Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation
D-80290 München

CoCheck Users' Guide V1.0
PVM Version

Georg Stellner
stellner@informatik.tu-muenchen.de
http://wwwbode.informatik.tu-muenchen.de/~stellner/

Jim Pruyne
pruyne@cs.wisc.edu
http://cs.wisc.edu/~pruyne/

November 17, 1995

"Who loves you babe?"

Theo Kojak

Contents

1 Introduction

Networks of workstations (NOWs) [1] are becoming a popular environment for executing parallel applications. An important argument for this approach is their overall availability and accessibility. Typically, workstations are assigned to a certain user, the *primary user (owner).* The machine resides on the owners desk and serves as an execution platform for the owner's applications. The network to which the machine is connected provides access to further workstations, where users can log in and perform work as *secondary users (guests).* Particularly, during times when the primary user does not use his machine secondary users might use the idle machine to perform useful work and borrow the cycles from the primary user's machine to execute parts of a parallel application on the machine.

A problem arises when the primary user wants to resume work on his workstation which is currently executing parts of a parallel program. Typical parallel applications are consuming many resources (CPU time and main memory). Hence, the machine he tries to reclaim for interactive work is occupied by the parallel application, and response time is longer. A solution to the problem would be to migrate the processes which are borrowing cycles from this machine to another available host. Currently, all popular programming environments like NXLib [12], p4 [5], PVM [7] or the implementations [2; 6; 3; 4; 10] supporting the MPI standard [9] do not provide process migration.

Another important demand for a process migration capability comes from parallel applications. It is highly desirable that all processes of a parallel application achieve the same progress in their computation. Otherwise, if one or more processes are lagging behind, they are reducing the performance of the complete application as all the other processes have to wait for those processes. In the worst case this means, that the slowest machine determines the speed of an application. On a classical parallel computer one node typically lags behind because it has to perform more computation than the other nodes, i.e. the load is not evenly distributed. In a multi-user environment such as a NOW, certain processes may also lag behind because of other users' processes which are executing on a certain node. In such cases of load imbalance — either because the application itself is unbalanced or other processes cause the imbalance — process migration could be used to accomplish an even distribution of the load by migrating processes from overload to less loaded hosts.

Finally, parallel applications have typically very long execution times. In workstation environments machine failures are very likely. Hence, it is important that such failures can be tolerated. Checkpointing is one technique to accomplish that.

In the framework of PVM the CoCheck environment tries to solve the above mentioned problems. It provides a means to create checkpoints of PVM applications which can be used to restart the application in case of a failure. In addition, CoCheck provides process migration. An application can benefit from the advantages of CoCheck by simply relinking it to the supplied CoCheck libraries. Then a supplied command can be used to invoke process migration or creation of a checkpoint. It is also possible to use the API functions of CoCheck to integrate checkpointing or migrating processes into the application itself.

2 Terminology

This section lists several frequently used terms and defines their understanding throughout the rest of this document.

PVM application consists of several components referred to as *tasks* or *application processes* which communicate using the PVM interprocess communication calls.

task basic units of a *PVM application* which are represented by conventional UNIX processes. Each task is executed on one *node* of the *VM* and is addressed via a unique *tid*.

tid unique identifier of each *task* which is created when a UNIX process joins the *VM*.

original tid refers to the *tid* of a *task* when it joined the *VM* for the first time. This original tid is invariant throughout the lifetime of the *PVM application*.

current tid is the *tid* of a *task* currently valid to address it. The current tid changes whenever the *task* leaves and then rejoins the *VM*. This is especially important when a *PVM application* creates a *consistent checkpoint* and is restarted afterwards.

virtual machine (VM) collection of *nodes* used to execute (multiple) *PVM applications*.

node basic building block of the *VM* consisting of a *host*. Each node executes a special *pvmd* which takes care to incorporate the *host* into the *VM*. Nodes may be added dynamically during the life time of the *VM*.

PVM daemon (pvmd) provides the basic services to the *tasks* on a *node*. Each *node* of the *VM* executes one pvmd and altogether the pvmds span the *VM*.

