Consistent Checkpoints of PVM Applications

Georg Stellner
Institut für Informatik der Technischen Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation
D-80290 München
Tel.: ++49-89-21052689, Fax:++49-89-21058232
stellner@informatik.tu-muenchen.de

1 Introduction

Currently PVM is the standard for developing parallel applications in workstation environments. One of its goals is to use the computational power of idling workstations. In practice many users refrain from opening their machine to other users' PVM processes. This is due to their experience that such a process, which usually requires a lot of resources (CPU and memory), increases the response time for short interactive commands, i.e. reduces his own productivity. In the following, we describe an approach which enhances existing queuing systems with a support for parallel PVM applications and allows the migration of PVM processes to other machines. Hence, users who start to work interactively on their machine are no longer bothered by resource consuming PVM processes.

2 Load Leveling with Queuing Systems

The interaction of the user with the queuing system is based on the notion of jobs. Formally, a *job* is the obligation of the queuing system to schedule a program for execution on any machine as soon as the required resources are available. Between the start and the termination of the program it is not necessary that it is constantly executing. Moreover, the queuing system can suspend the execution at any point and reschedule the program later. An important issue is that the basic scheduling algorithm is fair, e.g. the execution of all similarly prioritized jobs also progresses similarly with the goal to finish them as soon as possible.

Based on a complete overview on the current load situation of all machines in the cluster the queuing system can schedule the jobs on the least loaded machine. During the execution of a program the load on the machine might increase due to new processes from other users or might become interactively used. In this case the execution is suspended and the job is migrated to another, less loaded or idling machine. Usually the migration is based on a checkpoint which is created when the job must vacate the machine. Each checkpoint file is assembled from the executable and a core-dump of the executing process [6]. The checkpoint can be used to restart the job later on. This can also be done on any other machine in the cluster as long as it is binary compatible to the former host of the process. The queuing system Codine transports the checkpoint file via a standard network file system like NFS or AFS to the machine where the job will be executed next [4]. Thus, the load is dynamically balanced among the machines in the cluster, shifting part of work from heavier to lighter loaded nodes.

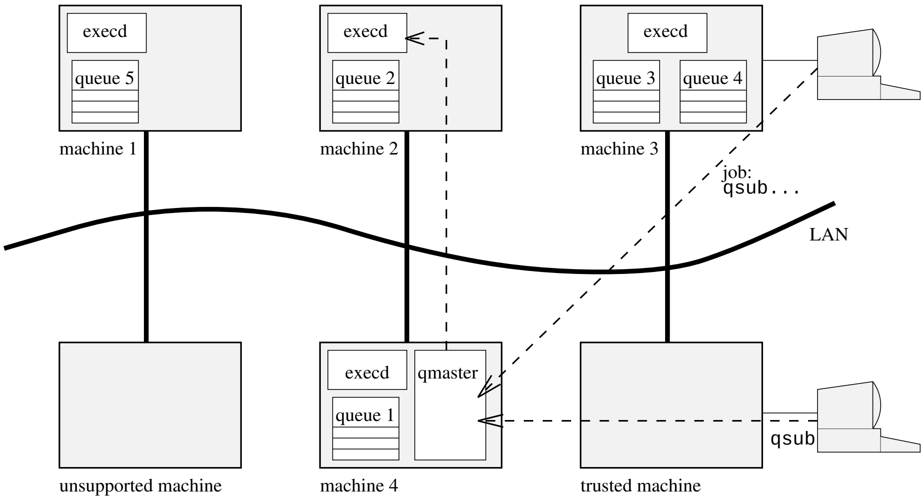

Figure 1: Cluster of workstation with the queuing system Codine

Figure 1 shows the main components of a cluster with Codine installed. The master daemon (qmaster) forwards submitted jobs to a queue matching the specified requirements. Local execution daemons (execd) remove single jobs from the queues and start the corresponding processes. In case of an occuring overload on a machine the execution daemon suspends the currently executing process and the master daemon finally decides about the migration of the job. Certain machines in the cluster (trusted machines) can be configured as sole clients: jobs can be submitted from them, but are not executed on them.

