Flexible group-based interprocess communication facilities for heterogeneous parallel architectures

Pedro D. Medeiros  
José C. Cunha
Universidade Nova de Lisboa – Departamento de Informática
2825 Monte da Caparica
Portugal

e-mail: pm,jcc@fct.unl.pt

Abstract

PDVM (Parallel and Distributed Virtual Machine) is a runtime environment targeted at the execution of parallel programs in a heterogeneous architecture composed of distributed-memory multiprocessors and uniprocessors connected by a local area network.

The most distinctive feature of PDVM is its support for group-based communication primitives. Several dimensions arise when defining a group-based communication facility. The type of process groups supported can range from groups with static composition and no guarantees of ordered delivery of successive broadcasts to dynamic process groups which guarantee an uniform and consistent view for all the members of the group, concerning communication and membership changes.

Of course, increased functionality is paid in more overhead. PDVM allows the programmer to specify the properties of the group in its creation, according to the needs of the application. On the other side PDVM uses techniques borrowed from Horus to achieve efficient implementations of richer semantics for groups.

1 Introduction

The work described here is integrated in a project that is running in the Departamento de Informática of Universidade Nova de Lisboa [8] [9] [7] having the main goal of developing a programming environment targeted at a heterogeneous set of machines, connected by a local area network, including distributed-memory multiprocessors.

This environment includes the support of several programming languages (namely a parallel and distributed Prolog and a group-oriented concurrent language based on C++) and programming tools for visualization, monitoring and debugging. It is built upon a software layer that offers a common programming interface, running over different kinds of hardware architectures and runtime environments. This layer, called PDVM, offers an unified model of concurrency, communication and synchronization, which virtualizes different types of hardware and system software and isolates upper software layers from such heterogeneity.

PDVM is intended to support a wide range of applications including artificial intelligence, robotics, fluid dynamics and system simulation. Many applications in these fields are characterized by the existence of a set of independent components that are created and
destroyed during the execution of the program and that communicate in “non regular” patterns.

This work was also inspired by our previous experimentation with systems like PVM [12] and Isis [2]. In fact PDVM tries to integrate in a single system several aspects which are only partially supported by the above systems:

- a simple, and yet flexible, interface for the management of the virtual machine environment as offered by PVM;
- a powerful set of group-oriented facilities as those supported by Isis;
- a more flexible internal architecture than the ones of PVM and Isis, targeted to heterogeneous networks with multicomputers, where special optimizations are possible in order to reduce the overheads involved in the implementation of group–oriented abstractions. This internal architecture is inspired in the Horus architecture [17].

The following sections discuss the basic functionalities of PDVM. For each component we present the motivations for its design and briefly describe the supported entities and their corresponding primitives.

2 Virtual machine management

PDVM integrates different ways of exploitation of parallel hardware, namely the ones that
use a homogeneous multiprocessor (sometimes called parallel systems) and those that exploit a network of workstations (usually called distributed systems)\(^1\).

The objectives of the integration are twofold:

- **type of problem**: different techniques may be suitable for the parallelization of different components of the problem, and these different techniques are, by nature, closer to different types of machines.

- **economic considerations**: in many problems a heterogeneous set of small multiprocessors and workstations can outperform a supercomputer: some times in absolute performance, almost always in the cost performance ratio [1].

According to these considerations PDVM manages a set of physical machines in order to offer an abstraction of a virtual machine, which is composed of several components. In this area PDVM is quite similar to systems like PVM [18] [12] or P4 [4]. Each component of the virtual machine is called *machine* and has one of the following types:

- **uniprocessor** - typically an UNIX workstation, which is characterized by its architecture and operating system.

- **multicomputer** - a distributed-memory multiprocessor composed of a homogeneous set of nodes and a host computer. PDVM needs to know the type, number and runtime environment of the nodes and the type and operating system of the host computer.

The initial set of the machines that compose PDVM is defined in a configuration file. PDVM is started from a console and launches a set of daemon processes in the uniprocessors and in the host computers of the multicomputers. Machines can be added and removed during the program execution.

An application is composed of a set of autonomous processes. A process owns the resources required by its execution (virtual processor, memory and communication devices). PDVM maps processes onto processors and allows the dynamic creation and destruction of processes. Process migration is not supported.

The following table summarizes the PDVM primitives related to machines and processes. In this and in other tables + stands for input argument and − stands for output argument; details of descriptors are omitted for simplicity.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Main primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine</td>
<td><code>pdvm_machine_add(+machine_description, −machine_id)</code></td>
</tr>
<tr>
<td></td>
<td><code>pdvm_machine_remove(+machine_id)</code></td>
</tr>
<tr>
<td></td>
<td><code>pdvm_machine_info(+machine_id, −machine_info)</code></td>
</tr>
<tr>
<td>Process</td>
<td><code>pdvm_process_create(+machine_id, +process_description, −process_id)</code></td>
</tr>
<tr>
<td></td>
<td><code>pdvm_process_delete(+process_id)</code></td>
</tr>
<tr>
<td></td>
<td><code>pdvm_process_info(+process_id, −process_info)</code></td>
</tr>
</tbody>
</table>

\(^1\)This very informal distinction is common in the parallel processing community.
3 Thread management

PDVM supports multiple threads of control within each process. This removes the well-known limitations of a programming model where the only device for concurrency specification is the process.

