CoCheck: Checkpointing and Process Migration for MPI

Georg Stellner
Institut für Informatik der Technischen Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation
D-80290 München
stellner@informatik.tu-muenchen.de

Abstract

Checkpointing of parallel applications can be used as the core technology to provide process migration. Both, checkpointing and migration, are an important issue for parallel applications on networks of workstations. The CoCheck environment which we present in this paper introduces a new approach to provide checkpointing and migration for parallel applications. In difference to existing systems CoCheck rather sits on top of the message passing library than inside and achieves consistency at a level above the message passing system. It uses an existing single process checkpointer which is available for a wide range of systems. Hence, CoCheck can be easily adapted to both, different message passing systems and new machines.

1. Introduction

Networks of workstations (NOW) [1] are a popular environment for executing parallel applications due to their overall availability and accessibility. But typically, parallel applications are heavily consuming resources which makes interactive work nearly impossible. Hence, it is important to vacate all processes of parallel applications, if the owner of the machine resumes interactive work either by migrating them to other hosts or checkpointing the parallel application and restarting it later on. As checkpointing also facilitates process migration (take a checkpoint, redistribute the checkpoint files and restart with a different mapping), migration is an optimized form of checkpointing with redistribution of processes and immediate restart. Therefore, throughout the rest of this paper we will refer to the term checkpointing as the primary operation, but meaning both checkpointing and process migration. Currently, all popular programming environments like NXLib [13], p4 [3], PVM [6] or MPI [11] do not provide checkpointing.

Process migration can also be used to accomplish an even load distribution among the nodes of the parallel application. This is particularly important, because otherwise the most loaded (and hence slowest) machine might determine the speed of the computation. On a parallel computer a node typically is slowed down when it has to perform more computation than others, i.e. the work is not evenly distributed. In a multi-user environment nodes may also lag behind because of other users' processes. In both cases the load imbalance can be resolved by migrating processes from overloaded to less loaded hosts.

Finally, parallel applications typically have long execution times, whereas failures in workstation environments are very likely. Hence, it is important to tolerate such failures by saving the state of the computation from time to time and resume execution from the latest state in case of a failure. Checkpointing is one technique to accomplish that automatically and transparently.

This paper introduces CoCheck, an environment that provides checkpointing transparent to the application for parallel applications on NOWs. As CoCheck sits on top of the message passing system, its approach can be easily adapted for different message passing systems. Due to the use of a single process checkpointer which exists for various machine types, CoCheck can be ported to different machine platforms.

The remainder of the paper is organized as follows. The next section illustrates the basic problem behind checkpointing. Then, an overview on similar existing systems is given. After that the basic ideas of our approach are explained. This is followed by a description of the implementation of CoCheck for MPI. Finally, we give a performance overview and conclude with an outlook on future research projects.

2. Determining Consistent Global States

We refer to a parallel application as a set of processes which cooperate by exchanging messages. In turn each process execution consists of a set of sequentially ordered events. Each event changes the local state of its process. Depending on the level of abstraction the events may for instance reflect the execution of machine instructions or subroutines. Special send and receive events represent message exchanges.

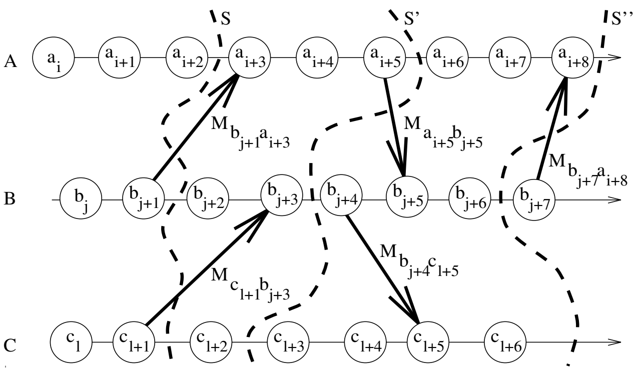

Figure 1. Global States.