Similarly to the migration of a job, which consists of a single process, parts of a parallel application could be shifted to new machines when interactive work starts on some workstations. Current implementations of queuing systems only support jobs which consist of a single sequential process. Parallel application with multiple execution threads cannot be submitted.

For two reasons a combination of both approaches seems to be very desirable. Only, if the users are not influenced in their personal work they are willing to open their machine to other users, so that in turn they can benefit from the idle cycles on that particular machine. On the other hand, parallel applications increase the number of processes in the cluster. Hence, the task of the scheduler of the queuing system to distribute the load evenly among the machines becomes easier.

3 Integrating PVM Applications in Queuing Systems

3.1 Supporting Parallel Jobs

A first step in combining PVM and queuing systems is an extension to the concept of jobs. In addition to jobs consisting of a single process, jobs with multiple parallel execution threads must be managed (*parallel job*). The scheduling algorithm which is used must respect that either all of the processes of a parallel job are in execution or none of them (gang scheduling [3]). This technique avoids unnecessary waiting conditions for results from processes which are currently suspended. The efficiency of a parallel application is hence preserved.

3.2 Consistent Checkpoints

As presented in section 2 checkpoint files are the basis of the process migration facility in Codine. Nevertheless, it is not sufficient for parallel jobs that each process takes its own checkpoint independent of all the others. Messages which are exchanged between two processes might get lost or become duplicated in this case due to the time they spend on the network.

A *consistent checkpoint* is a set of checkpoints of all processes of a parallel application which insures that the application can be restarted correctly. Particularly, a consistent checkpoint guarantees that no messages are dropped or duplicated. One method of creating consistent checkpoints is introduced in the next section. A more formal definition can be found in [1].

3.3 Implementation

Submitting a parallel job in our environment can be done in two different ways. Firstly, jobs can be clustered when they are submitted, e.g. the user explicitly specifies all processes belonging to the parallel job. The queuing system treats all these processes as a management unit and takes this into account for its scheduling policy. For complex parallel jobs this is not flexible enough. Consequently, we have introduced a second way to submit parallel applications. Usually one process of the PVM program acts as the master and creates the other processes dynamically. The ancestor relation can be used to define the set of processes in a parallel job by allowing that the number of processes in a job is not fixed but may increase and decrease during run time. Starting with a job which consists only of a single process all further processes that will be created by this process are added to the parallel job. For PVM a call was implemented that substitutes the original process creation call (`pvm_spawn`) with a new one allowing a dynamically varying number of processes in a parallel job. The process creation is thereby removed from the PVM system and shifted towards the queuing system.

Operating system objects like open files or signal handlers are mostly stored in system address space. A core dump does not comprise these address regions. Nevertheless, this information is necessary to insure a correct behavior of the program after a restart. Hence, the information must be duplicated in user address space to be stored in a core dump. *Overlayed functions* are used to accomplish this by substituting the original call with an enhanced version which logs the necessary information and exchanges parameters.

A strictly synchronizing approach has been implemented to create consistent checkpoints, which allows a straight-forward integration into the existing queuing system. Optimistic protocols require several checkpoint files and need to determine a consistent subset of files to restart the application. In the worst case this is the initial state of the whole application and hence, this approach cannot be used to dynamically balance the load. Logging messages on the other hand would reduce the performance of the message-passing calls which is not desirable because this decreases the efficiency of the parallel application.

The mechanism to create consistent checkpoints in our environment is divided up in three subsequent phases: the synchronization phase, the process termination phase and the checkpoint phase. The restart is done during the reconstruction phase. All four phases are briefly explained in the following and their timing is presented in figures 2 and 3.

Synchronization phase. It is initiated by the interception of a checkpoint signal (SIGTSTP) by one of the processes of the parallel application. Usually the queuing system delivers this signal to a process if a periodic checkpoint should be created or the process should vacate the machine. The receiving process becomes the *initiator* in the following protocol sequence. The initiator in turn requests the queuing system to synchronize all the other processes in the application. A synchronization signal (SIGWINCH) is then delivered to the processes of the parallel job.