PVM master daemon (master pvmd) is the first *pvmd* which was started by the user. Further *pvmds* on other machines can be started by the master pvmd to add more *nodes* to the *VM*.

host machine or workstation hosting a *pvmd*.

resource manager task (RM) special system *task* which can intercept certain calls of a *PVM application*. Hence, certain services which are usually provided by the *pvmds*, like for example adding *hosts* or creating *tasks* can be provided by the RM.

consistent checkpoint set of *checkpoint files* representing the *global state* of a *PVM application* from which the application can be correctly restarted later on. Specifically, this means that no messages are lost or duplicated.

checkpoint file a file representing the *local state* of a process from which the process can be restarted later on with the same state.

global state state in a *PVM application* which consists of the *local states* and the *state of the network* of a *PVM application*.

local state state of a given process at a given time.

state of the network messages which have been sent but not yet received at a given time.

overlayed function, wrapper a system call or function for which a new body has been provided. The original call is renamed and usually called from within the new wrapping function. This technique can be used to change or enhance the functionality of a system call without modifying the original source code.

3 Using CoCheck

3.1 Obtaining the Software

The current version of CoCheck is V1.0. To get the sources refer to the CoCheck homepage on the World Wide Web: http://wwwbode.informatik.tu-muenchen.de/~stellner/CoCheck/. There, a list of ftp servers and file names which must be downloaded is provided. The software is freely available under the terms of the GNU puplic license.

3.2 Installation Procedure

As checkpointing is highly machine dependent CoCheck does not support all machine types where PVM applications can be executed. CoCheck uses the Condor checkpointing library as single process checkpointer. Hence, the list of all supported machines and operating systems in table 3.1 is primarily limited to the machines supported by Condor.

Machine	Operating System	$PVM_ARCH
Sun - SPARC	SunOs 4.1.x	SUN4
DEC - ALPHA	OSF/1 V2.1B	ALPHA

Table 3.1: Supported machine architectures of CoCheck V1.0.

The CoCheck environment consists of different components: the Condor checkpointing libraries, the CoCheck libraries, a resource manager, a user command and an example PVM program. The Condor libraries are precompiled for every supported machine type and a library matching the machine where CoCheck shall be installed must have been copied via the CoCheck homepage. The rest of the software is generated during the installation procedure.

For the installation procedure we assume that a complete and correct PVM 3.3.8 installation exists on the machine you are planing to install CoCheck and that the environment variables PVM_ROOT and PVM_ARCH have proper values. References in the text to their values are made in conventional shell syntax. If you are for instance working on a Sun SPARC with SunOs the occurence of a “$PVM_ARCH” in the text would have to be replaced with the proper value “SUN4”. We will use this convention throughout the whole document in file names or path names to distinguish machine specific parts. For a description of how to get and install PVM refer to the description in [8].

3.2.1 Installing the Condor Checkpointing Library

The first step is to install the correct condor checkpointing library. Therefore you need to get the package CkptLib-$PVM_ARCH-V1.0.tgz[1]. Put this file into an appropriate location in the file system of your machine like for example /usr/local/lib or $HOME/lib. Later we will refer to this directory as the Condor installation directory and we will assume that we have decided to put the libraries in /usr/local/lib. The next thing after you have put CkptLib-$PVM_ARCH-V1.0.tgz in the installation directory is to unpack it. This can be done in the installation directory with the following command:

```
ibode2-2.1$ gunzip -c CkptLib-$PVM_ARCH-V1.0.tgz | tar xvf -
```

The necessarry libraries are then extracted from the archive file and placed under Condor-V1.0. We highly recommend to install the checkpointing libraries only once on each machine in a system directory such as /usr/local/lib, although it is also possible that multiple users privately install the libraries. The installation in a system directory saves disk space and makes sure that all users access the same version of the library, so that updates only need to be applied once. Finally, create a symbolic link in the condor installation directory with the command:

```
ibode2-2.2$ ln -s Condor-V1.0 Condor
```

This link has the advantage that later on a new version can easily be installed under a new directory and tested there. After all tests have been successful this new version can be made accessible for users by simply creating a new symbolic link to the new installation directory. The users need not be aware of the new version as they will rely on the versionless path name /usr/local/lib/Condor.

