                    // amount of new work
    double half_life,
    double& avg,                    // average work per day (in and out)
    double& avg_time,               // when average was last computed
    double& fakeTime                // new, for simulations
) {
    //double now = dtime();
    double now = fakeTime;

    if (avg_time) {
    double diff, diff_days, weight;

    diff = now - avg_time;
    if (diff<0) diff=0;
    diff_days = diff/SECONDS_PER_DAY;

    weight = exp(-diff*M_LN2/half_life);

    avg *= weight;

    if ((1.0-weight) > 1.e-6) {
        avg += (1-weight)*(work/diff_days);
    }
    else {
        avg += M_LN2*work*SECONDS_PER_DAY/half_life;
    }
}
else if (work) {
    // If first time, average is just work/duration
    //
    cout << "avg_time = " << avg_time << "\n";
    cout << "now = " << now << "\n";
    double dd = (now - work_start_time)/SECONDS_PER_DAY;
    avg = work/dd;
    }
    avg_time = now;
}

int main() {
    double RAC = 0;
    double timeOne = 1;
    double timeTwo = 1;
    double totalCredit = 0;
    double timeInterval = 3600; // new; time in seconds between each RAC update

    double work_start_time = 0;         // when new work was started // (or zero if no new work)
    double work = 200;                  // amount of new work
    double half_life = 604800;
    double& avg = RAC;                  // average work per day (in and out)
    double& avg_time = timeOne;         // when average was last computed
    double& fakeTime = timeTwo;         // new; for simulation

    for (int i=0; i<1500; i++) {
        if (1) {
            if (i % 24 == 0) {
                cout<<"week " << i/168 + 1 << ", day " << (i/24)%7 + 1 << ";
                current hour = " << i << "; ";
                cout<<"totalCredit = " << totalCredit << "; ";
                cout<<"RAC = "<< RAC << "\n";
            }
        }
        fakeTime += timeInterval;
        update_average(work_start_time, work, half_life, avg, avg_time, fakeTime);
        totalCredit += work;
    }

    cout<<"Final totalCredit = " << totalCredit << "\n";
    cout<<"Final fakeTime = " << fakeTime << "\n";
    cout<<"Final RAC = "<< RAC;

