UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

The implementation of dynamite: an environment for migrating PVM tasks

Iskra, K.A.; van der Linden, F.; Hendrikse, Z.W.; Overeinder, B.J.; van Albada, G.D.; Sloot, P.M.A.

Publication date
2000

Published in
Operating Systems Review

Link to publication

Citation for published version (APA):
Iskra, K. A., van der Linden, F., Hendrikse, Z. W., Overeinder, B. J., van Albada, G. D., & Sloot, P. M. A. (2000). The implementation of dynamite: an environment for migrating PVM tasks. *Operating Systems Review*, *34*(3), 40-55.

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:18 Jan 2025

Reference to this paper:
K.A. Iskra, F. van der Linden, Z.W. Hendrikse, B.J. Overeinder, G.D. van Albada, P.M.A. Sloot, "The implementation of Dynamite – an environment for migrating PVM tasks," Operating Systems Review, Vol. 34, No. 3, pp.40–55, (July 2000).
(Copyright Universiteit van Amsterdam).

The implementation of Dynamite — an environment for migrating PVM tasks

K. A. Iskra, F. van der Linden, Z. W. Hendrikse, B. J. Overeinder,
G. D. van Albada, P. M. A. Sloot

Informatics Institute, Universiteit van Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
email: {kamil,frank,zegerh,bjo,dick,sloot}@science.uva.nl

Abstract: Parallel programming on clusters of workstations is increasingly attractive, but dynamic load balancing is needed to make efficient use of the available resources. Dynamite provides dynamic load balancing for PVM applications running under Linux and Solaris. It supports migration of individual tasks between nodes in a manner transparent both to the application programmer and to the user, implemented entirely in user space. Dynamically linked executables are supported, as are tasks with open files and with direct PVM connections. In this paper, we describe the technical aspects of migrating message-passing tasks.

Keywords: cluster computing, task migration, message-passing, PVM

1 Introduction

As processors become more powerful and local area networks become faster, and as the prices of both decline, parallel computing on networked workstations becomes increasingly interesting and competitive with dedicated parallel machines. PVM [1] and MPI [2, 3] are two prominent examples of environments for parallel computing on clusters of workstations.

An important problem with using a cluster of workstations for parallel computing is that the available capacity of individual nodes changes in time, as other users execute their programs on the same nodes. The resource requirements of the individual tasks of the parallel application can also change in time. Due to the time-varying performance requirements and resources' availability, the optimal task allocation changes dynamically.

Building on earlier *DPVM* work by L. Dikken, F. van der Linden, J.J.J. Vesseur, P.M.A. Sloot, R.N. Heederik, and B.J. Overeinder [4, 5, 6], we have developed Dynamite[1] in the Esprit project 23499. Dynamite [7] attempts to maintain optimal task allocation for parallel jobs in dynamically changing environments by migrating individual tasks between the nodes. An additional advantage of task migration is that it is possible to free individual nodes, should they for example need to be serviced, without breaking the computations.

Dynamite supports applications written for PVM 3.3.x, running under Solaris/UltraSPARC 2.5.1 and 2.6 and Linux/i386 2.0 and 2.2 (libc5 and glibc 2.0 binaries are supported[2]). From the user's perspective, all that is needed is to relink the application with Dynamite's version of the *PVM* libraries and with the Dynamite dynamic loader. Moreover, the checkpointing mechanism can be used for non-*PVM* applications as well.

Essentially, Dynamite consists of three components: migration, monitoring and scheduling. This paper focuses on the first, on the technical aspects of task migration in *PVM*.

The rest of this paper is organised as follows: Section 2 presents an overview of the environment and of issues associated with task migration. Section 3 describes the mechanisms of process checkpointing and restarting. In Section 4, the modifications that had

[1] Dynamite is a collaborative project between ESI, the Paderborn Center for Parallel Computing, Genias Benelux and the Universiteit van Amsterdam. Of the many people that have contributed, we can mention only a few: J. Gehring, A. Streit, J. Clinckemaillie, A.H.L. Emmen.

[2] glibc 2.1 is not supported at this point.

to be made to *PVM* itself in order to support task migration are described. Section 5 describes the limitations of the approach used, and Section 6 presents an overview of Dynamite performance. Section 7 presents the related work, and Section 8 concludes the paper with a summary and future work.

2 Migration overview

Parallel *PVM* applications consist of a number of processes (also called *tasks*) running on interconnected nodes forming a *PVM virtual machine*. A *PVM daemon* runs on every node and communicates with other daemons using the UDP/IP protocol. *PVM* tasks communicate with each other and with *PVM* daemons using a message-passing protocol. *PVM* message passing is reliable: no message can be lost, corrupted or duplicated, messages between two individual tasks arrive in the order sent.

In Dynamite, a *monitor* process is started on every node of the *PVM* virtual machine. This monitor communicates with the local *PVM* daemon and collects information on the resource usage and availability, both for the node as a whole and individually for every *PVM* task. The information is forwarded to a central *scheduler*, which makes migration decisions based on the data gathered. *PVM* daemons assist in executing these decisions.

The term *task migration* is used to describe the action of moving a running process from one node to another.

First, the state of the running process must be consistently captured on the source node. This operation is known as *checkpointing*. The process is subsequently *restored* on the destination node, with its state initialised from the checkpointed process. Thus, its execution resumes from the point at which the source process was checkpointed. Typically, the original process on the source node is terminated.

An interface must be made available that allows to trigger the checkpointing asynchronously from the outside of the running process, for example from within the local *PVM* daemon. Signals are typically used for this purpose.

The items that contribute to the state of the running process include:

- memory segments: text (i.e. program code), data, dynamically allocated data, shared libraries (text and data segments) and stack,
- processor registers,
- signal handlers and the signal mask,
- open files.

Capturing the shared libraries is in general problematic, because no standardised interface is available that would provide information on what shared libraries are in use and where they reside.

Open files pose another difficulty when migrating tasks. Again, no standardised interface to get the list of the currently open files is available. Moreover, the open files might be local to the source machine, and thus not readily available after the migration.

Processes that are part of the parallel *PVM* application present additional difficulties. Every *PVM* task has a socket connection (typically, TCP/IP) with the local *PVM* daemon. This connection is used for the *indirect routing*, i.e. when messages between tasks are routed through the *PVM* daemons. *PVM* tasks can also establish point-to-point *direct* TCP/IP communication channels with each other, to improve the performance. Extra care must be taken when migrating *PVM* tasks to ensure that they do not permanently lose the connection with the rest of the parallel application, and that the *PVM* message protocol as outlined above is not violated.

3 Preserving the memory image

Preserving the memory image of a process boils down to writing the process's address space to a file (checkpointing) and retrieving its contents afterwards (restoring it to memory). This includes the memory segments of the process itself and those of the shared libraries used by the process, to make the checkpoint file self-contained. In addition, the contents of processor registers have to be taken care of, such as the program-counter and the stack pointer. Moreover, a comprehensive implementation should also consider communication channels such as open files and TCP/IP sockets.

