Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor

Derek Wright
Department of Computer Sciences,
University of Wisconsin, Madison

Abstract

Clusters of commodity PC hardware running Linux are becoming widely used as computational resources. Most software for controlling clusters relies on dedicated scheduling algorithms. These algorithms assume the constant availability of resources to compute fixed schedules. Unfortunately, due to hardware and software failures, dedicated resources are not always available over the long-term. Moreover, these dedicated scheduling solutions are only applicable to certain classes of jobs, and they can only manage clusters or large SMP machines. The Condor High Throughput Computing System overcomes these limitations by combining aspects of dedicated and opportunistic scheduling into a single system. Both parallel and serial jobs are managed at the same time, allowing a simpler interface for the user and better resource utilization. This paper describes the Condor system, defines opportunistic scheduling, explains how Condor supports MPI jobs with a combination of dedicated and opportunistic scheduling, and shows the advantages gained by such an approach. An exploration of future work in these areas concludes the paper. By using both desktop workstations and dedicated clusters, Condor harnesses all available computational power to enable the best possible science at a low cost.

1 Introduction

Science and industry are increasingly reliant on large amounts of computational power. With more CPU cycles, larger problems can be solved and more accurate results can be obtained. However, large quantities of computational resources have not always been affordable to the end user. Small research groups and institutions are often unable to acquire the resources necessary to meet their computational needs.

The Condor High Throughput Computing System has scavenged otherwise wasted CPU cycles from desktop workstations for more than a decade [1][2][3]. These inconsistently available resources have proven to be a significant source of computational power, enabling scientists to solve ever more complex problems. Condor efficiently harnesses existing resources, reducing or eliminating the need to purchase expensive supercomputer time or equivalent hardware.

In recent years, a new trend has emerged. Clusters of commodity PC hardware running Linux are becoming widely used as computational resources. The cost to performance ratio for such clusters is unmatched by other platforms. It is now feasible for smaller groups to purchase and maintain their own clusters. However, these clusters introduce a new set of problems to the end user. There are problems with both the dedicated nature of the resources in the clusters, and with the scheduling systems used to manage these resources.

1.1 Dedicated resources are not dedicated

Most software for controlling clusters relies on dedicated scheduling algorithms. These algorithms assume the constant availability of resources to compute fixed schedules. Unfortunately, due to hardware or software failures, dedicated resources are not always available over the long-term. Everything from routine system maintenance down-time and scheduled interactive use, to disk failures, network outages, and other unexpected problems, can cause significant problems for dedicated scheduling algorithms. No resource is always available.

As the price of hardware continues to decline, the size and computational power of these dedicated clusters continues to increase. However, the increase in size also increases the probability of hardware failure. This compounds the problems introduced by scheduling software that assumes continuous availability.

1.2 The problems with dedicated schedulers

Most dedicated scheduling solutions currently in use are only applicable to certain classes of jobs, and can only manage dedicated clusters or large SMP machines. If users have both serial and parallel applications to run, they are often forced to submit their jobs to separate systems, learn a different set of interfaces for each one, use different tools to monitor the jobs, and so on. Moreover, the system administration costs are increased, since the administrators must install and maintain separate systems.

It may be difficult or impossible to have separate schedulers managing the same set of resources. As a result, resource owners or administrators are sometimes forced to partition their resources into separate groups, one that serves parallel applications and one that serves serial applications. If there is an uneven distribution of work between the two different systems, users will wait for one set of resources while computers in the other set are idle.

1.3 The Condor solution

Condor overcomes these difficulties by combining aspects of dedicated and opportunistic scheduling into a single system. *Opportunistic scheduling* involves placing jobs on non-dedicated resources under the assumption that the resources might not be available for the entire duration of the jobs. Condor makes use of a procedure known as *checkpointing*, where the computation is suspended at a fixed point and the current job state is written out to a file[4][5]. The resulting checkpoint file can be used to restart the application at the point in its execution when the checkpoint was taken. Condor can insure forward progress of its jobs, even on non-dedicated resources, by using checkpointing.

Instead of forcing administrators to partition resources, Condor manages all resources within a single system. Users can submit a wide variety of jobs to Condor, from serial to parallel jobs, including both PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) applications. This flexibility allows the user to spend less time learning new interfaces and tools, yielding more time for doing the research they ultimately care about. It saves time and money for the administrators who only have to maintain a single system. The unified approach also yields high resource utilization with simple dedicated scheduling algorithms.