The parallel application in figure 1 consists of three processes. The vertices denote events whereas messages are depicted as arrows. The labels of the messages refer to the sending and the receiving event respectively. The lines S, S' and S'' represent global states of the application, i.e. a view of the application that an external viewer might perceive. Such a global state comprises all local states and the messages in transit, i.e. those arrows which are cut by the line. Global state S for instance comprises the local states of the processes and the messages $M_{b_{j+1}a_{i+3}}$ and $M_{c_{l+1}b_{j+3}}$. Global states can be used to checkpoint and restart an application or migrate processes. Unfortunately not any global state, i.e. any cut, can be used. In the example above restarting from the global states S and S' would lead to correct, whereas restart from global state S'' would result in erroneous behavior: message $M_{b_{j+7}a_{i+8}}$ would be duplicated.

Chandy and Lamport examine under which circumstances a global state can be used for correct restart [4]. Such global states are called consistent. From the results obtained in [4] we can derive the following two properties, which are sufficient, so that the individual states (of both processes and the network), which are recorded independent of each other, form a consistent global state. First, if a receive event is part of a local state of a process, the corresponding send event must be part of the local state of the sender or has not yet occurred at all (avoids duplication)[1]. The global state S'' in figure 1 violates this condition. Second, if a send event is part of a local state of a process and the matching receive event is not part of the local state of the receiver, the message must be part of the state of the network (avoids loss of messages). The former property will be used unchanged whereas the latter will be further optimized for our purposes.

[1]The second part of this allows non blocking communication. It is important, to avoid global states where a message is already in transit, i.e. part of the state of the network, but the local state of the sender does not record that the send was already carried out.

3. Related Work

DynamicPVM [5] supports PVM applications and uses Condor [9] to create a checkpoint of the local state of a migrating process. Usually, PVM uses an indirect message transfer via daemons, i.e. a message is first transfered to the sender's local daemon, then forwarded to the local daemon of the receiver and finally delivered to the receiver. The migration in DynamicPVM is done in cooperation with the local daemon on both the old and the new host of the process and requires a message forwarding technique. Before migrating the process the local daemon on the new host sets up a message buffer for its new process. Then the routing information on the relevant daemon processes is updated and the process can checkpoint with the Condor checkpointer. After restart on the new host it contacts the local daemon and can receive messages again. Messages which are still being sent to the daemon on the old host are forwarded to the daemon on the new host and the sender daemon is informed about the new location of the process. In future sends to this process, the daemon of the new host will directly be used. In addition to the indirect message delivery PVM also supports direct message exchange via TCP connections between processes. DynamicPVM currently cannot migrate such processes. A further problem might occur due to message forwarding. PVM usually guarantees that a process receives messages from another process in the sending sequence. If for some reason the message being forwarded is delayed, it might be overtaken by a newly sent message which already is directly sent to the daemon on the new host. A further restriction is, that it is only possible to migrate one process at a time.

In contrast to DynamicPVM, MPVM [8] also supports PVM applications with direct TCP connections. The migration is handled in concert by the local daemons of the current host and the new host of the process, the process that ought to migrate and a skeleton process on the new host. At the beginning of the migration the local daemon sends a signal to the process to be migrated. In turn the process starts to close down all direct TCP connections to other processes. Special precautions based on acknowledgments insure that no messages are being lost. After that the process quits the virtual machine. At the same time a new skeleton process is started on the new host of the process. It will finally become the migrated process. After the complete state has been transfered to the new process via a TCP connection, the process re-enrolls in PVM and is assigned a new address. PVM itself takes care of reopening the TCP connections, because if there is no connection existing, it will be set up automatically. For messages that are sent via the indirect route over the daemons, two additional things have to be done: providing a mapping table that maps the addresses the application uses to their current values and forwarding messages to the new process. The update of the mapping tables which are maintained on each daemon and the message forwarding technique are similar to the approach in DynamicPVM. Additionally, a message sequencing scheme has been introduced in MPVM to insure PVM's property of receiving messages in their sending sequence.

Although MPVM solves most of DynamicPVM's problems there is still the drawback that both require changes to the PVM library and daemon. This makes it difficult for both environments to keep up with steadily upcoming new versions of PVM. In addition both approaches only seem to be suitable for PVM as they highly depend on its implementation. Finally, neither of the above currently allows checkpointing.

