Task Exchange

Paid Volunteer Computing in a Market for Computing Resources

Peter Xu

Abstract

Volunteer computing, or alternatively grid computing, is often used by scientists to run computing-intensive tasks cheaply by using the spare computing resources of Internet-connected, volunteered devices. More recently, cloud computing has become popular due to its ease-of-use and quick setup and spin-down: servers are rented by the hour. We introduce Task Exchange, which combines the advantages of both cloud computing and volunteer computing. With Task Exchange, one does not have to recruit users and run one’s own servers as with volunteer computing, while still getting much cheaper resources than from cloud computing providers such as Amazon Web Services. Task Exchange uses data escrow, sandboxes, and market matching (constrained optimization) to create a market where task creators can easily write computing-intensive tasks and pay volunteers to run them. We discuss the complications needed for sabotage-tolerance and to prevent cheating, before showing in benchmarks the feasibility of Task Exchange to efficiently run large-scale computing tasks.

Long Abstract

Volunteer computing, or alternatively grid computing, is often used by scientists to run computing-intensive tasks cheaply by using the spare computing resources of Internet-connected, volunteered devices. More recently, cloud computing has become popular due to its ease-of-use and quick setup and spin-down: servers are rented by the hour.

We introduce Task Exchange, which combines the advantages of cloud computing and volunteer computing. Task Exchange represents a refinement of volunteer computing libraries such as BOINC, and is developed with ease-of-use and security in mind. With Task Exchange, one does not have to recruit users and run one’s own servers as with volunteer computing and BOINC. Meanwhile, task creators still get much cheaper resources than from cloud computing providers such as Amazon Web Services. Likewise, volunteers are paid, encouraging more of the world to connect idle computers to Task Exchange so that the world’s computing power is more fully tapped.

Task Exchange uses data escrow, sandboxes, and market matching (a form of constrained optimization) to create a market where task creators can easily write computing-intensive tasks and pay volunteers to run them. It takes into account client capabilities, task resource requirements, deadlines, and user compensation in scheduling tasks. We also provide several sample applications, including a web crawler that circumvents IP blocks, and a Bitcoin miner-like hash generator, and also propose other new uses such as censorship and uptime-checking or speed testing. We then discuss the complications needed for sabotage-tolerance and to prevent cheating. Finally, using a
prototype programmed in Python that uses the PyPy sandbox, we show with benchmarks the feasibility of Task Exchange to efficiently run large-scale computing tasks.

Introduction

Over the past 15 years, distributed computing has become a large part of scientific projects. Using the paradigm of volunteer computing, these projects solicit volunteers to donate their computers’ spare processing capability, saving researchers the costs of supercomputers and high-performance computing clusters (Larson, et al. 2002). However, only a small proportion of the world’s computers participate. Each project also must independently recruit users before tasks can be run, which is a long and labor-intensive process.

At the same time, Amazon Web Services, Google App Engine, Microsoft Azure, and other similar cloud computing services have made computer storage, processing power, memory, and network resources into commodities that can be purchased separately. These services all rely on machines in data centers, and provide a high level of isolation by running virtual machines. They also can be ramped up and down extremely quickly. Compared to volunteer computing, though, they are expensive, as they rely on dedicated hardware.

Many tasks either do not require this level of isolation, or must be run on end-user machines. For example:

- Tasks that can be broken up into discrete work units often do not require full isolation, unlike servers (the most common use case for cloud computing). This is often-used in the volunteer computing model. Potential tasks that could be run include:
  - Video and audio processing and transcoding.
  - Scientific computing.
- Tasks whose result depends on who’s running them. For example:
  - A task from Netflix to see how fast the transfer speeds are from different ISPs.
  - Tasks from large websites (e.g. Google, Twitter) to see if the sites are accessible and to quickly report downtime or censorship.
  - Tasks that crawl sites that limit requests by IP address or other legal- or network-based means.

