



HAL
open science

POM: a Virtual Parallel Machine Featuring Observation Mechanisms

Frédéric Guidec, Yves Mahéo

► **To cite this version:**

Frédéric Guidec, Yves Mahéo. POM: a Virtual Parallel Machine Featuring Observation Mechanisms. International Conference on High Performance Computing (HiPC'95), Dec 1995, New Delhi, India. hal-00495379v2

HAL Id: hal-00495379

<https://hal.science/hal-00495379v2>

Submitted on 5 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

POM: a Virtual Parallel Machine Featuring Observation Mechanisms

Frédéric Guidec

Yves Mahéo

IRISA
Campus de Beaulieu
F-35042 Rennes, France
guidec@irisa.fr

IRISA
Campus de Beaulieu
F-35042 Rennes, France
maheo@irisa.fr

We describe in this paper a Parallel Observable virtual Machine (POM), which provides a homogeneous interface upon the communication kernels of parallel architectures. POM was designed so as to be ported easily and efficiently on numerous parallel platforms. It provides sophisticated features for observing distributed executions.

1 Introduction

POM is a Parallel Observable Machine featuring mechanisms for observing distributed applications. It provides a homogeneous interface upon the many communication kernels available on current parallel architectures, and it can be easily and efficiently implemented on these architectures.

POM has not been designed in order to compete with communication interfaces and libraries such as PVM [2], MPI [7] or P4 [3]. These systems mainly aim at easing the task of the programmer of distributed applications by offering a wide variety of communication services (primitives for packing and unpacking typed data in messages, notion of communications within groups of processes, XDR coding for exchanging data in a heterogeneous network, etc.) and dynamic task management. Most of the time, such services are not provided directly by the communication kernels associated with the operating systems, which only offer basic low level communication primitives (*e.g.* untyped data exchanges between neighbouring nodes).

Implementing services more “comfortable” for the application programmer (dynamic task management, packing and unpacking typed data...) requires that complex mechanisms be grouped in software layers above the communication kernels mentioned earlier. Although they effectively ease the task of the appli-

cation programmer, these mechanisms are difficult to implement and their use is costly. They may turn out to be highly prohibitive in some application domains where the seek for performances prevails over the immediate comfort of the application programmer.

The prior goal of POM is not to offer numerous services to the application programmer. It mostly aims at masking the specificities of the various communication kernels of today’s machines with no significative degradation of performances. In that sense, our approach is quite similar to that of projects PICL [5] and PARMACS [4]. However, one of our main priorities while designing POM was to define a model of virtual machine and to clearly specify the semantics of the communications in this model. We also wanted to define an easily portable machine —*i.e.* a machine that can be ported on a given platform in a short time— and whose implementation can be achieved efficiently on many parallel platforms.

We also gave POM sophisticated observation mechanisms. Actually, we consider that any parallel programming environment should include a set of tools to help the programmer design and implement new distributed applications, be it for checking the correctness by detecting and removing bugs, or for improving performances. Considering that even the slightest perturbation in the execution of a distributed application can ruin its analysis, we decided to incorporate observation mechanisms at a low level in POM. Moreover, the observation technique fostered in POM is based on the analysis of execution traces, rather than on a direct observation of distributed applications (in which case the observation is achieved inside the application). The various observation mechanisms offered by POM can all be enabled or disabled separately. The intrusion in the distributed applications observed is thus kept as low as possible.

