Condor

By Douglas Thain, Jim Basney, and Miron Livny
Computer Sciences Department
University of Wisconsin

Condor is a system for performing high-throughput computing on a community of workstations.

High-throughput computing concerns maximizing the number of computations performed over a long period of time. Many computer users, ranging from scientists performing simulations to artists rendering images, are constrained by how much computation can be performed before a deadline. Such users measure computer performance in terms such as "simulations per year" or "video frames per month." High-throughput computing is not the same as high-performance computing, which seeks to maximize computation speed over short time frames and is measured in figures such as MIPS or FLOPS.

A community of workstations can be almost any collection of workstations that agree to work together. A community might be a cluster of identical machines in a single room, or it might be a loose collection of decentralized, distributed, heterogeneous, and non-dedicated machines. A good example of such a community is the set of personal workstations that sit on the desks of workers in an office building. This community is decentralized because each user is in command of his or her workstation – no central authority directs the minute-to-minute operation of any workstation. It is also physically distributed – some machines may be on the same network in the same room, while others may be on slow networks or separated by wide oceans. Such a community is more than likely to be heterogeneous. Because each workstation is specialized to its owner's needs, each workstation is likely to have a different CPU, a different amount of memory and disk, and even different software and operating systems. Finally, no single workstation is dedicated solely to serve the community. When the owner of a workstation steps out of the office, its resources become available to the community, but when the owner returns, the workstation must be released for his or her use.

A significant amount of throughput can be extracted from such a community [1]. A typical office workstation is left idle about 75 percent of the day [2]. This means the large majority of computing purchased for individual use goes to waste. Condor scavenges for these wasted CPUs and puts them to good use. Cycles scavenged from a community of inexpensive medium-performance workstations can often provide better throughput at a lower price than a single high-performance computer.

A community of workstations managed by Condor is known as a *pool*. A pool has a *central manager* that collects information about participating machines and the jobs to be run. Each machine in a pool periodically updates the central manager with the jobs it wishes to run and its availability to run other's jobs. A *matchmaker* runs on the central manager and pairs jobs with idle machines. Several Condor pools can be tied together in a *flock*, allowing jobs to migrate between pools.

The Condor project has grown steadily since its inception in 1985 at the University of Wisconsin. It was initially known as Remote Unix [3] and tied together about one hundred workstations in the Computer Science. From there, Condor pools were established at many institutions around the world. In 1993, 250 machines in five countries were linked together to form the first international flock of Condor pools [4]. By the year 2000, over 600 CPUs were available in the Wisconsin Condor pool alone. In that same year, several Condor pools and Globus resources (see Globus) provided surplus cycles to solve the NUG-30 optimization problem [5]. Over the course of one week, Condor supplied 11 CPU-years of surplus cycles. The largest number of CPUs applied to the problem at one time was 1009. The Condor project continues to study and support high-throughput computing systems and users today.

Opportunism

A key design principle of Condor is that a workstation must always be under the full control of its owner. For example, an owner may insist that a workstation only be used during certain times of the day, and then only by certain people. If an owner should return to his or her workstation and begin typing or working, Condor will suspend or migrate any running jobs in order to give the owner full use of the computer. Condor always defers to the owner's wishes, even at the loss of some throughput, for one simple reason: if the owner is unhappy due to load imposed by Condor, he or she will likely decline to donate any further cycles, thus dropping the throughput in the long run.

Because Condor must only use what resources are available at the moment, it is called an *opportunistic* computing system. Clearly, an office full of personal workstations requires an opportunistic system to deal with its constant changes. However, opportunism is also needed to provide high throughput on other systems. Even a professionally managed microprocessor cluster will suffer breakdowns, power failures, network partitions, full disks, and other troubles. As clusters increase in size, the mean time between failures decreases until a failure of some kind will always be present. An opportunistic system is needed to maintain high throughput over long time scales.

Condor provides three key mechanisms to achieve high throughput on opportunistic systems: matchmaking, checkpointing, and split execution.

Matchmaking

The Condor ClassAd Matchmaking framework [6] provides the flexible scheduling services required in a distributed and heterogeneous environment. The framework allows machine owners to define custom policies for how their machines can be used and allows job owners to specify the types of machines on which their jobs should run. Unlike other load-sharing systems, Condor jobs are not submitted into particular queues. Each job and each machine specify their exact needs and preferences, and Condor matches them together.