Task Exchange is a proposed system for running these tasks by taking advantage of the idle resources of the computers of the world. In short, task creators upload Task Exchange tasks to a centralized server, and specify deadlines, verification methods, and most importantly, the monetary reward for finishing the task. Task runners install the Task Exchange client software, which then runs these tasks. The client schedules tasks to maximize the earnings given the computer’s capabilities, while securely containing them so that tasks cannot access user data irrelevant to the task. In other words, we provide a market for computing power, using the idle CPU and GPU cycles, memory, hard drives, and network connections of the world.
Related Work

The most closely-related body of work is in volunteer computing, which has a similar model of recruiting non-dedicated machines for particular tasks. In particular, BOINC (Berkeley Open Infrastructure for Network Computing) is another middleware product that lets different scientific teams create tasks for volunteer computing (Anderson 2004).

BOINC

BOINC uses a similar model of servers (scheduling servers and data servers) distributing workunits to computers running the BOINC client. Each BOINC client signs up for one or more projects. The project’s scheduling servers schedule workunits with attached resource usage parameters and time limits, and then the client downloads the workunits from the project’s data servers. After processing, the client sends results back to the scheduling servers, which verifies them by comparing the results of the same workunit from multiple, different clients to ensure that a majority (quorum) of them match. If there is a quorum, the scheduling server saves the canonical result.

On the client side, participants can limit how the BOINC client uses resources such as bandwidth, CPU, GPU, and memory (Anderson, Christensen and Allen, Designing a Runtime System for Volunteer Computing 2006). The client also schedules tasks from the multiple projects to maximize resource usage, meet all deadlines, and in case a client signed up for multiple projects, allocate resources to these projects based on the participant’s preferences. Scheduling servers also give credit to clients for workunits they finish, but these credits are only displayed on leaderboards, and are not convertible to money.

Folding@Home

Folding@Home uses a similar design, with small architectural differences. The roles of the scheduling and data servers are combined into a work server, with optional redundant collection servers for collecting results when the work server is down (Beberg, et al. 2009). Clients, when
first joining the project, first contact the assignment server, which assigns them a work server; the assignment server is a load balancer. Clients only run one project (Folding@Home), avoiding the scheduling challenges.

Perhaps more insightful for us is the way credits, or points are allocated to clients. Clients receive both base points and bonus points (Pande Lab, Stanford 2013). The server benchmarks a test workunit in a group of similar workunits, and then assigns a base point value for the entire group of workunits. The client then receives a bonus based on the square root of early finishing; if the client took 1/5 of the time to the deadline calculating, its score is multiplied by \( \sqrt{5} \).

**Slicify**

Slicify is a commercial service that offers a “cloud computing platform powered by a global network of home computers” (Slicify 2014). At first glance, this is very similar to the system envisioned by Task Exchange. However, the computing model is very different. Rather than using a model of tasks and workunits, Slicify installs a VirtualBox Linux virtual machine, and then allows other people to rent it by the hour (Slicify 2014). Those who rent the machines (our task runners) are called compute sellers, and the lessors (our task creators) the compute buyers.

The virtual machine model is very flexible for power users, but it has its disadvantages. Compute sellers must download a large archive containing VirtualBox and a Linux image, and be willing to let go of a good portion of their computer’s computing resources for a fixed length of time. Turning off the service ends the “rental”, and as a result, one also needs to plan ahead. Buyers must also be willing to deal with VM instances that might shut down randomly when the seller decides that he needs his own computer back.

Tasks and workunits, run under a programming language-specific sandbox or a container, mitigates both of these problems, and is easier to write for compute sellers and easier to run, turn on, and turn off for compute buyers.

**Task Exchange Compared**

Task Exchange introduces a few novel features compared to previous volunteer computing implementations:

- **Data escrow**: As clients are now rewarded monetarily, we need an impartial and trusted third party (the Task Exchange administrator) to run servers, verify results, collect money from task creators, and award credits that are convertible to money to clients.
  - As an additional upside, task creators do not have to run their own servers, making it much easier than BOINC to write tasks.
  - We also introduce a new, generalizable way to verify the output of tasks.