4. CoCheck: Basic Concepts and Properties

In addition to designing a checkpointing facility for MPI, a further goal of CoCheck was to achieve portability. Apart from supporting various workstation platforms the basic approach should be easily adaptable to different message passing libraries. First, because checkpointing a running process is inherently machine dependent and difficult to implement, CoCheck should be layered on top of a portable single process checkpointing mechanism such as [7, 9]. Second, an implementation which is integrated into the message passing systems reduces the suitability for other message passing libraries. Consequently, CoCheck should be implemented on top of the message passing library rather than inside. A further decision was to start with global checkpointing, followed by improvements and optimizations to facilitate process migration.

An important issue from the applications point of view is transparency of checkpointing, i.e. the application must not be aware that either an application has been restarted from a checkpoint or some of its processes have been migrated. Particularly, it is inevitable that the application can use the same addresses for the processes throughout its lifetime. Hence, we introduced *virtual addresses*. This means that the address a process gets when it is started is used as the address in the application, whereas CoCheck actually uses the current addresses of the processes. Therefore, a mapping from original to current addresses has to be maintained. As we do not make any assumptions about the availability of additional daemon processes, we decided that each process of the parallel application has its own copy of the mapping table. Upon migration these tables have to be updated. To hide all the additional functionality like virtual addresses or message buffering (explained below), we are providing wrapper functions for all calls of the message passing library. When the user links his application these wrappers are used instead of the original functions. The wrappers in turn call the original functions to perform the corresponding service. The wrapper functions add additional functionality where necessary. In the framework of MPI the profiling interface [11] can be used, whereas for other message passing libraries the names of the calls are directly renamed within the library by a tool, so that no source code modification is necessary.

A major problem were the messages which are being in transit when a checkpoint should be taken or a process should be migrated. Messages can be found in various places such as the physical network, operating system buffers or buffers of the message passing library (which in turn can be located in an additional daemon or locally within the processes or both). Moreover, it is difficult to access these locations and to capture the state of the network to get a consistent global state of the application. In addition, accessing the messages in these places would also violate the requirement that the implementation of CoCheck sits on top of the message passing library and not underneath it. If the message passing library preserves the sending sequence of the messages when they are received, the following solution can be applied to get around this problem.

As in [4], we also assume that there is a unidirectional communication channel from one process of the application to all the other processes. In contrast to this approach where the state of the network is part of a checkpoint, we achieve a situation where no more messages are in transit. Therefore, we use special messages, so called "ready messages" (RM), to clear each channel. Each process in the application gets a notification that a checkpoint has to be taken or a set of processes must be migrated. It is important that this notification is not processed while a send operation is in progress. Otherwise the consistency of the global state could be violated. Consequently, the send operation has to be atomic, which is also achieved in the wrapper functions for the send calls. Each process then sends a RM to all the other processes. Besides, each process receives any incoming message. If it is a RM from a certain process, then it can be concluded that there will be no further messages from that process, i.e. that this channel has been cleared. If the message is not a RM, it is stored in a reserved buffer area of the process. This receive operation anticipates the receive operation of the application.

Figure 2 shows the effect of the RMs: all messages in transit before the RM are delivered to the process and buffered there. After restart receive calls of the process will first try to extract a matching message from that buffer. When finally a process has collected the RMs from all other processes, it can be sure that there are no further messages on the network which have its address as destination. All messages that were in transit are safely stored in the buffer and are available for later access. Hence, the process can be seen as an isolated single process for which it is easy

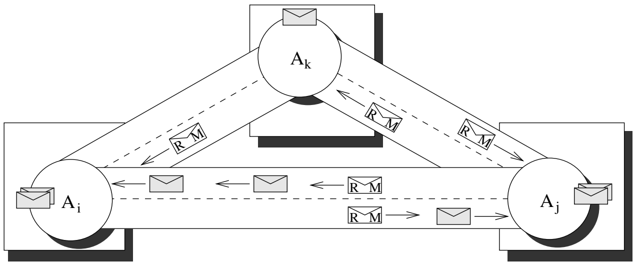

Figure 2. Clearing communication channels.

to determine its current state and create a checkpoint with the single process checkpointer. As no messages are being transmitted at checkpoint time, there is also no need to forward messages after restart.

