CS380L Project Writeup: Distributed Completion Service

Jason Z. Liang
Department of Computer Science
University of Texas at Austin
jasonzliang@utexas.edu

Abstract

Task parallelism is difficult to implement in a distributed setting due to machine unreliability and communication latency. HTCondor, an existing distributed computation framework, is insufficient for addressing these shortcomings. In this report, we present a high level abstraction on top of the HTCondor called the Distributed Completion Service (DCS) that improves upon reliability and latency. Experiment results reveal DCS's ability to scale and demonstrate a reduction in latency when using DCS compared to just using HTCondor.

1 Introduction and Background

Task parallelism, especially in a distributed setting where multiple tasks are running concurrently on a separate machines, has been an area of intense research recently. Tasks parallelism is useful tool for solving computationally heavy problems in many areas of computer science. For example, in machine learning, it is often leveraged to perform hyperparameter search concurrently on many different models [1]. It is also used to support essential web services such as web search and indexing [2]. However, distributed task parallelism is considered to be a challenging problem due to the following reasons: 1) Hardware and software are unreliable and any nodes/tasks may fail or become unavailable at any moment. 2) A small fraction of tasks sometimes may lag behind (also know as the straggler problem). 3) There might be high communication and startup latency for worker nodes. 4) Writing a distributed algorithm often requires a lot of boiler plate code.

Many distributed frameworks have tried to alleviate these issues and maximize the throughput of a computational cluster; these include Map Reduce [3], Spark [4], and Dandelion [5]. Most of these frameworks contain sophisticated algorithms that increase system reliability and minimize the performance cost and latency of executing tasks. The distributed framework in the UTCS Department, HTCondor [6], solves some of the low level issues with distributed computing such as job scheduling and user multiplexing; however it is not capable of solving the problems stated above. HTCondor incurs a rather high latency for submitting jobs and executing them and has no reliability guarantees for tasks. Lastly, submitting jobs to HTCondor requires creating a verbose submission file with many flags and options that are irrelevant to many users.

This report presents a novel distributed framework called the Distributed Completion Service and is built on top of HTCondor. It provides improved reliability against task failure, a method to detect and handle stragglers, reduced latency and overhead, and a simple and clean API for users.

2 Distributed Completion Service Design Overview

The Distributed Completion Service (DCS) is composed of four main components: 1) A server that represents an entry point for the user to submit task and get back results. 2) Clients that run either locally on the same machine as the server or on a remote machine using HTCondor. 3) A client

manager which allows the user to control the number of clients running locally or on HTCondor. 4) A pair of submit and return queues which is used to send tasks from the server to the client and results from the client back to the server.

The typical workflow for an user utilizing the DCS would be like the following: 1) The manager is used to spin up a fixed number of clients either locally or HTCondor. 2) The user submits tasks through the server, which the server then pushes into the submit queue. The clients which are running pull tasks from the submit queue in the order in which the tasks are submitted (FIFO) and begin working on them. Once the task is finished, the standard output of the task is placed onto the result queue. The server will then pull the results from that queue and return them back to the user. Exit codes are also passed back to the user as part of the results and are used to indicate whether the task is successfully executed or not (non zero exit code means failure).

2.1 User API

The API for the DCS has been explicitly designed to be easy to use. As such, there are only two main functions which the user can call. They are the following:

```
submitTask(task, task_data, estimated_time)
getResults(timeout)
```

In submitTask(), the user provides a string that points to the filesystem path of a shell runnable script or program (task), a list of arguments to the task (task_data), and optionally an estimate of how long the task will take (estimated_time). The getResults() function takes a single argument (timeout), which specifies how long getResults() should block for and returns the output of a corresponding task that was submitted. If a result is not returned before exceeding the timeout, a null reference is returned instead. Below is an example of how the user would use these two functions to perform computation in parallel.

```
x = CompletionServiceServer(clean_start=True)
for i in xrange(100):
  x.submitTask("test_tasks/square.py", i)
for i in xrange(100):
  print "i:", x.getResults().task_data
```

As the example above shows, the user can easily leverage massive task parallelism in only 5 lines of code. This is much less than the amount of boilerplate code needed to execute the same tasks using HTCondor.

2.2 Reliability

Unlike HTCondor, which provides no guarantee for successful completion of tasks, DCS improves reliability by allowing each task to be retried by the client up to K times, where $K > 1$ is a hyperparameter specified by the user (by default it is set to 3). Alternatively, if there are K more idle clients (clients not working on a task) than the number of tasks in the submit queue, the computer service will submit K copies of the same task.

