A Flexible State-saving Library for Message-passing Systems

A. Clematis
IMA - CNR
Via De Marini, 6
16149 Genova (Italy)

G. Deconinck
K.U.Leuven
Kardinaal Mercierlaan 94
B-3001 Heverlee (Belgium)

V. Gianuzzi
DISI -Università
Via Dodecaneso, 35
16146 Genova (Italy)

Abstract

Message passing applications on distributed computer require tools to integrate state-saving and rollback, to support dynamic program reconfiguration, fault tolerance and others. This paper presents the results of integrating two independently developed tools that combine flexibility and portability. The User-Triggered CheckPointing (UTCP) provides checkpointing and recovery while relying on the programmer to indicate the position of the recovery-line and the contents of the checkpoint. The tool PVMsnap provides an extension to PVM to obtain a consistent cut of the message-passing application. The combination of both tools results in a portable and flexible solution for fault tolerance which can be adapted to the applications' needs.

1. Introduction

Clusters of heterogeneous computers are increasingly being applied to solve scientific, long-running problems, and the issue of how to handle failures, possibly occurring in these collections of machines, cannot be ignored. Fail-stop failures can be masked using two major schemes: independent or co-ordinated checkpointing, followed by rollback. In the first case processes save their state independently, such that a (possibly non consistent) global state is recorded. Message logging or an additional communication protocol are then required to obtain a consistent state. In the second case, a consistent recovery line is obtained by synchronising the user processes, so that no inter-process communication is in transit through this line.

Checkpointing algorithms can be implemented at different abstraction levels i.e. at system-level, where memory binary images, including supporting kernel structures such as stack, heap and registers are saved, and at user-level, where the content of each checkpoint is indicated by the user process itself and where only information visible to the user is recorded.

System-level checkpointing is usually performed transparently to the user, by means of tools which are triggered periodically. Problems of this approach are the large size of the checkpoints and the need of a homogeneous network of machines.

User-level checkpointing tools requires more effort from the user: the application structure must be defined following some restrictions, checkpoint positions and recorded data must be defined so that consistency is not a worry.

If the possibility of working on heterogeneous networks and portability on different environments are considered as important goals, user-level checkpointing scheme should be preferred, since it offers three main advantages: tool portability, since no system dependent details are required; hardware independence of the checkpoint, which allows the tool to run on heterogeneous systems; smaller checkpoint size, since only significant variables have to be saved.

The ideal choice would be a tool supporting both user-level and co-ordinated checkpointing in order to reduce efforts required to the user, such as to find or to introduce synchronisation points inside her/his application, and to be applied also in non-symmetrical applications.

Looking at co-ordinated checkpointing schemes two components can be isolated: the protocol needed to define the consistent cut line and to detect in-transit messages, and the mechanisms needed to save and restore the global checkpoint.

This paper describes an experience performed within a collaboration between researchers at DISI-University and IMA-CNR Genova (I) and the Katholieke Universiteit Leuven (B). We started considering two independently developed systems, having different functionalities: UTCP, a non-co-ordinated user-triggered distributed checkpointing system implemented on top of Parix, and PVMsnap, a

0-8186-8332-5/98 $10.00

PVM-embedded implementation of a consistent cut protocol. The aim was to have an integrated system working on heterogeneous network of workstations, supporting the checkpoint-rollback technique for PVM applications.

The most important design goals of both systems were the same. First of all to provide the user a visible interface with the underlying system (user-triggered checkpoint and PVM receive functions extended to point out in_transit messages). Additionally, to maintain a high portability. Although the checkpointing system was implemented on top of Parix for Parsytec machines, then for a massively parallel system, its design concepts are applicable to any message passing environment, and it has been ported on a Sun network system with little effort. The consistent cut system was implemented inside PVM, which is by itself portable on numerous architectures. For others checkpointing rollback tools for PVM library see for example [6,7, 8].

In the following we describe the early results of our experiences which led to a system of two software components, providing a co-ordinated, user triggered checkpointing facilities for PVM applications.

The rest of the paper is organised in this way: sections 2 and 3 describe the two systems separately, while in section 4 their joint use is shown. Finally, some figures of merit about an implementation performed on Sun workstations are presented.

2. User-triggered checkpointing approach.

The User-Triggered Checkpointing Tool (UTCP) has been implemented at the Leuven University under Parix operating System, an extension of Unix, on different Parsytec systems [2,3]. These parallel computers are connected to a Sun workstation that has access to the disks.