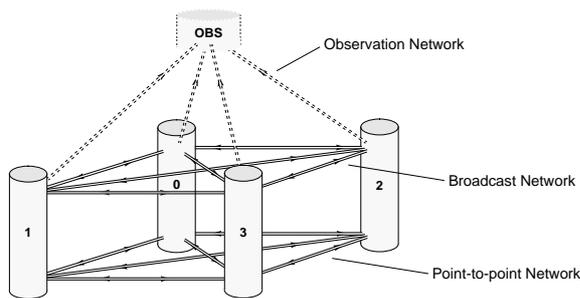


Figure 1: Model of the virtual machine

2 The virtual machine

2.1 Model of the machine

POM defines a model of virtual machine that consists of a set of *application nodes* numbered 0 to $N-1$. These nodes communicate via two distinct media. The first medium is a fully connected network devoted to point-to-point communications. The channels of this network are FIFO and reliable (messages are neither lost nor desequenced). The second medium allows broadcasting messages. It is also a fully connected network with reliable FIFO channels. With this medium, a node can send a message simultaneously on all output channels.

By defining fully connected networks, we intentionally avoided considering the actual physical topology of parallel architectures. This allows for the evolution of modern parallel machines in which messages are routed more and more efficiently by the hardware (or by the low level system). The underlying topology thus remains hidden to the programmer.

The distinction between the two networks is necessary because on many parallel platforms, it is difficult to ensure at low cost that virtual channels carrying both point-to-point messages and broadcast messages are FIFO. Actually, on these platforms, point-to-point and broadcast communications rely on distinct protocols, and sometimes on distinct physical devices too.

Besides the application nodes, POM can include a complementary *observation node*. When this observer is present, one must consider a third communication medium: a network of reliable FIFO channels linking each application node to the observer. Through this observation medium, communications only occur from the application nodes towards the observer.

Tasks are managed statically in POM: there must be a single process on each node of the virtual machine. However, we consider incorporating lightweight

processes in a future version of POM.

2.2 Communication model

The communication paradigm implemented in POM is that of asynchronous message passing. POM permits most of the variations around this kind of communication. Communications can be performed in point-to-point mode or in broadcast mode. Sends are non-blocking, that is, the sending process resumes its execution as soon as the message to be sent has been taken in charge by the underlying operating system. Receives can be deterministic (on a given incoming channel) or non-deterministic (on any incoming channel). Receives are blocking: the receiver resumes its execution only after the message awaited has been effectively received.

Other kinds of communication, such as the rendez-vous, can be built easily on top of these basic operations.

3 Observation mechanisms

POM makes it possible to combine application nodes together with an observation node, whose role is to collect and handle trace information relative to the behaviour of the application. The observation node can proceed to an “on the fly” analysis of the information received, or it can store this information for a *post-mortem* analysis. Actually, the observer can be just a part of a programming environment featuring software tools such as trace collectors, distributed application debuggers, performance analysers and graphical viewers.

Inserting “observation points”.

It is up to the application programmer to specify which events must be traced. To do so, the programmer must insert *observation points* in the code of the distributed application. During the execution of the distributed application, every time an application node runs through an observation point, a *trace message* is sent to the observation node. A trace message is typically composed of information for identifying and dating an event. POM offers several dating mechanisms, whose management remains fully transparent to the application programmer. When loading a distributed application, the programmer simply needs to specify which kind of dating mechanism must be used. The events traced can thus be stamped and/or dated,

and the dating can be achieved according to a local or global time reference.

Stamping events.

Stamping events makes it possible to analyse the synchronisations that occur between the application nodes during a distributed execution. These synchronisations are captured by the notion of causal dependency. Each application node manages a local stamp that is updated every time an application message is sent or received. POM ensures that the value of the local stamp of the sender is sent transparently together with the application message.

To date, POM allows the user to choose between two kinds of stamps: vectorial stamps whose size remains constant during an execution, or “adaptive” stamps whose size can vary dynamically [6]. POM was designed so that it can easily incorporate new kinds of stamps.

Physical dating of events.

