The Large MarLen Collider

Markus Peloquin et al.
markus@cs.wisc.edu

April 5, 2014

Abstract

Developing applications to run in the cloud presents a different set of challenges than developing for local machines. We implement a distributed checksum collision finder on top of Google App Engine, a platform for web applications. We examine the performance of our application, as well as the usability of the platform for such intensive uses. The problems we encountered are discussed, and our solutions to them are presented.

1 Introduction

Cloud computing is a term with various definitions; for the purposes of this paper, we will consider it to be the execution of applications on a cluster of machines owned by an external entity and allocated on demand, rather than on a machine or group of machines owned by the application writer. There are several major advantages of cloud computing over more traditional models. One advantage is that the machines can be shared between a number of users. It is likely that many will have little need of the resources they have, and unlikely that many will see a spike in need for resources at the same time. This is also an advantage to application developers; they do not need to pay for resources to meet their peak requirements continuously, but rather only for the resources that they actually use. This makes it possible to use many machines to execute tasks in parallel without actually paying for the machines all the time. Cloud computing is thus well suited for both applications with a large number of users and easily parallelized, computationally-intensive tasks. For parallelizable tasks, it is simple to scale up to many systems in order to reduce execution time.

There are also some disadvantages to cloud computing, which can generally be reduced to the loss of control. Such concerns include privacy, side-channel attacks, and vendor lock-in. Privacy is only protected by the provider's honesty. Side-channel attacks can be restricted to some extent, but the provider must anticipate future attacks and also make changes to mitigate current attacks. Vendor lock-in is definitely troublesome; the three platforms we investigated (Amazon EC2, Google App Engine, and Microsoft Azure) were significantly dissimilar, making porting impossible.

1.1 Distributed Computation in the Cloud

The goal of our project was to implement a distributed application to run in the cloud, and to evaluate the ease with which such an environment might support it. The application we implemented takes two files, appending data to the second in order to have identical CRC-32 hashes.[1] The cloud computing model should work well for this, because the problem is embarrassingly parallel; there is no need for communication between processes, as they all simply search a subset of potential appended data for collisions.

Cloud computing is not without its difficulties. This is especially the case with computationally intensive tasks, where shortcuts need to be made for reasonable performance. Debugging errors can be a problem given the separation between developers and the virtualized systems that execute the code. Providers may not make guarantees on the architecture code runs on. 64-bit architectures sometimes do not run even high level code in the same way as a 32-bit architecture would. The other characteristic of cloud computing that we consider a potential disadvantage is the constraint on execution environment. In our case, it would have been faster to compute the CRC-32s in a different language, such as C, but we were limited to using Java or Python. Though C may be much harder to debug remotely than Python, the potential benefits to application developers would be

[1] This is with the CRC-32-IEEE 802.3 algorithm used in Ethernet, POSIX `cksum`, and zlib.

much higher. However, the increased difficulty in preventing attacks on the framework when using a less safe language than Java or Python might explain why providers would be reluctant to provide these languages.

2 Methods

2.1 CRC-32

CRC-32 is a simple hashing algorithm used for error detection. It produces a 32-bit integer from an arbitrary stream of data, where the data is read one byte at a time. When appending a byte of data to a stream, recomputing the CRC-32 only involves one round of the CRC function. CRC-32 is fine for error detection, but it can easily be broken. We are breaking it by brute force only to show how well such distributed algorithms can work in the cloud.

2.2 Google App Engine

Google App Engine is a platform for cloud computing. We chose to use it to implement our CRC collision generator for several reasons. First, it provides a simple interface and does not require us to set up an entire virtual machine (as would have been the case with Amazon's EC2); although the flexibility to do so is useful sometimes, it was not necessary for this project. The services that App Engine provides were sufficient for our purposes, and having them already implemented rather than having to implement, test, debug, and optimize them ourselves was advantageous. Second, it allowed us to work in familiar languages (Python and Java are supported). Third, it is free to use for a limited number of CPU hours and use of other resources per day, making it much more attractive than EC2. Even with the paid plan, we only needed to pay for the overage charges: $.10 per CPU hour, where a CPU hour is misleadingly defined by Google with respect to a 1.2 GHz CPU, probably from Intel.

App Engine provides more features than we needed, but we tried to make use of as many as possible. Our desire was to become more familiar with the advantages of App Engine, rather than just its suitability to our chosen task. We used the *datastore* (database), the *memcache* (a key-value store), task queues, the *blob store* (stores uploaded files), and a cron-like task scheduler. It provides still more features for web application writers for authenticating users and sending emails, but these are irrelevant for the purposes of our application.