After restart the address mapping tables have to be updated with the new current addresses of the processes. The processes can then continue with their execution. When they are executing a receive operation, the receive first has to check the buffer for a matching message. If there is such a message stored, it is retrieved from the buffer and returned. Otherwise, a real receive operation intercepts the next matching message from the network.

Taking into consideration section 2, it is obvious, that this approach achieves a consistent global state. First, the successive (anticipated) receive operations capture any incoming message. As a result there is a corresponding (artificially introduced) receive operation (anticipated receive) for any send operation which is part of a local state of a process. Upon restart the receive requests are first served from the buffer with the messages being retrieved in the arrival order, then from the network. For the process this actually looks like the messages are directly received from the network. So far this avoids loss of messages. The checkpoint notification is either processed before a send operation or after it has completed. Hence, if the local state of process comprises a receive operation (user receive or an anticipated receive) two cases are possible. First, if the checkpoint signal is processed after the send completed, then the global state also comprises the corresponding send operation. The corresponding message has been transfered to the receiver by the RM. Second, if the checkpoint signal has been processed before a send, then there is obviously no message that must be stored. After restart a (possibly ongoing) receive operation (including non blocking operations) will continue just as if nothing had happened.

The approach described so far assumes one special process which serves as a coordinator of the activities. As this can either be a special system daemon being responsible for resource management or dynamic load balancing, or a user command triggering checkpointing or migration, this is no restriction. The outline of the algorithm now is as follows:

1. **central instance:** send notification to all processes
2. **each process:** send RM to all other processes
3. **each process:** receive incoming messages until all RMs have been received
   - store user messages in buffer & count RM
4. **each process:** disconnect from parallel environment
5. **each process:** create checkpoint or migrate

The restart can be summarized as follows:

1. **central instance:** restart all processes
2. **each process:** reconnect to parallel environment
3. **each process:** send new address to central instance
4. **central instance:** collect new addresses and distribute the new mapping table to all processes
5. **each process:** resume execution (using the new addresses and the messages in the buffer area)

This algorithm has been successfully implemented and tested with different single process checkpointers for PVM and proved to be viable and stable [12, 14]. In the meantime also reasonable performance could be achieved [15].

5. Implementation of CoCheck for MPI

As the standard for message passing programming is MPI we decided to use it to demonstrate the suitability of the CoCheck approach for a different message passing system. At that time it turned out that all publicly available MPI implementations were based on another message passing library or part of a larger environment. Hence, we decided to first bring up our own native MPI implementation. This could be done in reasonable amount of time as much code could be borrowed from MPICH [2] for the higher level MPI functionality and NXLib [13] for the communication part.

5.1. Implementation of tuMPI

Primarily tuMPI (Technische Universität München MPI) should be a research environment to investigate dynamic load balancing based on process migration. Other goals like efficiency or support for heterogeneity were not as important or left open for future work.

The process structure of a small sample tuMPI application is shown in figure 3. It consists of two application processes (AP) and a daemon (DP) which comprises three major components: the starter, the locator and the grouper. The starter is used to start APs, whereas the locator is an address server. It knows about the exact whereabouts of all APs. Finally, the grouper is used to manage MPI groups and contexts. Every AP has a local address table which is filled when the application is performing send and receive operations. TCP connections to other APs are set up on

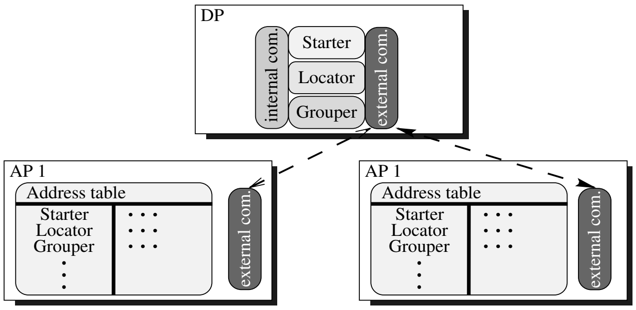

Figure 3. Process structure of tuMPI.

demand with the assistance of the locator. The mapping of MPI addresses to TCP addresses is entered in the mapping table of the two APs that start to communicate and the mapping table of the locator. With this scheme all APs that need to exchange messages are fully connected via TCP sockets.