- **Instant matching**: Clients do not have to join projects, and task creators do not have recruit users. Rather, clients can pick the generic types of tasks they are willing to run, and tasks automatically get assigned.
- **Profit-based scheduling**: Clients schedule tasks based on the new criteria of reward: they try to maximize the amount of credits they can get given their resource limits.
- **Client-dependent tasks**: Some tasks actually have results that depend on who is running the task. For example, Netflix may want to know the download speeds from different clients, or Twitter may want to check for censorship or downtime in particular countries. These tasks are runnable on Task Exchange, but not any of the other platforms.

**Design**

Task Exchange includes servers and clients.

The **servers** are run by a centralized, trusted **administrator**, and are responsible for distributing tasks to clients, collecting the results, and then forwarding the results to the task creators.

**Considerations**

Task Exchange is designed to be easy for developers. In particular, existing volunteer computing platforms are all designed with particular users in mind: BOINC assumes that task creators are large scientific organizations with significant technical know-how and hardware. Task Exchange hopes to makes it easier for task creators to write tasks. Tasks are simple scripts which take an input from standard input and output to standard output, and do not rely on a custom API. Similarly, the task creator needs no hardware: even the most basic users can upload tasks and download the results from a web interface.

Similarly, on the client side, we want to make installation and use painless. This involves allowing users to turn on or off the computation as they wish without hurting their earnings, and making the installation package small and widely compatible.

**Task Creator**

**Task creators** write **tasks**, which consist of a few parts that are uploaded to the administrator’s servers:

- **Client scripts**: scripts run by the clients that takes the input data (downloaded by the client from the servers) in standard input, processes them, and outputs the result.
- **Workunits**: data files to be stored on the server and used as the input of client scripts.
- **Verification scripts**: scripts run on the server that check that the responses returned by a client is valid, which marks the result as either correct or incorrect. This is more general than the BOINC method of just finding a quorum of identical responses, as certain tasks (e.g. checking website uptime) may have to be verified in other ways.
  o This flexibility is very helpful. For example, hard-to-solve but easy-to-verify tasks such as hash solving can be distributed to clients, and the servers can verify the result with just a single result. Alternatively, a web crawler may get different results
depending on when the page is visited; instead of checking the data for exact matches, we can simply check that certain parts of the page are correct.

Client

Meanwhile, users install the **client software**, and sets their preferences for:

- How much resources they are willing to dedicate (memory, CPU, GPU, bandwidth) to tasks, with additional fine-tuning for scheduling tasks at different times during the day.
- What kinds of tasks they are willing to run.

On startup, the client will:

- Benchmark the client’s host system, used for task allocation.
- Assign the client a unique ID and log in to an account to which we should credit its computations.

Complete Process

Putting the two together, we describe the complete process of running a task below:

![Diagram of the process](image)

*Figure 2: the full process of running a task in Task Exchange*

1. A task creator uploads a task’s scripts with its associated workunits, and specifies each task’s reward, as well as each workunit’s deadline, number of results needed for verification, and system permissions needed (e.g. network access, empty filesystem access, etc.)
2. The task runner’s client requests a task.
3. The server looks through all tasks, and assigns it tasks that maximize credits earned by considering:
   a. Each task’s reward in credits.
   b. The client’s computing resources and preferences.
   c. The resources needed by the task.
4. The client downloads the task’s workunit data and client scripts from the server.
5. The client runs the client script with a lower process priority than regular user processes.
   a. The client script must be sandboxed, possibly using a scripting sandbox (e.g. in PyPy), VM-like containers, or similar measures.
   b. Optimally, we would also be able to monitor resource usage (e.g. memory, CPU cycles, etc.) by intercepting system calls or checking the resource usage of the sandbox environment. Barring that, we can instead try running some sample workunits on the server to estimate their resource usage.
6. The client sends the result of the client script to the server, as well as other resource usage statistics.
7. The server stores the output in a temporary database that the task creator cannot access, and runs the verification script if there are enough completed results for the workunit.
   a. If correct, it awards the credit and saves it to a database where the task creator can view the result.
   b. If incorrect, it does not award a credit and has the workunit redone by other users.