Dedicated scheduling was introduced in the 6.3.X development series of Condor. Its initial release was version 6.3.0 (April, 2001). This release ushers in a new phase in Condor's history. Throughout most of its existence, Condor had only provided opportunistic scheduling for non-dedicated resources. Now, Condor has integrated dedicated and opportunistic scheduling into a single system.

The following sections cover the general architecture of the Condor system, how Condor manages both dedicated and non-dedicated resources with a combination of dedicated and opportunistic scheduling, why some of the traditional problems in dedicated scheduling do not apply to this hybrid approach, and how Condor handles various failure conditions. An exploration of future work in these areas concludes the paper.

2 Architecture of the Condor system

Condor is a distributed batch scheduling system. It is composed of a collection of different daemons that provide various services, such as resource management, job management, matchmaking, and so on. The goal of Condor is to provide the highest feasible throughput by executing the most jobs over extended periods of time.

Condor exercises administrative control over a *Condor pool*. A pool is a set of resources that all report to a single daemon called the `collector`. The `collector` is the central repository of information in the Condor system. Almost all Condor daemons send periodic updates to the `collector`. Each update is in the form of a *ClassAd*, a data structure consisting of a set of attributes describing a specific entity in the system. Each ClassAd can optionally specify requirements and preferences on the entities it is looking for, much like classified advertisements in a newspaper. Many tools and daemons perform queries against the `collector` to acquire information about other parts of the system. The machine where the `collector` runs is referred to as the *central manager*.

The central manager also runs a daemon called the `negotiator`, which periodically performs a *negotiation cycle*. This cycle is a process of *matchmaking*, where the `negotiator` tries to find matches between various ClassAds, in particular, resource requests and resource offers[6]. Once a match is made, both parties are notified and are responsible for acting on that match.

Computational resources within the pool are managed and represented to the system by a daemon called the `startd`. This daemon monitors the conditions of the resource where it runs, publishes resource offer ClassAds, and it is responsible for enforcing the resource owner's policy for starting, suspending, and evicting jobs. The policy is defined by a set of boolean expressions given in a configuration file that control the transitions between the possible states the resource could be in (idle, running a job, suspending the job, gracefully evicting the job, killing the job, and so on). Resource owners can specify requirements for which users or jobs the resource is willing to serve. Owners can also specify a rank (in an arbitrarily complex order) for which users or jobs a resource most prefers to serve. Any machine running a `startd` can be referred to as an *execute machine*, since it is able to execute Condor jobs.

A user's job is represented in the Condor system by a ClassAd. Users submit their jobs to a daemon called the `schedd`. This daemon maintains a persistent job queue, publishes resource request ClassAds, and negotiates for available resources. After it receives a match for a given job, the `schedd` enters into a claiming protocol directly with the `startd`. Through this protocol, the `schedd` presents the job ClassAd to the `startd` and requests temporary control over the resource. Once it has claimed a given resource, the `schedd` performs its own local scheduling to decide what jobs to run. Any machine running a `schedd` can be referred to as a *submit machine*, since users are able to submit Condor jobs from that host.

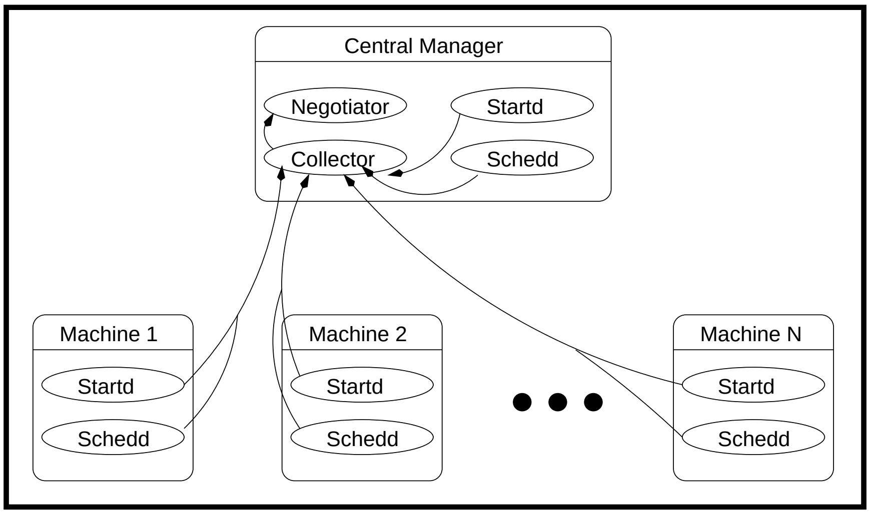

Figure 1: Architecture of a Condor pool with no jobs running

Several things can break a claim and either create a new claim with a different `schedd`, or give the resource back to the control of the `startd`:

1. The resource owner can start using the resource again (outside of the Condor system). By default, Condor is configured to evict any currently running jobs and release any claims. This situation is referred to as either *eviction* or *owner preemption*.