3.2.2 Installing the CoCheck Sources

After you have successfully installed the checkpointng library the next step is to unpack the CoCheck sources. Therefore, you will need the CoCheck-V1.0.tgz file. Again, a suitable location in the file system must be choosen such as /usr/local/src or $HOME/src and the sources can be unpacked with

```
ibode2-2.3$ gunzip -c CoCheck-V1.0.tgz | tar xvf -
```

The complete source tree can then be found under CoCheck-V1.0. Again, a symbolic link should be created with the command:

```
ibode2-2.4$ ln -s CoCheck-V1.0 CoCheck
```

3.2.3 Compiling the CoCheck Sources

In preparation for the compilation process the file $PVM_ARCH.config in the config directory has to be slightly modified. Assuming you current working directory is still the CoCheck installation directory the following command sequence can be used:

```
ibode2-2.5$ cd CoCheck

ibode2-2.6$ emacs config/$PVM_ARCH.config
```

[1] Assuming you are working on a Sun SPARC the name of the file you would have to get is CkptLib-SUN4-V1.0.tgz.

If you have gcc installed on your system then you simply need to assign the proper Condor installation directory to the macro `CONDORDIR`. If gcc is not available you must also change the `CC` macro to the appropriate compiler name. In addition you have to change the `STATIC` macro. There, you should insert the switches your compiler needs to link statically and to exclude the standard startup file (`crt0.o`).

After these preparations you can go ahead and compile CoCheck with:

```
ibode2-2.7$ make all
```

This will go through all directories and create the tools, the resource manager, the libraries and an example program. During the compilation process a version of the original PVM library will be copied and the names of the functions are renamed. This step requires a correctly installed version of PVM 3.3.8 so be present in the filesystem and the path name of the installation directory in the PVM_ROOT environment variable. In addition to the libraries a small test application (bin/$PVM_ARCH/m) has also been created. So, you can refer to section 3.3.2 and section 3.3.3 how to start this application and create a checkpoint respectively.

3.3 Running Applications

3.3.1 Linking an Application

In order to benefit from CoCheck's capabilities you need to relink your PVM applications with the CoCheck libraries. For correct operation the sequence of the libraries is highly important, so that we recommend that you use the `Makefile` of the example program as a template and adopt it to your needs. The libraries can be found in the found under the lib/$PVM_ARCH directory in the CoCheck top level directory.

If you nevertheless want to link your application by hand, then you have to obey the following things. This discussion assumes you are using gcc as C compiler and that the variable $COCHECKTOP carries the path name of the CoCheck top level direcory (such as /usr/local/src/CoCheck).

- make sure you are linking statically: `-static`
- do not link with the standard startup file: `-nostartfiles`
- link with the Condor startup file instead: `$CONDORDIR/condor_ckpt/condor_rt0.o`[2]
- link with the CoCheck libraries in exactly that order on the command line
  1. $COCHECKTOP/lib/$PVM_ARCH/libCoCheck.a
  2. $COCHECKTOP/lib/$PVM_ARCH/libCoCheckpvm3.a
  3. $COCHECKTOP/lib/$PVM_ARCH/libCoCheckgpvm3.a
  4. all the additional libraries your application needs
  5. $CONDORDIR/condor_ckpt/libckpt.a
  6. $CONDORDIR/condor_ckpt/c_plus_alloc.o

[2] The variable $CONDORDIR denotes the same value you have used to configure the `CONDORDIR` macro in the config/$PVM_ARCH.config file as explained in section 3.2.3 on page 7.

The different libraries contain the following routines:

condor_rt0.o modified condor startup file that allows restart.

libCoCheck.a wrappers for all PVM functions and the functions to achieve a consistent state of the application.

libCoCheckpvm3.a copy of the original PVM library with renamed PVM functions. The wrappers from the libCoCheck.a call these functions.

libCoCheckgpvm3.a copy of the original group library of PVM with renamed function names. Again, they are called from the wrappers in libCoCheck.a.

libckpt.a contains the Condor single process checkpointer which allows to create a checkpoint of a single UNIX process.

c_plus_alloc.o some additional functions which are called from libckpt.a.

3.3.2 Starting an Application

Again we are assuming that the variable $COCHECKTOP expands to the path name of the CoCheck top level directory (cf. section 3.3.1). The first step in running an application is to start both the virtual machine and the resource manager. This is done by the following comand:

```
ibode2-2.8$ $COCHECKTOP/bin/$PVM_ARCH/pvm_rm args
```

Where *args* are whatever command line arguments are normally given to the first pvm daemon. After that you can add further machines as usual with the console command. Now, the application can be started from a shell. Currently, we recommend to use absolute path names to start a PVM application because otherwise the restart of an application might fail. Later version will evaluate the ep= option of PVM.