The user-triggered checkpointing approach exploits the assistance of the programmer to implement backward error recovery for long-running number-crunching applications. User-triggered checkpointing does not require explicit co-ordination via communication protocols. Instead, it requires the co-operation of the programmer for indicating the checkpoint contents and the position of the recovery lines in the code of the application processes. This will co-ordinate the saving of a consistent state of the application. A set of library functions has been developed for this. Besides, a checkpoint control layer is running and is responsible for the management of checkpoint data. The checkpoint library provides the programmer with: *define-calls*, indicating from which elements of the application process the state must be saved (this defines the checkpoint contents); and *trigger-calls* indicating the position of the recovery line in the program-code.

At a trigger-call, the state of all elements, defined by the checkpoint contents, is saved to stable storage as the checkpoint data. Upon recovery, the same call triggers the restoring of checkpoint data into the application.

The checkpoint control layer consists of a set of processes running on nodes that have access to stable storage and is responsible for managing the recovery lines in a distributed way. It is informed about the progress of the checkpointing by the trigger-calls. From the saved checkpoint data, it keeps track (on stable storage) of which recovery line is valid. Upon recovery, it informs the checkpoint library about which checkpoint data must be restored.

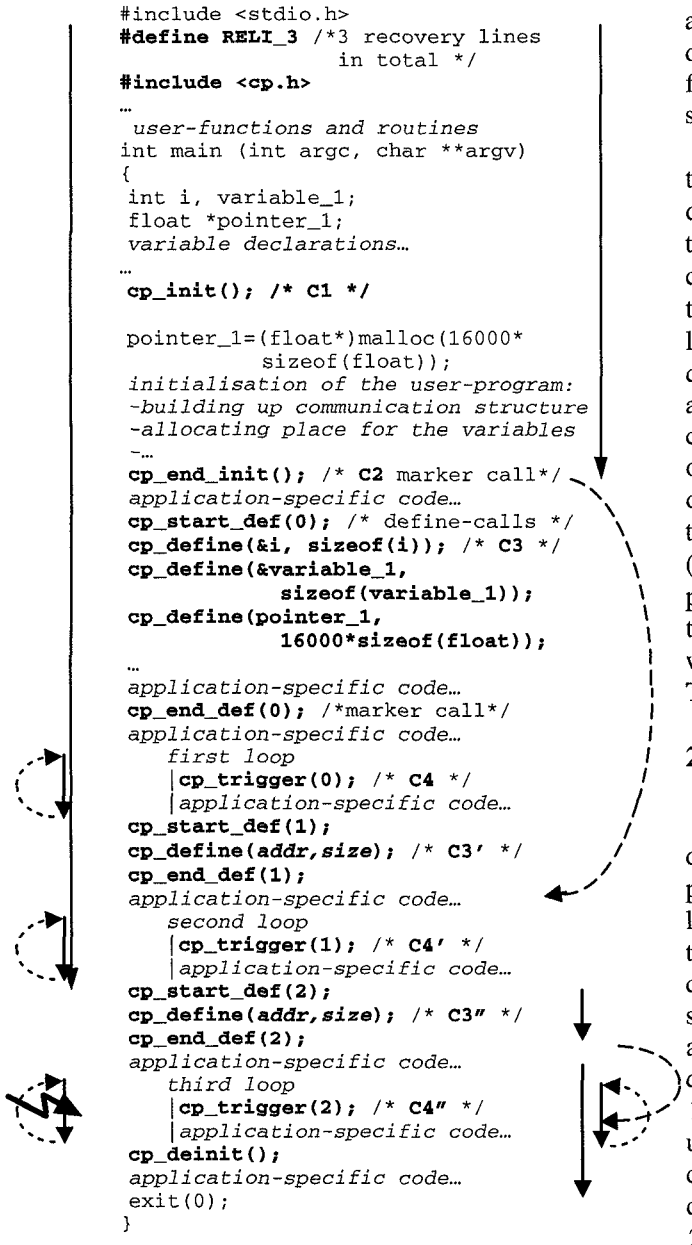
2.1 An Example Program

Hereafter we shortly present the behaviour of a SPMD (single program multiple data) example application program which employs the UTCP library for fault tolerance. The code of the example is presented in Figure 1. During normal execution, an init-function is called, where each process establishes a connection to the checkpoint control layer (location C1 in Figure 1). This connection will be used by the checkpoint library to inform this layer about the progress of the checkpointing or to retrieve information on consistent recovery lines from this layer. Simultaneously, some local data structures are set up at each node, where the checkpoint contents will be maintained together with management information, e.g. identification of the node. Then each process of the application continues its execution initialising all WORM data (Write Once, Read Many) like network topology, or application parameters, that does not change during the whole execution, and needs not to be included in the checkpoint contents. At the end of this phase, a marker-call is placed (position C2 in Figure 1).

