Distributed Algorithm Development with PVM-Prolog
José C. Cunha, Rui F. P. Marques
Universidade Nova de Lisboa
Departamento de Informática
2825 Monte de Caparica
Portugal
(jcc|rfm)@di.fct.unl.pt
March 27, 2001

Abstract
The design of parallel and distributed algorithms poses many difficulties. This has motivated proposals of high-level languages for parallel program composition, their physical distribution in multiprocessor architectures, specification of communication and synchronization, and failure handling. In this paper, we describe an approach for prototyping parallel and distributed programs, that is based on a logic programming system called PVM-Prolog. PVM-Prolog is a programming interface from Prolog to the PVM system, offering all PVM functionalities to the logic programmer such as process spawning and control, virtual machine management, and failure handling. We describe the PVM-Prolog model and illustrate its application.

Keywords: Distributed algorithms, logic programming, PVM.

1 Introduction
Explicit specification of parallelism and distribution at the user level is required by many applications. In order to overcome the difficulties in designing parallel and distributed processing systems, there is a demand for advanced programming environments providing an integrated view of the specification and execution stages, including debugging, performance evaluation and program visualization [5]. Therefore it is important to use very high-level notations for parallel program decomposition and communication which could also be integrated with low-level run-time tools. A compromise must be achieved between the required user transparency degrees, and the need of providing control of low-level execution, e.g. for performance tuning.
Among the high-level notations, logic programming models offer many interesting characteristics, namely clarity and ease of programming, and facility for developing rapid prototypes. On the other hand, classical logic languages such as Prolog, pose several difficulties concerning efficient implementations and support for large scale program development. The past decade has shown many relevant proposals to use logic languages for parallel and distributed programming, in search for an adequate model to express interactions among multiple autonomous components of an application. However, high-level logic concurrency definitions are difficult to conciliate with the requirements for efficient implementations and low-level control of the execution environment.
As a solution to these problems, we have proposed PVM-Prolog [1] [2] [3], a Prolog interface to the PVM environment [4]. PVM-Prolog allows programming using the Prolog style, and some degree of control of the system level abstractions. As it is an intermediate layer, it has simple concurrency semantics which are in fact inherited from the ones of PVM, suitably adapted to a logic framework. On the other hand this layer provides the basic platform upon which one can implement more high-level language models, or applications. Also for several Artificial Intelligence applications there is the requirement to model a system through a collection of cooperating agents. Current research in this area is studying the reasoning and behavioral models for such agents, and modified Prolog interpreters can be used for this purpose. The interactions between agents can be tested under real distributed settings by using PVM-Prolog as the implementation layer to support inter-agent communication, agent distribution, as well as concurrency within each agent. We must stress the fact that PVM-Prolog is a low-level programming layer that can typically be used to build the above mentioned high-level abstractions. By expressing parallelism, distribution, communication schemes, and virtual machine
management in PVM-Prolog, the user is freed from certain low-level details, and can concentrate mainly on the logical organization of the applications. In a first stage of application development it is easy to obtain a rapid prototype that can be used to test and debug logical correctness, and to experiment with alternative configurations of the parallel PVM virtual machine towards better performance. These requirements can also partially be met by other low-level distributed Prolog systems such as SICStus Prolog and IC-Prolog II which support TCP/IP socket based communication [11, 12]. However, the PVM-Prolog system is more easy to install and to use, and it is easily portable to a large number of hardware platforms.

At this point we have developed a fully operational PVM-Prolog implementation\(^1\), and we are currently developing tools such as a high-level distributed debugger, and a monitoring system. This paper illustrates the PVM-Prolog programming model through a few simple examples involving both parallel and distributed processing issues. In section 2 we describe the fundamental concepts of the PVM-Prolog model. In section 3 we give some simple examples. Finally we conclude by briefly discussing ongoing work.

### 2 The PVM-Prolog Programming Model

The PVM-Prolog system consists of two distinct components: the virtual machine (PPVM) and the process engine (PE). The process engine is the building block that represents the computing entities in a particular PVM-Prolog application. It supports a specific core inference engine for Prolog, or for a different language, as the PPVM interface allows the control and communication in programs composed of heterogeneous multilingual components. The virtual machine (PPVM - Parallel Prolog Virtual Machine) provides primitives for the activation and control of process engines, interprocess communication and synchronization, and management of the execution environment.

#### 2.1 The PPVM interface

All of the PVM functionalities are accessible at the Prolog level, through a raw interface to PVM, called PPVM0 that is implemented as a set of Prolog built-in predicates. Messages are interpreted as Prolog terms, and extensions to PVM pack and unpack functions are provided in order to convert term representations in hybrid heterogeneous applications. C data types such as character strings, arrays, and structures are converted, respectively, to Prolog atoms, lists, and compound terms, and vice-versa. All of the interface predicates exhibit a strict deterministic behavior, i.e. they all fail on backtracking.

On top of PPVM0, an upper layer PVM1 offers a higher-level semantics for process control and communication as described in [1] [2] [3].