2. A user with a higher priority (based on resource usage) can *preempt* the existing job. Once the existing job has been evicted, the `schedd` representing the new user will claim the resource for itself. This process is known as *priority preemption*.

3. A user or job that is ranked higher by the resource can enter the system and request access to the resource. The `startd` will release the claim from the lower-ranked entity, and grant a claim to the higher-ranked one. This process is referred to as *rank preemption*.

4. The `schedd` can decide it no longer needs the resource and release its claim.

To handle the low-level details of starting an application on a given resource, the `schedd` spawns a `shadow` process. To execute an application under a `startd`, the `shadow` contacts the `startd` and requests that a `starter` process be created. The `starter` and `shadow` communicate with each other to handle all the application-specific needs for staging input files, starting jobs, moving output files when the jobs complete, and reporting the exit status of the application back to the `schedd`.

The `shadow` also enables *remote system calls* for certain opportunistic jobs[5]. System calls performed by the job are trapped by a Condor library, sent over the network to the `shadow`, executed locally on the submit machine, and the results are shipped back over the network to the job. This allows remote access to files that exist only on the local file system of the submit machine. In addition to remote system calls, Condor provides mechanisms to transfer data files from the submit machine to an arbitrary execute machine. This allows any job to stage input data and retrieve its output without relying upon a shared file system or relinking the application with Condor's library.

The various daemons (`schedd`, `startd`, `collector` and `negotiator`) that provide different services in the Condor system are completely independent from each other. A given machine in a Condor pool can run any number of them, depending on what services are needed on that machine. The only requirement is that, by definition, only one `collector` and `negotiator` can run in a single pool.

In addition to the daemons described above, all machines in a Condor pool run the `master` daemon, which spawns any other Condor daemons a machine is configured to run, and acts as a server for a number of remote administration tools.

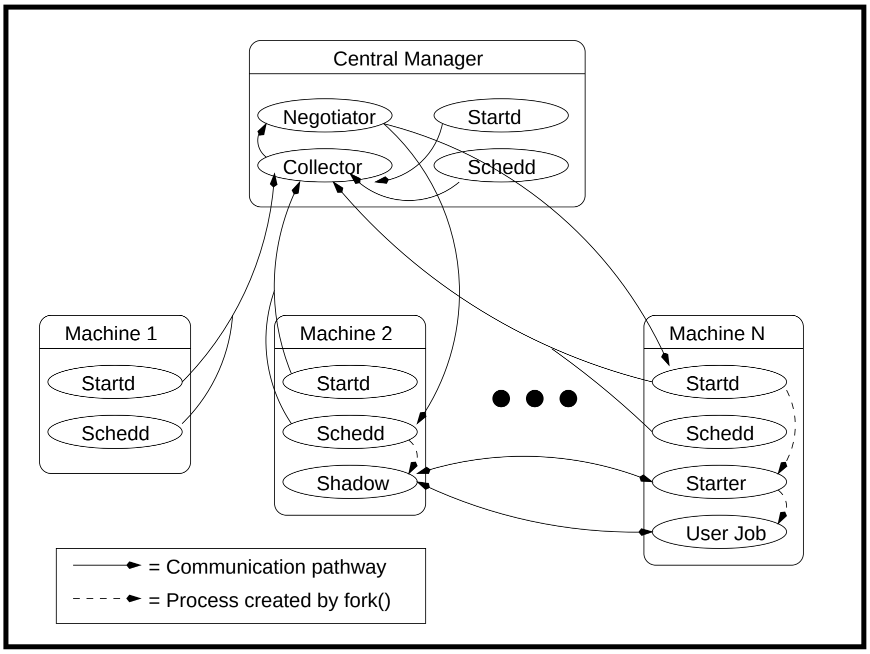

Figure 2: Architecture of a Condor pool with a job running

3 Condor's dedicated scheduling

The introduction of dedicated resources and dedicated scheduling to the Condor system created a few minor changes to the system architecture. A new version of the `schedd` contains the special dedicated scheduling logic. This `schedd` also performs all of the existing opportunistic scheduling services. For example a large Beowulf cluster with a single front-end node can run the dedicated `schedd` on that node and the `schedd` would handle both dedicated and opportunistic jobs.