Each process continues its execution and initialises the checkpoint contents at the define-calls (positions C3 in Figure 1). These calls define the elements for which the status must be saved. These elements correspond to the memory regions of variables, arrays, structures, pointers, and others that contribute to the checkpoint data when the checkpointing will be triggered, and are defined for each process by the define-calls.

Then, the process continues until a trigger-call is encountered (positions C4 in Figure 1). This trigger-call must be positioned in each process of the

application, such that no application messages cross the line connecting them. This initiates the checkpointing for this process.



```
#include <stdio.h>
#define RELI_3 /*3 recovery lines
                  in total */
#include <cp.h>
…
 user-functions and routines
int main (int argc, char **argv)
{
 int i, variable_1;
 float *pointer_1;
 variable declarations…
…
 cp_init(); /* C1 */

 pointer_1=(float*)malloc(16000*
           sizeof(float));
 initialisation of the user-program:
 -building up communication structure
 -allocating place for the variables
 -…
 cp_end_init(); /* C2 marker call*/
 application-specific code…
 cp_start_def(0); /* define-calls */
 cp_define(&i, sizeof(i)); /* C3 */
 cp_define(&variable_1,
           sizeof(variable_1));
 cp_define(pointer_1,
           16000*sizeof(float));
 …
 application-specific code…
 cp_end_def(0); /*marker call*/
 application-specific code…
    first loop
    |cp_trigger(0); /* C4 */
    |application-specific code…
 cp_start_def(1);
 cp_define(addr,size); /* C3' */
 cp_end_def(1);
 application-specific code…
    second loop
    |cp_trigger(1); /* C4' */
    |application-specific code…
 cp_start_def(2);
 cp_define(addr,size); /* C3" */
 cp_end_def(2);
 application-specific code…
    third loop
    |cp_trigger(2); /* C4" */
    |application-specific code…
 cp_deinit();
 application-specific code…
 exit(0);
}
```

Fig. 1: Sample code of a single process of an SPMD (single program, multiple data) application with integrated user-triggered checkpointing calls. Control flow during fault-free execution is on the left. Assuming that a fault occurred in the third loop, the control flow during recovery is shown on the right. The dotted lines show the jumps in the control flow

Therefore, the checkpoint library saves in a stable storage the state of each of the elements belonging to the checkpoint contents as checkpoint data. A communication is also sent to the control layer. When all processes of the application saved the checkpoint data at the trigger-call, this data forms a recovery line for the application, i.e. these calls saved a consistent state of the application.

Every next time this trigger-call is encountered in the application (e.g. in a loop), the new checkpoint data is saved. This new checkpoint data corresponds to the current state of the elements of the checkpoint contents. As such, a single instance of a trigger-call in the application code may lead to multiple recovery lines: every time this call is executed, the current checkpoint data is saved. Each process of the application may have multiple instances of trigger-calls (locations C4, C4' and C4" in Figure 1). For each of them, the different checkpoint contents may be different (locations C3, C3' and C3" ibid.). Each trigger-call may again lead to multiple recovery lines (e.g. in a loop). It is the task of the application programmer to indicate which elements contribute to the checkpoint contents at the position in the code where the recovery line is indicated by the trigger-call. This is often an obvious task for SPMD applications.

2.2 The Rollback

Upon recovery, the entire application is downloaded on a set of fault-free nodes and all processes are restarted from the beginning. The same library functions now act slightly differently to restore the state in the processes of the application. This control flow is shown on the right in Figure 1. As such, recovery relies on two aspects to bring the application back in a correct state from when the checkpoint data was saved:

1. The re-execution of the first part of the application updates those items, that are not included in the checkpoint contents (e.g. WORM data), to their correct state.
2. The restoring of checkpoint data brings those items that are included in the checkpoint contents back to the correct state. The checkpoint contents consists only of elements from the application, representing memory areas from the process's data space. No other information from the process's workspace %(program counter, registers, operating system information), has to be saved, as it is restored to the correct state by re-executing a portion of the application and then jumping to the appropriate code section (see Figure 1). This allows to decrease the dependence of the checkpointing tool from the operating system.