Other Design Details

As a platform, there will be initially a dearth of tasks and clients. Fortunately, there are easy ways to seed the platform.

For tasks, the rise of cryptocurrencies such as Bitcoin allow for a range of tasks mining each cryptocurrency. These tasks can have their credit values adjusted to account for fluctuations in value, and would be similar to participating in a mining pool. Alternatively, we can sign up major launch partners.

On the client side, we can provision some VPSs or servers with the client software as a starting point. Alternatively, we can recruit large institutions with many idling computers such as libraries and schools as a starting point.

Implementation

We implemented Task Exchange in Python as a prototype and proof-of-concept. For code reusability and interaction purposes, all three components (the server, client, and tasks) are written in Python.
Server

The server is a HTTP server written using Flask-SQLAlchemy. HTTP was chosen, rather than a custom protocol, because most developers are fairly familiar such APIs provided by Google, Facebook, Github, etc. Likewise, clients would have the easiest time accessing such services, as vanilla HTTP is rarely blocked by firewalls.

All APIs return JSON and, unless otherwise stated, return the **standard return** struct/dictionary/hashtable of:

- **success**: (Boolean) whether the operation was successful.
- **messages**: (list of strings) messages on why the operation was successful or not.

The API includes the following operations.

**Task creator: AddTask (POST)**

Tasks creators use this to create a new task by uploading its scripts and specifying its payout. The server takes the task’s script files, does a basic sanity check to make sure the right files are there, and then packs them into a .tar.gz file for download by clients later. Takes:

- **creator**: the creator’s username. (In a later version, we’ll add authentication.)
- **name**: name of the task.
- **pay**: the number of credits to pay for each verified result.
- Either: (for more details, see the **client section**)
  - **archive** (file): a .tar or .zip file containing the code for the task, including at least `client.py` and `verify.py`, the client and verification scripts
  - **clientScript** (file) and **verifyScript** (file): separately uploaded files of `client.py` and `verify.py`

Returns the standard struct.

**Task creator: AddWorkunits/task_id (POST)**

Task creators use this to add one or more workunits to an existing task specified by *task_id*. Takes:

- **deadline**: when this workunit’s results are needed by.
- **results_needed**: the number of results needed per workunit, as determined by the verification script.
- Input files for the client script, in either:
  - **input**: text to put in the input file for a single new workunit
  - **file** (file): either:
    - One file containing the input for one workunit, or
    - A .zip or .tar archive in which each file is the input for a workunit.

Returns the standard struct.
Client: GetTask (POST)

Task runners’ clients use this to ask for a workunit to run, and register that the requester is now running the workunit so that the workunit does not also get assigned to someone else. Eventually, this should also set a deadline on the workunit based on how long the task usually takes to run so that if a client dies, we assign the workunit to someone else. Takes:

- client_id: the unique ID of the particular client.
- username: the user running the client, who will get paid.

Returns: a struct containing:

- success (Boolean): whether the server successfully found a task to run.
  - If success is true:
    - task_id (int): ID of the task.
    - workunit_id (int): ID of the workunit.
  - If success is false:
    - messages (list of strings): messages on why the operation was successful or not.

Client: GetWorkunit/workunit_id (GET)

Task runners’ clients use this to get the input of a workunit, used as the standard input for client.py. Returns the binary input.

Client: static/tasks/task_id.tar.gz (GET)

The .tar.gz archives with the code for individual tasks are stored here, and can be requested.

Client: AddResult (POST)

After a client finishes computing a workunit, it calls AddResult to submit the result, as well as relevant metadata.

