An update on the scalability limits of the Condor batch system

D Bradley[1], T St Clair[1], M Farrellee[1], Z Guo[1], M Livny[1], I Sfiligoi[2], T Tannenbaum[1]

[1]University of Wisconsin, Madison, WI, USA
[2]University of California, San Diego, La Jolla, CA, USA

E-mail: dan@hep.wisc.edu

Abstract.
Condor is being used extensively in the HEP environment. It is the batch system of choice for many compute farms, including several WLCG Tier 1s, Tier 2s and Tier 3s. It is also the building block of one of the Grid pilot infrastructures, namely glideinWMS. As with any software, Condor does not scale indefinitely with the number of users and/or the number of resources being handled. In this paper we are presenting the current observed scalability limits of both the latest production and the latest development release of Condor, and compare them with the limits reported in previous publications. A description of what changes were introduced to remove the previous scalability limits are also presented.

1. Overview

Condor is a batch system designed for high throughput computing. It is used extensively in HEP for management of computing tasks on dedicated compute farms and in resource scavenging environments. The scavenging mentality and support for strong security mechanisms made Condor a good fit for the requirements of glideinWMS, which is a workload management system used by HEP experiments and others wishing to utilize resources distributed between multiple institutions in grids and clouds [1].

The scalability of Condor is an important consideration for administrators of large and expanding compute farms, grid middleware providers who depend on Condor to integrate large collections of computers across multiple institutions, and, of course, end-users with increasingly large workflows to process. Because the HEP community provides an abundance of all of these, it has been a large driver in recent efforts to understand and advance the scale at which Condor can operate. Large industrial users and those who support them face some of the same issues of scale, and so they have also joined in this effort [2].

For the purposes of understanding this paper, a brief overview of the Condor architecture is helpful. A Condor pool is defined by a *collector*, which serves as a registry for the rest of the distributed daemons in the pool. Each daemon is described by a *ClassAd* record, which the daemon periodically sends to the collector to advertise its presence and current status in the pool. Job execution nodes in the pool are represented by a *startd*, which is responsible for carrying out job execution requests. The startd divides the machine into one or more logical subdivisions called execution *slots*. A collection of jobs that have been submitted by users is

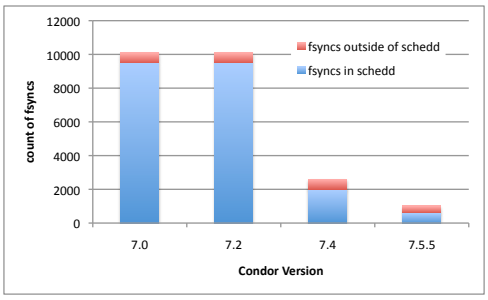

Figure 1. Fsync operations performed by Condor to run a sample workflow: 100 vanilla universe jobs were submitted in clusters of 10; they ran for 20 minutes, were evicted, restarted and ran to completion after an hour. Each job cluster had its own event log. To illustrate fsync optimizations that have been made in response to real-life HEP usage, the jobs each set match_list_length=5 and have five attributes with $$() references.

maintained by a *schedd*. The *schedd* obtains a lease to run jobs on an execution slot. The lease is obtained from the *negotiator*, which is the daemon responsible for pool-wide user priority management and for matchmaking. Matchmaking involves finding compatible execution slots for the resource requests (in the form of ClassAds) the schedd makes on behalf of the users.

2. Schedd Scalability

The schedd contains the queue of jobs submitted by users and it is responsible for communicating with other components in the Condor pool to run the jobs. Most job management commands available to users are clients that communicate with the schedd to get it to operate on the jobs—operations such as submitting, modifying, examining, and aborting jobs. In addition, automated time- and event-based job management policies specified by the user or administrator are largely the responsibility of the schedd to enforce.

A condor pool may contain more than one schedd, and these may each run on a separate computer if desired, so one can scale the number of jobs being managed by simply adding more instances of the schedd. However, for manageability and ease of use, it is desirable to have a small number of schedds. It is also desirable for a single instance of the schedd to be able to handle bursts in activity without seriously degraded performance or reliability, because current methods of provisioning additional schedds and load balancing across them are not instantaneous. For many users of Condor, instantiating another schedd involves numerous manual steps, including purchase of new hardware and configuring software that interacts with the schedd. Scalability of each instance of the schedd is therefore an important concern.

2.1. Disk Transactions

Interaction with the disk is one limiting factor for the schedd. The status of every job is logged to files on disk in a transactionally safe (ACID) way so that workload managers can rely on the semantics of queued jobs:

(i) the same job will not run in multiple instances at the same time

(ii) once submitted, a job will not disappear from the queue without at least one record in the job's event log indicating that it finished or was explicitly cancelled.