##### 2.1.1 Task identification, entry and exit to PVM-Prolog

PVM-Prolog processes correspond to PVM tasks.\(^2\)

The `pvm_gtid(-tid)` predicate allows a PVM task to determine its own unique PVM Task Identifier, and if the process is not already a PVM task, it becomes so. The `pvm_exit` predicate detaches a process from the PVM environment.

##### 2.1.2 Task creation and destruction

`pvm_spawn` creates new PVM-Prolog tasks.\(^3\) Here we assume an homogeneous PVM-Prolog system with the Prolog engine defined by default\(^4\).

\begin{verbatim}
pvm_spawn( +programe, +goal, +opt_list, +where, +ntasks, -tid_list, +StacksSize, +HeapSize )
\end{verbatim}

\(^1\)The system is available by contacting the authors.

\(^2\)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.

\(^3\)The raw interface provides a more basic form for this predicate, supporting the spawning of different types of process engines.

\(^4\)Our current prototype relies upon the WAM model as the process engine, based upon a system called NanoProlog, developed at our Department by A. M. Dias [7].
tasks are created to solve the given goal in the presence of the specified program. proglme is the name of the file containing the Prolog program. The task identifiers of these processes are returned in tid_list. A PVM task is spawned for executing an instance of the NanoProlog engine which will consult the specified Prolog file. Then the specified goal is activated. The new process is completely detached from its parent, and all its interactions with other processes must be explicitly programmed. All known PVM options are available in opt_list concerning the specification of a specific machine, or architecture type for running the newly created process.

A task can invoke pvm_kill to terminate another task. pvm_exit allows a task to exit from PVM.

2.1.3 Communication

AT PPVM1 level, a reasonable transparency is preserved concerning communication management. We have hidden all PVM buffer manipulation primitives [4] because all Prolog representations are based on terms. However, at the PPVM0 level, one must use PPVM0 predicates for packing and unpacking terms so that conversions between Prolog and C can be performed under program control. pvm_send( +tid, +msgtag, +term ) sends messages to another task, while pvm_mcast( +tid_list, +msgtag, +term ) multicasts messages to several recipients given by tid_list. pvm_recv( ?tid, ?msgtag, -msg ) performs a blocking receive, while pvm_nrecv performs a non-blocking receive and fails if there’s no pending message at invocation time. The message tag (msgtag) may be a positive integer, such that only messages with that tag will be accepted, or it can be an uninstantiated variable if we want to accept any incoming message. The tid may also be an uninstantiated variable, if messages are to be accepted from any sender.

2.1.4 Management of the PVM configuration

Some control and status information may be obtained concerning the configuration of the PVM host machines and tasks by invoking the following predicates: pvm_mstat( +host, -mstat ) provides information about host status by returning mstat instantiated to the host status (ok, down, unknown). pvm_config( -nhost, -narch, -hostlist ) reports on the global configuration of the PVM virtual machine, namely, number of hosts, number of distinct architecture types, and a list of terms, one for each host, with information on its characteristics. pvm_tasks( +which, -ntasks, -tasklist ) requests PVM task status information through the which numeric parameter: all tasks in the PVM environment; all the tasks in the given host; or just the task identified by tid. Dynamic configuration of the virtual machine is possible through insertion and removal of hosts: pvm_addhosts( +hostlist, -infolist ) gives a list of hosts to be included into the environment and infolist is a list returning the success codes of this operation, for each machine. pvm_delhosts( +hostlist, -infolist ) deletes the given hosts from the PVM environment.

2.1.5 Task Groups

PVM-Prolog also gives access to PVM grouping mechanisms by providing predicates for joining and leaving groups, as well as communication and synchronization operations [1] [4].

2.2 A Multithreaded Prolog Engine

It is possible to run a PVM-Prolog application consisting of multiple distinct Prolog engines, i.e. single or multi-threaded, or even supporting internal forms of implicit parallelism. They all use the same PPVM interface for communication and process control.

We have extended a core inference engine for sequential Prolog in order to support multiple threads of control, by managing multiple goal execution contexts. Threads are useful when there is a need for internal concurrency to an agent. In a multi-agent system, each agent may be required to react to some event when it is executing some other computation. Threads also allow to exploit concurrency with smaller overhead than conventional, heavy-weighted, processes.
2.2.1 Threads

A Prolog process is a place for the concurrent execution of multiple threads, each solving its own goal, and sharing the same logic program (set of Prolog clauses). Each thread has a unique Thread Identifier. Within each process threads execute one at a time. When a process starts, an initial thread with identifier 0 is created, which will usually, in an interactive process, run the Prolog interpreter. Threads are dynamically created and terminated as follows.

t_create( +Goal, +Stacks, +Priority, -ThID) creates a new thread, by specifying the goal for the new thread to solve, and a thread identifier is returned in ThID. The new thread is put into the ready state, and the invoking thread proceeds with its execution. t_kill(+ThID) allows a thread to terminate the execution of another (or even itself!). There is no variable sharing among individually created threads from a given clause. Furthermore, t_create is always successful; any failure in the child thread must be explicitly handled by the program. t_currentid(-ThID) enables a thread to determine its own identifier.