5.2. CoCheck for tuMPI

A basic property of MPI, namely that multiple calls to the enrollment function `MPI_Init` are undefined [11], made it impossible to go through the cycle of disconnecting and reconnecting in the checkpoint protocol. At this point, CoCheck had to be integrated into the sources of tuMPI. The same consequence will also hold for other MPI implementations in concert with CoCheck. Finally, as the tuMPI sources had to be modified anyway, the central instance of CoCheck was integrated as an additional component in the DP.

The algorithm to create a consistent state was optimized for process migration in the following way. After the notification about a migration has been delivered, the migrating process (MP) sends a RM to all its communication partners. All current communication partners can be determined from the local mapping table. All APs which have received a RM in turn send a RM back to the migrating process. Again until MP has received all RMs, incoming users messages are buffered. When MP has collected all RMs, it uses the Condor checkpointer to migrate to the new node by transferring local state over a TCP connection to a skeleton process (SP) on the new host. Figure 4 summarizes the migration of MP.

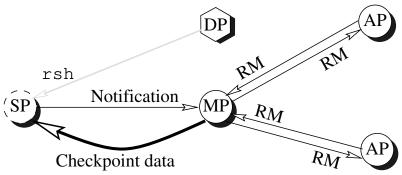

Figure 4. Migrating a tuMPI AP with CoCheck.

A slightly modified protocol can be used to take a checkpoint of the complete application. in that case, the notification is sent to all APs and instead of writing the checkpoint directly into SP, each process stores its checkpoint on disk.

5.3. Further Assumptions

The current implementation assumes that collective communication is based on point-to-point communication. In this case the CoCheck protocol perceives the collective communication just as code written by the user and no further precautions are necessary to guarantee correct behavior after restart.

Currently the semantics of synchronous sends can be changed by a checkpoint: an anticipated receive and not the actual receive of the application may trigger the return of the send. But this can be changed by a slight modification to the wrappers, where an additional synchronization is introduced at the end of the synchronous send and receive operation.

6. Performance of CoCheck with tuMPI

A performance criterion for process migration is the time it takes to move a process from one machine to another. This is important for dynamic load balancing and for vacating a machine because an interactive user reclaims it. In the first case, the migration time suspends the computation of the whole application, thus increasing the total runtime. As the goal of dynamic load balancing is the reduction of runtime the migration time must be kept minimal. In the latter case, the time it takes to migrate the process to the new machine is the time an interactive user is potentially still bothered by a process of a parallel application.

The measurements were done on a cluster of eight machines consisting of Sun SparcStation 2 and Sparc 10. The process migrated between Sparc 10s. The application we have used is a synthetic benchmark program that allows one to select a certain main memory size and a computation/communication ratio. Although the experiments were done during the night there was still other user activity on the Ethernet which was used as interconnect. Apart from system daemons the machines themselves were reserved for the experiments.

The results of the migration of a single process are shown in figure 5. Obviously the migration time is independent of the number of processes in the application. In contrast to that, the size of the process image is the dominating factor of the migration time. It increases linearly with the memory size. In addition it turned out that the communication/computation ratio of the application had no influence on the migration time.

A linear regression of the performance data gave the approximation shown in equation 1 for the migration time

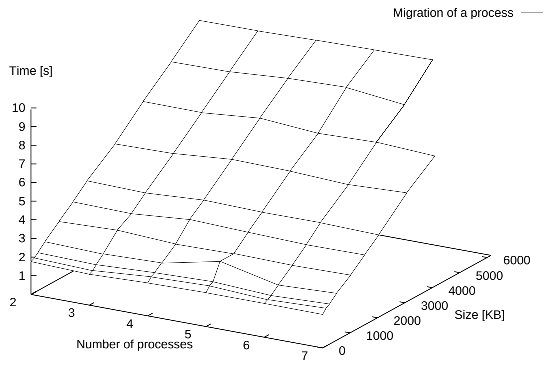

Figure 5. Migration time of a process.

