Mechanisms for High Throughput Computing

M. Livny, J. Basney, R. Raman, and T. Tannenbaum,
Department of Computer Sciences, University of Wisconsin–Madison
1210 W. Dayton St., Madison, Wisconsin 53706
Tel: (608)262-6611, Fax: (608)262-9777
{miron,jbasney,raman,tannenba}@cs.wisc.edu

May 9, 1997

1 Introduction

For many experimental scientists, scientific progress and quality of research are strongly linked to computing throughput. In other words, most scientists are concerned with how many floating point operations per week or per month they can extract from their computing environment rather than the number of such operations the environment can provide them per second or minute. Floating point operations per second (**FLOPS**) has been the yardstick used by most High Performance Computing (**HPC**) efforts to rank their systems. Little attention has been devoted by the computing community to environments that can deliver large amounts of processing capacity over very long periods of time. We refer to such environments as *High Throughput Computing* (**HTC**) environments.

The key to HTC is effective management and exploitation of all available computing resources. Since the computing needs of most scientists can be satisfied these days by commodity CPUs and memory, high efficiency is not playing a major role in a HTC environment. The main challenge a HTC environment faces is how to maximize the amount of resources accessible to its customers. Distributed ownership of computing resources is the major obstacle such an environment has to overcome in order to expand the pool of resources from which it can draw upon. Recent trends in the cost/performance ratio of computer hardware have placed the control (ownership) of powerful computing resources in the hands of individuals and small groups. These distributed owners would be willing to include their resources in a HTC environment only after they are assured that their needs are addressed and that their rights will be protected.

For more than a decade, the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison has been developing, implementing, deploying, maintaining, and supporting HTC software tools that can effectively harness the capacity of hundreds of distributively owned UNIX workstations. The Condor High Throughput Computing environment developed by the team was the first to address the challenges introduced by distributed ownership of computing resources. The satisfaction of computing resource owners is fundamental to the success of Condor. By addressing the needs and sensitivities of these owners, Condor enables scientists and engineers to simultaneously and transparently exploit the capacity of workstations they might not even know exist. These workstations can be scattered throughout the globe and may be owned by different individuals, groups, departments, or institutions.

From the outset, the sociological aspects of High Throughput Computing have been playing a major role in the design and implementation of Condor. Condor's mechanisms and policies enable each and every resource owner in the environment to control when, to what extent, and by whom, the resources of his or her workstation can be used as a HTC resource. In this paper we present three mechanisms employed by Condor to provide HTC services to its customers - the *Classified Advertisement* (ClassAd) mechanism, the *Remote System Calls* mechanism, and the *Checkpointing* mechanism. We view these three mechanisms as holding the key to the success of Condor in harnessing the capacity of large collections of distributively owned resources. ClassAds enable Condor to pair Resource Requests and Resource Offers, while Remote System Calls enable Condor to allocate resources across administrative domains. The Checkpointing mechanism enables Condor to revoke resources that must be freed due to owners' constraints and to resume the application from where it left off on another resource.

Condor was first installed as a production system in our Computer Sciences department over ten years ago. This Condor pool has since served as a major source of computing cycles to both faculty and students in our department. For many, it has revolutionized the role computing plays in their research. An increase in one, and sometimes even two, orders of magnitude in the computing throughput of a research project can have a profound impact on its size, complexity, and scope. In some cases, the almost unlimited amount of computing resources provided by the Condor pool enabled the exploration of more risky research directions. Over the years, the Condor Team has established collaborations with scientists from around the world and has provided them with access to surplus cycles (one of whom has consumed 50 CPU years). Today, our pool consists of more than 300 desk-top UNIX workstations. On a typical day, the pool delivers more than 180 CPU days.