2.2.1 Offline Development

Google makes available a limited SDK for development offline. It was very useful for getting the basics down, but it had significant limitations. It only allows one page to be loaded at a time, which becomes a problem when developers wish to access the development console at the same time. No tasks ever execute unless manually through the development console. These two conditions mean our design could not be run except using the online environment. Other differences are that the default database is completely unusable (we could not read our writes for more than five minutes), no `DeadlineExceededError`s (thrown after 30 s) were ever thrown, and the development console is wholly dissimilar to the online development console.

2.2.2 Quotas

Google provides a limited amount of various resources to developers, many of which can be increased by enabling billing. When quotas are increased by enabling billing, developers only need pay for the resources used beyond the free quota. One challenge we encountered was that the service quality provided to paying vs. non-paying applications differs greatly, but is not clearly specified in the documentation. We found that the per-minute quotas for several resources (e.g. datastore updates) are higher for users with billing enabled than users without, even if both were still using the free quota at the time. This caused tests run from our non-paying account to overwhelm some short-term limits or encounter transient errors; this led to a cascading failure that somehow squandered all CPU hours and made the tests unrunnable. In contrast, the account with billing enabled rarely experienced any errors.

2.2.3 30 Second Time Limit

There is a thirty second wallclock time limit for every request. When this limit has almost been exceeded, a `DeadlineExceededError` will be thrown by any App Engine API call. The application can then catch this error, and perform operations such as saving some state so that it can be restarted later or returning a different code to the user. If the application does not finish within an unspecified but short time after the `DeadlineExceededError`, a `CancelledError` is thrown. The latter error is not catchable, and the script will terminate.

2.2.4 Datastore

App Engine provides a transactional schema-less, non-relational, strongly-consistent database. Instead of tables, it has entity groups, which are defined in Python as needed. The datastore supports transactions, allowing an entity to be retrieved and updated atomically, or rolled back if a conflict occurs. Each entity is limited to 1 MB in size, and for non-paying users, the total datastore is limited to 1 GB.

We used the datastore to store several types of information. We had a *Job* class in which we stored one entity per job, containing the original data and data for the last task that was assigned. There is also a *Task* class, which holds information uniquely describing each task to be run.

One problem that we encountered with the datastore is that it is incredibly slow. According to the App Engine System Status page, during the time we were developing our application, the latency for a single-entity `get` was frequently greater than 50 ms and sometimes greater than 400 ms, while the latency for a single-entity `put` was often greater than 125 ms, and sometimes greater than 500 ms[1]. These latencies posed a problem when each script was limited to 30 seconds, especially when some of that time was already taken up by the overhead of starting the scripts. Additionally, the latency of the datastore was large enough that it was not possible to use it to store intermediate state in the period between a `DeadlineExceededError` and the final `CancelledError`.

2.2.5 Memcache

The memcache is similar in appearance to memcached. It stores key-value pairs, with no way to iterate through them. The latency for the memcache is far better than for the datastore; the App Engine System Status page showed it as generally less than 10 ms[1]. For this reason, we used it to store information we needed to access frequently, such as whether a task had completed. It is also the only option for storing data after a `DeadlineExceededError`.

2.2.6 Blobstore

The blobstore is an immutable store for larger objects than can fit into the datastore. It will take files uploaded from a web form, and inserts them into the blobstore. References to the an entry can be kept in the datastore with a blob key. The filesize limit for an entry in the blobstore is 50 MB, as compared to the 1 MB limit for the datastore. However, it is still only possible to retrieve these items 1 MB at a time, due to a limit on API calls. We used the blobstore only for files uploaded by the user, although direct input is also possible.

2.2.7 Task Queues

Because of the time limit for each script, and because there is little point in running this application in the cloud if we cannot take advantage of large amounts of parallelism, it was necessary to find some way to execute many tasks at once. App Engine provides task queues, an experimental feature to execute tasks asynchronously. Tasks are basically private scripts, and so have the same restrictions. There is a limit to how many tasks the queue can invoke per second, and they are still subject to the same 30 second time limit as any other script.

