Managing Checkpoints for Parallel Programs

Jim Pruyne and Miron Livny

Department of Computer Sciences
University of Wisconsin–Madison
{pruyne, miron}@cs.wisc.edu

Abstract

Checkpointing is a valuable tool for any scheduling system to have. With the ability to checkpoint, schedulers are not locked into a single allocation of resources to jobs, but instead can stop running jobs, and re-allocate resources with out sacrificing any completed computations. Checkpointing techniques are not new, but they have not been widely available on parallel platforms. We have implemented CoCheck, a system for checkpointing message passing parallel programs. Parallel programs tend to be large in terms of their aggregate memory utilization, so the size of their checkpoint is also large. Because of this, checkpoints must be handled carefully to avoid overloading the system when checkpoints take place. Today's distributed file systems do not handle this situation well. We therefore propose the use of checkpoint servers which are specifically designed to move checkpoints from the checkpointing process, across the interconnection network, and on to stable storage. A scheduling system can utilize numerous checkpoint servers in any configuration in order to provide good checkpointing performance.

1 Introduction

The ability to checkpoint a running program, whether it be a sequential or parallel program, is a valuable tool for a scheduling system. One common use for checkpointing is to provide fault tolerance. Checkpointing also allows the scheduler to re-allocate resources among both running and queued jobs without sacrificing any computations already performed. For example, Condor's [1] ability to checkpoint sequential programs has allowed it to effectively utilize the idle time of privately owned workstations for long running jobs. By checkpointing a program when an owner reclaims a machine, Condor is able to run programs which take much longer than any single idle interval at a workstation.

Parallel scheduling systems may also benefit from the ability to checkpoint programs in many ways. For example, most current parallel schedulers require the user to specify how long a job will run, and the scheduler simply kills jobs which do not complete in the specified time. By killing the job, the entire current state of the computation is lost, and therefore the resource time allocated to the job has been wasted. A more desirable approach would be to checkpoint the entire parallel application. The checkpointed program could then be re-submitted to the system, and computation would continue from the point where the job was forced to vacate the machine. In this way, the time already invested in the job will be preserved.

Another use for checkpointing in a parallel system is to perform *dynamic partitioning* which has been shown [2] to be more effective than static methods of scheduling parallel programs. In a dynamic partitioning scheme, the number of resources allocated to a job is changed while the job is running based on changes in load on the overall system. Without the ability to checkpoint and save the state of running processes, it would not be possible to move processes to perform a dynamic partitioning resource reallocation. The conditions under which dynamic partitioning is beneficial depend greatly on the overhead involved in doing resource re-allocation. When this overhead becomes high, the benefits of dynamic partitioning are lost. It is therefore important to perform checkpoint and restart operations as quickly as possible.

Techniques for checkpointing of parallel and distributed programs have been understood for quite some time. For example, Chandy and Lamport proposed a "distributed snapshot" protocol in 1985 [3]. Thus far, however, implementations of these techniques for real parallel systems have been rare. Building on the theory and the experience gained doing checkpointing for single process jobs in Condor, we have developed a system for checkpointing message passing parallel programs called CoCheck (for Consistent Checkpointing). CoCheck implements a network consistency protocol much like Chandy and Lamport's distributed snapshot protocol, and utilizes the single process checkpoint ability of Condor to save the state of each process in a parallel application.

In practice, checkpointing a parallel program tends to be a time consuming operation. The exact attribute which makes parallel programs desirable, their ability to perform computations which are extremely large in both computation and memory requirements, makes them difficult to checkpoint. In particular, a checkpoint must by definition include the entire state of the running program. A parallel program's state consists of the state of the interconnection network as well as the address space of each process. The combined memory of all of the processes often amounts to a huge overall state which must some how be moved into secondary storage. There are two potential bottlenecks in saving this data: the interconnection network over which the data must travel, and the secondary storage devices on which the data will be stored. Because the scheduling system makes decisions related to checkpointing, it must determine how and where the checkpoints will be stored. To give the scheduler flexibility in making these decisions, we have implemented a *checkpoint server* which performs checkpoint file store and retrieve operations at the request of the scheduler. Using checkpoint servers, the scheduler is able to precisely direct the movement of checkpoints, and is not at the mercy of an external mechanism, such as a distributed file system, in which it cannot impose policy.