Also, if the result means that we have enough results in this workunit, runs the verification script (verify.py) on existing results. If the results verify, credits are given to the task runners (clients) from the task creators. If not, the results are discarded. Takes:

- client_id: the unique ID of the particular client
- workunit_id: the ID of the workunit that just finished.
- result (file): a file containing the standard output of the task, from executing client.py.
- result_stderr (file): a file containing the standard error of the task, from executing client.py.

Returns the standard struct.

Task creator: Result/result_id (GET)

Task creators use this to get the completed, verified result of a workunit.
Miscellaneous

The web server provides the additional pages:

- **Register** (GET and POST): register new users.
- **AddTask** (GET): access the AddTask (POST) API call earlier
- **AddWorkunit/<task_id>** (GET): access the AddTask (POST) API call earlier
- **ViewTasks**: see existing tasks.
- **ViewWorkunits/<task_id>**: see existing workunits.

Unimplemented

Due to time constraints, some features were planned but not implemented.

- **AddCredits/WithdrawCredits**: a way to add credits to one’s account using a credit card, Paypal, and to withdraw credits to Paypal or a bank account.
- **RegisterClient (client_id, info)**: when a client comes online for the first time, it submits its system specifications and preferences to the server.
- Currently, there are no **user security checks**. In the future, we will ensure that:
  - Only clients assigned a workunit can request the task’s code (static/tasks) and the workunit data (GetWorkunit).
  - Only a task’s creator can view results through Result, and can only do so after the result has been verified.

Client

The client is relatively well-contained; in the main loop, the client:

1. Asks for a workunit by requesting **GetTask**.
2. Downloads the task’s scripts from **static/tasks/task_id.tar.gz**.
3. Downloads the workunit’s input from **GetWorkunit/workunit_id**.
4. Extracts the task’s scripts to **tasks/task_id** locally.
5. Runs **tasks/task_id/client.py** using a PyPy Python sandbox, which contains all of the code, with the downloaded workunit as the standard input.
6. Sends the result back to the server using **AddResult**

Before running the client, though, one must first initialize it with a unique ID and the username to whom credits should go by running **init.py username**.

Task writers and runners do not have to worry about the details of the client’s implementation. The only relevant detail is the specifications for tasks.

Tasks

Tasks must, as mentioned earlier, include a client script and verification script, both written in Python. Currently, due to sandbox limitations, the scripts cannot make any network requests.
When scripts are uploaded in a package, all the contents of the package will be placed in the sandbox. As a result, a task uploaded as an archive can contain `client.py`, which imports other Python files and packages in the directory tree for cleanliness of code and ease of use.

**Client Script**

If uploaded in a package, the client script must be named `client.py`. When run, the workunit’s input is supplied to the standard input (and can be read as any normal file), and the result should be output to standard output.

**Verification Script**

The verification script must contain at least a single function named `verify`, with the signature:

- **Parameters:** `def verify(input, results)`
  - `input`: a string containing the workunit’s input.
  - `results`: a list of strings, one for each result. There should be just as many results as specified in `results_needed` when the workunit was created.
- **Returns:** Boolean (True if successfully verified.)

It is also run in a sandbox, so it does not have access to the network. When it is uploaded as a part of a package, it will not be part of the .tar.gz file that clients can download.

**Sample Tasks**

**Hasher**

One working sample task is provided: hasher takes in a text string, and then tries to add strings to the end of it until we find a hash that ends with 20 zeros when represented in binary. This is analogous to the Bitcoin proof-of-work method of mining new blocks (Nakamoto 2008).

The verification script demonstrates how this hard-to-calculate but easy-to-verify problem’s results can be verified with just one script.

**Crawler**

This task takes in a list of URLs, and then systematically issues HTTP GET requests to each of them. It stores the responses and returns them as the result.

It also comes with a prepackaged, basic verification script which takes two inputs and checks if they are the exact same. This is the most common form of verification, similar to a quorum except with only two results.