The Dynamite checkpointing support has been implemented directly in the dynamic loader: the Linux ELF dynamic loader version 1.9.9 has been modified to provide the necessary functionality under both Linux and Solaris. The ELF dynamic loader is a low-level user-space component of a running UNIX process: this is where the checkpointing can be implemented most efficiently and transparently. The

```
00010000    8K r/x    dev:176,4341 ino:259415
00020000    8K r/w/x  dev:176,4341 ino:259415
00022000  232K r/w/x    [ heap ]
EF680000  592K r/x    /usr/lib/libc.so.1
EF714000   56K -        [ anon ]
EF722000   32K r/w/x  /usr/lib/libc.so.1
EF72A000    8K r/w/x    [ anon ]
EF780000    8K r/x    /usr/lib/libdl.so.1
EF790000    8K r/w      [ anon ]
EF7A0000   80K r/x    dev:176,4341 ino:651982
EF7C2000   16K r/w/x  dev:176,4341 ino:651982
EF7C6000  208K r/w/x    [ anon ]
EFFFA000   24K r/w/x    [ stack ]
```

a

```
00010000    8K r/x    dev:176,4341 ino:487872
00020000    8K r/w/x  dev:176,4341 ino:487872
00022000  232K r/w/x    [ heap ]
EF680000  592K r/x    dev:176,4341 ino:487872

EF722000   40K r/w/x  dev:176,4341 ino:487872

EF780000    8K r/x    dev:176,4341 ino:487872
EF790000    8K r/w      [ anon ]
EF7A0000   80K r/x    dev:176,4341 ino:651982
EF7C2000   16K r/w/x  dev:176,4341 ino:651982
EF7C6000  208K r/w/x    [ anon ]
EFFFC000   16K r/w/x    [ stack ]
```

b

Figure 1: Process memory map before (a) and after (b) the migration.

advantages of implementing checkpointing support in the dynamic loader include:

- the ability to run arbitrary code before the application starts running, like installation of the checkpoint signal handler: there is no need to modify the startup code in the `crt*.o` files nor to change the name of the `main` function in the application source code,
- the ability to wrap any dynamically-bound function calls, like file manipulation or memory management calls: there is no need for additional libraries when linking the application, the original library functions are invoked by the wrappers,
- straightforward support for shared libraries: the dynamic loader has full control over where to map them and can record this information for later retrieval.

When the application starts, the dynamic loader records the locations of all the memory segments: text, data and stack; it loads requested shared libraries into memory, also recording their locations, and performs dynamic linking. Afterwards, the handler for the checkpoint signal (currently `SIGUSR1`) is installed, and the user code starts executing.

Figure 1 (a) presents an example memory map of a running process, obtained with the `/usr/proc/bin/pmap` command under Solaris 2.6 (and minimally hand-edited to fit on the page). The first three lines denote the text, data and heap (i.e. dynamically allocated memory) segments (the `dev` entries indicate that the `mmap`ped file resides on a remote NFS server). There is a large space afterwards (please compare the addresses), since the heap can grow upwards. The next four lines are the entries of the dynamically linked `libc` library: text, an unused memory hole, data and uninitialised data (`anon` entries indicate zero-initialised segments), followed by the text segment of the `libdl` library. The next four entries belong to the dynamic loader itself: there is a page used as a pool for dynamic memory allocations early in the dynamic loader's startup procedure, followed by text, data and uninitialised data segments of the dynamic loader. Finally, there is the stack segment, which grows downwards, so there is some space left for it between the last two segments (again, please compare the addresses).

3.1 Checkpointing

Checkpointing takes place when the checkpoint signal is delivered (in case of *DPVM* applications, this is done by the local *PVM* daemon). Before the `ckpt_handler` signal handler starts executing, the operating system preserves the contents of all the CPU registers on the stack. This simplifies the checkpointing procedure.

The signal handler calls `sigsetjmp` in order to save the signal mask and essential registers, in particular the stack pointer, in a buffer — these will be used when restoring.

Next the location of the checkpoint file is determined. This depends on environment variables such as `DPVM_CKPTDIR` and `HOME`. For *DPVM* applications, the `dpvm_usersave`[3] procedure is invoked to prepare

[3]The `dpvm_usersave` function and its companion `dpvm_userrestore` are hooks for the user to provide code to be executed during checkpointing and restoration. We use these functions to preserve *PVM* communication status across a migration.

for the checkpointing, see Section 4.3 for details.

The state of open files is saved subsequently. For every open file, a position of the file pointer is obtained with the `lseek` call. Next, the state of signal handlers is obtained and saved using `sigaction`. Other relevant file status information is captured during program execution by the wrappers for functions like `open` and `chdir`.

Finally, the `ckpt_create` procedure writes the checkpoint file to disk. All the data segments must be stored, since the process has very likely modified them. This applies both to initialised and originally uninitialised segments, which are merged when checkpointing. The dynamically allocated heap segment, the top of which is obtained using the `sbrk` call, is also stored. Both the application data segments and those of the shared libraries are stored, as is the CPU stack segment. The text segments of the process itself and those of the dynamically linked shared libraries are also stored. While this is not strictly required, it makes the whole migration operation more robust, since it eliminates the danger of using a wrong version of some library when restoring, should the two machines be not strictly identical, as often is the case. Essentially, all of the process's address space is written to the file.

Certain low-level data is needed by the dynamic loader when restoring, most importantly the precise locations of all the memory segments and information on the state of open files. All such cross-checkpoint data is stored in the `privdata` structure, which is part of the data section, and is thus stored in the checkpoint file. A pointer to this data is stored separately, in a place easily accessible to the dynamic loader when restoring.

Once the checkpoint file is written, the task terminates.

3.2 Restoring

The checkpoint file is actually a complete ELF executable. The original executable file is not needed when restoring. Instead, the checkpoint file should be executed using standard UNIX `exec` call.

Just as with standard dynamically linked executables, the execution of the checkpoint file begins in the dynamic loader. The checkpoint file differs substantially from a standard executable, for example it has additional sections. This allows the dynamic loader to recognise that it is dealing with a checkpoint file and to invoke the restoring procedure `ckpt_restore`.

At this point, the text and data sections — the first two lines in Figure 1 (b) — are already loaded into memory by the operating system, while the rest is not. This is because these two sections are marked in the executable checkpoint file as `PT_LOAD`, the rest as `PT_NOTE`. The ELF specification requires that the operating system automatically loads the former before invoking the dynamic loader, but not the latter.

The dynamic loader finds the section that contains the address of the cross-checkpoint data structure, and goes on to restore the heap segment. A `brk` call is used to allocate the memory for the heap and its contents are initialised with a simple `read` from the checkpoint file. Subsequently, all the shared library segments are restored by `mmap`ping them from the checkpoint file to memory.

Restoring stack is more complicated, since there is a considerable danger of overwriting the frame and return address of the currently executing function. To prevent this from happening, the restoring routine calls itself recursively until its stack frame is safely beyond the dangerous area, at which point the stack can be restored with a simple `read` call.

Experiments with different approaches to restoring have been conducted, such as letting the dynamic loader "manually" load all of the sections, including the text and data sections, or letting the operating system do all the loading by marking all the sections as `PT_LOAD`. The heap segment turns out to be particularly sensitive to such changes: if shared libraries are loaded by the kernel, the heap is assumed to begin where they end, i.e., several GB from where it should. Similarly, if the heap segment is merged with the data segment and is `mmap`ped into memory, it is impossible to decrease it with `sbrk`, because for the UNIX kernel it is not a heap segment. The current behaviour most closely resembles that of the standard executables: text and data segments are loaded by the kernel, heap is allocated using `brk` and shared libraries are `mmap`ped. It is thus expected to be the most portable approach.

The address space of the process is basically restored at this point. The following differences between Figures 1 (a) and (b) can be observed:

- since the memory hole in the middle of `libc` is unused, it is "missing" in Figure 1 (b),
- initialised data / uninitialised data segments are merged, which results in the second "missing" entry — comparing the sizes proves that nothing is actually lost,

- shared libraries are `mmap`ped from the checkpoint file,
- stack usage is smaller — apparently, the process in question needs a lot of stack when initialising, but since that initialisation is skipped when restoring from the checkpoint, the stack usage is now smaller.