This policy guarantees that the user will have a high chance of getting a successful result for the task with bounded time. The probability of task failure that decreases exponentially as K increases. Assuming each task has C chance of failure $0 < C < 1$, the chance of getting at least once successful result back for a task is $1 - C^K$; as we see converges quickly to 1. In the worse case, the amount of time to get back all results for the tasks bounded by $K * \frac{TM}{C}$, where T is the number of tasks, K the number of clients, M the longest time for each task, and C is the number of clients available. In the best case, when $C >> T$, overall running time is $\frac{TM}{C}$. This time is no different than if each task is completely reliable and no retries are required. Thus in the worse case scenario, the overall running time when trying unreliable tasks is only K times longer than completely reliable tasks.

2.3 Detecting and Handling Stragglers

Straggler handling is another important shortcoming of HTCondor that the DCS deals with. Stragglers are tasks which complete much slower than expected and could be due to the slow performance of

the hardware where the client is running on, or if the client is frequently suspended by the HTCondor job manager. To deal with stragglers, DCS will automatically estimate the running time of all tasks submitted. To do so, it will keep a history of the running times of all past tasks that returned results. When submitting a new task, its running time is conservatively estimated as the the mean running time plus two standard deviations of the past running times. If the actual running time of the task is P (a hyperparameter set to 2 by default) times longer than the estimated running time, the task is deemed a straggler.

Straggling tasks are allowed to complete on their current client, however straggling tasks are also moved back into the submit queue which then other clients can pull and work on. Straggling tasks can only be restart a limited number of times. The number of times a task can be retried due to failure or due to flagged as straggling cannot exceed a total of K times. This ensures that the system will not wait indefinitely for an straggling task to finish.

2.4 Minimizing Task Latency

Compared to HTCondor, the DCS exhibits much lower latency for submission of tasks when all the clients are running. This is due to DCS avoiding the overhead of the HTCondor job scheduler, which depending on the overall system load, can add at least 30 seconds to several minutes of delay before jobs begin running. When using the DCS with clients running locally on the same machine with a SSD, the latency between submitting a task and having the task begin execution can be as few as several milliseconds. When on a machine with network file system (NFS) [7] and clients running on HTCondor, the latency is also only around several hundred milliseconds. However, if the user tries submit tasks to the DCS from a cold start (with no clients running), the user will still initially incur the heavy penalty of starting up clients on HTCondor.

To remedy this issue, DCS has an ability to running in a setting called "hybrid mode". In "hybrid mode", DCS will not only spin up clients locally, but on HTCondor as well. The clients on HTCondor will still pull tasks in a FIFO manner, but the local clients will prioritize pulling tasks from the submit queue that have short estimated running times. This allows short tasks to be executed and completed even before the clients on HTCondor are up and running. If the user is submitting a significant fraction of tasks that complete quickly, the fast completion of these tasks can help mask the startup latency incurred by running clients on HTCondor.

3 Implementation Details and Tricks

The DCS is implemented in Python [8] using mostly standard library modules. Python is chosen because its powerful standard library contains useful components for filesystem traversal, object serialization/deserialization, and managing/forking processes. Some preliminary benchmarking and profiling has shown that the performance bottleneck is usually due to filesystem reads and writes. Thus, Python is considered to be sufficiently fast enough as the language for the first version of the DCS.

When the DCS server starts up, it creates a base directory. This base directory contains the working directories of the server and clients. When the clients or server pulls tasks from the queues, these tasks are stored in their respective working directories. The working directories also contain log files and can be used to debug remote clients which crash.

3.1 Container Object for Tasks and Results

Submitted task and returned results are encapsulated into a container object before they are serialized and placed into a queue. Serialization and deserialization of task object is performed using the cPickle standard library module. Pickle files are binary encodings of arbitrary Python objects and do not require any schema or writing any additional code. An explanation of the fields in this container object is provided below:

1. **time_created**: time when task or result is created.
2. **uid**: random string that is the same for duplicate submitted tasks, retried tasks, and any results which come from the task.

3. **num_failures**: number of times that the task has failed and been retried.
4. **is_result**: boolean flag whether object contains a result or submitted task.
5. **task**: a filepath string of the command/program to run if object is task, None otherwise.
6. **task_data**: a string of command arguments for submitted tasks, otherwise output of submitted task for result.
7. **estimated_time**: estimated time for submitted task, None for result.
8. **metadata**: a dictionary that contains information about a completed task such as exit code, time elapsed, and standard error.

3.2 Submit and Return Queues

The submit and return queues are both filesystem based queues which allow atomic push() and pop() operations without the use of locks. Each queue is represented as a directory and tasks/results in the queues are represented as files inside the directory. Each task has a unique filename that is composed of a random string, the time when it is added to the queue, and the estimated time of the task. These information are included as part of the filename in order to simplify lookup when pulling tasks from the queue either based on earliest timestamp, or shortest estimated running time.