2 The ClassAd Mechanism

Match-making is the cornerstone of a HTC environment. It is the means by which *Resource Requests* and *Resource Offers* that satisfy each other are identified and paired together. The Classified Advertisement (ClassAd) mechanism of Condor is based on a flexible and expressive framework for match-making between entities. The ClassAd mechanism does not constrain which entities take part in the match-making process, how they wish to describe themselves, or what it takes to satisfy their requirements. Instead, the ClassAd framework rigorously defines a mechanism which arbitrary entities can use to describe themselves and their requirements. This approach facilitates semantic-free and extremely flexible match-making. To see why this is true, consider the elements that constitute a ClassAd.

A ClassAd is an aggregate of the following information:

1. Type information This is a tuple of two types, *self* type and *match* type, which characterizes the kinds of entities involved in the match.

2. Attribute List This is a list of arbitrary attributes which describe the properties of the advertising entity.

3. Requirement This is an expression which represents the constraints that the advertising entity places on the match ad.

4. Rank This is an expression which represents the advertising entity's assessment of the quality of a match.

5. Advertisement ID If an entity publishes several ads and a match is found, the match can be qualified with an advertisement ID of the matched ad so that the advertising entity can identify which ad was matched.

Two ClassAds **A** and **B** are said to match if and only if:

1. **A**'s self type is equal to B's match type and **B**'s self type is equal to **A**'s match type.
2. **A**'s requirement evaluates to TRUE with respect to **B**, and **B**'s requirement evaluates to TRUE with respect to **A**.

It is important to note the following:

1. All information required for matching two ClassAds is available in the ClassAds themselves. Thus, they are self-describing structures requiring no external policies or mechanisms.
2. The ClassAd mechanism does not specify (and thus possibly constrain):
   (a) which entities publish a ClassAd,
   (b) what information may be placed in a ClassAd,
   (c) who performs the match-making, or
   (d) what actions are to be performed in case of a match.
3. The use of expressions on boolean, integer, real and string values allow unambiguous and flexible expression of attributes and constraints.

These characteristics facilitate the light-weight and semantic-free properties of a ClassAd.

2.1 ClassAds in Condor

This section describes how ClassAds are used in the Condor HTC environment. The Condor "kernel" is composed of the following three entities: 1) Startds, which represent resources, 2) Schedds, which represent customers, and 3) The Central Manager, which performs match-making services.

Startds represent resources in a Condor pool. These resources are usually UNIX workstations. The startd periodically extracts the resource's state information and encapsulates this information in a Resource Offer ClassAd. Additionally, the resource owner's policy regarding when and who can use the workstation is also included in the ClassAd. This ClassAd is periodically refreshed with the current state of the workstation and sent to the central manager.

Schedds represent the customers of the Condor pool. Users submit their applications to their personal schedd, which then enqueues these requests in an application queue. The application queue is a list of ClassAds which are ordered by a priority scheme — each ClassAd represents the attributes and requirements of one application.

Periodically, the Central Manager (or CM) of the pool enters a negotiation cycle. At this time, schedds in the pool are contacted by the CM in priority order, and ClassAds from their application queues are extracted. These Resource Request ClassAds are then matched against the Resource Offer ClassAds sent in by the startds to determine if there is a match. If a match is found, the CM informs the schedd and startd of the match which in turn initiate protocols to place the application on the remote machine which can run it.

It is important to note that this model can be generalized to include resources other than workstations and customers other than applications. The main reason for this flexibility is the ClassAd mechanism itself which is the vehicle for bringing resources and customers together.

3 Remote System Calls

With the help of the ClassAd mechanism, Condor harnesses CPU and memory resources by placing applications on remote workstations that are capable and willing to serve these applications. One hurdle to overcome when placing an application on a remote execution workstation is data access. In other words, in order to utilize the remote resources, the application has to be able to read from and write to files on its submit workstation. A requirement that the remote execution machine be able to access these files via NFS, AFS, or any other network file system may significantly limit the number of eligible workstations and therefore hinder the ability of a HTC environment to achieve high throughput. Therefore, in order to maximize throughput, Condor strives to be able to run any application on any remote workstation of a given CPU architecture without relying upon a common administrative workstation setup. The enabling technology that permits this is Condor's Remote System Calls mechanism. In fact, due to this mechanism, Condor does not even require a user to posses a login account on the execute workstation.