A `siglongjmp` call is made to jump back to the checkpoint signal handler and to restore the signal mask.

The state of the signal handlers is restored using `sigaction`, and the previously open files are restored using `open` and `lseek`. For *DPVM* applications, the `dpvm_userrestore` procedure is invoked to recover from the checkpointing, see Section 4.2 for details.

When the signal handler returns, the operating system restores all the CPU registers and the application resumes its execution, unaware of anything that happened.

3.3 Call wrapping

As has been mentioned at the beginning of Section 3, one of the advantages of adding checkpointing support to the dynamic loader is the ability to wrap function calls. This is done at the application start-time, without doing any modifications to the shared libraries or to the object code of the application.

In the dynamic loader, the `_dl_find_hash` function is used to resolve unbound external references. This function gets the symbol name (a character string) as a parameter, and returns the address of the symbol: this address is stored in the text or data segment of the process. It is straightforward to modify this function to return faked addresses for certain symbol names — the addresses of wrapper functions in the dynamic loader. The wrappers can do their job, and call the actual function, the address of which is stored in a private variable of the dynamic loader.

The following functions are wrapped:

- file manipulation functions: `open`, `close`, `creat`, `dup`, `chdir`, `fchdir`,
- memory mapping functions: `mmap`, `munmap`, `mremap`[4].

The file manipulation functions must be wrapped in order to have the names of the open files: they are stored outside of the process's address space, so they can only be obtained when the file is being opened.

[4] `mremap` is Linux specific.

The need to wrap memory management functions became apparent when doing the port to Linux. Linux libc library uses `mmap` intensively as an efficient memory allocator. For example, `malloc` calls `mmap` when asked for large memory blocks (where "large" is defined to be more than or equal to 128 KB by default), for smaller blocks it allocates on the heap via `sbrk`. Therefore, the `mmap`ped memory regions must be written to the checkpoint file and restored as well, just like shared libraries.

4 Preserving communication

Checkpointing and restoring of the process's address space as described in Section 3 does not preserve communication. Extra precautions that need to be taken to preserve communication include:

- flushing and closing direct connections,
- disconnecting from the *PVM* daemon before checkpointing,
- reconnecting to the *PVM* daemon after restoring.

The protocol used for the above steps must meet basic correctness requirements with respect to the messages exchanged between the tasks, namely:

- no messages are allowed to be lost,
- message data received must be the same as sent,
- messages must be delivered in the right order.

4.1 *PVM* task identifier

In *PVM*, every task of the parallel application has a unique task identifier. Part of the task identifier denotes the node on which the task is running, and part denotes the task number within that node. The *PVM* daemons need to know which node the task runs on when routing a message to it.

It is essential that the task remain accessible through its old identifier after it is migrated to a different host, otherwise the messages sent by other tasks will not reach it. In *DPVM*, the identifier of the task never changes: it remains the same no matter how many migrations are made.

As a consequence, the node identifier encoded in the task identifier cannot be trusted. *DPVM* solves this problem by maintaining in the *PVM* daemons the routing database for migrated tasks, which contains the current locations of migrated tasks.

4.2 Migration protocol

The migration protocol of *DPVM* consists of 4 main stages, as shown in Figure 2, in which *task 1* is migrated from *node 1* to *node 3*. The nodes that are active at a particular stage are marked gray.

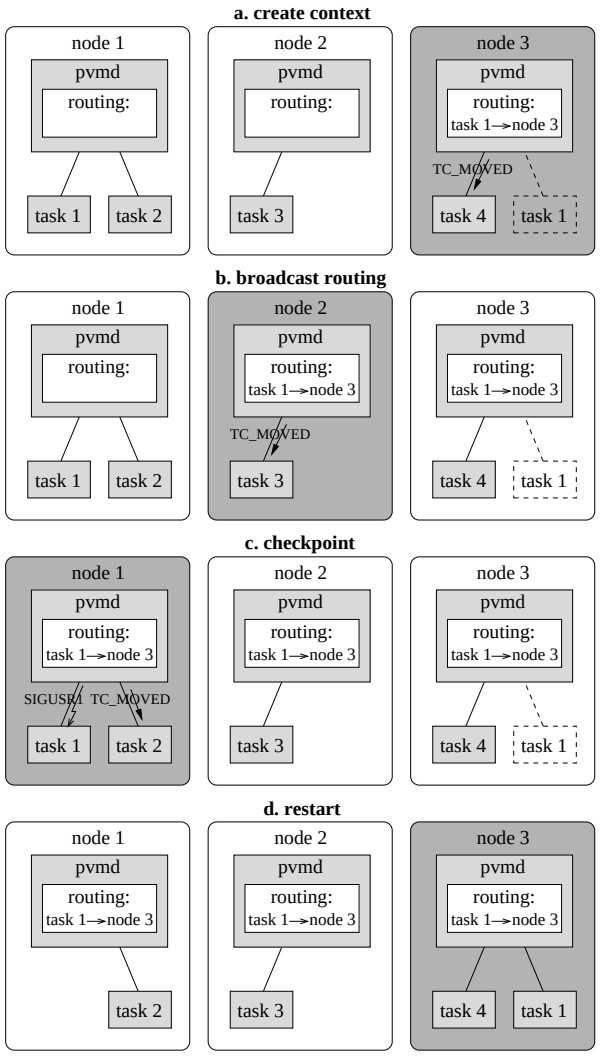

Figure 2: *DPVM* migration protocol. *task 1* is migrated from *node 1* to *node 3*. Gray nodes are *active*.

The 4 stages are executed in sequence, as requested by the *PVM* task that calls the migration function `pvm_move`. This is typically the interactive *PVM* console or an external scheduler.

In stage (a), a new task context for the migrating task is created on the destination node (*node 3*). The routing table on that node is updated to indicate that the task to be migrated is running on this node. While this is not really the case (no process was started, only control structures in the *PVM* daemon were allocated), the daemon will accept and temporarily store any messages it receives that are addressed to the migrating task. Along with updating the routing table, all tasks running on the node are notified of the approaching migration using the `TC_MOVED` message — the purpose of this will be explained in Section 4.3.

At this point, there are actually two nodes that claim to have the migrating task running on them, and are willing to accept messages for it. Messages from all the nodes but the destination node are still routed to the source node, where the migrating task is still running, completely unaware that it is soon to be migrated.

In stage (b), the new routing information is broadcast throughout the *PVM* virtual machine: all the nodes but the source node update their routing tables (*node 2*).

Stage (c) actually consists of two different operations executed on the source node (*node 1*). First, the routing table is updated. Once this is done, all new messages addressed to the migrating task are expected to go to the destination node. Next, the checkpointing signal is sent to the task. The task terminates its *PVM* connections (this will be described in detail in Section 4.3), creates the checkpoint file as described in Section 3.1 and terminates.

In stage (d) the checkpointed task is restored on the destination node, as described in Section 3.2. The dynamic loader invokes the `dpvm_userrestore` routine, which in turn calls `pvmbeatask`, a standard *PVM* startup function. This function reconnects the task to the destination *PVM* daemon using a somewhat modified connection protocol (there is no need to allocate a new task identifier, so parts of the initialisation can be skipped). Control is passed back to the application code, and the *PVM* daemon on the destination node can finally deliver all the messages addressed to the migrating task which it had to store during the migration.