The queues must be safe from race conditions even if the server and multiple clients are simultaneously using the queue at the same time. To ensure good performance of the push() and pop() operations, no locking is used for the queues. Instead, the push() operation is ensured to be atomic by using the rename() system call, which is guaranteed to atomic on all POSIX compliant filesystems, including NFS. The task is first serialized to a temporary file and then moved using rename() into the queue directory. Similarly, when pop() tasks from a queue, the task is moved into the working directory of the client or server and then deserialized.

It is interesting to note that non-filesystem system queues are also considered when designing the DCS. For example, the shared queue between clients and server could be implemented as a another server where all the clients and the first server connect to it through network sockets. The tasks on the queue server would be stored in memory and accessed through a multiple producer, multiple consumer thread-safe queue. However, because the performance of the NFS filesystem is sufficiently high and due to the complexity of implementing a protocol for communicating through the network, a filesystem based queue is chosen instead.

3.3 Client Heartbeat and Command Files

One issue with running clients on HTCondor is that their reliability are not guaranteed. The HTCondor job scheduler can kill or suspend clients at any moment without warning. Thus, one of the necessary tasks of the server is to detect clients which failed and move the tasks that the failed clients are working on back into the submit queue. This is achieved by having the clients maintain a heartbeat file in their working directory. Every H seconds, the client must refresh the heartbeat file with the current time. The server checks the heartbeat files of all the clients at regular intervals. If a heartbeat file is more than $I * H$ seconds old, then the client is deemed to have failed, and the server can proceed with cleanup. By default, $I = 5$ and $H = 3$.

Client also constantly scan for the presence of a command file in their working directory. If it detects a command file, it will open it and execute the commands inside. The currently supported commands are the following: 1) finish the current task and then shutdown and 2) shutdown immediately. The command file is used by the manager to shutdown clients and is much simpler than keeping track of IPDs of local clients and the job IDs of HTCondor clients. Clients also automatically shutdown if their working folder is deleted.

4 Experimental Results

All of the experiments are run on a UTCS instructional machine with 8 cores and 16 GB of ram. The filesystem on the machine that DCS depends on is NFS. The machine is also connected to HTCondor and can submit up to 150 jobs at a time. All the plots below show the mean results from 5 trials. Because the variance between each trial was very small, standard error bars are not shown.

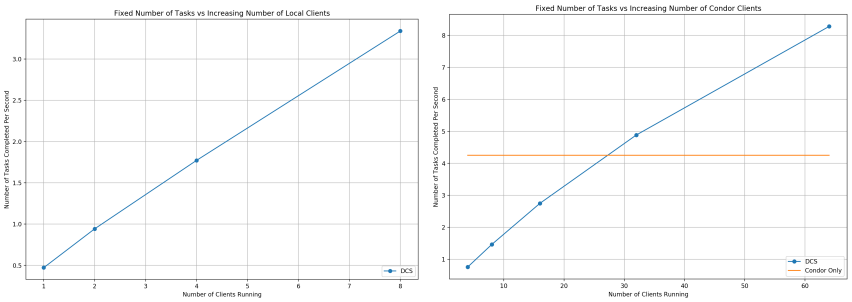

Figure 1: Left: The plot shows how task throughput increases vs number of local clients. Right: The plot shows how task throughput increases vs number of HTCondor clients. The orange line shows task throughput when submitting directly through condor for 128 tasks. Both plots show mean values from 5 trials.

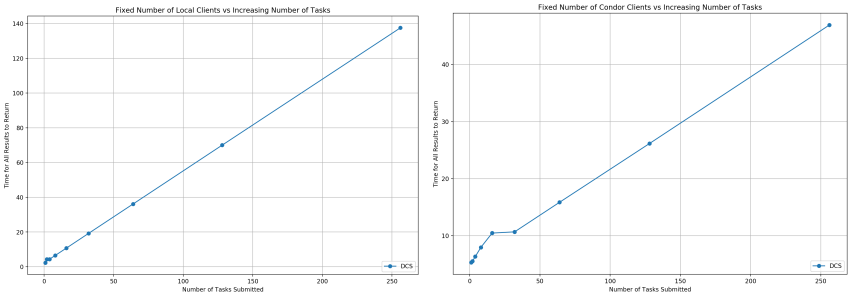

Figure 2: These two plots showing how total time to get back all results increase with number of tasks. Left: 4 local clients. Right: 32 HTCondor clients. Both plots show mean values from 5 trials.