Only MPI jobs require dedicated scheduling in Condor, since the PVM interface can handle dynamic resource management, with nodes coming and going from a PVM application[7]. To support MPI applications, modified versions of the `shadow` and `starter` were needed. Because of Condor's layered architecture, these were the only places in Condor that had to be modified to support this new kind of application.

3.1 Configuring a Condor pool for dedicated scheduling

To support dedicated jobs, a Condor administrator configures certain resources in the pool to be controlled by the dedicated scheduler. In general, there is no limit on the number of dedicated schedulers in a Condor pool. However, each dedicated resource may only be managed by a single dedicated scheduler. Therefore, running multiple dedicated schedulers in a single pool results in greater fragmentation of dedicated resources. This can create a situation where jobs will not run, because the jobs can not get needed resources. There is little benefit to having multiple dedicated schedulers, particularly at the cost of artificial resource fragmentation.

To configure a dedicated resource under a given scheduler, the resource owner or administrator sets a few lines in the `startd`'s configuration file. The resource advertises a special attribute in its ClassAd that says which dedicated scheduler it is willing to be managed by. The resource's policy for starting and stopping jobs must also be modified to always allow jobs from the dedicated scheduler to start, regardless of machine state, and to never evict dedicated jobs once they begin. Finally, the resource must be configured to prefer jobs from the dedicated scheduler over all other jobs. A dedicated resource in Condor is simply configured so that the dedicated scheduler of its choice has the highest rank. Condor's existing opportunistic scheduler (matchmaker) guarantees that if the dedicated scheduler wants to claim the resource, it will always be allowed to do so, regardless of any other conditions. Once the resource has been claimed by the dedicated scheduler, the opportunistic scheduler ensures that no other jobs can preempt it.

It is worth noting that Condor puts no other requirements on a resource for it to be considered dedicated. If the owners of desk-top workstations were willing to allow their machines to be configured in this way, those workstations would be dedicated resources in Condor, and would behave exactly like the nodes in a Beowulf cluster.

To aid in the definition of the policy expressions, the dedicated scheduler adds an attribute to all resource request ClassAds it generates, the Scheduler attribute. This attribute identifies each ClassAd as a request of a particular dedicated scheduler. The owners of the resources can easily define separate policies for dedicated and opportunistic jobs, simply by including two cases in each policy expression, one case for when the Scheduler attribute identifies the request as one belonging to the preferred dedicated scheduler, and one for if the Scheduler attribute is not defined or points to a different scheduler.

Once a set of dedicated resources have been configured and a new `schedd` is running, users are free to submit MPI jobs to the system. Once the `schedd` notices MPI jobs in the queue, the dedicated scheduler logic is activated.

3.2 Making resources dedicated

If the dedicated scheduler has idle jobs to service, it will query the `collector` for all resources that are configured to be under its control. The dedicated scheduler does its own matchmaking to find resources it could claim that will match idle MPI jobs. Once resources are identified, a ClassAd is generated and sent to the opportunistic matchmaker, requesting negotiation. During the next negotiation cycle, the `negotiator` contacts the dedicated scheduler and performs matchmaking for any idle resource requests (which are modified to include the Scheduler attribute described above, and to require that any resource matched with the request must be managed by the dedicated scheduler).

Once resources are matched, the dedicated scheduler claims them and puts them under its control. This moves the resources from the control of the opportunistic scheduler to the control of the dedicated scheduler. Once claimed in this fashion, the dedicated scheduler can run any jobs it wants on the resource, even dedicated jobs from a different user than the one whose job it originally negotiated for. The key is that the resource is claimed by the dedicated scheduler itself. So long as there are idle MPI jobs that match this resource, the dedicated scheduler will use the resource.

If the dedicated scheduler decides that a given resource is no longer needed, it can release its claim on that resource, thereby returning it to the opportunistic scheduler. The resource will then be treated as any other non-dedicated resource in the system. If more MPI jobs are submitted, the dedicated scheduler starts the process all over again. Depending on the state of the pool, the dedicated scheduler may be matched with the same resource, any currently running opportunistic job would be preempted, and the resource would be claimed by the dedicated scheduler again.