4.3 Connection flushing

When the migrating task receives the checkpoint signal and before it terminates, it must flush all open connections to ensure that no messages are lost during the migration. Figure 3 presents the connection flushing protocol used by *DPVM*, both for the direct task-task connections and the connection to the

PVM daemon. Two application tasks are presented in the figure — the migrating task and one *remote task*. The *remote task* represents every task of the parallel application but the migrating one: all the tasks perform the actions of the *remote task* when flushing the connections.

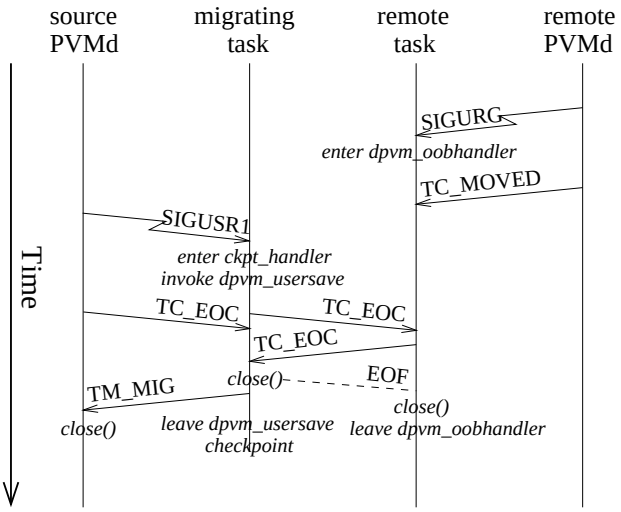

Figure 3: *DPVM* connection flushing protocol.

In Figure 2 (a) through (c), one could observe the `TC_MOVED` messages sent to all the tasks. These messages initiate the flushing protocol. Before the message is sent to the remote task, however, the *PVM* daemon local to the task signals the task using the `SIGURG` signal, and that task enters the `dpvm_oobhandler` function. The sole purpose of this is to assure a timely response from the task; otherwise, it could take arbitrarily long before the message was received, should the task be busy with computations at that time.

The remote task extracts the migrating task's identifier from the `TC_MOVED` message and checks if it has an established direct connection with the migrating task. If there is no direct connection, the remote task returns immediately from the signal handler. If the connection is only partially established, it is abandoned and the task returns immediately, too. Therefore, the rest of the protocol is only performed for tasks with fully established direct connections.

Once the `TC_MOVED` notification is sent to all the remote tasks, the migrating task receives the `SIGUSR1` migration signal. The `ckpt_handler` function from the dynamic loader invokes the `dpvm_usersave` routine, which is responsible for performing the migrating task's part of the flushing protocol.

The migrating task begins by sending a `TC_EOC` message via every fully established direct connection. The connections that were not fully established are abandoned.

Meanwhile, the remote task waits and reads all the messages from the direct connection to the migrating task. Once it gets the `TC_EOC` message, it stops, because this message is guaranteed to be the last message sent by the migrating task. The remote task replies with an identical `TC_EOC` message.

The migrating task reads the data available from all direct connections. Once it gets the `TC_EOC` message, it closes the connection. As a result of the close on the migrating side, the remote task gets an `EOF` when trying to read from the socket, and can close the connection from the other side, too, and, finally, return from the signal handler.

Thanks to this protocol it is guaranteed that, once the `TC_EOC` messages are exchanged, all the messages have been read: there are no more messages *on the wire* or in the kernel buffers.

Similar actions are performed for the connection to the source *PVM* daemon. After sending the migration signal, the source *PVM* daemon sends the `TC_EOC` message to the migrating task, as the end-of-connection marker. Once the migrating task closes all the direct connections and receives the `TC_EOC` message from the *PVM* daemon, it sends back the `TM_MIG` message to the daemon, which then closes its connection to the migrating task. The flushing is completed, and the migrating task can safely checkpoint itself and terminate.

4.4 Critical sections

The `SIGUSR1` or `SIGURG` signals can be sent by the *PVM* daemon at any time. However, there are situations in which the application task is not prepared to handle the signal immediately, for example in the middle of sending a message fragment.

DPVM acts conservatively in this respect. Most *PVM* function calls are protected against signals using `sigprocmask` system calls, which block the signals on the function entry and unblock them on exit. There are of course functions that clearly do not need to be protected, and are not, like the data packing/unpacking routines, which do not perform any communication at all.

However, some communication functions can block for an arbitrarily long amount of time. `pvm_recv` is

a classic example of such a call. But even `pvm_send` can block when direct communication is used. Direct communication requires active participation on the receiver side, both when establishing a connection and when sending large messages that do not fit in the kernel buffers.

In *DPVM*, the problem has been solved by modifying the lowest-level functions of the *PVM* library, in particular `mxfer`. This function handles all the communication, both when sending and receiving messages, and it is this function that the *PVM* application blocks in when communicating. The potentially infinitely blocking `select` call has been replaced by one that blocks for no more than a second. On return from `select`, the signal state is checked using the `sigpending` system call. Should there be a signal pending, it is unblocked and delivered. The advantage of this approach is that there is just a single, carefully chosen place inside communication functions where the signal can be delivered. This makes it simpler to resume the interrupted communication gracefully, once the signal handler returns, which will often be on a different machine, after migration.

Thanks to the modifications made to these low-level functions, it is safe to call the communication functions from within the signal handlers, i.e. the communication functions have been made partially reentrant.

An issue worth considering at this point is what happens to messages that were only partially sent or received when the migration took place?

Indirect communication does not need any special treatment. The beginning of the message can be sent from/received on one node, and the end of the message on another, without any complications.

Direct communication is more difficult to handle. Because of the way *PVM* is designed, it is not possible to receive a single message through two different communication channels. For indirect communication this is not a problem, because from the point of view of the *PVM* library there is just one indirect communication channel, the same before and after the migration. Direct connections are closed during the migration, so messages partially sent or received are discarded. However, after restoration the messages are fully resent using the indirect communication, so no data is lost. The direct connection is reestablished as soon as there are new messages to send.

4.5 Message forwarding

PVM daemons use the UDP/IP protocol to exchange messages. This is an unreliable protocol, so the message fragments sent can get lost or arrive out of sequence. While the *PVM* daemons can handle such problems under normal circumstances, task migration presents an additional burden.

Before going any further, the concept of *message fragments* must be explained. Large *PVM* messages are sent in smaller packets called fragments, typically no larger than 4KB. Every fragment is preceded by a *fragment header*, and the first fragment is also preceded by a *message header*. Fragment headers contain information such as the source and destination task identifiers and flags to identify the first and the last message fragment, while message header contains, among others, the message *code* value.

Going back to Figure 2: it could for example happen that a message fragment sent from *task 3* to the migrating *task 1* in stage (a) arrives at the *PVM* daemon of *node 1* after *task 1* begins checkpointing, i.e. in stage (c), improbable though it may be.

In *DPVM*, such a fragment is *forwarded* by the daemon to the new destination node. However, *task 3* might have sent other messages to *task 1* in the mean time, and if they went directly to the new destination node, they might have arrived there earlier than the forwarded message, although they were sent later. This is a violation of the *PVM* protocol [1].

To fix this problem, *DPVM* must put additional information in the message and fragment headers.

First, it puts the total length of the message in the message header — in standard *PVM* this information is not needed, since the end of the message is marked by a special flag in the last fragment's header. Because fragments (including the last fragment) can arrive in wrong order in *DPVM*, this would not be sufficient.

