PVM-Prolog: Parallel Logic Programming in the PVM System

Rui F. P. Marques, José C. Cunha

(rfm|jcc)@fct.unl.pt

May 30, 1995

Contents

1 Introduction ........................................ 1

2 System Overview .................................. 2
   2.1 Programming Model ............................. 2
   2.2 System Architecture and Implementation .... 2

3 The thread concurrency model ..................... 3
   3.1 Basic Concepts ................................ 3
   3.2 Thread Control ................................ 4
   3.3 Thread Communication .......................... 5
   3.4 Thread Scheduling Policy ...................... 6

4 The PVM Interface Layer .......................... 7
   4.1 Accessing a parallel virtual machine from Prolog . 7
   4.2 Summary of PVM functionalities ............... 7
   4.3 Definition of the Prolog interface predicates .... 7

5 Conclusions and future work ..................... 11

1 Introduction

In this document we propose a parallel programming model which encompasses the functionalities of a multi-threaded Prolog and the ones of a parallel programming interface from Prolog to the PVM [6, 7] system.

While the multi-threaded model was designed anew, the Prolog/PVM interface basically corresponds to the inclusion of additional system predicates which give full access to the PVM environment, from a Prolog program.
This document briefly introduces the basic concepts of the proposed model, and then describes in detail all the supported extensions and additional system predicates. For a detailed presentation of the rationale behind this research see [4].

2 System Overview

Our ultimate goal is to achieve an unified parallel logic programming system, supporting multiple levels of parallelism and concurrency.

The prototype offers a concurrent programming model for the Prolog language. Its implementation is supported by several layers, from the low level system and hardware levels to the higher level, a programming language interface.

2.1 Programming Model

Our programming model comprises a virtual machine environment supporting multiple levels of concurrency:

**Threads** Threads allow concurrency internal to a process.

**Processes or Tasks** Tasks define capsules or contexts for the execution of multiple threads, and correspond to a system protected or self contained environment. They are usually equivalent to UNIX processes, however they may differ from those in other environments, e.g. in a multicomputer.

**Host** Separate hosts correspond to separate physical machines interconnected through a communication network. The virtual machine environment allows a programmer to actually “see” the hosts, but it is also possible to ignore such a view and deal with the virtual machine as a set of processes.

By having access to multiple levels of concurrency, the programmer may better express different grains of parallelism, e.g. for tightly coupled problems one may exploit concurrency at the thread level, while for more loosely coupled problems a courser grain (such as the task level) may be more suitable.

The programmer has access to the functionalities provided by the model through deterministic interface predicates.

2.2 System Architecture and Implementation

The virtual machine is built by layers, relying upon existing systems, such as the PVM System [6, 7] and the WAM [8, 1], which in turn are built upon lower level systems and hardware.

This layered approach has many known advantages, which we feel outbalance some performance loss.
Particularly it enhances portability and greatly simplifies changes at each layer and the programmer’s job in general. This is specially important in parallel programming systems where there is a large diversity of multiprocessor architectures still in evolution.

PVM is used to support parallelism because of its popularity and its adequacy to deal with networks of UNIX workstations, which currently provide the most disseminated form of parallel hardware. PVM is most popular over TCP/IP protocols, but there are also implementations over other, more specialized communication protocols.

The WAM model is used as the core inference engine for Prolog. We have extended the WAM in order to support multiple threads of control, by managing multiple execution contexts on a single Prolog machine.

The WAM we have been using is NanoProlog, developed at our Department by Artur Miguel Dias [5]. It presents minimal Prolog functionalities but gives very good performance and has been designed to be highly modifiable.

## 3 The thread concurrency model

### 3.1 Basic Concepts

#### 3.1.1 Motivation

Threads are being used for some time in systems programming. In distributed systems, concurrent server processes may more easily be programmed using threads, which avoid the complex management of state machines embedded in a single process.

However, threads are also useful in other cases, where there is a need for internal concurrency to an agent or process. In a multi-agent system, each agent

![Virtual Machine Model](image)

**Figure 1: Virtual Machine Model**

<table>
<thead>
<tr>
<th>Virtual Machine</th>
<th>PVM Task 0</th>
<th>PVM Task 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 0</td>
<td>Thread 0</td>
</tr>
<tr>
<td>Thread 1</td>
<td>Thread 1</td>
<td>Thread 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Thread n</td>
<td>Thread n</td>
<td>Other PVM tasks...</td>
</tr>
</tbody>
</table>
may be required to react to some event when it is executing some other computation. Concerning performance, threads enable the programmer to exploit concurrency with smaller overhead than conventional, heavy-weighted, processes. Threads can also be used to model problems that exhibit natural concurrency features, such as pipelined or consumer/producer problems.