3.3 Condor's dedicated scheduling algorithm

Condor uses a basic scheduling algorithm for dedicated MPI jobs. A global FIFO queue of MPI jobs across all users is maintained. At the start of a dedicated scheduling cycle, the `schedd` performs a query to find all resources that either are claimed or could be claimed for dedicated jobs. Only resources in this list will be considered by the dedicated scheduler. Once the dedicated scheduler has the list of resource ClassAds, it sorts them by the time they would next be available for running a job.

The dedicated scheduler considers each job in the queue, in order. It tries to satisfy the requirements of the job, performing its own local matchmaking procedure. If the first job can be satisfied by resources that have already been claimed but are now idle, those resources are removed from the available list, allocated to the job, and the next job is considered. If the job at the head of the queue can not be served by currently claimed and idle resources, the dedicated scheduler sees if it could satisfy the job by negotiating for other resources that are currently unclaimed, or by a combination of already claimed/idle and unclaimed resources. If so, any resources used for the job are removed from the available list, and the next job is considered. If the top job cannot be satisfied at all by claimed/idle and unclaimed resources, the dedicated scheduler checks if enough dedicated resources exist to satisfy this job. If so, the dedicated scheduler stops scheduling jobs, and waits for its existing dedicated jobs to finish so that it can acquire enough resources to satisfy the large job at the front of the queue. If not, the job is too big to run (until more resources are configured as dedicated), so the scheduler skips over it and considers the next job in the queue.

Once all scheduling decisions have been made in a given cycle, the dedicated scheduler can begin to act on them. Any jobs that were satisfied by claimed/idle resources are spawned. The dedicated scheduler also sends out resource request ClassAds to the opportunistic scheduler for any unclaimed resources that were needed. If the dedicated scheduler

By maintaining a single queue across all users, this algorithm can lead to unfair resource allocation. If a user submits a large number of MPI jobs at the same time, once those jobs are at the front of the queue, the user will starve all other users until the jobs complete. This is obviously not ideal. It is an area of future work.

4 Why some traditional problems in dedicated scheduling do not apply to Condor

Some of the traditional problems found in dedicated scheduling do not apply to Condor because of its unique combination of dedicated and opportunistic scheduling. The following sections discuss these problems and how Condor avoids them.

4.1 Backfilling

In dedicated scheduling systems, one of the more difficult problems is what to do with holes in the schedule. For a variety of reasons, there will inevitably be periods of time when certain resources cannot be used by the scheduler. This fragmentation leads to poor utilization of resources and lower overall job throughput in the system.

The traditional solution to this problem is called *backfilling*. While backfilling can take many forms, in general, it attempts to fill holes in the schedule by running smaller parallel jobs of lower priority in an attempt to keep the resources utilized while the highest priority parallel job waits for enough resources to begin execution[8][9]. However, the holes remain if there are not enough small jobs, or if the jobs are of the wrong size and duration.

Another method for filling holes in the schedule would be to try to run serial jobs on the empty nodes. However, most scheduling systems do not provide support for opportunistic execution of serial jobs. If the serial jobs need to run longer than the available window in the schedule, they must either be killed, loosing the work they have completed, or they will delay the schedule, leading to a further reduction in utilization.

In Condor, since there is an existing infrastructure for managing non-dedicated resources with opportunistic scheduling, any holes in the dedicated schedule may be filled by releasing the resources the opportunistic scheduler's control. Because the jobs that will run on these resources are opportunistic, they are already prepared, even expecting, to be preempted at any time. When these resources are again needed for the dedicated schedule, they will be re-claimed by the dedicated scheduler. Condor's opportunistic scheduling system already handles this case, just as if a desktop workstation's owner had returned and moved the mouse or touched the keyboard. Condor will take a checkpoint of the job, and migrate the job to another host, or put it back in the queue if no other resources are available. Checkpointing the opportunistic jobs ensures forward progress during backfilling. This leads to high resource utilization, and greater overall system throughput.

4.2 Requiring users to specify a job duration

Another common problem that arises from traditional dedicated schedulers is that users must specify an expected duration for any job they submit to the system. Many users do not know how long their applications will run ahead of time, and in general, it is impossible to know without actually executing the code.[1]

The result of this requirement is that in practice users are forced to guess how long their programs will take. Unfortunately, there is a high price for guessing wrong. For most schedulers, a job that exceeds its expected duration will be killed, thereby losing all the work accomplished. Users are therefore encouraged to guess high. However, jobs with very long durations can often wait longer in the queue before a sufficiently large slot is available in the schedule. Some schedulers even charge users based on expected duration, not actual usage. So, guessing too high is also not in the user's interests. This leaves quite a dilemma for the user.