Then, every fragment is uniquely identified using the message identifier and fragment sequence identifier. This way fragments, or even whole messages, that arrive out of order can properly be taken care of. This feature is also needed to support reentrance in communication functions (see Section 4.4), since it allows a task to send a new message before it is finished with the previous one — something that the standard *PVM* communication protocol could not do, because it would interpret the new message as part of the previous one.

4.6 Establishing direct connections

Figure 4 presents the direct connection establishment protocol used by *DPVM*, which is a slightly modified version of the standard *PVM* protocol. Regions marked gray indicate places in which the given task can be migrated — in other places signals are blocked, see Section 4.4.

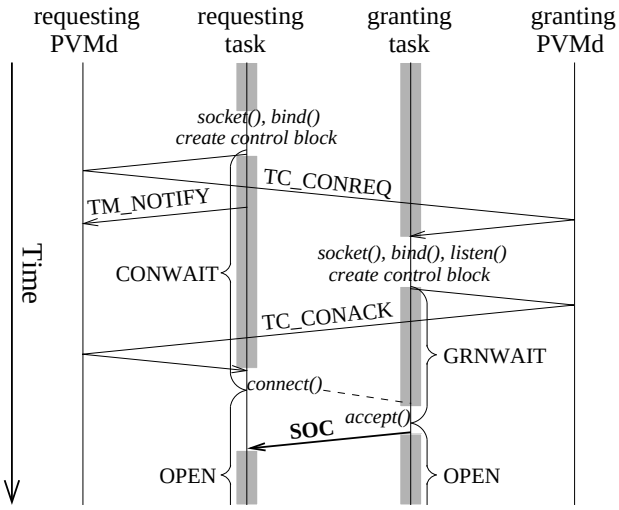

Figure 4: *DPVM* direct connection establishment protocol. In gray regions tasks can be migrated.

The connection establishment protocol is initiated by a task that wants to send a message to another task, if the user requested direct routing using `pvm_setopt`. The *requesting task* creates a port and a task-task communication control block that describes the connection. The connection, according to *PVM* nomenclature, is in `CONWAIT` state (waiting to connect). The task then sends `TC_CONREQ` connection request message to the other task (called from now on *granting task*) via the indirect route, and also requests the *PVM* daemon to notify it (`TM_NOTIFY`) when the granting task exits (at which point the connection should be closed).

When the granting task receives the connection request, it creates a listening port and the communication control block. The connection on this side is in `GRNWAIT` state (granted, waiting). The granting task replies via the indirect route with `TC_CONACK` message that contains the location of the created port.

The requesting task can now `connect` to the granting task and change the connection state to `OPEN`, i.e. fully established. At some later time, the granting task `accepts` the connection and changes the connection state to `OPEN`, too. That is the end of the standard *PVM* direct connection establishment protocol.

As shown in Figure 4, either of the two tasks can be migrated at practically any time while establishing the connection. Migrating a task can cause the protocol to fail. For example, if the granting task is migrated right after it sends the `TC_CONACK` message, the port location provided in that message might no longer be valid by the time the requesting task gets the message, so the subsequent `connect` will fail. Another example would be if the requesting task was migrated at that point: when it is restored and wants to send a new message, it will restart the protocol, sending another `TC_CONACK` message when the granting task already handled one. Modifications made in *DPVM* included making the code robust enough to recover gracefully from any such error conditions and protocol violations.

It was found that the most reliable way to do this is to immediately abandon partially established direct connections both on the migrating side and on the other side. Thanks to the fact that in *DPVM* all the tasks are notified about the migration, this is easily done.

One case has to be handled separately, though. In the standard *PVM* protocol, after the requesting task successfully connected to the granting task, it changes the connection state to `OPEN` and starts sending data via this channel. The connection on the granting task's side, however, is still in the `GRNWAIT` state, i.e. it is only partially established. Should the migration happen at this point, the connection will be abandoned and the data in the channel will be lost. *DPVM* prevents this by adding additional synchronisation step between the two tasks (marked in boldface in Figure 4). The requesting task does not send data via the channel immediately after connecting to the granting task — it waits. When the granting task accepts the connection, it sends the start-of-connection `SOC` message through it, which is read by the requesting task. This way, there is a guarantee that if a migration signal is delivered to a task and the direct connection is not fully established, no data will be lost if the connection is abandoned. Similarly, if a signal is delivered and the direct connection is in `OPEN` state, it is in the `OPEN` state on the other side, too (see Figure 4).

Another remark that can be made at this point is that establishing a direct connection requires active participation on both sides. The requesting task re-

acts in a timely manner (it has nothing else to do), but for the granting task this can take arbitrarily long, since it can execute application code both before it handles `TC_CONREQ` and before it calls `accept`. Therefore, even for a very small message a call like `pvm_send` can block, provided that it is the first call after direct routing has been requested.

4.7 Multicasts

The protocol used by *PVM* when sending multicast messages, i.e. by `pvm_mcast`, but also by `pvm_bcast` (which calls `pvm_mcast` internally) is significantly different from that used when sending standard point-to-point messages.

First, direct communication is never used in this case: messages are always routed through *PVM* daemons.

More importantly, sending the message itself is preceded by an initialisation step. First, the task sends the list of the destination task identifiers to the local *PVM* daemon. The daemon sorts the list by the nodes the destination tasks run on and sends parts of the list to the *PVM* daemons on the concerned nodes. Finally, it sends back a *multicast address* to the source task. The task then sends the real message, with the multicast address as its destination identifier, and the daemon forwards the message to other daemons, which in turn forward it to the destination tasks.

This presents various difficulties if the task is migrated in the middle of this protocol: the multicast address is only valid on one particular node, for example.

The easiest way to solve the problem is to disallow migrations while `pvm_mcast` is running, and that is what *DPVM* does. This is not a problem, since indirect communication is practically non-blocking, so the pending migration signal will be handled quickly.

5 Limitations

The mechanism of preserving memory image has considerable, although understandable, limitations. It is designed to preserve the memory image of the process and its open files, but nothing more than that. It is unable to preserve resources that are outside of the process's address space (in the kernel) or that are shared with other processes.

For example, processes that use any of the following features will not be migrated properly:

- pipes,
- sockets,
- System V IPC, like shared memory,
- kernel supported threads,
- file status as modified by e.g. `ioctl` and `fcntl`,
- `mmap`ping / `open`ing of special files (`/dev/...`, `/proc/...`, etc.).

Some of these, like sockets, might eventually be supported, but it is practically impossible to support migrating a process that communicates with other local processes via shared memory, for example[5].

Support for open files is limited to files residing on shared filesystems: the files must be available under the same pathname on the destination node as on the source node.

Also, the ability to migrate *PVM* tasks started from the terminal window, which is common for master tasks, is limited. While it is possible to migrate such tasks, this is probably not what the user wants, since the output of the task will from then on go to the *PVM* log files, instead of the terminal window. Shell output redirections cannot be preserved, either.

6 Performance evaluation

In a system like Dynamite, there are two easily measurable performance factors:

- how long it takes to migrate a task of a given size,
- what is the difference in communication performance compared to the standard *PVM*.

Experiments have been performed to measure the two factors, both under Linux and Solaris. In case of Linux, reserved nodes of a cluster have been used, equipped with PentiumPro 200 MHz CPU and 128 MB RAM, running kernel version 2.2.12. In case of Solaris, idle UltraSparc 5/10 workstations have been used, equipped with 128 MB RAM and running kernel version 5.6. In both cases, switched 100 Mbps Ethernet was used as the communication medium. However, it must be pointed out that in both cases the NFS servers used for checkpoint files were shared with other users, which could affect the performance

[5] Unless embedded within a protocol like *PVM* or *MPI*.