### 3.1.2 Threads

In our model a Prolog process is a place for the concurrent execution of a set of threads. Each thread solves its own goal, although all threads share the same logic program. Within a process each thread has an unique Thread Identifier (abbreviated to Thread Id.)

Threads execute one at a time; the thread currently executing is said to be in the running state. Other threads may be ready to execute, or blocked, waiting for some event to take place. Threads have associated priorities, so that more important threads may execute before less important ones.

Communication within a process is based on *Term Queues*. These are queues where threads may put and get terms, in FIFO order. Term queues are local to a process, having an unique name, and may be accessed by all the process’ threads.

Term queues also provide for thread synchronization as the get primitive is synchronous (blocking.)

### 3.2 Thread Control

Threads can be dynamically created and terminated. When a process starts, an initial thread with identifier 0 is created, which will usually, in an interactive
process, run the Prolog interpreter. A thread with the lowest priority, the watch
dog is also created at the start of the process and is used to keep the scheduler
running when all other threads are blocked 1.

t_fork(+Goal, +Stacks, +Priority, -ThID)
t_kill(+ThID)
t_mytid(-ThID)

The t_fork predicate creates a new thread. The parent thread must specify
the top goal which the new thread will solve, and a Thread Id (ThID) is returned,
which uniquely identifies the new thread within the process. The programmer
may optionally specify the execution stacks size, and the priority of the new
thread. The new thread is put into the ready state, and the invoking thread
proceeds with its execution (see section 3.4 concerning thread scheduling).

The predicate t_kill allows a thread to terminate the execution of another
(or even itself) identified by its Thread Id.

Note that there is no variable sharing among individually forked threads
from a given clause. The t_fork predicate is always successful; any failure in
the goal evaluation thread must be explicitly handled by the program.

The predicate t_mytid enables a thread to determine its own Thread Id.

3.3 Thread Communication

The Prolog clause database in a process is shared by all its threads. Thus, they
may communicate by Prolog assertion/retraction of terms, which are guaranteed
to be indivisible actions, but do not support any form of thread synchronization.

Another mechanism for communication, also providing for synchronization,
is called “Term Queues”: these are FIFO queues where a thread can put terms,
to be retrieved later by another.

The following predicates are available for Term Queues access:

queue_put(+Queue, +Term)
queue_[n]get[p]+Queue, -Term
queue_[n]select[p]+ListQueues, -Term
queue_empty(+Queue)

queue_put puts a Term into Queue. This predicate is non-blocking, i.e., the
thread “puts” the term into the queue and proceeds with its execution 2.

queue_get gets a term from Queue, instantiating Term to it. If the queue is
not empty the term is consumed from it, and the current thread doesn’t block.
However, if the queue is empty, there are several possibilities for the reader
thread:

1 In predicate definitions we use the convention of putting a + for input arguments, - for
output and ? for input/output ones. Square brackets ([]) denote optional parts on predicate
names, compacting the definition of similar predicates.

2 Unless some higher priority thread has become ready in the meanwhile, see section 3.4
Non Blocking The predicate fails. This is the case of queue\_get.

Normal The reader thread is put into a FIFO queue in the blocked state, waiting for a term to be put into the queue. This is the case of queue\_get.

Priority The reader thread is put into a priority queue, waiting for a term to arrive. These threads have higher priority to get at the terms than the threads in normal mode. If several threads are in priority mode, the one selected to get the term is the one with higher priority. This the case of queue\_get.

A thread can test if a queue is empty (it does not have any pending terms) by calling the predicate queue\_empty.

A thread may also non-deterministically select a term from several term queues, using queue\_select: the behavior of its variants when all the queues are empty is similar to the ones of queue\_get.

Note that there is no variable sharing between threads: once a term is passed onto another thread, all its uninstantiated variables are “freshened” (replaced by newly created ones.) No backtracking coordination is implied concerning thread communication through term queues.

3.4 Thread Scheduling Policy

Threads with the same priority are scheduled round robin.

The currently running thread may explicitly yield the control to another ready thread by calling the primitive t\_switch, or else run until either it blocks, or until it invokes a control predicate and some higher priority thread becomes ready. In a given moment each thread can be in one of the following states:

Running The thread is executing, i.e. it “owns” the control, within the enclosing process.

Ready The thread is ready to execute and will become running as soon as possible, given the scheduling constraints.

Blocked The thread is awaiting some event, and will become ready as soon as that event occurs. Blocked threads are usually waiting in a term queue or awaiting for I/O.

Priority levels are numeric, and range from 0 to 15, 0 being the highest priority and 15 the lowest. Each priority level has an associated FIFO queue for scheduling purposes, and higher priority threads always preempt lower priority ones (at the above mentioned preemption points.) So a thread with a lower priority only runs when there are no higher priority ready threads.