2.2.8 Fault Tolerance of App Engine

Although App Engine is to some extent presented as providing guarantees of fault tolerance and scalability, in reality, it does not provide complete fault tolerance and not all features scale. For example, when we attempted to run many tasks at once, we ran into quota problems; even if an application is scalable in theory, if it exceeds the quota limits or stresses the datastore or task queues, it will not be scalable in App Engine. There are also few guarantees when it comes to tasks completing. We encountered several errors, including ‘transient errors’. These are passed on to the user scripts, to handle as they choose; App Engine does not guarantee error-free execution of user scripts, so it is the developer’s responsibility to build in fault tolerance if it is necessary. Although tasks have the option of returning a failure status code and being automatically re-executed, this is not always a good option because in some cases the problem might be quota-related, in which case automatically re-executing the tasks would only exacerbate the problem.

3 Implementation

3.1 Master

Our model for finding a collision is to have a single master to create tasks. When the interface web page contacts the master script, the master will pick up where it left off. The solution was to use the datastore for persistence and fault tolerance. The datastore contains for each Job entry the last *end* value written to the Task (notional) table. Each time an entry is added to the Task table, a task is added to the queue (the private task page and some parameters). The expectation is that a Task entry will exist until the task has completed. When the master runs out of time it saves its state and returns a short document that indicates it has not found a solution. The client then polls the master again.

We added tasks to the task queue from the master script. Each task contained a subset of the suffixes to test. To ensure that the tasks did not continue running after a solution was found, each one initially checks the memcache to determine if a solution has been found, and if so, exits immediately. They also periodically check that a solution has not been found while they were executing; we chose to do this once per 16,384 checksum calculations to balance the latency of a memcache request against the time spent unnecessarily checking checksums. This was done to avoid wasting CPU time, since it was a finite resource.

3.2 Tasks

The Tasks do the work of locating collisions. They can be described as a simple tuple: a unique CRC (though a pair of CRCs is perhaps better), a beginning string, and an ending string. It might say 'try appending each string from `a---` to `b---`'. Tasks calculate the CRC of the original text appended with the current string. Failing a collision, they increment the current string and repeat. If a task happens to find a collision, it says as much to the datastore and memcache. Upon task completion, it removes its entry from the Task table.

As an optimization, tasks are sent their arguments as CGI parameters. They need not ask the datastore or memcache for anything before beginning. Still, tasks may sit in the queue for a long time, so they will look in the memcache to see if a collision has been found.

3.3 Fault Tolerance

In order to deal with `DeadlineExceededErrors`, we needed to provide a way for tasks to be resumed. Because the time to access the datastore is so high, it was not possible to simply update the job or task entry in the datastore. What is needed is a way to both save progress and restart.

3.3.1 Progress

Progress is generally stored in the datastore, but when that is not an option only the memcache is used. Upon resumption, the datastore can be updated to reflect what was written in memcache. Tasks store the last value they attempted to check, while masters store the last *end* value written to the datastore.

3.3.2 Resumption

Tasks need to store more than just their last value: they need to indicate their liveness. They also store a timestamp and a flag to indicate if they timed out. When the flag indicates they are running, it serves as a heartbeat. When the master starts, it will check the Task table for tasks that have not completed. If any of them have not been running in the last five minutes, or have timed out, they are restarted.

Restarting master processes is much easier, as there is only ever one. The web interface will restart the master whenever it gets an 'uncompleted' response. The master will first check for progress in the memcache. It then updates its table entry, checks for dead tasks, and then creates tasks until the deadline is exceeded.

It is thus not necessary that a process successfully save its state to memcache, as long as the service usually works. Tasks can not escape from the master's attention, and the master cannot escape from the browser refreshing it. Though it is rarely necessary, our system loses no progress even when the system load is higher than normal and our tasks fail to complete.

3.4 Difficulties

Though Python is cross-platform, it uses libraries that are not. We ran into some problems with the CRC function returning different results when running the application on our own machines or in the cloud. Zlib produces purely 32-bit integers, so when the size of a Python `int` is 4, as it is only on 32-bit architectures, the number is negative. Since Google makes no guarantee on the architecture code runs on, we needed to occasionally perform type checks.

One major complication we ran into is far too irrational to be intended: the mysterious `DeadlineExceededError`. There are in fact two such exception types in the App Engine library. The first is easy to access, but the second in a module that is not quite so easy to find, and certainly not documented anywhere. Perhaps Google simply refuses to help users that use a full 30 s.

4 Evaluation

We evaluated several factors of our implementations on machines we had access to as well as the App Engine machines.