Users of Condor do not need to guess a duration. Because Condor's dedicated scheduler can release or reclaim nodes at any time and still expect them to be utilized, it does not need to make decisions far into the future. At each scheduling cycle, Condor's dedicated scheduler can make the best decisions possible with the current state of all the resources under its control and the jobs in its queue. If an idle job in the queue could run if more resources were claimed, the dedicated scheduler can send out a ClassAd to the opportunistic scheduler to request more resources. The dedicated scheduler does not need to know how much longer the currently running jobs will take, since it can make all of its scheduling decisions assuming those resources are unavailable. As resources become available, they will be put to use for any dedicated jobs in the queue. If the job at the head of the dedicated queue needs more resources than are currently available, any currently claimed but idle dedicated resources can be released.

5 Condor handles failures of dedicated resources

The Condor system has been managing non-dedicated, unreliable resources for over 10 years. All levels of Condor are designed to recover gracefully from network or hardware failures, machines disappearing from the network, and other failures. Condor's support for managing dedicated applications is built on top of this foundation and takes advantage of all of the existing robustness.

If a submit machine crashes, all resources claimed by its `schedd` will shut down any currently running jobs. When the submit machine is brought back up, a persistent log file of the job queue enables the system to automatically re-spawn the jobs without user intervention. If the jobs were opportunistic, they will be restarted from their last checkpoint file, minimizing lost computation, otherwise, they are restarted from the beginning. The dedicated scheduler is just a special

[1] If you had a generic, deterministic method to decide if an application would complete **at all** (much less how long it would take), you would have solved the so-called 'Halting Problem' and proved that P=NP, one of the longest outstanding problems in all of Computer Science.

case of the `schedd`, so the same failure mode is used in the event of a hardware failure at the dedicated scheduler.

The Condor system is also resilient in the face of a hardware failure of the central manager. Once matches are made and resources are claimed by local schedulers (opportunistic or dedicated), the central manager is no longer needed for any part of the job execution protocol. Currently running jobs would not even know that the central manager was down. Once jobs complete, the local schedulers attempt to use the resources they have claimed to service other jobs in their local queues. This is true of the dedicated scheduler as well, since it will first try to use resources it has already claimed to service any idle jobs in its queue. Certain pool-monitoring tools will fail if the central manager is down, but running jobs continue to run, and new jobs may be scheduled.

Parallel applications create their own challenges in the face of failures. If a node in a parallel computation crashes due to a hardware failure, Condor notices that the resource went away and shuts down the rest of the computation. Since Condor does not yet support checkpointing of parallel applications, it must restart the job from the beginning when enough resources are available. This is an acceptable failure mode in the absence of checkpointing. So long as the job is re-queued and not lost as a result of a hardware failure, the scheduling system is doing its job.

If a node crashes due to a software failure on the part of the parallel application itself (for example, a segmentation violation in the user's code), Condor supports two methods to propagate the error back to the user: a log file and an email message to the user. When users submit jobs, they specific which form of error reporting, if any, they wish to use. In addition to notifying the user of the error, any core files produced are shipped back to the submit machine, and the job leaves the queue.

6 Future work

Although Condor's approach to managing dedicated resources by a combination of dedicated and opportunistic scheduling has advantages over traditional dedicated schedulers, there is a considerable amount of future work that is both possible and desired.

6.1 Incorporating user priorities into the dedicated scheduler

One of the first steps towards improving Condor's dedicated scheduling would be to incorporate user priorities into the system. Condor currently uses a basic scheduling policy, with a global queue of jobs. An improvement would be to have a separate logical queue per user, and utilize Condor's user priority system to determine the order in which queues were serviced. While this would not effect overall system efficiency or throughput, it would increase fairness.

6.2 Knowing when to claim and release resources

The unique combination of opportunistic and dedicated scheduling in Condor introduces a new problem: when should the dedicated scheduler claim or release its resources? Condor currently relies on relatively simple heuristics to make these decisions. System efficiency (and therefore, job throughput) could be enhanced by exploring more accurate and sophisticated methods for determining the best time to initiate these resource state changes.