However, the Crawler does not work right now because the PyPy sandbox does not have networking enabled. It exists as a proof-of-concept.
Discussion

Sandboxing

Task runners must know that their personal data is safe before they are willing to run arbitrary scripts on their computer. There are a wide variety of ways to implement sandboxing, including using jails, deploying virtual machines, and intercepting system calls. Slicify (Slicify 2014) uses virtual machines, and so does a majority of cloud computing services, as virtual machines offer good combinations of resource isolation and sandboxing functions (Armbrust, et al. 2010). Additionally, they impose a uniform base to build applications on, since we can deploy any operating system inside a virtual machine.

However, virtual machines are very bulky, with the operating system and the main multiplatform virtual machine host, VirtualBox, taking hundreds of megabytes to install and hundreds of megabytes of RAM. Recently, browser sandboxing and seccomp, a Linux module, has revived the popularity of system-call based sandboxing (Edge 2009). This concept dates involves in restricting system calls such as `read` and `write` so that only certain resources are available to the sandboxed process, with system calls funneled through another layer either in the kernel (as in seccomp) or another process (Goldberg, et al. 1996). These have the advantage of being much more lightweight than virtual machines. They are also more suitable to the tasks-and-workunits model that we use, as we do not need dedicated resources, and flexible resource usage is a plus.

Eventually, we settled on the PyPy sandbox, which is a Python runtime (PyPy Project 2014). The sandbox itself relies on funneling system calls through a helper process to isolate the program. Additionally, PyPy is highly performant, and Python is widely used in the academic and research community. Most importantly, PyPy is cross-platform. However, this is more limiting than the other models, as the sandbox has limited Python and external library support. Likewise, one cannot run an already-compiled application or something in a different language.

Scheduling

We have not fully implemented scheduling yet, but scheduling each workunit for Task Exchange is an optimization problem where we have to take into account:

- The workunit’s deadline as specified by the task creator.
- The task runner’s (client’s) computer’s capabilities.
- The task’s resource requirements.
- The amount of pay the task gives.

This specification by itself is a challenge, as while the workunit’s deadline and pay are well-known, it is up to us to find the client computer capabilities and the task’s resource requirements. In short, we use benchmarking for both of these.

After a user uploads a batch of workunits onto a machine, we serve as the task runner for a small, randomly-selected subset of these workunits first. Using Unix process monitoring (through
Python’s psutil library), we can monitor the CPU time and memory usage of particular tasks. The average and maximum run times are saved as the “computational cost” of the task, while the maximum memory usage is also saved to be used as a constraint.

Similarly, when a client comes online for the first time, we use the sysinfo libraries for each operating system to ask for CPU and memory characteristics. We then run a standard CPU benchmark to determine the computer’s capabilities. This information is then attached to the unique client_id assigned on client initialization.

To summarize, the additional information saved for scheduling are:

- Through benchmarking the task:
  - Maximum memory usage.
  - Longest CPU time used.
  - Mean CPU time used.

- Through the client:
  - Memory available.
  - CPU performance.

The actual scheduling is a constrained optimization problem. Whenever a client requests a task, we try to:

- Maximize \[ \frac{\text{Pay}}{\text{Mean CPU time used}} \]

- Constrained by
  - Maximum memory usage * 1.5 < Client’s memory available
  - Longest CPU time used * Client’s performance < Time until deadline

This best captures the mechanisms of a market. However, this also means that certain tasks that do not pay enough may not be assigned.

Preventing Cheating

A major challenge is that both the task runner and task creator are possible adversaries, especially since money is involved. The task runner may try to submit results and get paid without actually spending the compute time by using a modified client. The task creator may want to get results without having to pay.

By the Task Runner

The main problem with the task runner involves submitting incorrect results. In some cases, this is easily dealt with; for example, the hasher can be verified extremely quickly (similar to a majority of NP-hard problems, even though hashing itself is not one). However, many other problems must be solved to verify.