The rest of this paper is organized as follows. The next section describes the design of CoCheck. Section 3 provides discussion of alternatives which led to the development of the checkpoint server, and how it may be used by a scheduler. This is followed by some practical experience with the overall system and some conclusions and thoughts on future work.

2 CoCheck

CoCheck [4] is a freely available system for creating checkpoints of parallel programs which communicate via a Message Passing Environment (MPE). It has been developed via a collaboration between researchers at the Technical University of Munich and ourselves. The implementation available today works with PVM [5] on workstation clusters. Work is ongoing in Munich to extend CoCheck to support MPI [6].

We started with a number of important design goals when developing CoCheck. The first goal of CoCheck was to remain portable. That is, although the first implementation was done on top of PVM, the concepts used in CoCheck should be applicable to any MPE. Another important aspect of CoCheck is that it does not require modifications to the MPE implementation. This helps with portability both across different message passing systems, and for maintaining compatibility with updated releases of a single system. It also permits us to implement CoCheck on systems for which source code is not available. Additionally, we required flexibility in the degree of checkpointing to be performed. For example, it may be desirable to create a global checkpoint all processes in a parallel application, or it may only be necessary to checkpoint a little as one process to perform a migration. The degree of checkpointing to be performed is put under the control of the application or the scheduling system. Finally, CoCheck must have no *residual dependencies* on resources after a checkpoint is complete. This is to say that it is unacceptable to require continued participation in the life of a parallel program by a resource on which all processes have been checkpointed.

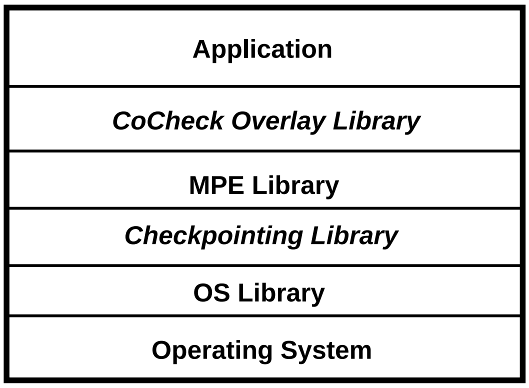

Figure 1: Layering of the CoCheck components

2.1 CoCheck Components

The state of a message passing parallel program at any given time consists of the state of each process in the application as well as the state of the communication network which may be carrying or buffering messages in transit. To capture this state, and to meet our design goals, CoCheck has been designed in three components: an overlay library for the message passing API, a single process checkpointing library, and a resource management (RM) process which coordinates the checkpointing protocol. The two libraries are linked into every application process generating a service layering as shown in figure 1. External to the application is a RM process which runs as part of the scheduling system. By using these three components, we have been able to meet our design goals, and at the same time leverage much pre-existing technology.

The overlay library is the key to doing checkpointing without modification to the underlying MPE. This library provides a stub for every function defined by the MPE. These stubs trap all application calls to the MPE, and perform communication identifier or other translations which must be made as a result of previous checkpoints and restarts. In most cases, the stub will in turn call the original MPE function to get the actual service performed. The overlay library also implements the protocol to capture the network state which is described below.

Single process checkpointing libraries have existed for quite some time. CoCheck utilizes the checkpointing library which was developed as part of Condor [7, 8] which, among others, provides this functionality without any modifications to the operating system on which it runs. The technique used for performing single process checkpointing is similar to the message passing overlay library described above (indeed, the techniques used in single process checkpointers were an inspiration for CoCheck's overlay approach). The state of a single process includes its memory (the bounds of its address space), the state of the processor registers, and any state within the operating system kernel such as the set of open files and their current seek position. Determining the bounds of the address space and saving the registers of a process are typically easy to perform. However, the increasing use of techniques such as dynamically loaded libraries have made address space lay-outs more complex making this a more difficult task. The overlay functions in a checkpointing library catch calls to the kernel which modify the kernel state of a process (for example, opening a file), and record this information so that it can be saved in the checkpoint and restored upon restart. Not all state of a process can be saved. For example, the parent-child relationship of processes following a `fork()` system call, or inter-process communication outside the scope of the MPE (e.g. pipes or sockets) cannot be retained. The Condor checkpointing library therefore disallows these system calls by trapping them and returning an error.