6.3 Scheduling into the future using job duration information

While there are problems that arise from trying to force users to specify a duration for their dedicated jobs (described above), there are benefits that can only be achieved by having duration information. Certain resources are only available for a fixed period of time. Condor's dedicated scheduler could not use these resources to service dedicated jobs unless it knew (or at least expected) that those jobs would complete before the resources became unavailable again.

Condor would not require users to specify a job duration. Duration information can improve response time by allowing Condor to use resources for dedicated jobs that would otherwise only be available for opportunistic scheduling.

6.4 Allowing a hierarchy of dedicated schedulers

The current Condor system only allows a resource to be controlled by a single dedicated scheduler. If Condor supported job durations as described above and was making schedules into the future, Condor could also allow a hierarchy of dedicated schedulers to control a single resource. The dedicated scheduler at the top of the hierarchy would have ultimate control over the resource. However, when that scheduler decided it no longer needed the resource, instead of returning it to the opportunistic scheduler for the indefinite future, the dedicated scheduler could specify how long that resource should be considered available. Other dedicated schedulers lower in the hierarchy could then claim the resource, if they could use it within the given time constraint.

A hierarchy of dedicated schedulers would be particularly useful in a situation where multiple Condor pools are flocked together, allowing jobs to migrate between the pools[10][2]. The dedicated resources in each pool would rank their local dedicated scheduler at the top of their hierarchy. If the resources were not used by the local dedicated scheduler, they would allow the other dedicated schedulers within the flock access. This would enable execution of parallel applications that were larger than any of the individual pools, assuming there were sufficient dedicated resources available.

6.5 Allowing multiple executables within the same application

Condor supports a single executable (which therefore is bound to one platform) within a given MPI job. Users should be able to specify multiple executables within the same job. This allows users to take advantage of resources of different platforms within a single computation. It also allows for more complex parallel architectures where not all nodes in the computation use the same executable.

[2] The paper referenced here is out of date. Unfortunately, no existing publications describe Condor's current flocking mechanism.

Many MPI applications circumvent this problem by including multiple sets of instructions within a single executable, conditionally executing one set based on the unique identifier of each node in the computation.[3] This solution is inefficient, as it requires larger executables to be moved around the system, and more (unused) data loaded into memory. Users could split these separate instruction sets into different stand-alone executables, so that each node would only incur the cost of bringing in the instructions it needs.

6.6 Supporting MPI implementations other than MPICH

MPICH is a widely used and portable implementation of the MPI standard, developed at Argonne National Laboratory and Mississippi State University[11][12]. While it is possibly the most used implementation of MPI, it is certainly not the only one (for example: LAM[13], ScaMPI, MPI/Pro, and a number of vendor-specific MPI implementations). One of the great strengths of MPI is that it is a standard[14][15]. A single computation can use different executables, linked against different implementations of the standard, and all nodes will be able to communicate with each other. Many of the vendor-specific implementations provide better performance than the generic, portable implementations on the given platform. Different implementations of MPI often use different methods for spawning the nodes, and Condor only supports the method used by MPICH.

6.7 Dynamic resource management routines in the MPI-2 standard

The de facto MPI standard (version 1.2) provides no dynamic resource management. There are no methods for adding resources to an existing MPI computation or for gracefully removing resources. The new version of standard, MPI-2, supports a number of dynamic resource management routines[16][17]. Unfortunately, there are no implementations of MPI that fully support these routines at this time. When implementations of the new standard are available, Condor will have to be modified to support the dynamic resource management interface. MPI-2 jobs will allow more flexibility in scheduling, including the use of non-dedicated resources.

6.8 Generic dedicated jobs

Some parallel applications that require dedicated resources do not use MPI. Certain serial applications desire access to dedicated resources. Condor could be modified to allow users to specify any job in the system as a dedicated job. This would enable users of Condor to submit generic parallel dedicated jobs, not just applications linked against MPI. It would also allow dedicated scheduling of serial jobs that can not be checkpointed. Dedicated scheduling of non-opportunistic serial jobs would increase the likelihood that the jobs would execute until completion, thereby improving overall system throughput.

6.9 Allowing resource reservations

Many users desire the ability to make future reservations on dedicated resources. For example, a user might need to run an interactive application on a number of

[3] Unfortunately, MPI uses the term *rank* for this unique identifier, which is completely different from the *rank* used in the rest of Condor to specify preferences.

dedicated resources at a certain time. Condor should be able to support reservations and ensure that no jobs are running on any of the requested resources at the specified time. Since Condor needs knowledge of the future to support resource reservations, job duration information is required.