Fortunately, existing grid computing and volunteer computing projects have already solved this problem in a number of ways, including creating checkpoints and proof-of-work, sampling parts.
of the results, and simple replication and majority voting. (Domingues, Sousa and Silva 2007). As a result, we provide the flexibility for task runners to implement each of these in their custom verify.py script. Nevertheless, we will also provide default verification scripts for convenience.

By the Task Creator

Since the whole concept of a task creator has not been a part of volunteer computing and grid computing, there is no literature so far on how they might possibly cheat. The main methods of cheating involve avoiding the escrow by leaking information, and abusing the verification scripts. As task runners do not get credited until their results are verified, task creators can cheat by crafting a task that directly discloses results to a server that they own, and then failing all results with the verification script. This is prevented by disallowing networking in the sandbox. Optimally, we would allow networking, as there are tasks that legitimately require it. However, this is really difficult. We could further sandbox network operations so that we do not allow any program variables other than the input be used in crafting an external connection or request, but this may require a lot of code. Perhaps the best method involves human inspection, similar to an app-store system, of all socket calls.

Task creators can also cheat by placing all the expensive operations in the verification script. Fortunately, we can easily monitor the runtime of the verification script, and if it takes longer than a reasonable amount of time (e.g. 5% of the task execution time or 10 seconds, whichever is longer), we can discard the task while fining the task creator.

There is another whole class of problems that do not involve gaming, but rather just sabotaging the system. For example, task creators can make tasks that never return True in the verification script. This is addressed through server-side pre-testing of each task: we run a randomly selected workunit and ensure that it verifies in a trusted environment. Similarly, task runners can try to steal tasks and simply never run them. To mitigate this, Task Exchange automatically sets deadlines for individual work units of a multiple of the time it takes to run them (e.g. 4x), and if it does not receive a result by the deadline, assigns the workunit to another client. As long as most clients are honest, the system would function.
Benchmarks

![Benchmark results (hasher with "abcddd")](image)

**Figure 3:** benchmark results for running the hasher task. The hasher was run using "pypy -m taskexchange.client.run_task...", while the vanilla PyPy was run using "pypy taskexchange/sampletasks/hasher/client.py < input" on an Intel(R) Xeon(R) CPU E5-1620.

Using the sandbox on the standard hasher task, we still are able to obtain a competitive result that runs 3.5 times as long as the original, partly due to system call filtering, but also partly due to the fact that PyPy’s sandbox reimplements certain Python functionality in slower pure Python rather than C.

While this is slower than running the task outside of a sandbox, it is not intolerably slow. Parallelizing multiple workunits on multiple computers, which is the whole point of Task Exchange, also negates the speed disadvantages that the sandbox, as long as the actual monetary cost of running it is lower. All signs indicate that it will be, as even the most powerful Slicify instances sell for no more than a few cents per hour of computation.

**Conclusion**

Task Exchange builds on past work in grid computing and volunteer computing to create a new platform for buying and selling compute time. It combines the flexibility, ease-of-use, and quick setup and dismantling times of cloud computing instances with the low cost and efficient use of spare computing power that characterize volunteer computing.

Many new complications arise because the task creator, in addition to the task runner, cannot be trusted. Consequently, we introduce a form of data escrow where results are withheld from the task creator until they are verified. We also introduce a sandboxing model and API to allow general tasks to be run in a secure environment.
Similarly, Task Exchange introduces the concept of a market for computing power. We outline a constrained optimization method of assigning tasks as the mechanism for matching buyers and sellers, reminiscent of Google AdWords and AdSense pricing methods. Benchmarks show that tasks on Task Exchange have performance comparable to those run on a standalone server, and that this could be a viable new way of building supercomputing applications without owning or renting dedicated hardware or VPSs, or recruiting volunteers.

Works Cited


8. Larson, Stefan M., Christopher D. Snow, Michael Shirts, and Vijay S. Pande. 2002. "Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in." Computational Genomics.