The predicate t\_priority( +ThID, ?priority ) allows to examine/change the priority of a thread. Note that changing thread priorities may cause a sudden shift in the control flow.
4 The PVM Interface Layer

4.1 Accessing a parallel virtual machine from Prolog

This layer provides an interface between Prolog and the PVM system [2]. It provides a simple, yet powerful tool to parallel logic programming over common and relatively inexpensive hardware.

For past experiences and a discussion of more general issues around this layer see [4].

4.2 Summary of PVM functionalities

PVM (Parallel Virtual Machine) is a parallel and distributed programming environment which provides dynamic process management, communication and synchronization over heterogeneous computer networks.

Different degrees of visibility are provided to the programmer regarding process (task, according to PVM terminology) creation: to choose the actual physical machine where a task is spawned, to specify only the architecture type where the task is to be run or to let PVM do all the choices.

Communication is asynchronous and buffered, from the point of view of the sender. Point-to-point communication primitives specify the partner task identifier (i.e. sender or destination) and a user-defined message tag. The receiver may choose to block waiting for a message or may use a nonblocking receive primitive. It may also use wild-cards or specify the sending task-id and message-tag.

Process groups are also supported, although there is no guarantee of a global or causal ordering for group events (such as ISIS virtual synchrony [3].) Message ordering is only preserved in point to point communication, between a pair of processes.

4.3 Definition of the Prolog interface predicates

A simplified operational definition of the major interface predicates is given in the following. All of these predicates exhibit a strict deterministic behavior and, as such, they are not backtrackable.

4.3.1 Task identification, entry and exit to PVM

\begin{verbatim}
pvm_mytid( -tid )
pvm_exit
\end{verbatim}

The predicate \texttt{pvm\_mytid} allows a PVM task to determine its own unique PVM \textit{Task Identifier}. If the process is not already a PVM task, it becomes so. \texttt{pvm\_exit} allows a process to detach from the PVM virtual machine.
4.3.2 Task creation and destruction

\texttt{pvm\_spawn( +programe, +goal )}
\texttt{pvm\_spawn( +programe, +goal, +opt\_list, +where, +n\_tasks, -tid\_list,}
\hspace{1cm} +StacksSize, +HeapSize )
\texttt{pvm\_kill( +tid )}

\texttt{pvm\_spawn} allows for the dynamic creation of new PVM tasks.
In general, \texttt{n\_tasks} (possibly one) are created to solve the given \texttt{goal} in the
presence of the specified program. \texttt{programe} is the name of the file containing
the Prolog program. The directory where Prolog program files are to be found
is indicated by the environment variable \texttt{PVM\_NANOHOME}.

The size and configuration of the process address space may be specified
through the \texttt{StacksSize} and \texttt{HeapSize} (in Kilobyte units) parameters whose
names hint their corresponding WAM areas.

Total location transparency is obtained in the first form of this predicate.
If so desired, an indication may be given regarding a specific machine for the
created process(es), or an architecture type depending on the given options in
\texttt{opt\_list} (‘host’ or ‘arch’ specifications.) See [6] for details.

Other options include triggering commands for debug and trace in the created
process.

The task identifiers of the created processes may be obtained in \texttt{tid\_list}.
\texttt{pvm\_exit} terminates another PVM task.

[6] recommends that a task should not use this primitive to commit suicide.
\texttt{pvm\_exit} should be used for a task to exit pvm, so that proper cleanup actions
are taken before task completion.

4.3.3 Communication

\texttt{pvm\_send( +tid, +msg\_tag, +term )}
\texttt{pvm\_mcast( +tid\_list, +msg\_tag, +term )}
\texttt{pvm\_[n]recv( +tid, +msg\_tag, -msg )}

\texttt{pvm\_send} allows for sending messages to another task, while \texttt{pvm\_mcast} al-

\texttt{pvm\_recv} is the predicate for a blocking receive, while \texttt{pvm\_[n]recv} allows a
non-blocking receive and fails if there's no queued message.

The message tag \texttt{(msg\_tag} which is used while receiving) may be a positive
integer, where only messages with that tag will be accepted or -1 which accepts
any message. The \texttt{tid} which is specified in the receiving predicates may also be
-1, specifying that messages are to be accepted from any sender.
4.3.4 Visibility and control of the underlying PVM configuration

If one so desires, some control may be exercised and status information may be obtained concerning the actual configuration of the PVM execution environment, e.g. host machines and existing tasks.

\[ \text{pvm\_mstat}( \text{+host, -mstat} ) \]
\[ \text{pvm\_config}( \text{-nhost, -narch, -hostlist} ) \]
\[ \text{pvm\_tasks}( \text{+which, -ntasks, -tasklist} ) \]

\text{pvm\_mstat} provides information about real machine (host) status. \text{mstat} may be instantiated to:

\text{ok} in case host is up

\text{fail} in case host is down or connection is lost

\text{noHost} in case host is not recognized as such.