In fact, if there is no possibility of specifying parallelism inside a process, one must use non-blocking communication primitives in order to avoid having the (virtual) processor stay idle waiting for replies or acknowledgements. The use of non-blocking message-passing primitives leads to difficult to understand and error-prone programs[10]. On the other hand, the availability of threads helps in the internal housekeeping of the process (see section 5).

Thread management is implemented at user-level, in order to enhance the portability of PDVM. The thread scheduler supports two priority levels:

- **system threads** - high priority, without preemption.
- **user threads** - low priority, with round-robin scheduling and time slicing.

The programming interface is similar to Pthreads [15]:

<table>
<thead>
<tr>
<th>Main primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread management</td>
</tr>
<tr>
<td>pdvm_thread_create(+thread_function, +thread_args, -thread_descriptor)</td>
</tr>
<tr>
<td>pdvm_thread_exit()</td>
</tr>
<tr>
<td>pdvm_thread_block()</td>
</tr>
<tr>
<td>pdvm_thread_unblock(+thread_descriptor)</td>
</tr>
<tr>
<td>Mutexes and conditions</td>
</tr>
<tr>
<td>pdvm_mutex_init(-mutex_descriptor)</td>
</tr>
<tr>
<td>pdvm_mutex_acquire(+mutex_descriptor)</td>
</tr>
<tr>
<td>pdvm_mutex_release(+mutex_descriptor)</td>
</tr>
<tr>
<td>pdvm_condition_init(+condition_descriptor, +mutex_descriptor)</td>
</tr>
<tr>
<td>pdvm_condition_wait(+condition_descriptor, +mutex_descriptor)</td>
</tr>
<tr>
<td>pdvm_condition_signal(+condition_descriptor)</td>
</tr>
<tr>
<td>pdvm_condition_broadcast(+condition_descriptor)</td>
</tr>
<tr>
<td>Timers and alarms</td>
</tr>
<tr>
<td>pdvm_create_timer(-timer_descriptor)</td>
</tr>
<tr>
<td>pdvm_read_timer(+timer_descriptor, -time_value)</td>
</tr>
<tr>
<td>pdvm_create_alarm(-alarm_descriptor)</td>
</tr>
<tr>
<td>pdvm_start_alarm(+alarm_descriptor, -time_value)</td>
</tr>
</tbody>
</table>

The synchronization facilities are used in order to guarantee a thread-safe library regarding memory allocation and input-output.

4 Interprocess communication

In a parallel application, related processes can be encapsulated in process groups. Process groups are structuring units that may act as kinds of meta processes and that can be used for the support of parallel libraries. Several recent message-passing systems like PVM, MPI
[11] and CHIMP [6] support this concept in one form or another. See [14] for a more in-depth discussion of process grouping functionalities and of the limitations of approaches like the ones of PVM and MPI in this regard.

4.1 Messages, addresses and visibility

A message is composed in three steps “à la PVM”. A send buffer is initialized, and an arbitrary number of data elements of different types are inserted in the buffer. Finally the message is sent to its destination. No conversion of data is performed: the receiver converts it, if necessary.

The destination of a message is a pair \((\text{group}_id, \text{tag})\): \text{group}_id uniquely identifies a group in the PDVM environment and \text{tag} is an integer between 0 and some positive number which is implementation dependent.

When communicating with a group we must distinguish two cases:

- internal communication: in this case, there are no limitations. A process can send a message with any tag that will be received by all the processes of the group (except itself).

- external communication: tags are invisible to the outside of the group unless explicitly publicized by a member of the group. The members of the group don’t receive messages with non-public tags. If the tag is public all the group members receive the message. The objective of this facility is to define an interface of the group with its external environment and to assure that some tags are private to the group (this allows a group to support a parallel library without interference from the outside).

A special group identifier (PDVM\_WORLD\_GROUP\_ID) allows the addressing of all the processes, thus bypassing the structuring introduced by the process groups.

PDVM supports asynchronous and synchronous forms of message sending. In the latter case one can specify the number of required answers (between 1 and the number of group members).

4.2 Flexible properties of groups

A group is characterized by:

- an unique identifier;

- a non-empty set of members;

- a communication interface that is used by external processes to communicate with the group and that is defined by the set of public tags; if this set is empty, we say that this is a closed group.

- the functionality of the group model, defined by the dimensions below;

Several dimensions are present in the definition of a process grouping model [3][13]:

- static or dynamic process grouping- the composition of a group can be unchanged during the execution of the program (static grouping) or membership changes can be allowed (dynamic groups).
• uniform ordering of the message delivery - supposing static groups, is there a guarantee that multicasts of messages addressed to a group are not interleaved?