    return 0;
}
```

B ER Approximation

The code for the ER approximation can be found here.

As described in Section 1.1, the computational contribution of all of the machines under a cruncher's CPID to a project is grouped into a single measurement called RAC, which is the only statistic currently provided to the public by projects. Increasingly granular disaggregation of the RAC yields an increasingly accurate estimate of the ER. There are three main levels of disaggregation:

1) *Machine-level disaggregation.* A single CPID can contain many machines, and those that do can be massive outliers in the data and completely distort the regression, to the extent that $\widehat{\rho}$ might not even intersect the axes on their positive rays.

2) *Application-level disaggregation.* Most projects have multiple applications, each of which has different computational requirements. Projects can normalize their credits across their respective applications – however, even a single project not normalizing their credits across their applications would prevent an accurate estimate of ρ. Furthermore, even if every single project normalized their credits across their applications, the current credit computations are still flawed, which is one of the motivations for basing the ER on WUs completed, rather than credits. Additionally, normalizing based on all of the machines on the network provides a much larger sample size than normalization based on a small subset of hardware to which projects have access.

3) *WU-level disaggregation.* A number of revisions to BOINC's credit-granting mechanism have been explored, see [1] for an example. However, the difficulty involved in taking into account all the factors about the machines means that there have been no comprehensive solutions for this problem yet. As explained in Section 1.1 and the preceding paragraph, this is why WU-level data is preferred.

C Game-Theoretic Simulations

The code for the game simulations can be found here.

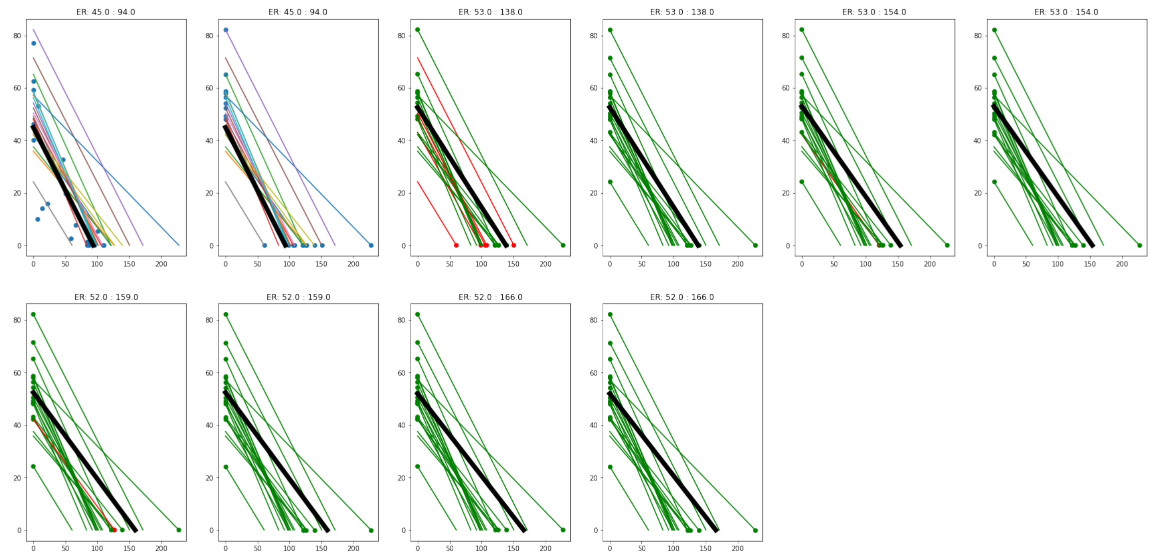

Figure 5: Game that converges

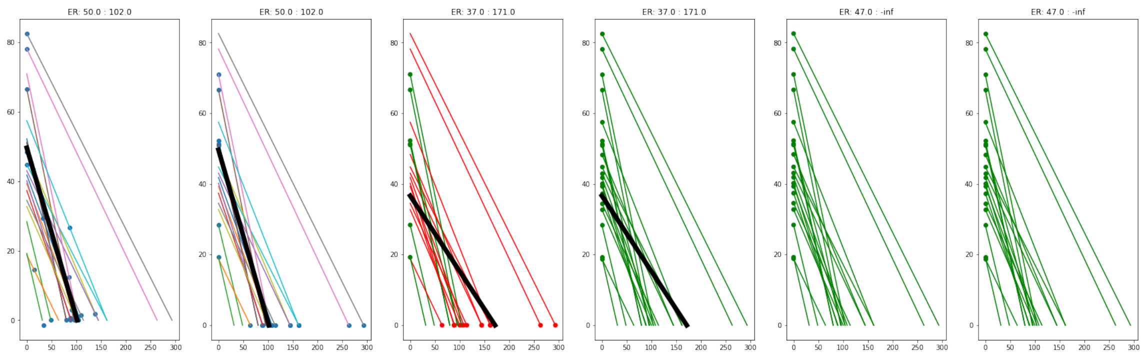

Figure 6: Game that diverges

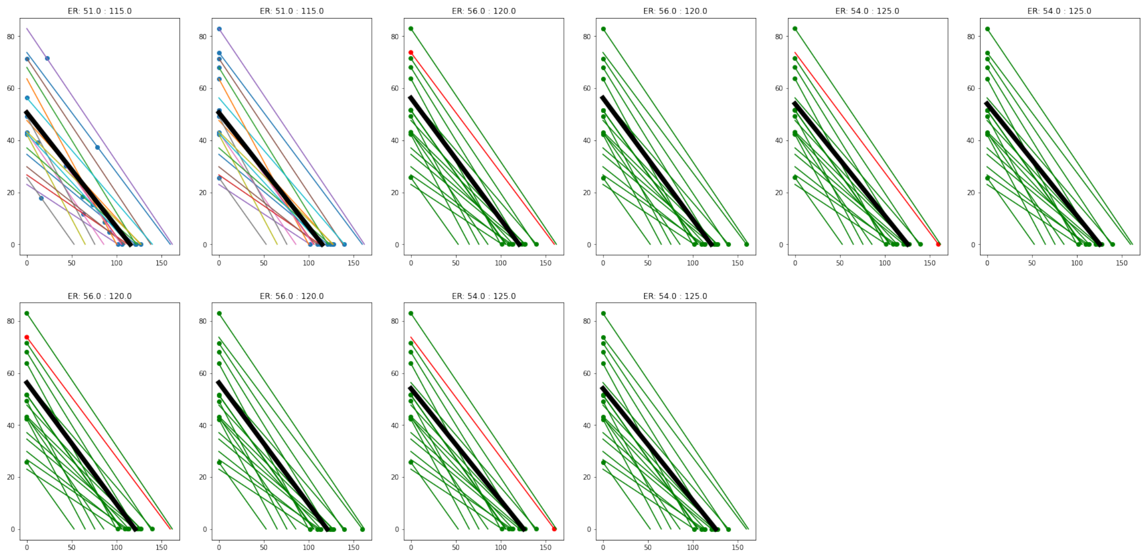

Figure 7: Game that cycles