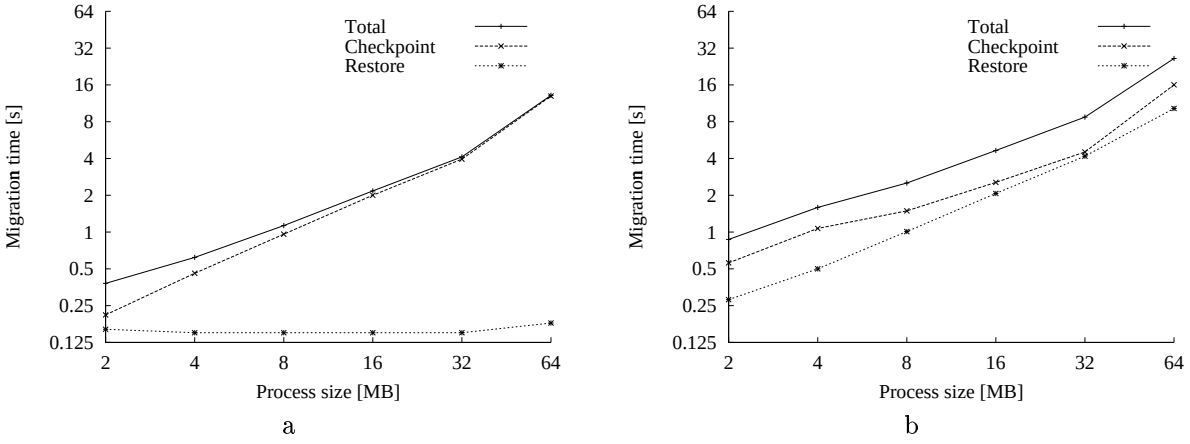

Figure 5: Migration performance of *DPVM* for (a) Linux and (b) Solaris.

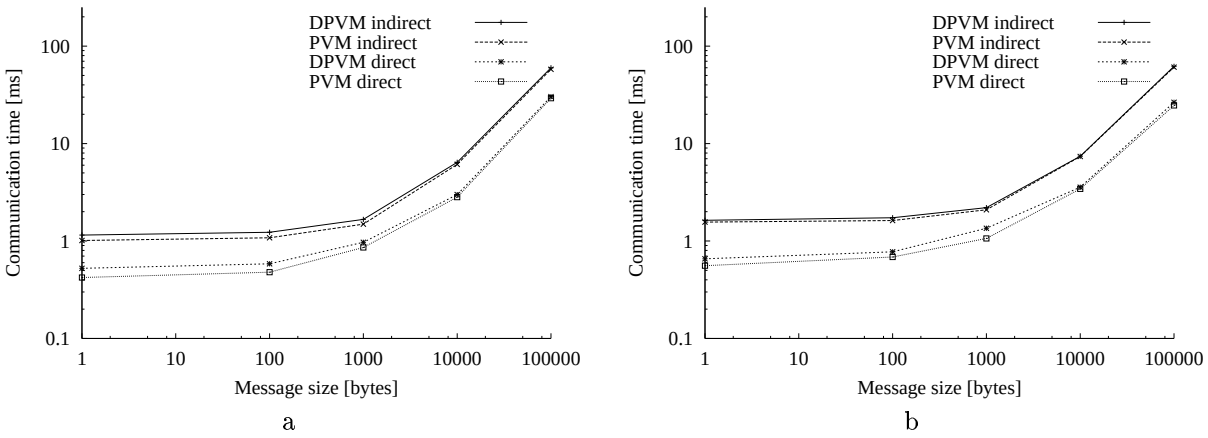

Figure 6: Communication performance in *DPVM* and *PVM* for (a) Linux and (b) Solaris.

to some extent.

Figure 5 presents the performance of migration in *DPVM* for various process sizes. A simple ping-pong type program communicating once a few seconds via direct connection was used, process size was set with a single large `malloc` call. Execution time of each of the four migrations stages (see Section 4.2) was measured. In general, it was found that the major part of the migration time is spent on checkpointing and restarting, the migration protocol and connection flushing amount to approximately 0.01 – 0.03s, and hence are not shown. The speed of checkpointing and restarting is limited by the speed of the shared filesystem. On our systems this limit lies at 4–5 MB/sec for NFS running over the 100Mbps network. It can be observed, however, that the restoring phase under Linux takes an approximately constant amount of time, while it grows with process size under Solaris, resulting in twice as large migration times for large processes. This is a side effect of differences in the implementation of `malloc` between the two systems. For large allocations, Linux creates new memory segment (separate from the heap) using `mmap`, whereas Solaris always allocates from the heap with `sbrk`. When restoring, the heap and stack are restored with `read` (see Section 3.2), which forces an immediate data transfer. However, for the other segments our implementation takes advantage of `mmap`, which uses more advanced *page on demand* technique, delaying network transfer until the data is actually needed. Since the allocated memory region is not needed to reconnect the task to the *PVM* daemon,

the time it takes to restart the task is constant under Linux. Clearly, delays may be incurred later, when the `mmap`ped memory is accessed and loaded.

In Figure 6, comparison of communication performance between *DPVM* and *PVM* is presented. Both indirect and direct communication performance has been measured. A ping-pong type program was used, which exchanged messages between 1 byte and 100KB in size. With *DPVM*, a slowdown is visible in all the cases. It stems from two factors:

- signal blocking/unblocking on entry and exit from *PVM* functions (function call overhead),
- extra header in message fragments (communication overhead).

The first factor adds a fixed amount of time for every *PVM* communication function call, whereas the second one increases the communication time by a constant percentage. For small messages the first factor dominates, since there is little communication. An overhead from 25% for direct communication under Linux to 4% for indirect communication under Solaris can be observed. While particularly the first difference in speed is significant, it must be pointed out that it represents a pathological, *worst case* scenario. The difference is due to the fact that the overhead percentage is larger for direct communication, since the communication is faster while the overhead from signal blocking/unblocking stays the same.

As the messages get larger, the overhead of signal handling becomes less significant, and the slowdown goes down to 2–4% for 100KB messages.

Tests have been made to compare the communication speed in *DPVM* before and after the migration, but no noticeable difference was observed (±1%).

7 Related work

Throughout the years, since the original publication on DPVM by Dikken et al. [4], a significant number of solutions for load balancing parallel jobs on networks of workstations by migrating individual tasks has been proposed. For *PVM* applications, this includes MPVM [8] (also known as MIST), CoCheck [9], tmPVM [10], ChaRM [11] and DAMPVM [12]. There are also similar solutions for *MPI*, including the already mentioned CoCheck [13] and Hector [14]. Systems that migrate sequential jobs have also been studied, e.g. Codine [15] and Condor [16] (the original DPVM was based on Condor).

7.1 Preserving memory image

Source code modifications. All of the above listed systems implement user-level checkpointing, i.e. no kernel modifications are needed. The user must link the application with additional or replacement checkpointing-aware libraries/startup files. Often, the source code must be modified by changing the name of the `main` function (MPVM, libckpt [17] (used by ChaRM)), or by adding a checkpoint initialisation function invocation at the start of `main` (tmPVM, DAMPVM). Contrast this with the dynamic loader approach used in Dynamite, where relinking with custom dynamic loader is all that is needed.