The final component of CoCheck, the resource manager process is the coordinator for the entire system. The RM process provided with CoCheck is an extension to the external RM process first designed for use with PVM [9]. This process receives requests for checkpointing services, and initiates the CoCheck protocol between itself and the overlay library of each the application processes to perform these services. The standard RM also writes a meta-checkpoint file which can later be re-read by a new instance of the RM to provide the information needed to restart the entire computation. Because PVM allows new resource manager processes to be defined, CoCheck can be used with any RM process which implements its protocols.

2.2 CoCheck Protocol

The CoCheck protocol (shown in figure 2) is responsible for ensuring that the entire state of the network is saved during a checkpoint, and to insure that communication can be resumed following a checkpoint. The CoCheck protocol begins when the RM determines that a checkpoint is required. This may be due to an application request, or because of a change in the state of a resource or due to a scheduling decision (such as the end of the time quanta allocated to a job). The RM begins by sending a signal and a message to each of the application processes. The combination of signal and message is required because each process may be either computing or communicating. The signal will interrupt a process which is computing causing it to enter the CoCheck library to participate in the checkpoint protocol. The overlay library of a process which is communicating will simply see the checkpoint request message, and interpret it as a request to begin checkpointing.

The checkpoint request message sent from the RM to each process contains two pieces of information. The first is how this process should participate in the checkpoint. The most common alternative is for the process to checkpoint itself. In this case, the message contains a World Wide Web style Universal Resource Locator (URL) which specifies where the checkpoint is to be written. This may be simply a file, an ftp site, or it may specify a *checkpoint server* which will be described later. When the URL does not specify a local file, the checkpoint is written directly to the network, and is never stored on the local disk. Avoiding the local disk operation allows checkpoints to occur at the maximum speed the network protocols permit. Instead of specifying a checkpoint destination, the message may tell the process not to checkpoint at all, or it may request that the process generate a new, CoCheck specific, URL from which another process may read its checkpoint. This last alternative provides a means of performing a direct process migration without the need to create an intermediate checkpoint file.

The checkpoint request message sent by the RM also includes a list of communication identifiers of processes which are also checkpointing. The checkpointing processes sends a *ready* message to each of these processes, and then waits for ready messages from all the other processes. Any other messages which are

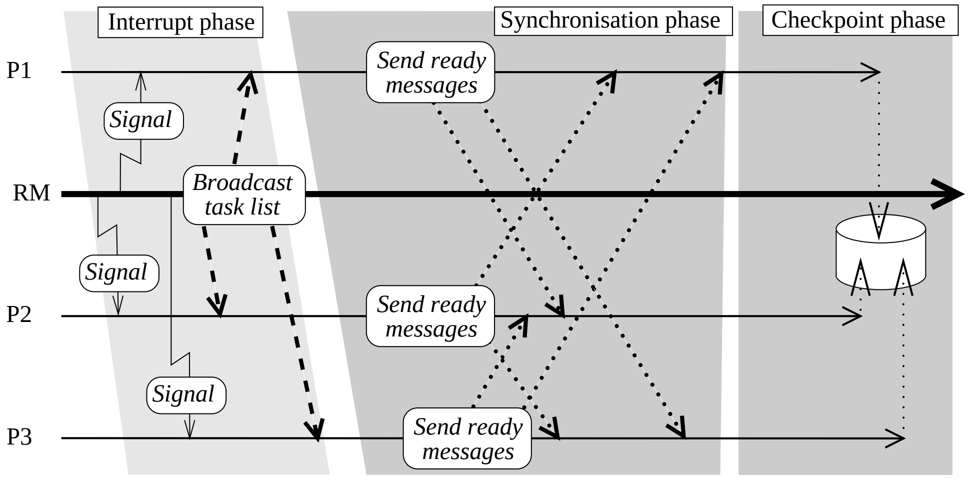

Figure 2: CoCheck's network cleaning protocol

received while waiting for readys are assumed to be part of the application's communication, and will be buffered in the process' address space so they become part of the process' checkpoint. With the provision that messages are delivered in order between any two processes, it can be assumed that the network has been drained when all of the ready messages are received. At this point, it is safe for each process to disconnect from the MPE (if the MPE requires it), and to invoke the Condor checkpointing library to save its state.