3.3.3 Creating a Checkpoint

Creating a checkpoint is easy. Therefore, the `kojak` command of CoCheck has to be used in the following way:

```
ibode2-2.9$ $COCHECKTOP/bin/$PVM_ARCH/kojak -c masterckptfile
```

This will create a checkpoint file of every process of the application and provide a file with the name “masterckptfile”. This file contains the locations of the single checkpoint files and additional information which is necessarry to restart the application from this checkpoint.

3.3.4 Restarting from a Checkpoint

To restart an application from a previously taken checkpoint with the master checkpoint file “masterckptfile” the following command should be used:

```
ibode2-2.10$ $COCHECKTOP/bin/$PVM_ARCH/kojak -r masterckptfile
```

3.3.5 Migrating Processes

Again the `kojak` command can be used to migrate processes. Currently, you have to specify `-m <tid>` for each task you would like to migrate. So if you have for instance an application consisting of four tasks with the tids 262147, 262148, 262149 and 262150 then you could migrate tasks 262149 and 262150 with the following command:

```
ibode2-2.11$ $COCHECKTOP/bin/$PVM_ARCH/kojak -m 262149 -m 262150
ckpt
```

Although no master checkpoint file gets written you have to specify its name. This will change in later revisions of the software. There, we will also include a means to specify the destionation hosts of migrating processes.

3.4 Caveats and Troubleshooting

The following list contains some caveats and hints if you are using CoCheck V1.0 for PVM.

1. all processes running in a virtual machine are forming a single application. hence, If you have two applications simultaneously running on a virtual machine a checkpoint will comprise both applications. Then both applications have to be linked with the CoCheck libraries.

2. when you create a checkpoint or try to migrate some processes no `pvm` console must be running. Otherwise, the checkpointer tries to include the console into the checkpointing/migration protocol. This will fail, as the console is not linked with the CoCheck libraries (cf. item 1 in this list)

3. PVM applications using group functions can currently not be checkpointed or migrated. This will be improved in future versions.

4. Support for interactive applications is limited. Following a restart, it is not possible for any application process to interact with a user via a tty. Ttys are not re-connected after a restart.

4 Outline of the Implementation

4.1 CoCheck: Checkpoints and Process Migration

Our protocol to create consistent checkpoints described in [13] proved to be viable and stable but was lacking good performance. This was due to several reasons. First, the checkpoint files which were produced by the single process checkpointer were too large. Then the migration of a single process required that all processes had to write a checkpoint file. Finally, all checkpoints were written to a network file system producing burst network traffic.

Hence, in the latest version of the CoCheck checkpointer for PVM applications we have addressed those problems to improve the performance. First, we have incorporated the latest version of the Condor single process checkpointer. It reduces the size of the checkpoint file of each process [14]. Then, the protocol has been enhanced to allow that checkpoint files can also be written to local discs and that only migrating processes write a checkpoint file.