2.3 The Checkpoint Control Layer

The checkpoint control layer is running on the nodes of the system which have direct access to a stable storage device. Each control process keeps track of the progress of checkpointing for those processes of the application that are closer to that stable storage node than to any other; these application processes are called the controlled set.

During fault-free execution, a control process establishes a connection to each of the processes of the application of its controlled set; this is done when the application process calls the init-function. It is the task of the checkpoint control layer to compose recovery lines from all the checkpoint data saved by the checkpoint library.

This recovery line management requires a global view on the application. Thus, the control processes are connected to a higher control layer, the master control process that is responsible for application-wide decisions that are forwarded to all other control processes. Each control process (including the master) keeps a local database on stable storage with information on the status of the recovery lines of the application, representing the checkpointing progress as seen by the lower layer [3].

Upon recovery, the control processes of the checkpoint control layer read their local databases from stable storage. Then, they determine which VALID recovery line can be restored in the application. The master control process proposes a recovery line to roll back to, and all lower level control processes have to agree. If one of the control processes disagrees (e.g. because a fault affected the checkpoint control layer) the previous VALID recovery line will be restored. An agreement protocol is executed among the control processes, and the master control process forwards the decision down the hierarchical tree of control processes.

At the init-call, the connection to the application processes of the controlled set is re-established. As the control processes know about the VALID recovery line to be restored, and where the corresponding checkpoint data can be found, they provide this information to the application process. These application processes will then restore this checkpoint data at the trigger-call. If no such a VALID recovery line can be found (e.g. because of an early failure), the checkpoint libraries are informed via the init-call that the application should continue as a normal execution and that nothing will be restored.

3. PVMsnap: a tool for visible snapshot

The second tool considered in this paper is PVMsnap, an extension of PVM version 3.3, in which send and receive functions have been extended to make a consistent cut visible to the user. It has been developed at the Genova IMA-CNR and DISI-University. PVMsnap can be used jointly with a non co-ordinated checkpoint system to mask fail-stop failure, to evaluate monotonic distributed functions such as the Global Virtual Time in Time Warp distributed simulation systems, or to support dynamic program reconfiguration.

Processes in PVM communicate and synchronise mainly through daemons, which also co-ordinate global operations. The connectionless UDP protocol has been chosen as the fundamental interprocessor communication vehicle to guarantee the scalability. However, because of its unreliability, message acknowledgement is performed by the daemon, in order to achieve higher reliability. In turn, each daemon is interfaced with the user processes residing on the same node using UNIX domain sockets which ensure order preserving communication.

In case of host or task failure, PVM does not crash until the master host is up. However, in such a case, it is responsibility of the programmer to act appropriately, since no other support is provided by the system.

The cut protocol implemented in PVMsnap is based on the Lai-Yang algorithm [5] to position the local cut event for each process, then, each message carries one bit information (the colour, red or white). Each message is supposed to be acknowledged, and such a property is used to detect the cut termination. A more detailed description of the algorithm is given in [4,1].

Since message acknowledgement is already provided by the underlying communication system the number of additional messages required by the protocol is O(n) where n is the number of processes. Moreover, control messages do not carry any information besides their type.

The cut protocol has been embedded in both the PVM daemon and library: the major responsibility, that is the co-ordination of the cut with the starter and the cut termination detection, is ascribed to the daemon, while the PVM library send-receive functions perform the correct message delivery to the user. Process spawning and their exit from PVM are handled as well. Acknowledge messages exchanged by the daemons are used by PVMsnap to detect the

termination of the cut reducing the number of additional control messages exchanged for each cut.

Each user message is painted with a colour: the additional bit is obtained from its tag field, without any additional transmission cost. The color is added when an output message is built after a call to PVM_INIT_SEND(), and it is filtered when received by the appropriate PVM receive operation, so that the user is not aware of it.