4.0.1 Comparison of Single-Task Execution

In order to be able to compare the time to find a collision on the App Engine and on our private machines, we determined the relative time to execute just the computation phase of one task. We measured this time on several machines, with results shown in Figure 1.

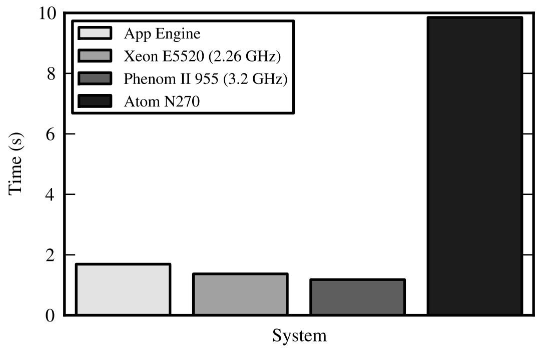

Figure 1: Time to execute one task in Python. The N270 is clocked at 1.66 GHz, but this is misleading when comparing to an x86 CPU.

These measurements do not include the overhead of starting the script; it is measured from before the function call to calculate the checksum for 1,048,576 possible values until the function call returns. It also does not include any calls to the datastore or memcache, as they would lead to an unfair comparison.

Language	**Time (s)**
C++	0.0126
Python	1.18

Table 1: Time to execute one task

Each test was run 100 times, and the results were averaged. The machines that the test process was running on for App Engine gave reasonably good performance, in line with our own machines.

4.1 Python vs. C++

One of the disadvantages of App Engine was that it limits the developer to using Java or Python, languages which might not be ideally suited for the task at hand. We were curious about the difference in performance between a Python implementation of the collision finder and a C++ implementation. Both implementations are using the same zlib implementation, though probably different versions. To test the performance of each one, we again ran one task, once in C++ and once in Python. Both tests were run on a Phenom II 955. The results can be seen in Table 1. We found that the C++ implementation is approximately 100 times as fast as the Python implementation.

4.2 Time to Find a Checksum

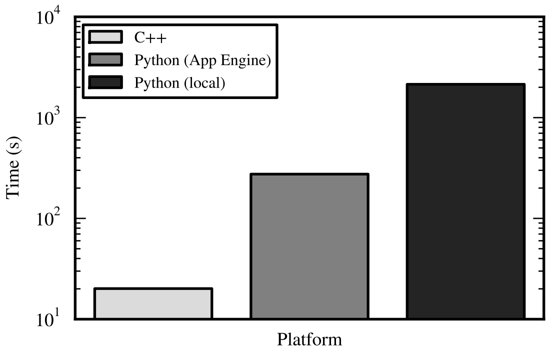

Figure 2: Time needed to find collision for ‘alpha’, ‘beta’ on a log scale.

We compared the time taken to find a colliding checksum on three different implementations: a sequential C++ implementation, a sequential Python implementation, and the distributed App Engine Python implementation. The results can be seen in Figure 2. The App Engine version was approximately 7.8

times as fast as the sequential Python implementation, indicating that we were finding some benefit in executing multiple tasks in parallel. After three trials, the cloud implementation had used 7.1 'CPU hours'. Assuming the GHz rating is a true indicator of performance, it takes 37% more CPU hours in the cloud than with the sequential Python implementation. We also found that although the App Engine version was faster than the sequential Python version, the sequential C++ version was over 10 times faster than even the App Engine version.

5 Discussion

5.1 Ease of Development

As previously discussed, we ran into a variety of problems with developing our application, many of which can be attributed to the fact that App Engine is designed as a platform for low-latency web applications, rather than computation-intensive applications like this one. This direction App Engine has taken is apparent in a few places: the supported languages, task queues being an experimental feature, and a 30 s limit for all scripts. The most annoying problem we ran into was the time limit; however, dealing with the errors caused by it and the ones caused by attempting to queue or run too many tasks at once forced us to incorporate fault tolerance in our design where we had not anticipated needing it.

Another problem we encountered was that we were limited in methods of communicating between different processes. This was not a large problem in this case, since we mostly only needed to communicate whether a solution had been found and that could be done simply through the memcache, but for applications that are less embarrassingly parallel, the lack of easy, fast, and reliable interprocess communication would pose a challenge. Synchronization is another possibility, which can be achieved with the datastore, but at a high cost. Transactions are slow, and timeouts make it quite difficult to release locks.