POM offers services for dating traced events. The default dating mechanism is based on the local time on each application node (value returned by the physical clock of the processor). POM also incorporates a mechanism for dating events globally. We opted for an approach based on a statistical method that consists in estimating the drift of the physical clock of each application node with respect to a reference clock [?]. Once the characteristics of the clock drifts have been determined for each application node, it is possible to relate the dates taken on the clocks of the application nodes to that of the reference node. The accuracy of the global time obtained this way is sufficient to ensure a coherent dating of events. The advantage of this approach is that it is not intrusive. The measurements required for evaluating the clock drifts are performed before and after the actual execution of the distributed application. This execution is thus not altered by the global dating mechanism. On the other hand, it is necessary to wait till the end of the distributed execution before global dates can be computed. The mechanism for global dating implemented in POM is thus only appropriate for a *post-mortem* trace analysis.

Generic observers.

POM provides the programmer with a set of primitives for developing observation programs. These primitives make it possible to receive trace messages

in a deterministic or non-deterministic way, and to extract from these messages significant information, such as the name of the event traced, the physical date of this event (local or global), the value of the associated stamp, etc.

The application programmer does not have to design a new observation program for each distributed application. Actually, the primitives of POM permit the design of generic observation programs that can perform the most simple observation functions, such as collecting, filtering and storing trace information. Generic observation programs can easily interface with analysis tools such as those designed in our laboratory (visualization of dependency graphs or graphs of global states, concurrency measurement, evaluation of predicates) [1].

When loading a distributed application, the user must specify what kind of trace information must be generated every time an observation point is reached, and which observation program must be used to collect and deal with this information.

The user describes the kind of observation required by passing *ad hoc* parameters to POM’s *loader*, which is detailed in section 4.2.

4 Interface

4.1 Modules APS and OBS

The services offered by POM are made available to the application programmer as a set of around forty primitives that forms two distinct modules. Module APS (*Application Services*) permits the development of application programs, whereas module OBS (*Observation Services*) is devoted to the implementation of observation programs.

The number of primitives has been intentionally limited in order to obtain a simple and easily implementable interface. For example, POM does not propose any non-deterministic receive because such a function can be easily obtained by combining some of the existing primitives. Likewise, we decided not to type messages. The interpretation of the content of a message is thus left to the application programmer and neither packing nor unpacking is necessary.

APS primitives.

The primitives of module APS are mostly communication primitives. Two primitives are available for sending messages, in point-to-point mode (APS_send) or in broadcast mode (APS_bcast). The corresponding

receives (`APS_recv_from` and `APS_recv_bcast_from` respectively) are blocking primitives that necessitate that the incoming channel be identified explicitly. Non-deterministic receive can be realized thanks to the two primitives `APS_probe_from` and `APS_probe_bcast_from` which are non-blocking primitives that test an incoming channel belonging to either the point-to-point communication network, or the broadcast communication network. Primitives `APS_probe` and `APS_probe_bcast` detect pending messages on any incoming channel for one or the other communication network. Once a pending message has been detected, the primitives `APS_info_pid` and `APS_info_length` can be used to get its origin and its length.

Module `APS` additionally incorporates a few primitives that provide information such as the number of application nodes, the identity of the local node, the local time returned by the physical clock of the processor, etc.

Module `APS` provides a primitive `APS_trace` that allows the programmer to insert observation points in the application program. When this primitive is invoked, a message is automatically generated and sent to the observation node. This message contains the information passed as parameters to `APS_trace` by the programmer, namely a string identifying the event observed and optional untyped data whose interpretation is left to the programmer. To this basic information, POM automatically adds dating data, whose nature depends on the observation options specified by the user when loading the application (see section 4.2 for more details).

OBS primitives.

Module `OBS` allows the observer to receive trace messages thanks to a primitive that blocks, waiting for pending messages on a given incoming channel (`OBS_recv_trace_from`). A non-deterministic receive can be done using either `OBS_probe_trace` or `OBS_probe_trace_from`, together with `OBS_info_pid` and `OBS_info_length`. Module `OBS` offers no send primitive. The observation node can only collect trace messages and extract data fields from these messages. For this, some functions of module `OBS` give access to the user data as well as to the value of the stamp and the local date embodied in a trace message. The global date of an event can only be obtained after the distributed application has completed by using function `OBS_convert_to_gclock` which converts the local date of an event into the corresponding global date.