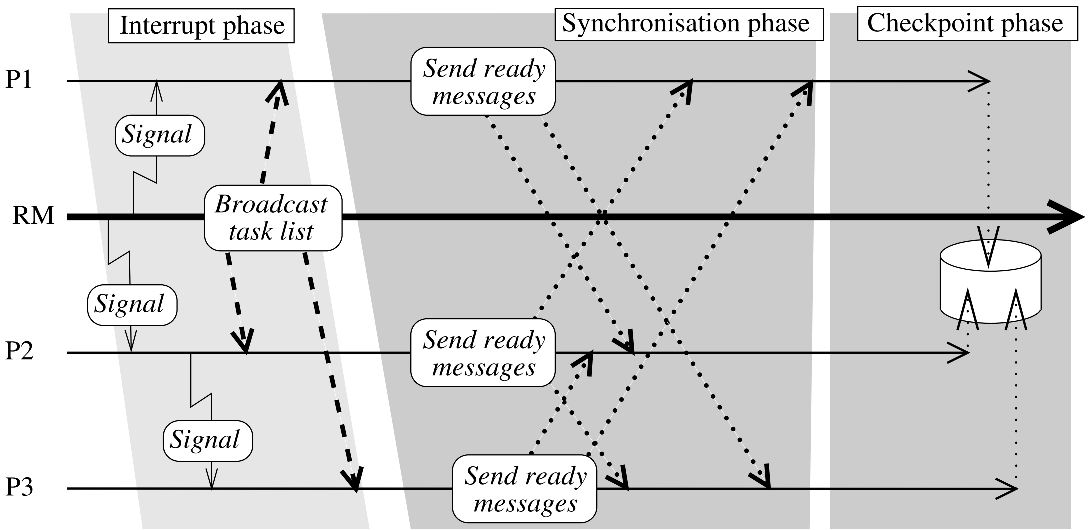

Figure 4.1: Steps to create a consistent checkpoint in CoCheck.

Both the creation of a consistent checkpoint of an application and the migration of a set of processes is initiated by the RM of the virtual machine by delivering a combination of a signal and a message to each process of the PVM application (cf. Figure 4.1). As the wrappers for the PVM functions guarantee that the PVM calls cannot be interrupted by a signal, it is possible to use PVM functions within the signal handler. PVM calls such as `pvm_recv` present a problem because they cannot safely be interrupted by a signal. Here, the message which is sent together with the signal can be used to force the interception of the signal. The wrapper functions for the blocking PVM calls have been implemented in that way. In addition this message contains information about all tasks of the application, which tasks need to create a checkpoint file, the location of the checkpoint file and if the process should exit. The location can either be a file on a local or network file system or a network address. The network address can be used in case of process migration, where the checkpoint can be directly transfered to the new process on another host. Exiting is only necessary if a process must vacate a machine.

After the signal and the task list have been intercepted all the tasks send specially tagged "ready" messages to each other. As PVM guarantees the delivery of the messages in sending sequence each task can draw the following conclusion: after all ready messages from all other tasks have been received there are no more outstanding messages on the network for this task. Hence, it can be seen as a normal stand alone process without any network state. While each process waits for the reception of the ready messages, it collects all other incoming messages in its address space, so that they become part of the checkpoint file when it is produced and can be retrieved after restart from the wrapper functions.

After each process has collected all the ready messages it disconnects itself from the PVM system and uses the Condor single process checkpoint library to create a checkpoint. Depending on what was specified in the task list message the file is either written to disk (local or via a remote file system) or its state is transferred directly to a remote process which will become the migrated process. If the checkpoint has to vacate the machine it exits after the creation of the checkpoint is complete. Otherwise it directly begins the restart protocol (cf. Figure 4.2).

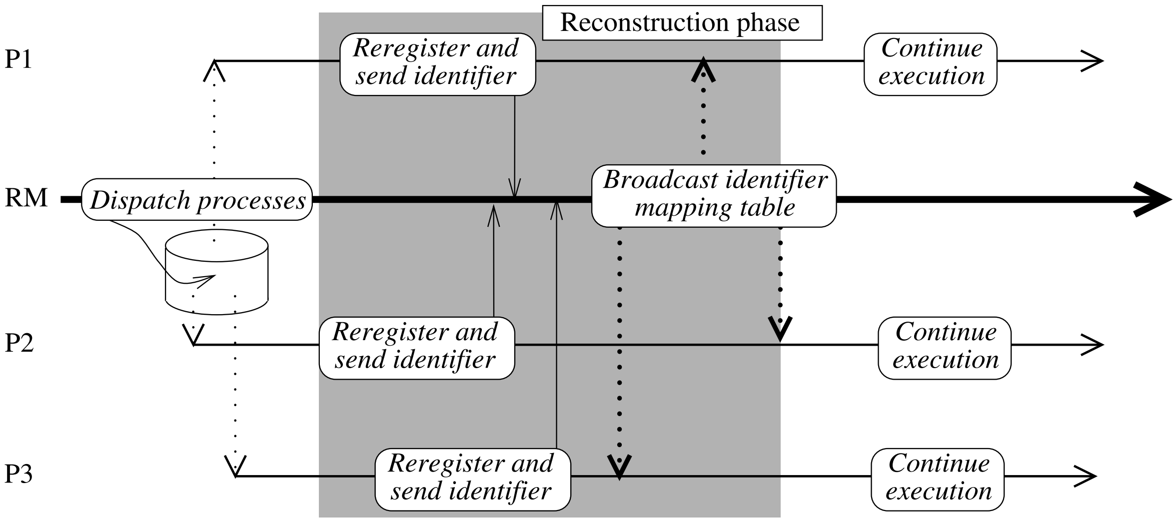

Figure 4.2: Restarting from a checkpoint.