These guarantees are intended to hold in the face of system crashes and restarts, as long as data that has been synced to disk is not corrupted or lost.

Because of the semantic guarantees about job logging, transactions requiring durability are synced to the disk. Frequently syncing transactions to disk can be quite expensive, so it is not surprising that in many situations this has been found to be a limiting factor for scalability of the schedd. Reducing the number of separate transactions that require durability has therefore been a productive area of work.

As shown in Figure 1, there has been recent progress in reducing the number of fsync operations. In some cases, this was accomplished by combining separate transactions into one.

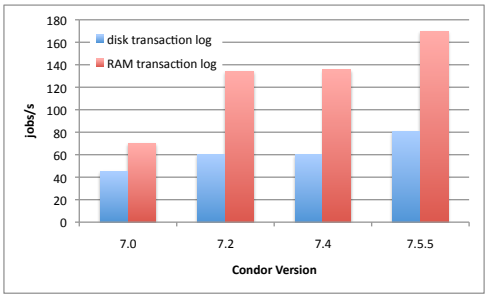

Figure 2. A "full throttle" test of the schedd that compares maximum sustained job completion rates. The schedd was started with a queue of 120,000 sub-second jobs, and it was tasked with attempting to keep 500 running at a time. In one case, the job state was logged to a 7200 RPM disk. In the other case, the job state was logged to system RAM (tmpfs). The simple high-frequency workflow used here differs from the HEP-inspired workflow in Figure 1, so the performance when disk-bound does not directly compare.

In other cases, transactions were found that could be safely changed from durable to non-durable treatment without sacrificing job management guarantees.

What this means for scalability is that the rate at which the IO system can process separate transactions places a bound on the rate at which a single schedd can manage submission, running, and completion of jobs. This effect is demonstrated in Figure 2. Higher rates of job completion can be achieved when job state is kept on a RAM disk verses a 7200 RPM disk. Given jobs of a characteristic duration, the bound on IO rate limits the number of simultaneous job execution slots that the schedd can efficiently utilize.

Another conclusion is that any other activity that generates significant IO operations on the device used by the schedd to record job state can reduce the rate at which Condor operates. The reduction in rate can be quite non-linear, because when the schedd is not able to keep up with the rate of events being generated, timeouts in communication between the schedd and other components can cause failures that ultimately increase the number of transactions required per job, which just makes the situation even worse. Condor 7.5 contains optimizations that significantly reduce this vicious feedback loop, which is why job eviction was included in the test shown in Figure 1; eviction happens to generate many of the same behaviors that happen when responding to failures that occur when the schedd is overloaded.

Fortunately, flash SSDs are becoming more common in server-class systems. Since these devices can often support over ten times the rate of IO operations that traditional disks can, they are attractive solutions for storing job state.

2.2. Port Usage

In Condor 7.4 and earlier, network port usage placed a limit on the maximum number of simultaneously running jobs that a single schedd could manage. With more than 23k running jobs, the schedd machine ran out of network ports. There are 65k TCP ports per IP address. In Condor, each running job normally requires 2, occasionally 3 ports during its life—all associated with the IP address being used by Condor for communication. In addition, the rate at which ports are opened and closed leaves a fraction of ports in the TIME_WAIT state at any given time.

Condor 7.5 offers a new service called condor_shared_port, which accepts connections on behalf of all Condor daemons running on the same machine and passes incoming TCP connections to the intended recipients over named sockets [3]. This makes it easier to run the schedd behind a firewall. It also reduces by one the number of ports required per running job. In addition, if the Condor Connection Broker (CCB) is being used to connect from the schedd machine to the job execution machines, the use of shared_port on the schedd machine eliminates one more port per running job. This mode of CCB usage is the norm in glideinWMS, where the job execution machines are often in private networks that can only be accessed from the outside

by using CCB. The extra reduction in port usage in this mode of operation is due to the change in direction of a TCP connection: instead of connecting from an ephemeral port on the schedd machine to a daemon port on the job execution machine, it connects from an ephemeral port on the job execution machine to the shared daemon port on the schedd machine.

So in Condor 7.5, it is possible to eliminate all port usage for running jobs except for one shared port to accept incoming connections and transient ports that are needed for communication between the schedd and sub-processes it creates to manage running jobs (condor_shadow). These are ephemeral ports only used for a fraction of a second for communication internal to the schedd machine when jobs are having their status updated. The main concern with these ports is that at high rates of communication, ports that have been recently used are unavailable while in the TCP TIME_WAIT state. This leads to temporary port exhaustion at sufficiently high rates of job completion. In Linux, with a typical system ephemeral port range of 28k, this limits the maximum job completion rate to 90-240 Hz, depending on details of file transfer usage. The ephemeral port range can be extended to at most 64k, which allows a maximum job completion rate of 220-550 Hz under a single schedd, given present communication patterns.