To restart from a CoCheck checkpoint each process is provided with a URL from which to read its individual checkpoint by the RM. Prior to performing a restart operation, the process connects itself with the MPE to establish a new communication identifier. It next performs a single process restart operation while preserving its new communication id. Following the restart, each process sends its new communication id to the RM process. The RM collects all of these ids from the restarting processes, and sends a mapping of old ids to new ids to each process. When this is received, it is installed in the CoCheck library, and all future communications will go through this mapping before passing into and out of the MPE implementation. In this way, processes are able to continue to use the original communication ids which were in use prior to the checkpoint. After the new mapping is installed, the application processes resume from the point at which the initial checkpoint notification was received.

2.3 CoCheck API

CoCheck was designed to be flexible in the number of processes to be checkpointed and in where the checkpoint is to be written. To leverage this flexibility, the interface to CoCheck must also be flexible. The basic interface to CoCheck is with the `GeneralCkpt()` function. `GeneralCkpt()` takes arguments for specifying three groups of processes: those that should be checkpointed, those that should block while the processes remain checkpointed in order to maintain a consistent communication identifier space, and those processes which should neither checkpoint nor block. This last group simply insures that the network is clear between itself and the checkpointing processes before continuing. The complement to the `GeneralCkpt()` function is `GeneralRestart()` which requires only specifying the first two groups of processes in order to get the checkpointed ones restarted, and to get the blocking ones new communication identifier mappings.

The checkpoint and restart functions are asynchronous remote procedure calls against the resource manager process. As with CARMI [10], they immediately return an integer request identifier. In this way, a process requesting a checkpoint need not block while the CoCheck protocol is running, and while the individual checkpoints are being stored. Processes may, though, include themselves in any of the set of processes defined by `GeneralCkpt()`. When a request is complete, the RM sends a completion notification message to the requesting processes.

Using these two functions as a basis, a variety of more special case checkpointing functions can be developed. For example, it is easy to design calls which checkpoint an entire parallel application or a single process. With only slight extensions, it has been possible to provide requests for migrating a single process, requesting a checkpoint to take place when another event occurs (such as a privately owned workstation being revoked), or allowing the user to specify where checkpoints should be written.

3 Methods of storing Checkpoint Files

When a scheduler makes a decision that an application must be checkpointed, it must also determine how that checkpoint will be stored. Checkpointing a parallel application creates a very large burst of data which must be stored reliably and as quickly as possible. This bursty pattern is exactly the set of circumstances under which most communication and storage systems perform poorly.

The tolerance to latency in performing a checkpoint will depend on the environment in which the parallel application is running. In a situation in which use of a resource may be revoked (such as for privately owned workstations), there is a degree of real-time constraint in saving the data. Condor, for example has a user configurable upper bound on the time allowed for a checkpoint when a workstation is reclaimed. If the checkpoint is not complete within this interval, Condor kills the job rather than waiting for the checkpoint to complete. In an environment where resources are completely under the control of the scheduling system there may be no hard constraint, but it is still very important to complete the checkpoint as quickly as possible in order to free the resources for other jobs. Time spent checkpointing is time when useful computation is not taking place.

The simplest, and perhaps most desirable method of storing a checkpoint of a parallel program is to simply use an existing distributed file system. Examples of these include the Network File System (NFS) [11] and the Andrew File System (AFS) [12]. Using these systems for storing checkpoints is quite attractive because it allows them to be stored in the same way as other files. The problem of where and how data is stored is handled by the file system. Unfortunately, these systems were not designed to perform well on operations which involve one time transfers of large files such as checkpoints.