Process termination phase. After all the processes have intercepted the synchronization signal they start to prepare their checkpoints individually. Therefore, it has to be insured that no messages are still being transfered between any of the processes. All processes send a specially tagged message (*ready message*) to each other. After receiving the ready message from all the other processes of the parallel job, a process can conclude that there are no further messages on the net, as the PVM systems insures the delivery of the messages in the sending sequence. Messages which arrive at the process before the last ready message has been received are buffered. After the processes have been restarted receive operations are served with the buffered messages first. The send and receive operations have been enhanced in this way with the overlayed function technique briefly described above. After each process has received all ready messages

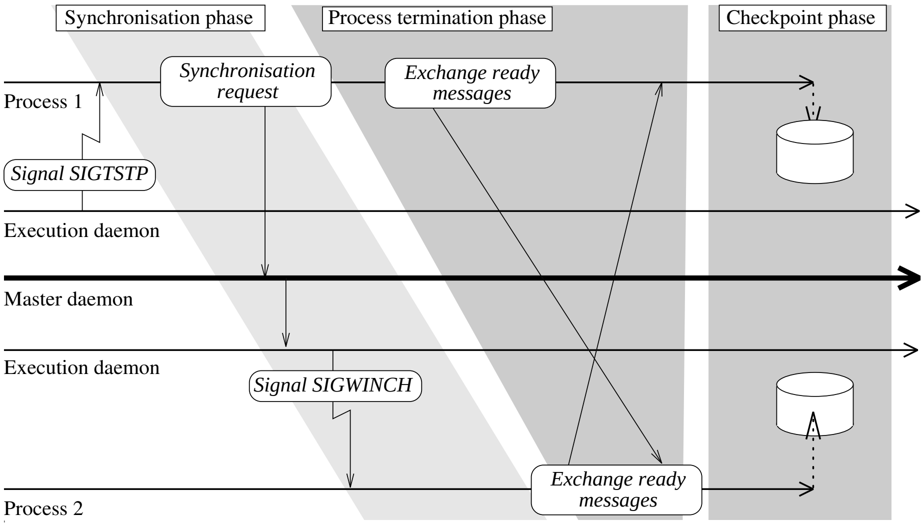

Figure 2: Timing of the synchronization, the process termination and the checkpoint phase

it quits the membership in all PVM groups and finally disconnects itself from the PVM system.

Checkpoint phase. During this phase all processes of the parallel application create independently their checkpoint file with the mechanism which is usually provided by the queuing system.

Reconstruction phase. The restart of an parallel job is steered by a modified startup routine (`crt0.o`). Firstly, all operating system objects are reconstructed based on the information which has been stored in the checkpoint file. After that, each process has to reestablish the PVM environment by enrolling at the local PVM daemon. This results in a new identifier for each process. To complete the PVM environment of each process it rejoins all groups it belonged to and forwards its new identifier to the queuing system. There, all new identifiers are collected and a mapping table is set up, which is in turn broadcasted to all processes of the parallel job. The identifier mapping table can then be used in the overlayed communication calls of PVM to map the old identifiers which the application uses to address communication partners to the new, now valid identifiers. Hence, the new identifiers are kept hidden in the system and the migration is completely transparent for the user.