At the beginning of the restart phase all processes are stand alone processes. As the first step to continue the parallel PVM application, each process rejoins PVM and gets a new task identifier (tid). This tid is then sent to the RM which can set up a tid mapping table from original tids to current tids. As the application should not be aware of the fact that a process has migrated or a checkpoint was taken and the application has been restarted from that checkpoint, the tids which each task got, when it enrolled for the first time, are used throughout the lifetime of the whole application. To achieve this transparency of the tids the wrappers use a mapping table each time a tid is referenced in a PVM call. This mapping table is set up and distributed to all tasks by the RM. After each task has received its copy of the mapping table it can continue normal execution. The wrapper functions take care of retrieving messages that have been stored in the restored user address space before other incoming messages.

As already mentioned one requirement is to have a signal safe extension of PVM. CoCheck achieves this by providing wrapper functions for all PVM calls which block signals. This introduces an additional overhead of about 2 percent during normal operation for sends and receives.

4.2 The Resource Manager Interface

As described above, one of the key components of the CoCheck protocol is the Resource Manager (RM) process. The use of a RM process was introduced to PVM in version 3.3 [11]. The goal of creating the RM interface was to allow users to customize the handling of resource management decisions, and to provide an easy way of interfacing to existing scheduling systems. A RM process assumes responsibility for making resource allocation and process placement decisions. In order to make these decisions, the RM must have information on the global state of the virtual machine. Because it already maintains this state, it is natural for it also to control the CoCheck protocol.

The Resource Manager acts in two ways during an execution of the CoCheck protocol. The low level responsibility is to provide the list of running tasks to the application processes, and to establish a new task identifier mapping at restart. Here, the RM takes advantage of its global state information. Since the RM needs to maintain a list of all active processes in order to make load balancing decisions, it can easily provide this list in the CoCheck protocol. At restart time, the RM collects information as tasks reconnect to PVM, and provides the new mappings to each task.

At a higher level, the Resource Manager must determine when a checkpoint or migration is required. This can be done either automatically or as the result of a user request. An automatic checkpoint would take place when the RM determines that one or more hosts being used by an application must be released to their owners or to other secondary users. When sufficient machines are available to replace those which must be released, a migration rather than a checkpoint can be performed. It may also be desirable for a user to trigger a checkpoint or migration for fault tolerance or load balancing purposes. Therefore, we expect to develop a user interface to the CoCheck mechanisms.

One of the goals of introducing the notion of the RM process to PVM was to interface to existing workstation scheduling systems such as Condor, Codine, LoadLeveler, etc. All of these systems are able to monitor resources and determine when they have been reclaimed by their owners, and allocate individual machines to user applications. Using these existing facilities, a resource manager can determine when a host must be vacated, and determine whether a migration is possible or whether a full checkpoint is needed. The ability to use the services already available in these scheduling systems allows us to bring CoCheck into large, existing clusters, and gives these systems much more flexibility in running PVM programs than ever before.

A Supported PVM calls

A.1 List of Overlayed Functions

`pvm_spawn`

- make call interruptible,
- record parent relation.

`pvm_send` substitute original tid with current tid.

`pvm_psend` substitute original tid with current tid.

`pvm_mcast` substitute original tids with current tids.

`pvm_recv`

- substitute original tid with current tid,
- make call interruptible.

`pvm_precv`

- substitute original tid with current tid,
- make call interruptible.

`pvm_nrecv` substitute original tid with current tid.

`pvm_recvf` record comparison function to be able to install the function again after restart.

`pvm_probe` substitute original tid with current tid.

`pvm_sendsig` substitute original tid with current tid.

`pvm_kill` substitute original tid with current tid.

`pvm_mytid` record the original tid of the task.

`pvm_parent` substitute current tid with original tid

`pvm_bufinfo` substitute current tid with original tid.

`pvm_notify`

- substitute original tids with current tids,
- record notification events and original tids of tasks to be notified.

`pvm_pstat` substitute original tid with current tid.

`pvm_tasks` substitute current tids with original tids.