NFS has stateless servers which handle file requests a single page at a time. This leads to poor performance because the file must be moved across the network via a series of page size requests to the server. AFS uses a more complex, full file caching scheme in which all files accessed are moved in their entirety between the server and client disks. Practice has shown that AFS is not adequate for parallel systems. For example, the Cornell Theory Center recommends that AFS not be used when data transfers become large [13]. The caching scheme used by AFS is particularly poor at writing results such as checkpoint files. These results generally will not be re-used on the node where they are generated, so caching them locally provides no future benefit, and in fact may cause other, useful data to be flushed from the cache. The AFS scheme also ends up causing two disk writes (one locally and one on the server) for the entire file. With fast interconnection networks, the latency of disk accesses becomes a bottleneck. A final difficulty with AFS is the inflexibility in placing file servers. AFS servers are considered insecure unless placed in a "locked room" to which users do not have access. This limits the ability to place AFS servers such that they will be close to the processes generating checkpoints.

3.1 Checkpoint Servers

Due of the perceived shortcomings of the existing solutions, we have developed a *Checkpoint Server* specifically suited for the problem of storing and retrieving checkpoints. The goal of the checkpoint server is simply to move data between the network and the local disk as quickly as possible. It is the scheduler's job to determine when a checkpoint should take place, and what checkpoint server should be used for storing which checkpoint files. When a checkpoint or restart is to be performed, the RM process, as the schedulers representative in the CoCheck protocol, starts by contacting the required servers to request a store or retrieve operation. The server responds by generating a URL on which it will transfer the checkpoint, and forks a child process to perform the transfer. The URL service prefix (e.g. "http:" or "ftp:") is unique to our checkpoint server, and is understood by the Condor checkpointing library (as described previously). This URL contains an Internet Protocol (IP) address and port number pair to perform a TCP transfer of the checkpoint. TCP is used because it is the fastest reliable protocol available in our network of workstations environment. In other environments, other transport protocols could be used by generating URL's with different service prefixes, and implementing them in the URL component of the Condor library. The checkpoint server uses a child process to perform the transfer to insure that it will be ready to receive the next service request.

Like other simple components, checkpoint servers can be combined to form more complex structures. A scheduling system can use multiple checkpoint servers as building blocks to provide good checkpointing performance. In putting the blocks together, one must consider a number of factors. Perhaps most important is the topology and characteristics of the underlying communication network. In a large, fragmented network with high latencies and low bandwidth, checkpoint servers should be scattered about to insure that any checkpointing process has as fast a link as possible to some checkpoint server. In a smaller, more tightly connected network, it may not be necessary to have many checkpoint servers since every potential checkpointing process will always have a fast path to a server.

One must also consider the characteristics of the checkpoint servers themselves. Particularly when attempting to reduce the total number of checkpoint servers, it is important to look at issues such as the bandwidth of the disk. When a fast network delivers many checkpoints to the same server, the disk will become the bottleneck. Also, the capacity of the disk is important. A server with a small disk should not be placed in a location where it will be expected to store many checkpoints. A scheduling system must understand these sorts of characteristics of its checkpoint servers, and schedule the checkpoint servers much like it would schedule compute or other resources.

A final consideration when deciding how to use checkpoint servers is how frequently checkpoint operations take place. In an opportunistic system such as Condor, the return of a single user may cause a multi-node parallel application to checkpoint. For this environment, it is worthwhile to allocate significant resources to checkpointing because they will be needed frequently. In all environments the frequency of checkpoint operations is going to be determined by the way in which the scheduler utilizes checkpointing.

A scheduler may trigger checkpoints periodically to provide fault tolerance. The degree of checkpointing in this case is going to depend on the scheduler's level of trust for its resources. When resources are reliable, the interval between checkpoints may be large, and there will be little load placed on the checkpoint servers. When the resources are less reliable, checkpoints may be taken more often in order to reduce the amount of computation lost due to a failure. A checkpoint may also be invoked based on the priority of jobs in a queue. There may be a preemption policy that running jobs will be checkpointed and replaced by newly submitted jobs with higher priority. Checkpoints may also be used to perform re-allocation of resources among running jobs to implement a dynamic partitioning strategy or, for example, to move processes which communicate frequently close to one another. In all of these cases, the variety of jobs is going to influence the frequency of checkpoints and therefore the level of checkpoint servicing required. It is therefore extremely important that the scheduler have flexibility in the number and placement of checkpoint servers.