PVMsnap makes the cut visible to the user. Before the cut, every process and message is painted with the same colour. A starter process requires a cut by calling the special non blocking function PVM_STARTCUT(). The starter daemon starts the protocol broadcasting a message to every daemon, itself included. Each daemon generates and multicasts a control message (*PvmStartCut*) to each local process, as soon as it receives the starter control message or a regular message painted with a different colour. Such a message has a high priority, then the user process will receive it before any other message whose sending event belongs to the future of the cut, independently of any selective receive performed. After the receipt of the *PvmStartCut* message, every outgoing messages will be painted with the new colour, since their sending event is in the future of the cut.

Messages received after the cut notification could be in transit, that is, sent in the past of the cut. This property is notified to the user process. When all in_transit messages have been received by the process, another control message (*PvmTermCut*) is sent from the daemon, with the same high priority of the *PvmStartCut* message. User processes are responsible for taking the appropriate action after the receipt of any control message or regular in-transit message. The starter process receives its *PvmStartCut* message only after the cut protocol termination, that is when all the messages in transit through the cut have been received.

The following additional negative values have been provided to the integer value returned by the PVM receive functions, to distinguish whether the message is a regular or control one:

- *PvmCutEvent*: a new cut has begun,
- *PvmTermCut*: all in_transit messages have been received.

In Figure 2, a startcut call example is given.

To allow the user's program to know if a message is in transit, an additional return parameter (*transmsg*) has been provided to the PVM receive functions. The returned value is IN_TRANSIT if the message is in transit, NOT_IN_TRANSIT otherwise. As well as for the other parameters (*tid* and *msgtag*), selective receive is allowed also on this last argument. The user supplied value ANY matches both of the previous values. In Figure 3, an example of use is shown. The other PVM receive primitives have been changed accordingly.

```
.....
    info= pvm_startcut();
      while ((bufid = pvm_recv(tid, msgtag)) !=
            PvmTermCut)
        {
         /* application-specific code... */
        } /* the cut is terminated */
    /* application-specific code... */
     info = pvm_startcut()  /* now a new cut can
                               start */
```

Figure 2 A starter process calls for a cut. Then, every received message is checked for the cut termination.

```
transmsg=ANY;
bufid = pvm_recv( ANY, ANY, &transmsg)
if (bufid >= 0)  /* regular message */
{
  if (transmsg)  /* in-transit message */
  {
   /* application-specific code... */
  }
  else  /* non in-transit message */
  {
   /* application-specific code... */
  }
}
else
 switch(bufid)
 {
    case PvmCutEvent :  /*cut begin code... */
      break;
    case PvmTermCut :   /*cut termination
                          code... */
      break;
    default :          /* error code... */
 }
```

Figure3: An example of message receipt in PVMsnap

4. The integrated approach

Integration of UTCP and PVMsnap allows the consistent cut algorithm to define the positions of the recovery lines, and the checkpoint contents.

Joint use of UTCP and PVMsnap allows the user to position consistent recovery lines following different

schemes, the more simple of which implies the postponement of the local checkpoint after the receipt of the cut termination message. After the cut begin message receipt, the user process selectively receives only in_transit messages until the end of the cut. Triggering the local checkpoint is now possible, since the incoming channels do not contain any in_transit message, and the cut is consistent.

In Figure 4, part of the sample code of above has been extended to show a possible joint use of the two systems for a non starter process.

```
int main(int argc, char **argv)
    {
      variable declarations...
      int bufid, transmsg;
      pvm_spawn(.....);              /* P1 */
      .....
      cp_init();                     /* C1 */
      .....
      transmsg=ANY;
       /*first loop */
       bufid=pvm_recv(ANY, ANY, &transmsg);
        switch(bufid)
          {
          case PvmCutEvent:      /* P2 */
            transmsg=IN_TRANSIT;
            break;
          case PvmTermCut:       /* P3 */
            cp_trigger(0);       /* C4 */
            transmsg=ANY;
            break;
          default:
           if (bufid>=0)  /* regular message */
            {
             /* application-specific code */
            }
            else
            {
             /* error code */
            }
          }
        end /* first loop */
    }
```

Figure 4: An example of joined UTPC and PVMsnap use

PVM processes can also be started by non-starter process, before the first marker call (see position P1 in Figure 3). Each received message must be checked whether it is a control or a regular message (positions P2 and P3). **cp_trigger**() function is called when every in_transit message has been flushed (position C4).

5. Conclusions

Although the UTCP prototype implementation is based on Parix as parallel operating system, the user-triggered checkpointing approach can make use of any public domain message passing API's (PVM, MPI) for their intercommunication. A version of it is now running under Unix on Sun Networks.