`pvm_joingroup` records the group that is joined to be able to rejoin the group after restart. The instance number returned is also recorded to set up the instance number mapping table after a checkpoint.

`pvm_lvgroup` removes the group from the joined group set which is used for restart to avoid joining a group which has

`pvm_gettid`

- substitute original instance number with original instance number,
- substitute current tid with original tid.

`pvm_getinst`

- substitute original tid with current tid,
- substitute current instance number with original instance number.

`pvm_barrier` make call interruptible.

`pvm_reduce` substitute the original instance number with the current instance number

`pvm_scatter` substitute the original instance number with the current instance number

`pvm_gather` substitute original instance number with current instance number.

`pvm_settmask` record current state of the trace mask to be able to reset it after restart.

`pvm_setopt`

- as far as tids are supplied as parameters these original tids are substituted with their current counterparts.
- options are recorded to be accessible during restart.

`pvm_getopt` as far as the results concern tids they are mapped to their original value. For certain options this mapping may fail and incorrect values may be returned (cf. Appendix A.2).

all calls checkpoint signal disabled at the beginning and enabled again at the end to insure reentrancy.

A.2 Restricted PVM Functions

`pvm_catchout` no support. Refrain from using.

`pvm_getfds` no mapping for the file descriptors provided. Anyway, this is a very ugly function providing information to the user which should be totally hidden! Nevertheless, the call to it provides the currently valid descriptors, but they are not transparent across checkpoints. Refrain from using.

`pvm_getopt` values representing a tid may be incorrect. As far as possible current tids are mapped to their original counterpart. For those values restricted use is recommended. All other options are not influenced and the call can be used without restrictions.

`pvm_hostsync` delivers the correct values but if migration takes place during subsequent calls to the function the time the task was not executing is added. Carefull use is recommended.

`pvm_trecv` timeouts are not supported. Refrain from using.

`pvm_tidtohost` delivers the correct name of the host but the host may change after a checkpoint is restarted. Hence, the name might be incorrect then. Carefull use is recommended.

`pvm_mytid` **and** `pvm_exit` task with multiple calls to `pvm_mytid` and `pvm_exit` might not work correctly, if a checkpoint is being created during the task is not a member of the VM (after `pvm_exit` was called and the call to `pvm_mytid` was not yet made). Carefull use of that sequence is recommended.

`pvm_reg_hoster` no support. Refrain from using.

`pvm_reg_tasker` no support. Refrain from using.

`pvm_reg_rm` it is assumed that a RM task is registered before an application is started. The RM will not be checkpointed and can also not be substituted. Refrain from using.

`pvm_start_pvmd` no support. Refrain from using.

Bibliography

[1] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum. *Users' Guide to* `mpich`*, a Portable Implementation of MPI*. Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4801, July 1995.

[3] R. A. A. Bruce, J. G. Mills, and A. G. Smith. *CHIMP/MPI User Guide*. University of Edinburgh, June 1994.

[4] G. D. Burns, R. B. Daoud, and J. R. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of the Supercomputing Symposium '94*, Toronto, Canada, June 1994.

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

[6] P.-C. Cheng, P. Vaughn, D. Reese, and A. Skjellum. The Unify System. Technical report, Mississippi State University, September 1994.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. The MIT Press, Cambridge, MA, 1994.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User' s Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, TN 37831-8083, 1. edition, May 1994.

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

[10] J. Meyer. Message-Passing Interface for Microsoft Windows 3.1. Master's thesis, Department of Computer Science and the Faculty of the Graduate College, University of Nebraska, Omaha, NB, December 1994.

[11] J. Pruyne and M. Livny. Providing resource management services to parallel applications. In J. Dongarra and B. Tourancheau, editors, *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, SIAM Proceedings Series, pages 152–161. SIAM, May 1994.

[12] G. Stellner, A. Bode, S. Lamberts, and T. Ludwig. NXLib — A parallel programming environment for workstation clusters. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*, pages 745–748, Berlin, July 1994. Springer Verlag.

[13] G. Stellner. Consistent Checkpoints of PVM Applications. In *Proceedings of the 1st European PVM Users Group Meeting*, http://www.labri.u-bordeaux.fr/~desprez/CONFS/PAPERS/abs010.ps.gz, 1994.

[14] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, (2):40–48, February 1995.