4.2 The loader

The procedure for loading and running a distributed application on a given platform is most of the time highly dependent on the characteristics of this platform. Each architecture imposes its own requirements when it comes to allocating a partition of processors and loading executable programs on this partition. The POM environment includes a *loader* tool named `pom_load`, whose implementation may depend on the platform considered but whose interface remains homogeneous on all platforms.

The syntax recognized by the loader permits complex loadings. One can for example load a different executable program on each application node (or on a subset of the application nodes). One can also pass parameters to the various executable programs (including the observation program), specify which kind of observation must be achieved during the execution, etc. The following example shows how to load and start a distributed application based on the master-slave model, with a single master task running the executable `master` and six slave tasks running the same program `slave`. The master program takes as a parameter the number of slave tasks. Moreover, the behaviour of this distributed application must be observed by the observation program `my_obs`. The trace information must include vectorial stamps (option `-stm VECT`) and events must be dated according to a global time (option `-gtm`).

```
> pom_load -s 7 -on 0 master 6 -on 1..6 slave -stm VECT
   -gtm -obs my_obs
```

In this example the application nodes are given identifiers which are logical identifiers. It is necessary to map these logical identifiers with the physical nodes of the target platform. On some machines, such as the Intel `ipsc` and Paragon `XP/S`, the operating system gives each physical node of a partition of size N a logical identifier ranging from 0 to $N - 1$. Therefore, the user of POM does not always need to describe explicitly the mapping of the application nodes. Yet, this can be done thanks to the option `-map` recognized by `pom_load`. The explicit mapping remains mandatory when the platform considered is composed of a set of workstations. It is then necessary to name explicitly the workstations that will support the distributed application. Hence, for loading and starting on two workstations named `excalibur` and `durandal` a distributed application composed of two programs `ping` and `pong`, the command line can be as shown below:

```
> pom_load -s 2 -on 0 ping -on 1 pong -map 0 durandal
  -map 1 excalibur
```

5 Implementation

To date, POM has been ported on the distributed memory parallel computers of IRISA, that is, the Intel machines iPSC/2 and Paragon XP/S. POM was implemented on these platforms using the communication kernels NX/2, OSF/1 and SUNMOS. It was also implemented so as to allow the execution of distributed applications on a network of workstations (*e.g.* Sun Sparc workstations), using TCP-IP and UDP sockets. Another version makes it possible to simulate parallelism on a single workstation. These two versions are quite useful because they allow the design and the experimentation of new distributed applications without monopolizing the iPSC or the Paragon. Applications can thus be loaded and run on real parallel machines only after they have been tested and corrected. We also implemented POM above PVM [2]. However, this version does not exhibit very good performances on parallel supercomputers: the mechanisms of PVM are too complex and imply too many memory copies to be able to compete with the performances obtained with the more “direct” implementations of POM.

Performances.

This section reports the performances observed when experimenting POM on several platforms. We tested several versions of POM corresponding to alternative implementations on a network of Sun workstations and on the Intel Paragon XP/S.

We also measured the influence of the observation services on the performances of the communications. The technique we use for computing the global time has no effect upon the behaviour of the application, as explained in section 3. On the other hand, the stamping mechanisms can alter the communication performances, although measurements show that this alteration remains negligible.

Figures 2 and 3 shows the maximal bandwidths observed on two Sun Sparc 4/50 IPX workstations connected to the same Ethernet trunk. They also show the bandwidth observed when the stamping service is enabled in POM-TCP. Figure 3 does not distinguish between the performances of POM-TCP with and without stamping, because the difference cannot be measured.