6.10 Checkpointing parallel applications

Checkpointing parallel applications is a difficult task. The main challenge in supporting parallel checkpointing is checkpointing the state of communication between nodes. Any data in transit between nodes is both difficult and essential to checkpoint. Preliminary research at UW-Madison has demonstrated the ability to checkpoint and restart MPI applications, including network state and data in transit[18]. Other methods for checkpointing parallel applications attempt to flush all communication pathways before checkpointing[19].

Supporting parallel checkpointing would significantly alter (and improve) many of the scheduling problems explained above. If a parallel job can be preempted without loosing all previous work, the scheduling system does not require dedicated resources. Resources could be configured to be dedicated and Condor would steer parallel jobs towards those resources. However, Condor could schedule parallel jobs on any resources it had access to, including a combination of dedicated and non-dedicated nodes. If Condor supported parallel checkpointing, it could provide resource reservations without any knowledge of job duration. Finally, Condor could provide periodic checkpointing of parallel applications, which would reduce the computational time lost in the event of hardware or software failures.

7 Conclusion

By using both desktop workstations and dedicated clusters, Condor harnesses all available computational power in a single system. This provides a consistent interface for users. Since the same resources can be used for both dedicated and opportunistic jobs, Condor provides a high degree of system utilization. Some of the traditional problems with dedicated scheduling are avoided by this unique combination of scheduling models. After years of managing non-dedicated resources, the Condor system is resilient in the face of many different types of hardware and software failure. This robustness is essential for high job throughput on dedicated resources as well, since all resources are prone to failure. With its combination of dedicated and opportunistic scheduling, and its robustness, the Condor system provides significant computational resources to end-users, thereby enabling better science. There are many areas of future work relating to the interaction of dedicated and opportunistic scheduling that will enhance Condor's ability to provide cheap, effective cycles.

References

[1] Michael Litzkow. Remote UNIX: Turning Idle Workstations into Cycle Servers. In *Proceedings of the 1987 Usenix Summer Conf.*, pages 381–384, 1987.

[2] Michael Litzkow & Miron Livny & Matthew Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[3] Jim Basney & Miron Livny. Deploying a High Throughput Computing Cluster. In R. Buyya, editor, *High Performance Cluster Computing*, volume 1, chapter 5. Prentice Hall PTR, 1999.

[4] Michael Litzkow & Todd Tannenbaum & Jim Basney & Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, April 1997.

[5] Miron Livny & Jim Basney & Rajesh Raman & Todd Tannenbaum. Mechanisms for High Throughput Computing. *SPEEDUP Journal*, 11(1), June 1997.

[6] Rajesh Raman & Miron Livny & Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[7] A. Geist et al. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA, 1994.

[8] D. G. Feitelson & A. M. a. Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backfilling. In *Proceedings of the 2nd International Parallel Processing Symposium*, pages 542–546, Orlando, Florida, April 1998.

[9] Brett Bode & David M. Halstead & Ricky Kendall & Zhou Lei. The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[10] D. H. J. Epema & M. Livny & R. van Dantzig & X. Evers & J. Pruyne. A Worldwide Flock of Condors : Load Sharing among Workstation Clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[11] W.D. Gropp & E. Lusk & N. Doss & A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] W.D. Gropp & E. Lusk. *User's Guide for mpich, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[13] Greg Burns & Raja Daoud & James Vaigl. LAM: An Open Cluster Environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[14] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3–4):165–414, 1994.

[15] Marc Snir & Steve W. Otto & Steven Huss-Lederman & David W. Walker & Jack Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core,* 2nd edition. MIT Press, Cambridge, MA, 1998.

[16] Message Passing Interface Forum. MPI2: A Message Passing Interface Standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.

[17] William Gropp & Steven Huss-Lederman & Andrew Lumsdaine & Ewing Lusk & Bill Nitzberg & William Saphir & Marc Snir. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, 1998.

[18] Victor C. Zandy & Barton P. Miller. *Reliable Sockets*. http://www.cs.wisc.edu/paradyn/papers/index.html#rocks. University of Wisconsin – Madison, 2001.

[19] G. Stellner. Cocheck: Checkpointing and Process Migration for MPI. In *Proceedings of the International Parallel Processing Symposium*, pages 526–531, Honolulu, April 1996.