Parallel schedulers also need to take the placement of checkpoint servers into consideration when they are allocating processes to compute nodes. Processes should be spread around the resources such that no single checkpoint server will be overloaded in case there is a need to checkpoint. Knowledge of the characteristics of the checkpointing infrastructure should be used. The scheduler must balance its desire to distribute checkpoints evenly with the application's need for high bandwidth and low latency communication which generally are achieved by clustering the application processes. Applications which do not do intensive communication may be scheduled based on the expected checkpointing requirements, while communication intensive applications may be scheduled to reduce application communication time at the cost of higher checkpoint times.

4 Experience with the deployment of checkpoint servers

As described in the previous section, before deploying checkpoint servers in our department, we had to understand the need for checkpointing services as well as the characteristics of our communication infrastructure. The Computer Sciences department at the University of Wisconsin has around 200 desktop workstations most of which are available to the Condor resource management system for executing long running sequential applications. Each of these workstations is also available to users via CARMI [10] the resource management and parallel programming interface to Condor. Condor has always supported checkpointing of sequential applications, and CoCheck has recently been integrated with CARMI to provide checkpointing services to parallel applications. Because checkpoints in this environment are triggered by owners returning to their workstations, they occur relatively frequently. We therefore require a checkpoint server architecture which can service numerous checkpoints.

The principle limitation in our environment, as in many other environments, is the available network bandwidth. Each of our workstations lies on an Ethernet class sub-net. Each Ethernet is connected to one or two routers which in turn directly connect each subnet to three to five other sub-nets as well as an FDDI backbone. The path between any two workstations, therefore, is at best at the Ethernet rate of 10 $\frac{Mbit}{sec}$, and may require crossing one or two routers. The department also has AFS available to all of the workstations, and the AFS servers are connected directly to the FDDI ring. We therefore wish to explore the alternatives in placing checkpoint servers on sub-nets

Checkpoint Route	Time to Checkpoint
Checkpointer and server on same sub-net	46
Checkpointer and server on separate sub-nets connected to the same router	64
Checkpointer and server on separate sub-nets with FDDI in between	79
Checkpointer on Ethernet, server directly connected to FDDI	49

Table 1: Times, in seconds, to write a 32Mb checkpoint file

as well as sharing the AFS servers which are directly on the FDDI ring.

Table 1 summarizes the results of experiments to determine how the network topology affects the time to write a checkpoint. In each of these experiments, a 32Mb checkpoint file was generated on a SPARC workstation running SunOS 4.1.3. The checkpoint files were received at checkpoint servers running on Dec Alpha workstations running OSF/1 V2.1. The results reported are the average of a number of checkpoint operations. In all cases, the variance in the time to checkpoint was low. As would be expected, placing the checkpoint server and checkpointing process on the same sub-net produced the best results. Placing the checkpoint server on FDDI performed nearly as well. In the tests where the checkpoint had to move off of one sub-net and onto another, the time increased markedly.

From these results, it seems that the most desirable method of placing checkpoint servers would be one per sub-net. In this way, every workstation will have the fastest available path to a server. There are two disadvantages to this. First, the number of sub-nets is large (approximately a dozen containing user's workstations), so many resources would have to be established as checkpoint servers. Also, although placing a checkpoint server on each sub-net will improve checkpoint times, to gain the same advantage at restart time would require re-scheduling a job on the same sub-net as when it last checkpointed. This severely limits the number of resources available for a restarting job. We wish to investigate ways to circumvent this problem by building hierarchies of checkpoint servers. A small checkpoint server could be placed near to the resource on which the checkpoint is taking place, but after the checkpoint is complete, it could be moved to some larger higher level server from which the restart will occur. This movement to the higher level server could take place off-line, when there is no immediate need for the checkpoint at any particular site.

Placing checkpoint servers directly on the FDDI ring appears to be nearly as desirable as having a checkpoint server per sub-net. In our environment, there are administrative barriers to this, but it appears that, in general, it would be wise to dedicate some resources on the highest bandwidth portion of a network to provide good checkpoint performance.