Condor maintains a classified advertisement (ClassAd) for each machine and job in the system. ClassAds are analogous to classified advertisements in a newspaper. They describe jobs in search of machines and machines in search of jobs. Each ClassAd contains a set of named attribute definitions written in the ClassAd language that describe the machine or job and specify compatibility between them. Attributes may be simple integer, real, or string constants, or they may be complex expressions constructed with arithmetic and logical operators. ClassAds have no fixed schema, so machines and jobs can be described as deemed appropriate by their respective owners.

Each ClassAd contains a Boolean Requirement expression and a numerical Rank expression that refer to both machine and job attributes to indicate compatibility. The Condor matchmaker searches for compatible machines and jobs, as defined by these expressions. If, when a machine is paired with a job, both the job and machine's Requirements expressions evaluate to true, they are compatible and Condor can run the job on that machine. The matchmaker searches for matches that result in the highest values for the Rank expressions, to satisfy the machine and job owners' preferences. Once the matchmaker decides on a set of matches, the matchmaker notifies the processes responsible for managing the matched machines and jobs. These processes then communicate directly to first verify that they are willing to proceed (since system conditions may have changed), and if they are willing, to start executing the job on the machine.

Figures 1 and 2 present example machine and job ClassAds. In Figure 1, the Requirements attribute specifies that the machine will only start a Condor job when its CPU load average is below 0.3 and there has been no keyboard activity for over 900 seconds. The Rank attribute specifies that the machine prefers to run a job owned by jbasney@cs.wisc.edu. In Figure 2, the Requirements attribute specifies that the job will only run on machines with a Intel-compatible CPU running the Linux operating system with 64 MB of memory or greater. The Rank attribute specifies that the job prefers to run on the machine with the greatest memory.

Figure 1. Example Machine ClassAd

```
[
   Arch = "Intel";
   OpSys = "Linux";
   Memory = 128;
   Disk = 2810834;
   KeyboardIdle = 1254;
   LoadAvg = 0.02;
   Requirements =
      (LoadAvg < 0.3) &&
      (KeyboardIdle > 900);
   Rank =
      (Owner=="jbasney@cs.wisc.edu")
      ? 1 : 0;
]
```

Figure 2. Example Job ClassAd

```
[
   Owner = "thain@cs.wisc.edu";
   Requirements =
      (Arch == "Intel") &&
      (OpSys == "Linux") &&
      (Memory >= 64);
   Rank = Memory;
]
```

Checkpointing

Condor provides a checkpointing (see Checkpointing) service to executables linked with the Condor checkpointing library [7]. Condor uses checkpointing to guarantee forward progress of jobs executing on non-dedicated machines. The machine owner's policy determines when Condor can run jobs on that machine. When the owner reclaims the machine, Condor must preempt any jobs running on it. Condor can checkpoint the job and preempt it without losing the work it has already accomplished. Condor can then resume the job from the checkpoint when it allocates a new machine to the job. Checkpointing also allows the Condor scheduler to preempt lower priority jobs and run higher priority jobs in their place as necessary without losing the work the preempted jobs have accomplished so far. Additionally, Condor periodically checkpoints jobs to provide fault tolerance. After any interruption in service, the job can always continue from its most recent checkpoint.

Since the entire memory state of a job is saved in the checkpoint, transferring and storing checkpoints often requires significant network and disk capacity. In an opportunistic environment, it is imperative that a preempted process vacates the machine quickly, to maintain the goodwill of workstation owners. Condor stores checkpoints on checkpoint servers, which have disk space allocated for this purpose. Condor pools are typically configured so jobs checkpoint to the nearest checkpoint server on the network. This helps to distribute the checkpointing load across the network, improving checkpointing speed and reducing the load on backbone or wide-area network links. Additionally, when jobs are preempted, they can request in their ClassAd Requirements or Rank attribute that they be re-scheduled on a machine close to the checkpoint server where they wrote their last checkpoint, so they can quickly restore their checkpointed state and resume computation on the new machine.

Split Execution

Although Condor is capable of distributing a user's jobs to machines spread around the world, not all machines have the correct configuration to run a particular job. For example, the needed input files may be missing, there may not be sufficient space to store output files, or the machine may not be capable of making the necessary network connections. Such a machine is said to be unfriendly to the job.