$t(x)$ of a process with memory size x. The assumption that $763\frac{KB}{s}$ is the achievable TCP bandwidth on our Ethernet was corroborated by a ping-pong test, where we obtained TCP bandwidth values between 740 and $790\frac{KB}{s}$.

$$t(x) = 1.77s + \frac{x}{763\frac{KB}{s}} \qquad (1)$$

7. Conclusion and Future Work

The basic idea of CoCheck is to achieve a global consistent state where no messages are part of the state of the network. Then, all processes can independently of each other create a single process checkpoint. It is possible to either create a checkpoint of the complete application on disk or to migrate a subset of processes. In the case of migration the checkpoint is directly transfered to a skeleton process on the new host. Implementations for both tuMPI and PVM have been done. During the adaptation for MPI it turned out that modifications to the sources were only necessary, when the processes re-enroll after restart. Apart from that porting CoCheck to MPI was not more difficult than porting any other parallel application. To obtain the software refer to the CoCheck homepage:

http://wwwbode.informatik.tu-muenchen.de/CoCheck

The performance of migration is determined by the main memory size of the process to be migrated. Other factors, like the number of processes or the communication/computation ratio proved to be negligible. The major limiting factor is the speed of the underlying network. As a consequence, high-speed interconnects could further reduce migration times.

Future research will be going in several directions. First, CoCheck's usability as a global scheduler for resource management systems will be evaluated. Therefore, an integration of CoCheck into existing resource managers is planed. The second important research topic is load balancing. Components will be added to the system such that it is capable of performing load detection on the machines in the network and deciding about which process should migrate to what machine. Finally, CoCheck will be incorporated in THE TOOL-SET [10], where it will provide checkpointing, dynamic load balancing and cyclic debugging

References

[1] T. E. Anderson et al. A case for NOW. *IEEE Micro*, 15(1):54–64, Feb. 1995.

[2] P. Bridges et al. *Users' Guide to* `mpich`*, a Portable Implementation of MPI*. Argonne National Lab., July 1995.

[3] R. M. Butler and E. L. Lusk. Monitors, messages and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, Apr. 1994.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computer Systems*, 3(1):63–75, Feb. 1985.

[5] L. Dikken. DynamicPVM: Task Migration in PVM. Technical report, Shell Research, Amsterdam, Nov. 1993.

[6] A. Geist et al. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, MA, 1994.

[7] Genias Software GmbH. *CODINE User's Guide*, 1993.

[8] R. Konuru et al. A user-level process package for PVM. In *Proc. SHPCC*, pages 48–55, Knoxville, TN, May 1994. IEEE CS Press.

[9] M. Litzkow et al. Condor — A Hunter of Idle Workstations. In *Proc. 8th Int. Conf. on Distributed Systems*, pages 104–111, Los Alamitos, CA, 1988. IEEE CS Press.

[10] T. Ludwig et al. THE TOOL-SET — An integrated tool environemnt for PVM. In *Proc. 2nd Euro. PVM Users Group Meeting (Short Papers)*, Lyon, Sept. 1995.

[11] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, May 1994.

[12] G. Stellner. Consistent Checkpoints of PVM Applications. In *Proc. 1st Euro. PVM Users Group Meeting*, 1994.

[13] G. Stellner et al. NXLib — A parallel programming environment for workstation clusters. In C. Halatsis et al., editors, *Proc. PARLE*, volume 817 of *LNCS*, pages 745–748, Berlin, July 1994. Springer.

[14] G. Stellner and J. Pruyne. Providing Resource Management and Consistent Checkpointing for PVM. In *Proc. PVM Users Group Meeting*, May 1995.

[15] G. Stellner and J. Pruyne. Resource Management and Checkpointing for PVM. In *Proc. 2nd Euro. PVM Users Group Meeting*, pages 131–136, Lyon, Sept. 1995. Editions Hermes.