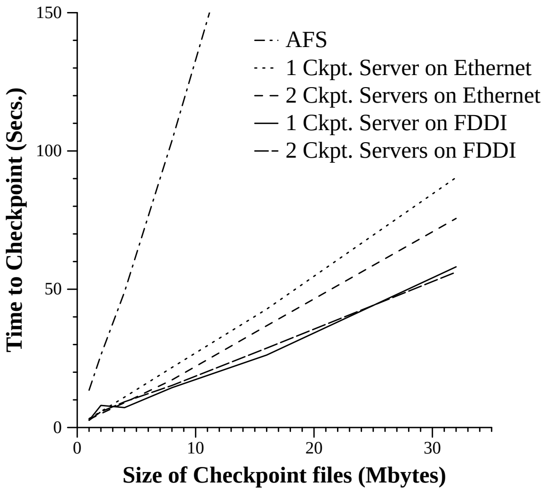

Figure 3: Time to checkpoint 2 processes of various sizes

Our second set of tests was intended to determine if checkpointing performance scales as the number of checkpointing processes, servers and size of individual checkpoints is increased. We also wanted to see exactly how well an existing file system, AFS, performs on these operations. Once again, our tests were limited by the bandwidth on our network. It is clear that no single checkpoint can occur faster than the bandwidth of a single sub-net, so we did not want any two checkpointing processes to lie on the same sub-net. This constraint limited us to checkpointing no more than two processes simultaneously. Figure 3 shows the results of different checkpoint server configurations and checkpoint sizes.

In all cases, the time to checkpoint scaled nearly linearly with the size of the checkpoint files. The most striking result is how poorly AFS performs for checkpoint operations. As mentioned previously, this is due to the method in which AFS caches files. As a checkpoint is being written, it is stored entirely on the local disk. When the file is closed, it is read off of the disk, and transfered across the network to the file server where it is written to the server's disk. This requires three disk I/O's as opposed to one for the checkpoint server. Nonetheless, it is surprising that AFS required

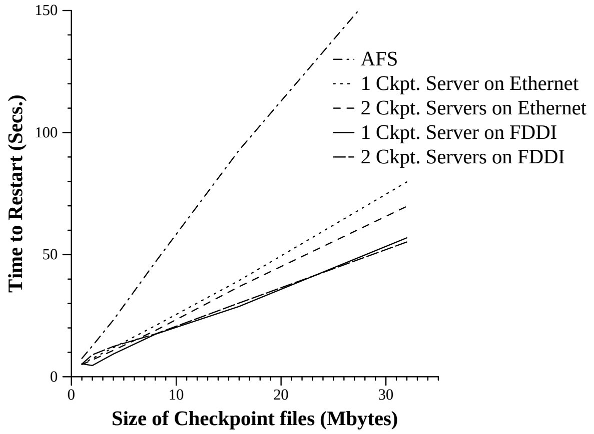

Figure 4: Time to restart 2 processes of various sizes

approximately an order of magnitude more time to produce a checkpoint.

In the checkpoint server cases, the results are what one might expect by extrapolating from the single checkpoint tests. Sending two 32Mb checkpoints to the same checkpoint server, takes almost exactly twice as long (90 seconds) as sending one checkpoint to one server. When sending to two checkpoint servers on separate sub-nets, we were constrained by our environment to go across the FDDI ring. The last of two checkpoints completed in virtually the same time as one checkpoint taking a route across FDDI. Sending two checkpoints to one server on FDDI took only slightly longer (58 seconds) than sending one checkpoint onto FDDI (49 seconds). This implies that the network is still the bottleneck in this operation, and that the processor and disk are still able to keep up. As more sub-nets feed the same FDDI connected checkpoint server, we would expect the disk to become the bottleneck. The fact that an additional server on the FDDI ring does not improve checkpoint performance further shows that the single server is not yet a bottleneck.

Figure 4 shows results of similar experiments for restarts. The same configurations of checkpoint servers and checkpoint sizes were used for the restart tests as for the checkpoint tests. The results for restarts are similar to those for checkpoints. Once again, AFS performs poorly, though restarts are significantly better than checkpoints. Typically, restart times are slightly higher than checkpoint times. This is due to the fact that in order to perform a restart, the executable file for the restarting process must first be moved to the executing machine. Executables are moved from the checkpoint server to the local disk of the executing machine using the same mechanism as checkpoint files.