Condor uses split execution to create a friendly execution environment on an unfriendly machine. In this model, two pieces of software work together to support the job. On the unfriendly machine, an agent process traps some of the job's procedure calls and converts them into remote procedure calls. (See also Remote Procedure Calls.) These RPCs are sent over the network to a shadow process executing on the user's home machine. The shadow process executes the system calls on behalf of the job and sends the results back. In this way, the remote CPU is used for the bulk of the job's computation, but operations that depend on the home environment are sent to where they can be performed correctly.

Not all of the job's system calls need to be trapped. For example, system calls that expand and contract the job's memory space must be executed at the unfriendly machine where the memory is actually in use. System calls that deal with files are generally trapped, but they are only sent to the shadow if the file in question is not stored on a shared filesystem available at the executing machine. The agent is responsible for tracking which files have been opened on the executing machine and which files have been opened on the home machine.

The "trapping" of system calls is accomplished by requiring the user to link his or her jobs with a specialized Condor library that implements the agent. This library simply gives new definitions for the system calls that must be trapped. It is linked with the job before the standard libraries and thus overrides the standard definitions. The same library implements the checkpointing mechanism

Not all programs can take advantage of this special library.   Because the program must be re-linked with the Condor library, programs that are distributed only in executable form cannot take advantage of split execution. Due to the single-threaded nature of the RPC link between the agent and the shadow, multi-process and multi-threaded programs cannot be used. Other complicated features such as file locking and memory-mapped file access are not supported.

Programs that cannot be linked with this library can still be used within Condor. They may take advantage of the matchmaking and opportunistic computing features, but will execute only on friendly machines. Without the Condor library, they cannot checkpoint and will be restarted from the beginning if a migration is necessary.

Conclusion

High-throughput computing is closely related to reliable computing. (See Reliability of Distributed Systems.) An opportunistic system designed to deal with unplanned events such as workstation owners reclaiming their machines is equally capable of continuing in the face of hardware and software failures. Even tightly controlled systems, such as microprocessor clusters and multiprocessor supercomputers, suffer downtime from incremental hardware and software failures. These systems are equally in need of opportunistic software. Condor is capable of improving the reliability and throughput of systems ranging from office computers to supercomputers.

References

[1] M. Litzkow, M. Livny, and M. W. Mutka, *Condor - A Hunter of Idle Workstations*, Proceedings of the 8th International Conference of Distributed Computing Systems, 104-111, 1988.

[2] M. Mutka and M. Livny, *The Available Capacity of a Privately Owned Workstation Environment*, Performance Evaluation, 12(4), 269-284, 1991.

[3] M. Litzkow, *Remote Unix: Turning Idle Workstations Into Cycle Servers*, Proceedings of the 1997 Usenix Summer Conference, 381-384, 1987.

[4] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne, *A Worldwide Flock of Condors: Load Sharing among Workstation Clusters,* Journal on Future Generations of Computer Systems, 12(1), 53-65, 1996.

[5] *NetWatch: Milestone for Linked Computers*, Science, 289(5479), 503, 2000.

[6] R. Raman, M. Livny, and M. Solomon, *Matchmaking: Distributed Resource Management for High Throughput Computing,* Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, 140-146, 1998.

[7] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, University of Wisconsin-Madison Computer Sciences Technical Report #1346, 1997.

Cross Reference

Checkpointing see Condor
ClassAds see Condor
Cluster Computing see Condor
Flocking see Condor
Heterogeneous Computing see Condor
Reliability of Distributed Systems see Condor
Remote Unix see Condor
Scheduling, Distributed see Condor

Dictionary Terms

High-performance computing. The study of executing a single program in as short a time as possible.
See Condor.
See High-throughput Computing.

High-throughput computing.   The study of executing as many programs as possible over a long period of time.
See Condor.
See High-performance Computing.

Load sharing. The process of distributing programs from a heavily loaded machine to less loaded machines.
See Condor
See Migration

Matchmaking. The process of matching requests for a computing resource with offers for a computing resource.
See Condor.

Migration. The process of moving an executing program from one CPU to another without affecting the computation in progress. Migration can be implemented using checkpointing. Migration is sometimes performed for the sake of load balancing.
See Checkpointing.
See Condor.
See Load Balancing

Opportunistic computing. A computing model where a system takes advantage of resources as they become available, instead of scheduling them in advance.
See Condor.

Split execution. A model for providing a friendly execution environment on an unfriendly machine. In this model, an agent process traps a program's system calls and sends them via RPC to a shadow process on another machine to be executed. This allows a migrating program to execute exactly as if it were on its home machine, regardless of where it is actually executing.
See Condor.
See Migration.
See Remote Procedure Calls.