Tests of Condor 7.5.5 confirm that the previous limit of 23k running jobs can be exceeded. With glideinWMS, a peak of 49k running jobs and sustained operation at 41k running jobs under a single schedd has been demonstrated. In pre-releases of 7.5.5, occasional episodes of system instability at scales of over 30k running jobs were encountered, requiring the machine to be rebooted. This was caused by contention for a lock file used by the many shadow processes created by the schedd—a problem which was addressed before the final release of 7.5.5 by sharply reducing the frequency of locking.

2.3. Memory Usage

Availability of memory on the schedd machine limits the maximum number of queued and running jobs that are possible under a single schedd. The entire job queue is stored in memory as a collection of ClassAds. In 64-bit Condor 7.4.4 and 7.5.5, a minimum of 30 kbytes per queued job cluster is required—more if the job description contains a lot of data such as a large environment specification. Jobs with similar descriptions may be submitted in clusters to save memory. 32-bit Condor requires 30% less memory, but the schedd is then limited to an address space of 4GB.

Running jobs require significantly more memory, because a sub-process, condor_shadow, is created for each running job. At the time this paper was presented, using a pre-release of 7.5.5, a system with 32GB of memory was able to support 27k running jobs using 64-bit Condor and 37k running jobs using 32-bit. Since that time, as 7.5.5 nears a public release, the memory used by the shadow has been reduced by 30%, making it equivalent to the memory requirements of 7.4.4.

3. Matchmaking Scalability

Figure 3 compares speed of matchmaking in Condor 7.4 and 7.5. This task is dominated by CPU usage in evaluating ClassAd expressions. These two versions of Condor use different implementations of the ClassAd library. The "new" implementation used in 7.5 is the ClassAd library that has been released to the public for many years but which was not used in Condor itself until now due to the difficulty of converting to a different API. This new implementation was actually significantly slower at matchmaking, but after its adoption by Condor, a number of optimizations were made to speed it up, with the result that it is about 20% faster than matchmaking with old ClassAds.

Matchmaking involves pair-wise comparison of jobs and all candidate machine ClassAds. The compatible machines are then sorted according to the job owner's and pool administrator's

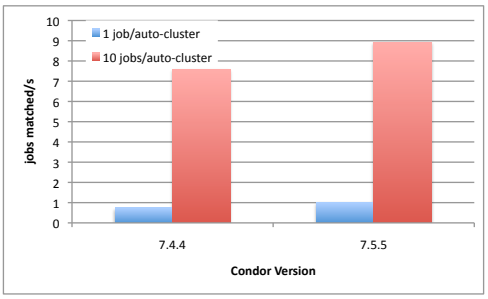

Figure 3. Average number of jobs matched per second while matching 1000 jobs to a pool of 10,000 slots. Higher rates are possible when jobs can be grouped into equivalent sets (auto-clusters).

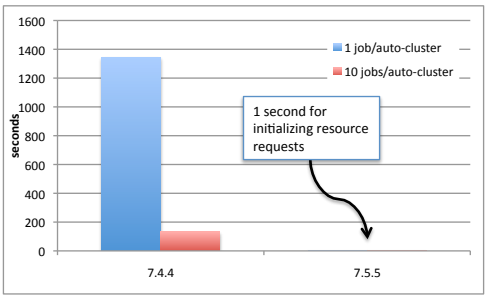

Figure 4. Time spent in the schedd while matching 1000 jobs to a pool of 10,000 slots. Asynchronous matchmaking in 7.5.5 significantly improves responsiveness and scalability of the schedd, because this allows the schedd to do other work while waiting for a response from the negotiator.

preferences. Since it is common for many jobs to be identical or nearly identical for matchmaking purposes, caching of results is done to avoid repeatedly re-evaluating the same search. Jobs are grouped into *auto-clusters* and matchmaking results are cached for each auto-cluster.

Note that the average rate at which jobs may start is not necessarily bounded by the matchmaking rate. When a user is expanding the number of running jobs, each new job that begins running must be matched to a new machine, so startup is bounded by the rate of matchmaking. However, once an existing job completes, rather than going through matchmaking to reclaim the machine, the schedd may choose another job to run on the same machine until the user's lease of the machine is taken away. When there are many idle machines or when preemptive resource reallocation is enabled, matchmaking tends to be expensive, because many potential matches must be evaluated. In such cases, the schedd can typically sustain a much higher rate of job completion than the matchmaking rate. For example, the rates reported in Figure 2 would have been much lower if each new job that ran went through the full matchmaking process.