5 Conclusions and future work

Parallel job schedulers are faced with an increasingly difficult task because the type of jobs and the types of resources are becoming more and more diverse. By providing schedulers with new techniques, such as checkpointing, we make it possible for more efficient schedules to be created. With each new technique, however, comes additional complexity of determining when and how to use it. For checkpointing, the problem is determining both when to checkpoint and how to most efficiently move large checkpoint images.

Today's methods of storing files on parallel systems, namely distributed file systems, do not provide adequate performance for storing checkpoints. These file systems are also implemented completely outside of the scheduling system, so the scheduler has very little means of controlling how they move data. By implementing checkpoint servers, we have given the scheduling system control over where and when data will be transferred. The scheduler can then treat the checkpoint servers like other resources which must be scheduled. Checkpoint servers provide the scheduler a great deal of flexibility in how checkpoints are stored. Techniques such as hierarchical checkpoint servers or striping a single checkpoint across multiple servers have not yet been investigated, but may provide higher levels of performance.

Initial experience with CoCheck has been very good, and the current work to support MPI with CoCheck is a good sign of its portability. Further experience with the checkpoint servers, and how best to utilize them is needed. Our existing testing environment is severely limited by the bandwidth of our network. We hope to gain further experience with the checkpoint servers on hardware which is dedicated to parallel processing and which contains a faster interconnect. Our department's Cluster Of Workstations (COW) which consists of forty dual processor SPARC workstations connected by a Myrinet is a likely target. The current obstacle to this is porting the Condor checkpointing library to the Solaris operating system which runs on these nodes. The simplicity of the checkpoint server should allow us to easily tailor it to use the best available communication protocol as we move to new hardware.

In addition to checkpoints, users' data sets must be distributed among the nodes of a parallel system. Integrating the distribution of this data into a scheduling system may allow faster start up of jobs. Instead of jobs waiting for the nodes to be loaded after being scheduled, the scheduler could load the data using techniques similar to the checkpoint server. This would require additional submit time information from the user specifying what data is needed on which node. Further integration with parallel I/O systems would also be desirable.

Acknowledgements

We wish to thank Georg Stellner of the Technical University of Munich for his initial design and collaboration during the development of CoCheck. The principle work on the implementation of the Checkpoint Server was done by Hsu-lin Tsao as part of a class project for Prof. Marvin Solomon.

References

[1] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor: A hunter of idle workstations," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 104–111, June 1988.

[2] M. Squillante, "On the benefits and limitations of dynamic partitioning in parallel computer systems," in *Job Scheduling Strategies for Parallel Processing* (D. G. Feitelson and L. Rudolph, eds.), vol. 949 of *Lecture notes in Compter Science*, Springer-Verlag, 1995.

[3] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, Feb. 1985.

[4] G. Stellner and J. Pruyne, "Resource management and checkpointing for PVM," in *Proceedings of the 2nd European Users' Group Meeting*, pp. 131–136, Sept. 1995.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guid and Tutorial for Networked Parallel Computing*. Cambridge, MA.: The MIT Press, 1994.

[6] G. Stellner, "CoCheck: Checkpointing and process migration for MPI," in *Proceedings of the International Parallel Processing Symposium*, IEEE, April 1996.

[7] M. J. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the Unix kernel," in *Proceedings of the Winter Usenix Conference*, (San Francisco, CA), 1992.

[8] T. Tannenbaum and M. Litzkow, "The Condor distributed processing system," *Dr. Dobb's Journal*, pp. 40–48, February 1995.

[9] J. Pruyne and M. Livny, "Providing resource management services to parallel applications," in *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing* (J. Dongarra and B. Tourancheau, eds.), SIAM Proceedings Series, pp. 152–161, SIAM, May 1994.

[10] J. Pruyne and M. Livny, "Parallel processing on dynamic resources with CARMI," in *Job Scheduling Strategies for Parallel Processing* (D. G. Feitelson and L. Rudolph, eds.), vol. 949 of *Lecture notes in Compter Science*, Springer-Verlag, 1995.

[11] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network file system," in *Proceedings of the Summer Usenix Conference*, pp. 119–130, 1985.

[12] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, pp. 51–81, February 1988.

[13] J. Gerner, "Input/output on the IBM SP2–an overview." `http://www.tc.cornell.edu/SmartNodes/Newsletters/IO.series/intro.html`.