When a UNIX process needs to access a file, it calls a file I/O system function such as *open()*, *read()*, *write()*, etc. These functions are typically handled by the standard C library which consists primarily of stubs that simply generate a corresponding system call to the local kernel. Condor users can link their applications with an additional enhancement to the standard C library called the *condor_syscall_lib* library. This library does not duplicate any code in the standard C library; instead, it augments certain system call stubs (such as the ones which handle file I/O) into remote system call stubs. The remote system call stubs package the system call number and arguments into a message which is sent over the network to a "shadow" process that runs on the submit workstation. Whenever Condor is executing an application, it also runs a shadow process on the initiating host where the user originally submitted the application. This shadow process acts as an agent for the remotely executing program in performing system calls. The shadow, which is linked with the standard C library, then executes the system call on behalf of the remotely running application in the normal way. The shadow then packages up the results of the system call in a message and sends it back to the remote system call stub in the condor_syscall_lib on the remote machine. The remote system call stub then returns its result to the calling procedure which is unaware that the call was done remotely rather than locally. In this fashion, calls in the user's program to *open()*, *read()*, *write()*, *close()*, and all other file I/O calls transparently take place on the machine which submitted the application instead of the remote execution machine.

In Condor, the shadow process not only acts as an agent for performing remote system call operations but also acts as a manager of policy over the application, directing how the application runs according to administrator or security policies. This allows Condor to optimize certain remote file I/O operations. For example, consider the case of an user submitting an application from Machine **A**, and the application is currently running on Machine **B**. Under Condor, Machine **B** can access any data files available to the user on Machine **A**. But suppose that some large data files needed by the application are accessed by Machine **A** via a network file-system from a NFS (or AFS) file server. In this case, the data files would have to make two hops over the network: one to Machine **A** via NFS (or AFS), and then a second hop via Condor's remote system call mechanism to Machine **B**. In order to avoid this type of situation, the stub for the open() system call in the condor_syscall_lib linked with the user's application first contacts the shadow to ask "how shall I open this file?". The shadow then compares Condor setup and configuration information on both machines (such as if AFS/NFS is configured, if both machines share the same /etc/passwd file information, etc.) and informs the condor_syscall_lib if it should open the file directly via a local system call (and therefore via NFS or AFS) or via a remote system call.

By handling file I/O via remote system calls, Condor is able to run applications linked with condor_syscall_lib on the remote execution machine as user "nobody" which has extremely limited access permissions, and/or in a protected subdirectory (such as via the chroot() system call). Furthermore, the user does not need to even posses a login on every machine in the pool, as only the shadow process on the submission machine needs to run with the user's file access permissions. Finally, the shadow process can be instructed to keep copious log files detailing all file I/O activity.

4 The Checkpointing Mechanism

Checkpointing an executing program is taking a snapshot of its current state in such a way that the program can be restarted from that state at a later time. Checkpointing gives a scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. If the scheduler decides to no longer allocate a machine to an application (for example, when the owner of that machine returns and reclaims it), the scheduler can checkpoint the application and preempt it without losing the work the application has already accomplished. The application can then be resumed later when the scheduler allocates it a new machine. Additionally, periodic checkpointing provides fault tolerance. Snapshots can be taken periodically, and after an interruption in service the program can continue from the most recent snapshot.

Due to the opportunistic nature of resources in a distributively owned environment, any attempt to deliver HTC has to rely on a checkpointing mechanism. Therefore, Condor employs such a mechanism to provides checkpointing services to single process applications and PVM applications on almost all major UNIX platforms. To enable checkpointing, the user must link the program with the Condor system call library (condor_syscall_lib). This means that the user must have the object files or source code of the program to benefit from the Condor checkpointing mechanism. However, the checkpointing services provided by Condor are strictly optional. So, while there are some classes of applications for which Condor does not provide checkpointing services (or applications that rely on their own checkpointing mechanism), these application may still be submitted to Condor to take advantage of Condor's other resource management services.