4.1 Scaling vs Number of Running Clients

In the first experiment, we examine how DCS scales with increasing number of local and HTCondor clients. First we examine task throughput of DCS(defined as number of tasks finished per second) when we increase the number of local clients. We submit 64 tasks (each task taking 2 seconds to finish) with 1, 2, 4, and 8 local clients running. The left plot in Fig. 1 shows that task throughput increases nearly linearly. This is good news as it indicates adding more clients always improves performance. Next, we take a look at throughput with clients running on HTCondor. We submit 128 tasks (each task taking around 5 seconds) with 4, 8, 16, 32, and 64 HTCondor clients running. The right plot in Fig. 1 shows very similar results, with throughput increasing almost linearly as well. For comparison sakes, we also show the task throughput when we submit all 128 tasks directly through HTCondor without DCS and have then all be executed simultaneously. As we can see, the task throughput when only using HTCondor is exceeded by DCS with just 32 clients; this is due to the delay and overhead when starting jobs on HTCondor.

4.2 Scaling vs Number of Tasks

In our second experiment, we take a look at how running time of DCS increases when the number of running clients is fixed and the number of tasks submitted increases. In the left plot of Fig. 2, we submit anywhere from 1 to 256 tasks (each task taking 2 seconds to complete) and 4 local clients running. In the right plot of Fig. 2, we submit anywhere from 1 to 256 tasks (each task taking 5 seconds) with 32 HTCondor clients running. In both cases, the scaling is highly linear when the number of tasks submitted exceed the number of clients running. When the number of tasks is less than the number of clients, the total time it takes to finish all the tasks is much more noisy. This is due to the fact that we summit duplicates of the same task as part of the DCS's policy to improve reliability.

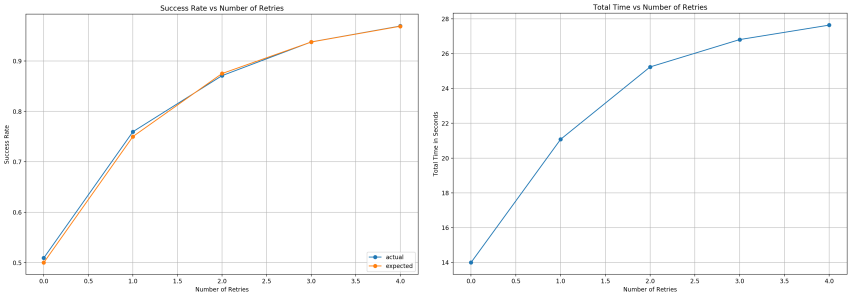

Figure 3: Left: The plot shows hows success rate increases as the number of retries increases. Right: The plot shows the total time to get back all results vs the number of retries. Both plots show mean values from 5 trials.

4.3 Success Rate vs Number of Retries

Finally, in our third experiment, we examine how the task success rate (defined as at least one successful result for a task) increases as the number of retries K increases as well. We run 500 tasks where each task has a 50 percent chance of failure and exits immediately with 4 clients running locally. The left plot of Fig. 3 shows that the actual success rate (blue line) is very much in line with the theoretical success rate (orange line) calculated earlier. Interestingly enough, the right plot shows the overall running time does not increase linearly with the number of retries. This is probably because most tasks do not need the maximum number of retries to achieve success and thus, only a small fraction of the tasks would suffer the worst case scenario.

5 Limitations and Future Work

As this is the first version of the DCS, there are several inherent limitations due to how it is built on top of HTCondor. For example, there are no real way to prevent clients from being killed or suspended by the HTCondor job scheduler. Similarly, depending on the system load and priority of the current user, the clients are not guaranteed to ever starting after being submitted as job to HTCondor.

Currently, only one server and multiple clients are supported, however future versions of the DCS will be able to support an arbitrary number of servers and clients. This is achieved by having a single submit queue which all the clients pull tasks from and a separate result queue for each server. Each submitted task will be identified by the server that submitted it; once the task is done, clients will put the results in the corresponding results queue of that server.

The first version of DCS also does not support passing environmental variables as part of the task or setting hardware requirements for the machines the clients are running on. Future versions will include this functionality. Furthermore, future versions will have an API that will allow tasks executing on the clients to report their progress back to the client. The clients in turn, will be able to report this progress back to the server. Progress information will be very helpful for any long running tasks.

6 Conclusion

In this report, we introduced a novel abstraction for task parallelism called the Distributed Completion Service. It not only provides a easy, simple to use API, but also additional reliability guarantees and mechanisms to hide and decrease the latency behind running tasks on HTCondor. This is only the first version of the DCS and we hope that subsequent versions will be open sourced and be widely used by a more general audience.

Acknowledgments

I would like to thank Chris Rossbach and Patrick MacAlpine for providing examples and tips on how to submit jobs using HTCondor.

References

[1] James Bergstra, Dan Yamins, and David D Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in Science Conference*, pages 13–20. Citeseer, 2013.

[2] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[5] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.

[6] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[7] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.

[8] Guido Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, page 36, 2007.