Shared libraries. Also, all of the above systems use the original executable file when restoring from the checkpoint, started using standard operating system `exec` call, only with special command-line arguments to force the process to read checkpoint data available in a file on a shared filesystem or via a TCP stream to the original process. The weak point here is the assumption that the new process's memory map will be the same as the old one. This is not guaranteed with ELF binaries — for example, in recent releases of Solaris, this assumption fails in case of shared libraries, which can be mapped into different memory regions, rendering the restored process unusable. While giving up shared libraries in favour of the static ones would solve the problem (and, indeed, MPVM requires static linking), static libraries are considered to be an obsolete concept. For example, Solaris 7 does not contain static versions of the 64-bit (sparcv9) libraries at all.

Condor checkpointing [18], as used in CoCheck, seems to be the only one attempting to tackle the problem with shared libraries. Like in other solutions, the original executable file is used when restoring, but the checkpoint file also contains the text and data segments of shared libraries. Condor's recovery code essentially performs the function of the dynamic loader on its own, by undoing the job done by the system dynamic loader and mapping the segments into appropriate memory regions.

Using the dynamic loader in Dynamite avoids the problem completely, since the restoring code has full control over where shared libraries are mapped.

Checkpoint file transfer. In general, two different methods of transferring checkpoint state are avail-

able: direct TCP stream transfer and a shared filesystem. MPVM and ChaRM use the former, while tmPVM, CoCheck and Dynamite use the latter. Unless checkpoint and restore are overlapped, a checkpoint-restart mechanism using a shared filesystem will incur a delay corresponding to at least two network transfers (from source to server and from server to target). The shared filesystem may add additional overhead, so that the direct TCP transfer can be faster by a factor of two or more. On the other hand, the solutions relying on shared filesystem are more modular, and thus more general and reusable. The checkpointing/restarting mechanisms of Condor and Dynamite are thus not limited to *PVM* applications.

Other approaches. An entirely different approach is used in DAMPVM. This system supports heterogeneous checkpointing, but the users pay a high price for that. Memory checkpointing/restoring is entirely directed by the user, who must create memory packing / unpacking routines. The restored process is restarted from the beginning of `main`, and it is the user's responsibility to recover gracefully and restart the computation at the point were it was left. The stack is not preserved, so local variables practically cannot be used, and mechanisms like recursive function calls are not practical, either. These limitations are not really surprising for heterogeneous checkpointing, but in practice they make the whole concept unusable for any larger applications.

MOSIX [19] uses yet another approach: kernel-level checkpointing. The solution, of course, is inherently unportable, but once implemented, the results can be impressive. The approach is geared for high performance and transparency, and completely unmodified binaries can be migrated at any time. Only the user-level part of the process is migrated, the kernel-level part, called *deputy*, stays on the old node and communicates with user-level part via a network socket. When the user code invokes a non-local system call, like signal delivery or an I/O operation, it is actually executed on the old node, the request and result are forwarded via the socket. Only lowest-level direct I/O device manipulation is not supported, and neither is writable shared memory. On the other hand, leaving the kernel-level part on the old node means that that machine cannot be rebooted, and many trivial non-local system calls, like `gettimeofday`, become extremely expensive due to network communication.

Portability. One drawback of the dynamic loader approach as used in Dynamite is its limited portability: it can only support operating systems that use *ELF* executable format, since that is what the dynamic loader has been designed for, and extending it is impractical. However, given the problems with shared libraries discussed above, any checkpointing system that attempts to be usable on concurrent operating systems must get very low-level and thus poorly portable at some point. While the checkpointing mechanism of Condor is more general, it only supports shared libraries if `/proc` filesystem is available, and the other user-level checkpointers (MPVM, tmPVM, libckpt) do not provide satisfactory support for shared libraries. The approach used in DAMPVM is the most portable, just opposite to the one in MOSIX.

7.2 Preserving communication

Moving on to the communication aspect of process migration, the solutions available use different mechanisms here, as well.

In **tmPVM**, a task to be migrated creates a checkpoint file (actually, it spawns an external extractor to do that), but does not exit immediately. It waits until the task is restarted on the destination node, and in the mean time forwards any messages that it receives to the destination (new) instance. The destination instance has a new *PVM* task identifier after the migration, so a special *TID alias directory* is maintained in the *PVM* daemons to translate between the original and current task identifier. Only indirect routing is supported. Moreover, message forwarding is not accompanied by sequencing, so messages can be received out of order.

A different protocol is used in **CoCheck** and **ChaRM**. The protocol is implemented outside of *PVM*, so it is a version-independent add-on. All *PVM* functions are wrapped, and the migration signal is blocked while they are executing. When a migration is about to happen, every task receives a signal and a message. This way, if a task is performing computations, a signal handler is invoked, which can safely call non-reentrant *PVM* functions. If, on the other hand, a task is blocked inside a *PVM* communication function, the message sent together with the signal can cause the *PVM* function to return, and the signal can be unblocked. However, this raises concerns about the transparency of such a solution: essentially, extra messages are sent to application tasks,

and care must be taken to ensure that these messages are never seen by the application code. This is easy in a Dynamite-like approach, which uses a modified *PVM* library, but is definitely much more difficult in a system based solely on wrappers.

In **CoCheck**, once a task receives information about the migration, it performs message flushing protocol: it sends *ready* messages to all the other tasks of the application, and then waits for such messages from all the other tasks. When a task receives messages from all the other tasks, it assumes that no other messages are pending, and the connections can be broken. A disadvantage of CoCheck is that all the tasks break their connections, even if only one is to be migrated, which is clearly suboptimal. The tasks then reconnect to the *PVM* daemons. Like in tmPVM, tasks get new task identifiers, so a mapping table must be maintained. Unlike in tmPVM or Dynamite, the table is maintained in every task of the application, and any wrapper for a *PVM* call that uses task identifiers is responsible for translating them back and forth.

The protocol used in **ChaRM** is far more optimal, since message flushing is only performed for tasks to be migrated: other tasks can communicate with each other without problems. Even messages sent to the migrating tasks during the migration do not block, because they are stored in a *delaying buffer* in the source task, and are resent when the migrating tasks resume their execution (at which point all the other tasks once more receive the signal/message pair with new task identifiers of the migrated tasks).

The key advantage of **CoCheck** and **ChaRM** is that no modifications to the *PVM* source code are needed. However, implementing such a sensitive operation like preserving communication across migration outside of the code that handles this communication is very difficult to do right and might be prone to suboptimal behaviour. For example, the flushing protocol used is rather suboptimal: all tasks must take part in it, even if some of them never exchanged any messages with the migrating task — in Dynamite, only tasks that have an established direct connection need to send flush messages. Also, any message send operations must be completed before a task can be migrated — Dynamite, in contrast, does not need to complete such operations. Finally, the correctness of the flushing protocol used by both CoCheck and ChaRM depends ultimately on the proper order of messages. While *PVM* does guarantee that in its documentation [1], the implementation actually violates this promise in case of direct communication and multicasts, as noticed in [8]: if a message is sent via `pvm_mcast`, which always uses indirect routing, and is quickly followed by another, point-to-point `pvm_send` call using direct routing, it can happen that the second message will arrive at the destination first. That is because *PVM* only takes care of proper message order across one communication channel, while in this case the two messages use two different channels. Fixing this essentially involves implementing message sequencing in *PVM*. A workaround would be to send two "ready" messages: one via point-to-point communication and one using multicasts.