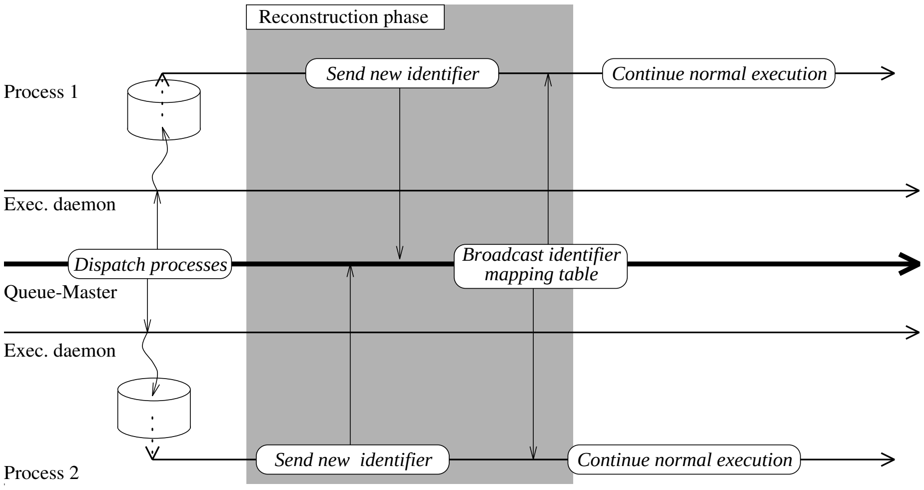

Figure 3: Timing of the reconstruction phase

4 Protocol Performance

A research prototype of the software environment has recently been finished. It has been tested on three DEC Alpha AXP 300 with OSF/1 1.2A and PVM 3.1. Creating a consistent checkpoint (first three protocol phases) of a communication intensive application requires between 5 and 15 seconds depending on the size of the process images. The reconstruction phase needs another 2 to 5 seconds which means that the pure protocol overhead for our test application is in the worst case about 20 seconds. The times which Codine needs to reconfigure its data structures have to be added to those figures.

5 Related Work

Dikken et al. [2] present a migration facility for PVM processes which has been integrated into the PVM daemons. Hence, only the slower indirect communication via the daemons can be used in PVM application which should benefit from the load balancing. The integration into the PVM system requires to integrate the checkpointing code into every new PVM version.

The UPVM environment for PVM applications is suggested by Konuru et al. [5]. Here, the PVM tasks are mapped onto lightweight UPVM processes, which can migrate within the cluster. The memory which was allocated on a machine for an executing UPVM process is still occupied after its migration to another machine, where, a corresponding slot had been reserved during the start of the application. Hence, the number of processes of a UPVM application is limited.

6 Conclusion

The experiences gained with the implementation of the research prototype, which integrates PVM and Codine, will be used to improve future versions of the software. One focus thereby will be laid on performance improvements. Currently all processes write their checkpoint files via NFS to the file system which means that a large number of bytes have to be serially transfered over the network. In addition, our protocol, which guarantees consistent checkpoints, synchronizes the creation time of the checkpoints hence creating burst network traffic. This leads to an additional decrease of network performance. In future versions local file systems will be used to store the checkpoints. This allows to write the checkpoint files in parallel and reduces the network traffic.

A further reduction of network traffic and performance improvement can be achieved tuning the synchronization protocol where currently $O(n^2)$ messages have to be exchanged to synchronize n processes. This amount can be reduced by restricting the ready messages to those processes which communicate with each other. For many parallel applications the number of interacting processes is restricted to a constant number of neighboring processes. Hence, only $O(n)$ messages have to be exchanged to insure consistency.

Finally, research will be undertaken to investigate the suitability of queuing systems to solve the needs of programmers for process creation and management. Therefore, an interface between the queuing system and PVM has to be defined. The interface will include functions to provide dynamic load balancing methods based on consistent checkpoints.

References

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

[2] Leen Dikken, Frank van der Linden, Joep Vesseur, and Peter Sloot. DynamicPVM. In Wolfgang Gentzsch and Uwe Harms, editors, *High-Performance Computing and Networking, International Conference and Exhibition Volume II: Networking and Tools*, volume 797 of *Lecture Notes in Computer Science*, pages 273–277. Springer Verlag, Berlin, April 1994.

[3] R. Esser and R. Knecht. Intel Paragon XP/S - Architecture and Software Environment. Technical Report KFA-ZAM-IB-9305, Forschungszentrum Jülich GmbH, Zentralinstitut für Angewandte Mathematik, D-52425 Jülich, April 1993.

[4] Genias Software GmbH, Erzgebirgstr. 2B, D-93073 Neutraubling, Germany. *CODINE User's Guide*, 1993.

[5] Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole. A user-level process package for PVM. In *Proceedings of the SHPCC'94*, pages 48–55, Knoxville, TN, May 1994. IEEE, IEEE Computer Society Press.

[6] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In *Proceedings of the USENIX Winter Conference*, San Francisco, CA, January 1992.