5.2 Performance

Although the performance per individual task was slightly worse than what could be obtained for a desktop machine in the case of the Python implementation, the hypothetical advantage of using App Engine was that we could have large numbers of machines running tasks in parallel, allowing the checksum to be found more quickly. However, in practice we ran into numerous problems. The short per-script time limit meant that we spent more time starting tasks and made the fault tolerance more difficult and time-consuming. This was compounded by limits on the number of tasks that could be added to the task queues per second, the number that could be invoked per second, and a limit on CPU-hour and other quota usage per second.

The variability in service is another difficulty. One simple solution for job partitioning might be to just reduce task size, but as can be seen in 3, this often will not work. It may be that the spike is caused by task allocation. It seems likely that there is high variability even in the systems Google provides. The first 100 requests see roughly the same performance, but the task run time drops sharply thereafter. Perhaps this hints that tasks are usually allocated to the same systems for data locality.

One interesting point is that the performance of even the computation for one task was far worse on the Atom processor than on a machine in the cloud. This is because netbooks are designed to provide long battery life at the cost of slower performance. The idea is that netbooks can be used for connecting to the internet, and then using applications in the cloud for anything computation intensive (for example, using Google Docs rather than running OpenOffice locally). In our case, you are better off trying to look for checksum collisions using a App Engine application than running it on the netbook, at least when using Python.

In terms of performance compared with a fast sequential implementation and fast App Engine implementation, the App Engine performance was significantly hurt because of the necessity of using Python. If the goal is to provide the best performance, Python is not an ideal language, and by limiting the code that can be run on the App Engine to Python or Java, the platform's maximum performance is limited compared to an environment where any language might be used.

Furthermore, although the particular application, language, and platform were in our case not well suited to obtaining the best performance, the fact that we obtained some speedup over the sequential case (for the same language) indicates that there is no fundamental flaw in the approach we took: dividing up the work into a number of subtasks, letting each execute in parallel in the cloud, and returning the results to the user.

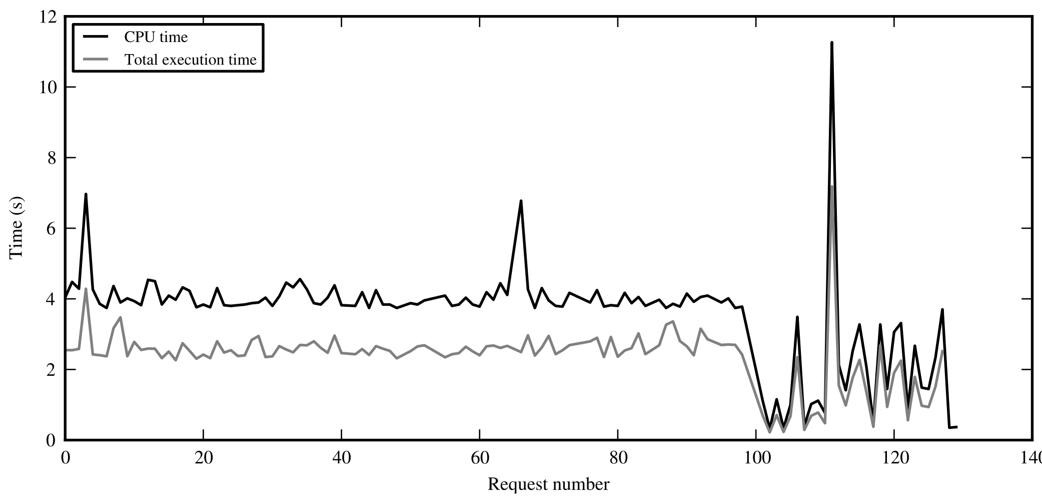

Figure 3: CPU time of tasks, and the amount of time they *actually* run for. The trace was generated at off peak hours with the words 'zeta', 'beta', a pair with a quicker solution.

6 Conclusion

Google App Engine is a platform that is still under development, and it is furthermore intended for low-latency web applications. For our project, we attempted to make use of its features, some of them experimental, to develop a computation-intensive distributed application. In doing so, we encountered several problems, which could be attributed to a mismatch between our goals and the intentions of the platform. These problems included script time limits which were too short for our purposes, throughput limits, and communication limits. Despite these difficulties, we implemented a checksum collision creator and demonstrated that it showed speedup over a sequential implementation in the same language. We also showed that it provides fault tolerance and progress is guaranteed, so long as the underlying framework continues to execute some of the tasks.

References

[1] Google app engine system status. http://code.google.com/status/appengine/.