The protocol implemented in **MPVM** is the closest to the one used in Dynamite. In both cases, after the migration the task has the same task identifier, so translation tables are not needed. The price paid is the need for routing tables, but these are only needed in the *PVM* daemons. Unlike in other solutions, the routing database in MPVM is distributed: only the daemon on the node that the task was originally started on has authoritative information about the current location of the task — other daemons can cache this information, but cache coherence is not guaranteed, as a lazy updating algorithm is used. Therefore, in MPVM the message forwarding and sequencing algorithm plays a major role in ensuring the correct delivery of messages, while with the current protocol of Dynamite, it is only used in case of race conditions stemming from network delays. Also, in MPVM the source node cannot be shut down, even if no more tasks are running on it. The migration protocol in MPVM is designed to be *fully asynchronous*, i.e. only the migrating task is notified, the state of other tasks is supposed not to matter. It is questionable whether this is really the case. MPVM cannot migrate a task while it is sending a message. As pointed out in Sections 4.4 and 4.6, `pvm_send` can block if using direct communication, hence the migration can be delayed for an arbitrarily long time. The problem is not so apparent in CoCheck and ChaRM, because they force other tasks to read data from the connection, which guarantees timely reaction. Dynamite does the same and, in addition, it does not need to wait until the whole message is sent. The protocol of flushing direct connections is also different in MPVM: it relies on *out-of-band* data and the `shutdown` system call. In Dynamite, a higher level protocol has been implemented, because these low-level socket features were found to be too limiting. For example, only one byte at a time can reliably be transferred using out-

of-band data, and the `shutdown` call not only refuses to send new data, it also refuses to retransmit the data packets already sent, should they be lost [20], which essentially makes the TCP connection unreliable. Another interesting feature of MPVM is that it does not protect critical sections with signal blocking / unblocking, but has a global flag that marks if the process is in critical section. The first thing that the signal handler does is to check the state of the flag, and return immediately if in critical section (the signal is raised again when leaving the critical section). While this solution is bound to have lower overhead, it raises some concerns: every system call in *PVM* can be interrupted as a result — does *PVM* handle all such cases gracefully?

8 Conclusions

The concept of implementing the checkpointer in the dynamic loader and using it to migrate *PVM* tasks has been proven to work in practice. Many thousands of migrations have been successfully performed for multimegabyte processes of real-world applications.

Dynamite puts extra overhead on the communication functions of *PVM*. Even for frequently communicating real-world applications this overhead was measured to be below 5%, however.

The architecture of Dynamite is modular. It is possible to use just the dynamic loader of Dynamite and get checkpoint/restart facilities for sequential jobs that do not use *PVM*. Even when using *PVM*, it is not required to use the Dynamite monitor/scheduler: the user can migrate tasks manually from the *PVM* console (using the new `move` command) or from custom programs (using the new `pvm_move` function call). This gives Dynamite extra flexibility, and makes its components reusable for different projects.

Dynamite aims to provide a complete integrated solution for dynamic load balancing of parallel jobs on networks of workstations. A number of challenges still need to be resolved to accomplish this, among them:

- support for MPI,
- generic support for the migration of the TCP/IP sockets,
- support for the latest versions of Solaris and GNU libc.

Acknowledgments

We acknowledge financial support by the European Commision through the ESPRIT initiative and by the Netherlands Organization for Scientific Research (NWO) through the MPR programme.

References

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, Massachusetts, 1994.
`http://www.epm.ornl.gov/pvm/`

[2] *MPI: A Message-Passing Interface Standard, Version 1.1*, Technical Report, University of Tennessee, Knoxville, TN, June 1995.
`http://www-unix.mcs.anl.gov/mpi/`

[3] W.D. Gropp, and E. Lusk, *User's Guide for mpich, a Portable Implementation of MPI*, Technical Report, ANL-96/6, Argonne National Laboratory, 1996.

[4] L. Dikken, F. van der Linden, J.J.J. Vesseur, and P.M.A. Sloot, *DynamicPVM: Dynamic Load Balancing on Parallel Systems* In Proceedings of High Performance Computing and Networking, in series Lecture Notes in Computer Science, v. 797, n. II, Networking and Tools, pp. 273–277, Springer-Verlag, 1994.

[5] J.J.J. Vesseur, R.N. Heederik, B.J. Overeinder, and P.M.A. Sloot, *Experiments in Dynamic Load Balancing for Parallel Cluster Computing*, In Proceedings of the Workshop on Parallel Programming and Computation (ZEUS'95) and the 4th Nordic Transputer Conference (NTUG'95), in series Transputer and Occam Engineering Series, Parallel Programming and Applications, pp. 189–194, IOS Press, 1995.

[6] B.J. Overeinder, P.M.A. Sloot, R.N. Heederik, and L.O. Hertzberger, *A Dynamic Load Balancing System for Parallel Cluster Computing*, Future Generation Computer Systems, v. 12, n. 1, pp. 101–115, 1996.

[7] G.D. van Albada, J. Clinckemaillie, A.H.L. Emmen, J. Gehring, O. Heinz, F. van der Linden, B.J. Overeinder, A. Reinefeld, and P.M.A.

Sloot, *Dynamite — blasting obstacles to parallel cluster computing*, In Proceedings of HPCN Europe '99, Amsterdam, The Netherlands, Lecture Notes in Computer Science, n. 1593, pp. 300–310, Springer-Verlag, 1999.

[8] J. Casas, D.L. Clark, R. Konuru, S.W. Otto, R.M. Prouty, and J. Walpole, *MPVM: A Migration Transparent Version of PVM*, Usenix Computing Systems, v. 8, n. 2, pp. 171–216, 1995.

[9] G. Stellner, and J. Pruyne, *Resource Management and Checkpointing for PVM*, In Proceedings of the 2nd European Users' Group Meeting, pp. 131–136, 1995.

[10] C.P. Tan, W.F. Wong, and C.K. Yuen, *tmPVM — Task Migratable PVM*, In Proceedings of the 2nd Merged Symposium IPPS/SPDP, pp. 196–202.5, 1999.

[11] P. Dan, W. Dongsheng, Z. Youhui, and S. Meiming, *Quasi-asynchronous Migration: A Novel Migration Protocol for PVM Tasks*, Operating Systems Review, v. 33, n. 2, ACM, pp. 5–14, 1999.

[12] P. Czarnul, and H. Krawczyk, *Dynamic Assignment with Process Migration in Distributed Environments*, In Proceedings of the 6th European PVM/MPI Users' Group Meeting, Barcelona, Spain, September 1999, in series Lecture Notes in Computer Science, n. 1697, pp. 509–516, Springer-Verlag, 1999.

[13] G. Stellner, *CoCheck: Checkpointing and Process Migration for MPI*, In Proceedings of the International Parallel Processing Symposium, pp. 526–531, Honolulu, HI, 1996.

[14] J. Robinson, S.H. Russ, B. Flachs, and B. Heckel, *A task migration implementation of the Message Passing Interface*, In Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing, pp. 61–68, 1996.

[15] `http://www.genias.de/products/codine/`

[16] J. Pruyne, and M. Livny, *Managing checkpoints for parallel programs*, In Proceedings of IPPS Second Workshop on Job Scheduling Strategies for Parallel Processing, 1996.

[17] J.S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent Checkpointing under Unix*, Proceedings of the Usenix Winter 1995 Technical Conference, New Orleans, LA, pp. 213–223, 1995.

[18] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, University of Wisconsin–Madison Computer Sciences Technical Report #1346, 1997.

[19] A. Barak, O. La'adan, and A. Shiloh, *Scalable Cluster Computing with MOSIX for LINUX*, In Proceedings of Linux Expo '99, pp. 95–100, Raleigh, N.C., 1999.

[20] S. Loosemore, R.M. Stallman, R. McGrath, A. Oram, and U. Drepper, *The GNU C Library Reference Manual*, Free Software Foundation, Inc, Boston, MA, USA, 1999.