Until 7.5.5, slow matchmaking could result in the schedd failing to efficiently utilize machines already allocated to it. This was due to time spent idling in the schedd while waiting for matchmaking results. Figure 4 shows the reduced matchmaking cost to the schedd that resulted from the implementation of asynchronous matchmaking requests. In numerous occasions, we have observed matchmaking wait-time cause badly degraded performance for busy schedds in HEP environments. We are pleased to report that for the Wisconsin CMS Tier-2 computing center, upgrading to a pre-release of Condor 7.5.5 turned this chronic problem into a non-issue.

4. Collector Scalability

In the case of glideinWMS where communication is across untrusted and sometimes high-latency networks, it is necessary to deploy the collector in a two-tier arrangement in order to scale to large securely-authenticated pools [4]. The daemons in the pool each report to one of the collectors in the bottom tier, and a single top-level collector aggregates the information from the bottom tier. Figure 5 shows a test that successfully operated this scheme at a scale of 90k execution slots without problems other than a temporary setback caused by the default 1024 file descriptor limit preventing the bottom tier collectors from supporting more CCB clients.

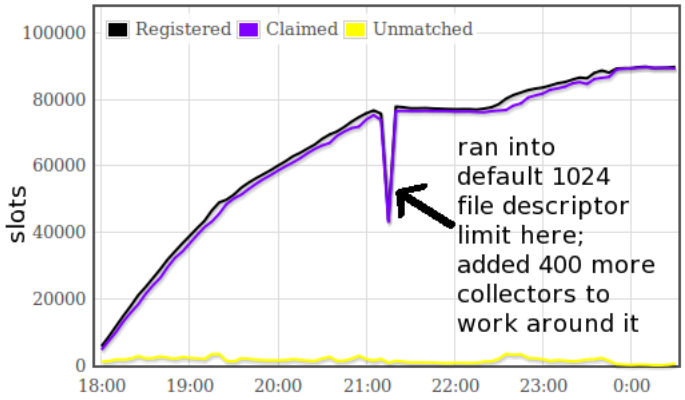

Figure 5. A glidein pool of 90k GSI authenticated execution slots using a two-tier collector configuration. 800 collectors were used in the bottom tier, which also provided CCB service. 5 schedds were used to keep the pool busy. The collectors and negotiator ran on a single machine and used 16G of memory.

5. Conclusion

Condor continues to evolve in response to the demands of today's large-scale computing problems. Key measures of performance demonstrate recent improvement in handling workflows of O(100,000) jobs, pools of 10-50 thousand execution slots, and job throughput rates around 10-100 Hz. As a demonstration of the overall improvement, Figure 6 shows the increasing level of throughput that has been achieved over the years in tests of glideinWMS, using configurations and workflows similar to real HEP usage.

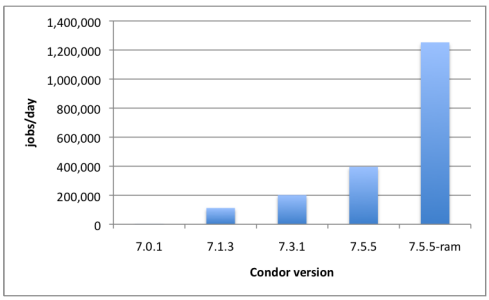

Figure 6. Sustained job execution rate in jobs/day achieved in trials of glideinWMS using a Condor configuration suitable for secure operation across global networks. These trials, performed in different years [4], were not done on identical hardware or in identical conditions, but they give a flavor of the expanding capabilities of the whole system over time. Job durations were 30 minutes or more. The 7.5.5-ram test was done with job state logged to a RAM disk; all others logged to a traditional disk.

Acknowledgments

This work was partially funded by the U.S. National Science Foundation grant number 0621704 (Condor), PHY-0533280 (DISUN), PHY-0612805 (CMS Maintenance & Operations), OCI-0943725 (STCI), and the US Department of Energy grant DE-FC02-06ER41436 subcontract 647F290 (OSG). Many thanks to the FNAL CMS Tier 1 team for providing the test hardware.

References

[1] Sfiligoi I, 2008 glideinWMS—a generic pilot-based Workload Management System, *J. Phys.: Conf. Series* **119** 062044

[2] Wagner M and Trieloff C, 2010 Red Hat Enterprise MRG messaging performance seminar *Summit JBoss World.*

[3] Tannenbaum T, 2010 What's new in Condor? What's coming up? *Condor Week 2010.*

[4] Bradley D, Sfiligoi I, Padhi S, Frey J and Tannenbaum T, 2009 Scalability and interoperability within glideinWMS, *J. Phys.: Conf. Series* **219** 062036