• consistent view of events by the members of the group - supposing dynamic groups, the members of the group must have an uniform view of the communication and membership events.

The most notable proposal of a group–oriented model is the one of the Isis distributed programming systems. In our research on the design of PDVM we were greatly influenced by the Isis model. However, unlike Isis, we focused our attention on a more modular definition of the group communication interface. We are also particularly interested in letting the programmer choose the group functionality according to the application requirements, only paying higher overheads when richer group semantics are really needed.

4.3 Group-oriented IPC primitives

The following table summarizes the IPC facilities of PDVM. These routines are thread-safe, in the sense described in [5].

<table>
<thead>
<tr>
<th>Group management</th>
<th>pdvm_group_create( +group_properties, −group_id )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pdvm_destroy( +group_id)</td>
</tr>
<tr>
<td></td>
<td>pdvm_group_enter( +group_id)</td>
</tr>
<tr>
<td></td>
<td>pdvm_group_leave( +group_id)</td>
</tr>
<tr>
<td></td>
<td>pdvm_group_publish( +tag, +group_id)</td>
</tr>
<tr>
<td>发送/接收</td>
<td>pdvm_nbsend( +group_id, +tag, +buffer )</td>
</tr>
<tr>
<td></td>
<td>pdvm_sendrecv( +group_id, +tag, +buffer, +number_of_replys, −reply, ...)</td>
</tr>
<tr>
<td></td>
<td>pdvm_recv( +tag, −buffer, −sender )</td>
</tr>
</tbody>
</table>

5 Implementation of the flexible process groups

We are basing our implementation of the process groups in ideas from Horus [16] [17]. According to its creators Horus can be viewed ([16]) “... a box of LEGO blocks. Although each type of block implements a different communication feature, the blocks have standardized top and bottom interfaces that allows them to be stacked on top of each other in runtime ...”.

Below we give an outline of our approach for implementing flexible process groups. This is being implemented on a network with several SUN stations, two different Transputer-based distributed-memory multiprocessors and a cluster of DEC Alpha stations.

Our implementation, currently under development, has the following “LEGO blocks”:

• COM - basic communication: this layer supports the native communication mechanism offering a common communication interface;

• FIFO - this layer assures reliable FIFO multicast;

• MEMBERSHIP - implements the group membership management service;
- **CAUSAL** - message broadcasts preserve causality in the delivery of messages.
- **TOTAL** - message broadcasts preserve a global and uniform ordering in the delivery of messages to all the members;

Each “LEGO box” is supported by a thread residing in the same address space of each process that joined the group. These threads are mapped onto the system threads of PDVM. So, the entrance of a process into a group corresponds to the creation of the corresponding threads inside the process. Each thread is responsible for taking care of the part of the headers of the message corresponding to its role. As all these threads do share memory the associated overheads are kept small.

![Diagram of two groups with different functionalities](image)

Figure 2: Two groups with different functionalities

## 6 Conclusions

We have claimed that the availability of process grouping facilities is very useful in the development of certain classes of parallel applications and we have introduced flexible group-based interprocess communication facilities. This flexibility has three dimensions:

- **group communication interface level**: the communication interface offers encapsulation facilities by only making visible to the outside tags which were explicitly publicized. On the other hand, this interface can be modified during the execution of the program.

- **specification of group semantics**: the user can define the semantics of a group.

- **flexibility in the internal architecture of PDVM**: the Horus layered approach can be useful when supporting process groups over the native interprocess communication facilities of multicomputers. According to the characteristics of these native message-passing systems (regarding ordering and multicast facilities) some layers can be simplified or even eliminated, thus reducing the overheads.
What remains to be proved is if the performance degradation associated with the enhanced facilities of process grouping is acceptable. By letting the user to decide which facilities to include in the virtual machine environment we are contributing to reduce the importance of the overheads, but a real answer to the question can only be found by experimenting with PDVM, at the following levels:

- directly using the interface primitives from the C and PROLOG languages, for programming parallel and distributed applications in the domains mentioned in section 1; our short term attention will be directed to the domain of distributed diagnosis.

- use of the PDVM interface to support the implementation of shared data space abstractions within process groups. Those abstractions are being developed for the mentioned concurrent extensions to C++ and PROLOG, as a form of supporting the structuring of an application and a higher level form of process communication using shared structures which are encapsulated within process groups. [9].

- use of PDVM group–oriented facilities for the implementation of parallel programming tools such as distributed monitoring for parallel machines, visualization tools and a parallel debugger. Some of this work is done in the context of a related project [19][8] [7]. This work exploits the semantics of group communication to ease the management of event orderings in such tools. It also uses the group abstractions to handle the heterogeneity of the underlying hardware environment.

Acknowledgments

This work was partly supported by the CIENCIA and PRAXIS XXI (project PROLOPPE) Portuguese Research Programmes, and DEC EERP PADIPRO project.

References