In Table 1 and Table 2 some performance results are presented. The first Table shows time values obtained running a test without PVMsnap, and with PVMsnap, performing 100 cuts. In such an experiment, a set of 61 tasks, with a random communication pattern and dominant communication time (each one receives a 2 byte message and immediately sends another message to a randomly chosen task) has been distributed on 6 Sun workstations.

	PVM	PVMsnap
No. of tasks	61	61
No. of workstation	6	6
Total time	6628 sec.	6712 sec.
Average time (between 2 cuts)		65.93 sec.

Table1 : PVMsnap vs PVM performances

The second experiment has been performed with 6 tasks working on 6 Sun workstations, during light load, performing both snapshot (with PVMsnap) and checkpoint (with UTCP). Three evaluations are given: the minimum and maximum time needed for executing the test with PVM alone (without checkpoints), with 50 cuts and a checkpointed state of 100 bytes, and with 50 cuts and a checkpointed state of 1000 bytes. Times are varying in a wide range during the first run, while they present a lower variance in the other cases. May be, this is a consequence of some loose synchronisation during the checkpoint algorithm.

	PVM	Checkp (100)	Checkp (1000)
No. of tasks	6	6	6
No. of workstation	6	6	6
Total time min.	369 sec.	421 sec.	443 sec.
Total time max	411 sec.	438 sec.	459 sec.
Average time (between 2 cuts)		8,57 sec.	9 sec.

Table2 : Checkpointing cost with UTCP and PVMsnap

The programmer involvement is an important issue for the user-triggered checkpointing approach. It requires awareness of the programmer of fault tolerance, and his/her input to indicate the checkpoint contents and the recovery lines in the application. UTCP does not include user facilities to assure consistency to the cut, thus, the user is responsible for correctly triggering the checkpoint, in order to create consistent recovery lines. In some parallel processes with substantial geometrical symmetry, it is easy for the user to find existing synchronisation points which allow the definition of a consistent recovery lines, while in other applications it could be quite difficult or even impossible.

PVMsnap provides the PVM programmer the facilities needed to define consistent recovery lines in a suitably way, independently of the geometry of the application. Its features makes particularly suitable its joint use with UTCP, since both of them are user-triggered and non-blocking. The continue-before-validate approach provided by UTCP is then preserved, avoiding the possible I/O bottleneck arising when all application processes reach the recovery line at the same moment.

Acknowledgements

Geert Deconinck acknowledges the support of a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.) .

Andrea Clematis and Vittoria Gianuzzi acknowledge the support of CNR under contract "Virtual libraries for computational problems"

References

[1] A. Clematis, V. Gianuzzi, G. Sacchetto, 'Implementing Snapshot Protocol for Message Based Communication Libraries', *Proc. 3th . ACPC Conf.*, LNCS, 1996.

[2] G. Deconinck, J. Vounckx, R. Cuyvers, R. Lauwereins, B. Bieker, H. Willeke, E. Maehle, A. Hein, F. Balbach, J. Altmann, M. Dal Cin, H. Madeira, J.G. Silva, R. Wagner, G. Viehvver, "Fault Tolerance in Massively Parallel Systems", *Transputer Communications*, Vol. 2(4), pp.241-257, 1994.

[3] G. Deconinck, R. Lauwereins, "User-Triggered Checkpointing: System-Independent and Scalable Application Recovery", *2nd IEEE Symp. on Computers and Communications (ISCC'97),* Alexandria, Egypt, Jul. 1-3, 1997.

[4]http://www.disi.unige.it/person/GianuzziV/soft_Gianuzzi .html.

[5] T.-H. Lai, T.-H. Yang, "On distributed snapshots", *Inform. Process. Lett.*, 25, pp.153-158, 1987.

[6] J. Leon, A.L. Fisher, P. Steenkiste, "Fail-safe PVM: a portable package for distribuited programming with transparent recovery", *CMU-CS-93-124*, 1993.

[7] K. Skadron, "Asynchronous checkpointing for PVM requires message-logging", Tech. Rep. Rice Univ., Dept. Comp. Sc., 1996.

[8] G. Stellner, J. Pruyne, "Resource management and checkpointing for PVM", *Proc. 2nd Europ. PVM users' Group Meeting*, pp.131-136, 1995.