Threads with the same priority are scheduled in a round robin fashion. The currently running thread may yield the control to another ready thread by calling t_yield, or else run until either it blocks, or until it invokes a control predicate and some higher priority thread becomes ready, or until it its timeslice period expires. Each priority has an associated FIFO scheduling queue, and higher priority threads always preempt lower priority ones (at the mentioned preemption points).

2.2.2 Thread Communication

Thread communication and synchronization within a process is based on Term Queues, where threads may put and get terms, in FIFO order, by using the following predicates.

queue_put( +Queue, +Term ) puts a Term into Queue in a non-blocking way. queue_get( +Queue, -Term ) consumes a term from Queue, instantiating Term to it. If the queue is empty, the thread blocks. If queue_get (non-blocking) is used, the predicate fails when the queue is empty, otherwise it behaves like queue_get. A thread can also test if a queue is empty without blocking, by calling queue_empty or can non-deterministically select a term in a list of term queues by invoking the predicate queue[n]select (see [1]).

As 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 automatic backtracking coordination is implied concerning thread communication through term queues.

2.2.3 Binding Messages and queues

Signals and messages can be explicitly received by a PVM-Prolog program, through a mechanism for automatic handling of signal and messages. A message sent to a PVM-Prolog process may be automatically saved in a term queue, with msg_bind(+FromTID, +Tag, +Queue, -BID). The given queue is bound to the event corresponding to the arrival of any message coming from the specified process with the given tag. In case these two arguments are uninstantiated, then any messages can be caught by this mechanism. Messages are automatically put into the queue as they arrive, and they are simply consumed by invoking queue_get.

3 Examples

In this section we present a few simple examples which illustrate the PVM-Prolog model.

3.1 A multithreaded remote query evaluator

We describe how to spawn remote “goal servers.” Each goal server corresponds to a rule database (Prolog program). It executes remote queries and delivers answers to a client program. For simplicity, only a single client is assumed, but it is easy to change the program to handle multiple clients.

The server is created with the command:

start_server( ProgName, TID ) :-
  pvm_spawn( goal_server, server(ProgName), □, □, 1, [TID] ).

ProgName is the file containing the database. The server’s TID (task id) is later used by a client to start new goals:
start_goal( TID, Goal, Tag, Number ) :- pvm_send( TID, 1, Goal/Tag/Number ).

The term that is sent in this message carries information for the server process: Goal, the goal that must be evaluated, Number the number of solutions to be obtained, possible values being one or all, and Tag, the message tag that is used later to get the answers from the server:

get_answer( TID, Tag, Answer ) :- pvm_recv( TID, Tag, Answer ).

The server process waits for the arrival of new goals, and executes them, using t_create to ensure that multiple queries are evaluated concurrently.

server(ProgName) :- [ProgName], msg_bind(AnyTask,1,goalqueue,_,!),
    repeat, queue_get( goalqueue, Goal/Tag/Number ),
    t_create( answers( Goal, Tag, Number ) ).
    fail.

The answers predicate computes one or all answers, sending them to the server. After producing all the answers, a fail atom is sent, meaning no more answers.

answers( Goal, Tag, Only1 ) :- Goal, deliver_answer( Goal, Tag ), Only1=one.
answers( Goal, Tag, _ ) :- deliver_answer( fail, Tag ).
deliver_answer( Answer, Tag ) :- pvm_parent(PTID), pvm_send(PTID, Tag, Answer).

3.2 PVM controlling predicates

The PVM-Prolog interpreter can be used as an interactive shell to PVM, through the direct use of interface predicates. More powerful control predicates are easily implemented, e.g. pvm_reset, a predicate that kills all user processes in the current environment (except the killer process).

pvm_reset :- pvm_tasks(0, L, _),
    member(taskinfo(TID,_,_,_), L), not pvm_mytid(TID),
    pvm_kill(TID),
    fail.
pvm_reset.

4 Conclusions

We have described a programming interface supporting several forms of explicit parallelism and distribution of programs in a PVM environment.

Multiple applications of the PVM-Prolog model are being exploited in the scope of several research projects in which we are participating as the implementation language to build a parallelized Prolog interpreter for non-monotonic reasoning with explicit negation [8] [9], with application to the domain of model-based diagnosis [10]; to implement shell and controller processes that are able to activate and manage the software components of a distributed processing system, as well as to build specialized intelligent interfaces that must access services which are implemented in distinct languages; as the distribution and communication control platform that supports cooperation among multiple multiprocessor sites, each supporting a specific computation model [6].

Acknowledgments

This research was supported in part by Programa Ciência, project PROLOPPE (PRAXIS/3/3.1/TIT/24/94) and DEC EERP PADIPRO, (P-005).

References


