Cloud Based Optimization with Preemptible VMs
Masters of Engineering Project

Paul West (prw54)

May 29, 2017

Introduction

Plumbing for Optimization with Asynchronous Parallelism (POAP) is a Python package written by Professor David Bindel for the asynchronous execution of callback-based optimization routines. Generally, POAP uses a centralized controller to disseminate commands from customizable strategies to a pool of workers, and return the results of the worker's computations to the strategies. My project was to create a framework on top of POAP to allow the easy creation and management of workers on preemptible virtual machines on the Google Compute Engine (GCE) platform.

Implementation

This project is broken down into two parts: creating workers that can gracefully handle preemption events, and creating a framework for managing a pool of preemptible workers on the Google Compute Engine platform.

Preemptible Workers

Google Compute Engine allows the creation of inexpensive virtual machines that use excess computing resources at less than the cost of normal virtual machines on the condition that GCE can shut down preemptible virtual machines and claim back their resources with 30 seconds of notice. In order for a POAP worker to operate in that environment, it should be able to detect a preemption event, gracefully shut down any ongoing work, and inform the controller before the virtual machine on which it runs is forced to shut down. Ideally, the worker should also be able to save its state so that a different worker can pick up where it left off.

POAP allows for the creation of many different types of workers. In deciding how to tackle this problem in a general way, I realized that POAP workers have three main defining characteristics:

1. Function Evaluation Style: how POAP executes units of optimization work. Two examples are simple evaluation, in which a worker calls a Python function, and process evaluation, in which a worker spawns a subprocess from an executable.

2. Communication with the Controller: workers operating on different servers from the controller require different communication protocols with the controller than do local workers, and different types of remote workers have their own differences.

3. Other Evaluation Capabilities: this includes interruption detection and preemption detection among other things.

I decided that instead of implementing the complete set of capabilities in each worker, these attributes should be factored apart in such a way that creating a worker with any combination of attributes would be a matter of picking the correct components. Because evaluation style boils down to a function call and communication with the controller involves putting a wrapper around the worker interface, I decided to start by defining a set of classes to implement the Other Evaluation Capabilities. I call these the BaseWorkerTypes. They are:

1. BaseEventWorker: a worker that handles commands from the controller and internal events through an event queue. The BaseEventWorker defines the callback structure that subclasses use.

2. BaseInterruptibleWorker: a worker that allows interruption of eval instructions from the controller. Evaluation is run in a separate thread that adds a callback to the worker's event queue when evaluation is complete. In the future, this should be done with the multiprocessing module instead of the threading module. The worker includes hooks for handling terminate and kill instructions from the controller as interrupts.

3. BasePreemptibleWorker: a worker that adds detection and handling of preemption events as interrupts to the BaseInterruptibleWorker. Functional BasePreemptibleWorkers must implement preemption detection. I have added Google Compute Engine specific preemption detection as a class in PreemptionDetectors.

Adding communication with the controller is the next step. I provide example implementations of thread workers and TCP socket workers, and then use multi-inheritance to combine communication structure with interruption and preemption capabilities in brief classes. These may be found in ThreadWorkers and SocketWorkers. SocketWorkers have two components: the remote worker, and a local handler. Local handlers are found in SocketWorkerHandlers. The controllers and servers corresponding to the different types of worker are found in PreemptibleControllers.

The last step is adding function evaluation style. I provide example implementations of simple python function evaluation in SimpleWorkers and the framework for subprocess evaluation in ProcessWorkers. All final workers are built as small classes built using multi-inheritance of higher-level components.

Recoverable Preemptible Workers

Because of differences in the interfaces of the default POAP local and remote workers, I opted to implement recovery for preemptible workers only for TCP socket workers. The implementation should be easily modifiable to work with other types of remote worker. The general idea is that recoverable workers pass state and lock objects to the evaluation function. The lock modulates write access to the state for the evaluation function, and read access to the state for checkpointing. State objects are created with a record ID and an optional string which tells the state whether and how to recover. State objects also know how to save themselves, and output a recovery string upon successful saving. Recoverable workers include the recovery string in their preemption message to the controller. A recovery strategy that I built as an extension of the POAP Retry strategy handles retrying recoverable preempted computations by telling a different worker to recover evaluation with that state object. It is up to specific state object implementations how to save and recover themselves, and what to store. I have provided an example implementation of a self-pickling and self-unpickling Python dictionary.

Google Compute Engine Workers

The second section of this project is streamlining the creation and management of workers on virtual machines on the Google Compute Engine platform. I decided that worker management should be handled separately from the worker pool that the controller knows about, and that a single virtual machine should be able to host multiple workers. Thus I implemented a virtual machine monitor, called GCEVMMonitor, that wraps the creation and deletion of VMs within GCE. The GCEVMMonitor communicates with a worker manager called a GCEWorkerManager that boots up on VM startup. The GCEVMMonitor tells the GCEWorkerManager when to start up new workers, and provides the GCEWorkerManager with the information with which to do so. The GCEWorkerManager starts workers as their own processes with communication information for the controller. The GCEWorkerManager is not responsible for gracefully terminating workers. As implemented, the GCEVMMonitor has a hard-coded startup configuration, but this is easily modifiable. On startup, a VM executes startupscript.sh, which installs necessary packages for the worker to operate, and then creates the GCEWorkerManager. So that a GCEVMMonitor knows which workers are associated with it, a hook to add workers is implemented in the GCEVMMonitor class, and a callback on creation of SocketWorkers is included in all of the TCPServers in this project. An example of the use of this final project can be found in tests/test_gce_execution, which starts a single recoverable TCP socket worker through a GCEWorkerManager on a Debian server created by a GCEVMMonitor that is notified by a RecoverableThreadedTCPServer when a new RecoverableSocketWorker connects to it. The example uses simple function evaluation and a version of the POAP FixedSampleStrategy modified for use with the recovery strategy. The

user must execute this script from a virtual machine running as the controller on a GCE server, as my TCP workers only handle connections to other GCE VMs.

Future Work

BaseInterruptibleWorkers should run optimization function evaluation in a subprocess instead of a thread. GCEVMMonitors should allow for more configuration of the VM parameters, such as number of cores, amount of memory, etc. TCP workers should be given the ability to communicate over the public internet instead of only inside GCE. Communications over TCP should be encrypted. Management of a pool of GCEVMMonitors should be formalized, along the lines of the test_gce_execution example. Monitoring worker liveliness via heartbeats should be implemented. Preemptible workers on remote servers other than over TCP connections should be implemented.

Conclusion

I have created a highly modular, easily customizable, relatively easy-to-use, and hence highly extensible framework for using POAP on unreliable machines.