Party Distributed Processing

Johnson Lam   Justin Chen

Boston University

April 21, 2017

Abstract

In the past few years, machine learning and deep learning have burgeoned from theoretical research to hobbyist computing. Training algorithms require extremely large datasets and users without access to large amounts of resources must purchase them from cloud services, such as AWS, which can be extremely expensive over time. We propose an alternative solution using commodity hardware spread across the internet to tackle this barrier to entry. This could allow any user the opportunity to do computationally intensive work

I. Introduction

Open source software libraries, such as TensorFlow, Torch7, and Theano, have made machine learning more accessible than ever before. However, training learning algorithms require massive datasets in the order of several gigabytes to several hundred of gigabytes, which require expensive, specialized GPUs. These models, however, can be just as effectively trained on a large number of CPUs. Cloud services provide these resources at a cost and their interfaces can be difficult to navigate for new users.

The average computer is typically under-utilized or idle in a typical day. These available resources could be altruistically allocated to agreed party members for distributed computation. We propose an affordable and easy-to-use distributed data computation paradigm for processing datasets across a party of commodity hardware connected by the internet. Our goal is to lower the cost of entry and simplify the interface users interact with. We use the MapReduce architecture as a guide for our own design and implement a distributed word counter as a proof-of-concept on BU's csa servers.

II. Design

We propose utililzing hardware owned by a trusted network, like a set of computers owned by a group of friends or a trusted organization. This can even be extended to locally connecting a group of machines that a user accumulates over the years. Our current design utilizes a "master" cloud server that will coordinate work across the party, combine intermediate results from each worker, and return the final result to the client that submitted. Each computer in this network can act a client wanting to do work or as a server available to do work for others.

Each computer begins by stating their available resources (cores, RAM, etc.) and their restrictions (bandwidth) to the master server when it comes online. At any point in time a computer on the network can make a request to the master server that it'd like to do some work on a dataset and pass the dataset to the master. Similar to how the master node took care of distributing work in MapReduce, the master node in our system will distribute work amongst the available servers in the network based on each computer's available resources and restrictions. The bottleneck of this design will be the network speed and reliability of each computer. Given this restriction, distribution of work will be in smaller chunks such that sending data over the networkw will not be too cumbersome. The master server will be responsible for handling network failures and unresponsive or slow workers in the same way as if these servers were in the same data center.

Our current work achieves the abovementioned by distributing work for counting words in a large document, except it does not take into account the available resources and restrictions for a worker. A worker can come and go at any time, and work will be distributed accordingly to this dynamic network inspite of worker failure.