\text{pvm\_config} reports on the global configuration of the PVM virtual machine:

\text{nhost} gives the total number of hosts.

\text{narch} gives the total number of distinct architecture types.

\text{hostlist} consist of a list of terms, one for each host, with some information on its particular characteristics:

\[ \text{host}( \text{pvmd\_tid, host\_name, arch\_name, cpu\_speed} ) \]

\text{pvm\_tasks} requests PVM task status information.

The request for information is specified through the \text{which} numeric parameter, which can assume the values:

\text{0} : show all tasks in the PVM environment

\text{pvmd\_tid} : show all the tasks in given pvmd’s host

\text{task\_id} : just show the task as identified by tid

\text{ntasks} returns the number of tasks, while \text{tasklist} returns a list of terms in the form:

\[ \text{task}( \text{tid, parent\_tid, pvmd\_tid, flags} ) \]

PVM also provides support for dynamic configuration of the virtual machine, allowing for assertion and retraction of hosts:

\[ \text{pvm\_addhosts}( \text{+hostlist, -infolist} ) \]
\[ \text{pvm\_delhosts}( \text{+hostlist, -infolist} ) \]
hostlist gives a list of hosts to be included or deleted from the PVM environment and infolist is a list returning the success codes of this operation, for each machine.

PVM also allows for other settings to be examined/changed by the user task:

\[
pvm\_setopt( +\text{option}, -\text{val} )
pvm\_getopt( +\text{option}, +\text{val} )
\]

Most used values for \text{option} are:

- \text{route} routing strategy. Allows the programmer to suggest message rout-
ing policies to PVM. It may assume the following values:
  - \text{routedirect} Requests PVM to set up direct task-to-task links
    (using TCP) for all subsequent communication.
  - \text{dronroute} All communications are done through the pvm daemon.
  - \text{allowedirect} Lets PVM choose to routing policy.

- \text{debugMask} Sets the PVM debug mask so that specific PVM actions are
  traced.
- \text{autoErr} When \text{arg} is 1 enables automatic error messages, when it’s 0
  disables it.

4.3.5 Event notification

\[
pvm\_notify( +\text{about}, +\text{msgtag}, +\text{tidlist} )
\]

A message with the given tag (\text{msgtag}) is broadcasted to every task in
\text{tidlist} when a particular event occurs, according to the \text{about} argument:

- \text{taskExit} Task exits or is killed
- \text{hostDelete} Host is deleted or crashes
- \text{hostAdd} New host is added

4.3.6 Task Groups

PVM supports grouping mechanisms, although they do not enforce global or
causal orderings of events.

\[
pvm\_joingroup( +\text{gname} )
pvm\_lvgroup( +\text{gname} )
pvm\_gettgid( +\text{gname}, +\text{instnum}, -\text{tid} )
pvm\_getinst( +\text{gname}, +\text{tid}, -\text{instnum} )
pvm\_gsize( +\text{gname}, -\text{nelems} )
pvm\_barrier( +\text{gname}, +\text{count} )
pvm\_bcast( +\text{gname}, +\text{msgtag}, +\text{term} )
\]
pvm_joingroup allows a task to become a member of group name. pvm_listgroup is used for detaching from the group.

pvm_gettid and pvm_gettid allow to determine a task id, given its group name and instance number, and vice-versa.

pvm_size determines the number of elements in a group.

pvm_barrier supports barrier synchronization for count processes within the given group.

pvm_bcast allows to broadcast a message to all members of a group.

5 Conclusions and future work

We have described the concepts and primitives of a programming interface which supports several forms of explicit parallelism and distribution of programs in a PVM environment.

The interface consists of two basic separate components: the multi-threaded process model and the Prolog/PVM interface. The former corresponds to a self-contained extension to the WAM, with its own built-in thread scheduler, and posing no requirements to the underlying operating system environment. The later provides all of the existing PVM functionalities to a Prolog programmer, in the form of system predicates which offer great flexibility for the explicit user control of a distributed/parallel multi-processed Prolog application.

The integration of the above mentioned components into a single unified model is our current goal. A layer that combines multi-threading and PVM capabilities is being developed.

The current prototype is also expected to evolve towards improved performance. Capabilities for performance evaluation and debugging support are also to be incorporated.

Acknowledgments

Thanks to all the members of the Parallel and Distributed Processing Group of our Department. Special thanks to Artur Miguel Dias, designer of NanoProlog, for his support to this work.

This research was supported in part by Programa Ciência, project PRO- LOPPE contract no. PRAXIS/3/3.1/TIT/24/94 and DEC EERP PADIPRO, contract no. P-005.

References