Process checkpointing is implemented in the Condor system call library as a signal handler. When Condor sends a checkpoint signal to a process linked with this library, the provided signal handler writes the state of the process out to a file or a network socket. This state includes the contents of the process's stack and data segments, all shared library code and data mapped into the process's address space, all CPU state including register values, the state of all open files, and any signal handlers and pending signals. On restart, the process reads this state from the file or network socket, restoring the stack, shared library and data segments, file state, signal handlers, and pending signals. The checkpoint signal handler then restores the CPU state and returns to the user code, which continues from where it left off when the checkpoint signal arrived.

Condor processes for which checkpointing is enabled perform a checkpoint when preempted from a machine. When a suitable replacement execution machine is found (of the same architecture and operating system), the process is restored on this new machine from the checkpoint, and computation is resumed from where it left off. Jobs that can not be checkpointed are preempted and restarted from the beginning.

Condor's periodic checkpointing provides fault tolerance. Condor pools are each configured with a periodic checkpointing expression which controls when and how often applications which can be checkpointed do periodic checkpoints (examples: never, every three hours, every six hours if the process's image size is greater than 20 megabytes, etc.). When the time for a periodic checkpoint occurs, the application suspends processing, performs the checkpoint, and immediately continues from where it left off. There is also a condor_checkpoint command which allows the user to request that the application immediately performs a periodic checkpoint.

In all cases, Condor applications continue execution from the most recent complete checkpoint. If service is interrupted while a checkpoint is being performed, causing that checkpoint to fail, the process will restart from the previous checkpoint. Condor uses a commit style algorithm for writing checkpoints: a previous checkpoint is deleted only after a new complete checkpoint has been written successfully.

By default, a checkpoint is written to a file on the local disk of the submit machine. A checkpoint server has also been developed to serve as a repository for checkpoints. When a host is configured to use a checkpoint server, applications submitted on that machine write and read checkpoints to and from the server rather than the local disk of the submitting machine, taking the burden of storing checkpoint files off of the submitting machines and placing it instead on server machines (with disk space dedicated to the purpose of storing checkpoints).

5 Conclusions

In this paper we briefly described three mechanisms - the ClassAd match-making mechanism, the Remote System Calls mechanism, and the Checkpointing mechanism - that are the three pillars of the Condor High Throughput Computing environment. Through the use of these mechanisms, Condor can provide resource management services which enable scientists to harness the capacity of very large collections of distributively owned UNIX workstations. The generality and flexibility of these mechanisms allow the Condor Team to expand these services to new operating systems and computing resources. The team is currently engaged in an effort to port Condor to the WindowsNT operating system and to add support for multi-processor, communication, and software resources.

6 References

[C97] Condor Team, "The Condor High Throughput Computing Environment," *http://www.cs.wisc.edu/ condor/*

[ELDE96] J. Epema, M. Livny, R. van Dantzig, X. Evers and J. Pruyne, "A Worldwide Flock of Condors : Load Sharing among Workstation Clusters," in *Journal on Future Generations of Computer Systems*, Volume 12, 1996

[L87] M. Litzkow , "Remote Unix - Turning Idle Workstations into Cycle Servers," in *Proceedings of Usenix Summer Conference*, 1987.

[LLM98] M. Litzkow, M. Livny, and M. W. Mutka, "Condor - A Hunter of Idle Workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June, 1988.

[LTBL97] 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*, April 1997.(http://www.cs.wisc.edu/ condor/doc/ckpt97.ps)

[PL96] J. Pruyne and M. Livny, "Interfacing Condor and PVM to Harness the Cycles of Workstation Clusters," in *Journal on Future Generations of Computer Systems*, Volume 12, 1996