The master keeps track of workers currently available to it by sending periodic heartbeats and removes workers from the list who are unresponsive. When finished, the master returns the final word count to the worker that submitted the request. This implementation is currently running on csa1, csa2, csa3, and ks-sesa with csa1 being the master server.
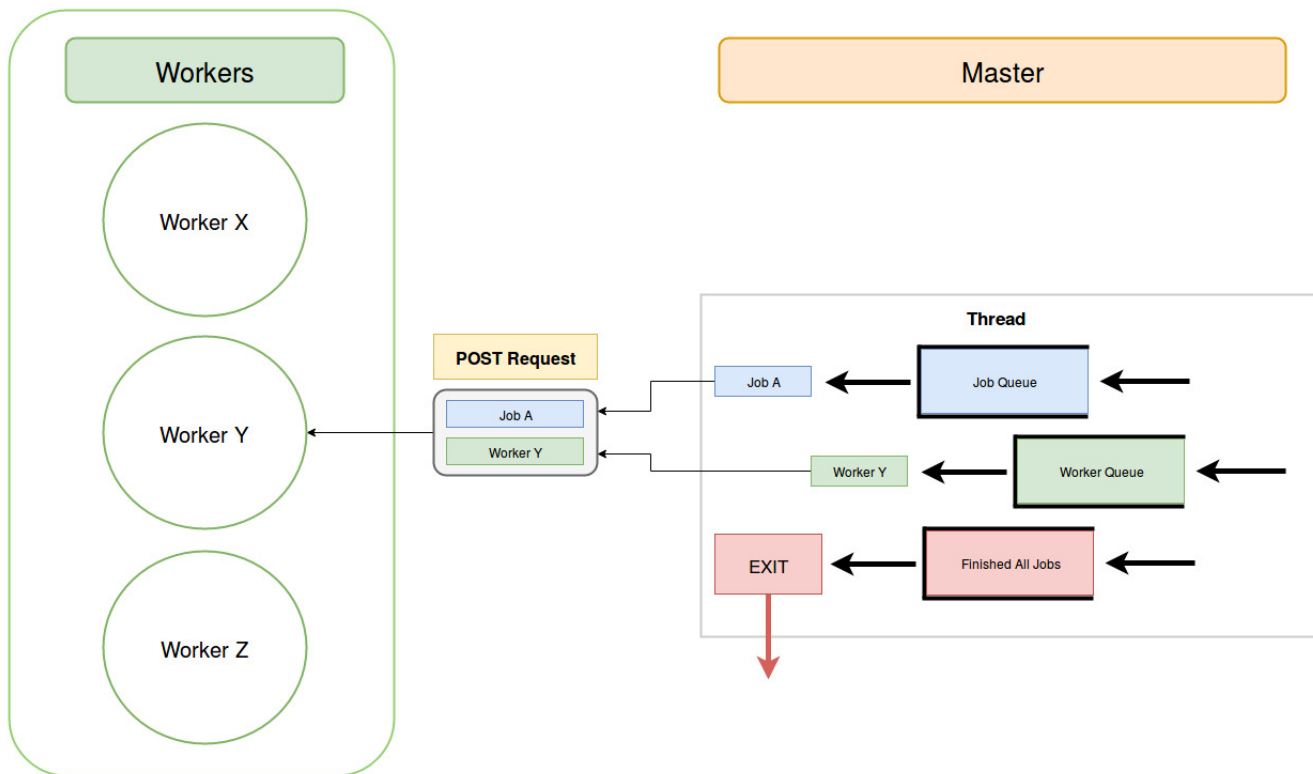
III. ARCHITECTURE

Our program is written in Go and utilizes the "net/http" package to communicate between the worker and master. The master has four endpoints: join, heartbeat, job_request, and job_chunk. The worker has three endpoints: heartbeat, process_chunk, and finish_job.

When workers first come online, they notify the master they are joining the network. The worker then idles and waits for either a heartbeat or a job from the master and responds to the master accordingly. When a worker submits a request to the master, the user specifies a URL containing the text document and a name for the document. The worker contributes its resouces for work its given to do by the master. It will then be notified of the final result of its request when the master is finished distributing work.

The master handles all of the coordination on its fleet of machines. When it first boots up, it starts a thread to send periodic heartbeats to workers that have joined the party. When the master sees a new worker joining, it adds it to its map of current workers and starts a thread responsible for making sure it's responsive by using a timer. If a worker does not respond to a heartbeat, this timer will not be reset and will eventually fire. When the master receives a job request, it makes a GET request on the URL provided, downloads the page onto its machine, and splits the file up into 1000 line file chunks. It then distributes the file chunks to the available workers, combine their results, and returns it to the submitter

The key components of our work are heartbeating and distribution of work for a request. We use heartbeating as a form of checking on the availability of workers and checking on worker progress on a file chunk. The former case is simple: if the worker is unresponsive after three timeouts, delete the worker. In the latter case, if the worker is unresponsive or fails to make satisfiable progress, delete the worker and add the file chunk back into the queue of jobs that need to be processed.

We distribute work by creating a separte thread with two channels and treat them as queues, one for jobs to be worked on and one for available and idle workers to process these jobs. We then insert the file chunks and workers into their respective queues. A third channel is used to exit this thread when all jobs have been completed, indicated by when the last file chunk is received. As mentioned earlier, a job is reinserted back into its queue if unsatisfactory progress is made by the worker and the worker is deleted. If a new worker joins in the midst of a request being processed, it adds itself to the worker queue and will be given a job when one becomes available. When a worker returns the result for a job, the master adds it back into the worker queue. The insertion of jobs and workers into the queue is done on separate thread to ensure that it does not block the master- adding things to the channel will be blocked if a reader is not present on the otherside. The following code illustrates this:

IV.   BENCHMARKS

preliminary testing further rigorous testing

V.   FUTURE WORK

i.   Functionality

ii.   Architecture

REFERENCES

[Figueredo and Wolf, 2009] Figueredo, A. J. and Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330.