Figure 4 shows the transmission times measured with several versions of the POM library developed

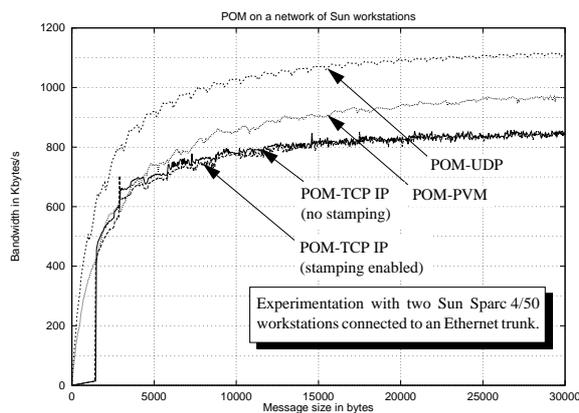


Figure 2: Bandwidth observed for short messages exchanged between two workstations.

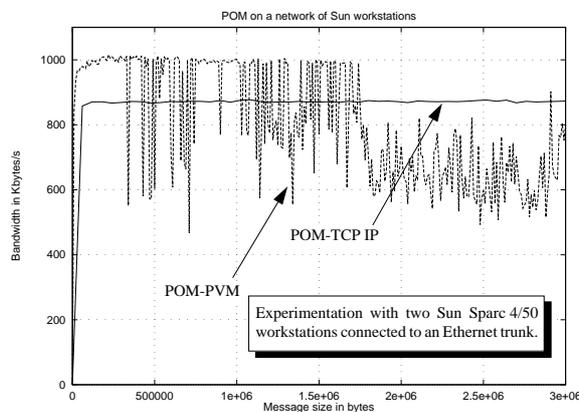


Figure 3: Bandwidth observed for long messages exchanged between two workstations.

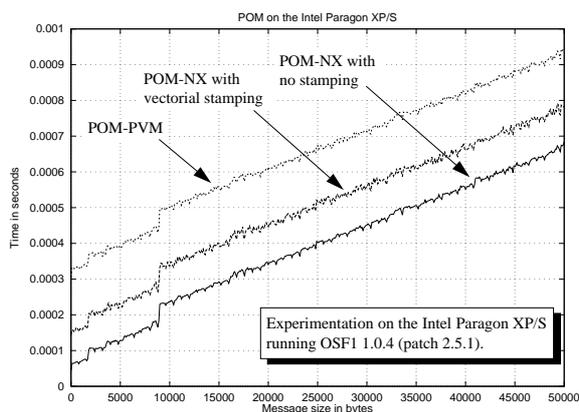


Figure 4: Transmission times observed for short messages on the Intel Paragon XP/S.

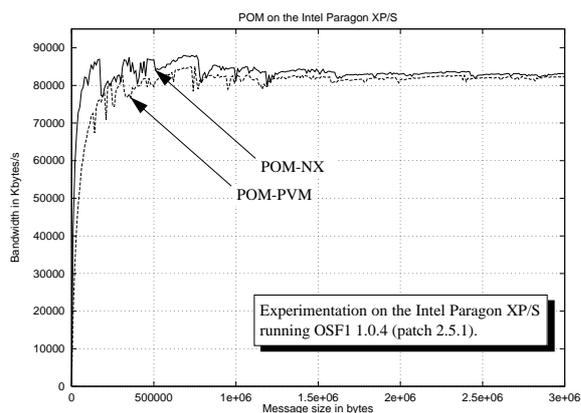


Figure 5: Bandwidth observed on the Intel Paragon XP/S.

for the Intel Paragon XP/S. This figure shows that low latency that can be obtained with POM-NX (implemented directly above the NX-OSF/1 kernel). We also incorporated in figure 4 the transmission times observed with POM-NX when the stamping service is enabled. It turns out that with the current versions of POM-NX, the global cost of stamping mechanisms remains acceptable.

Figure 5 shows the maximal bandwidths observed on the Paragon for long messages. The performances observed when the stamping service is enabled do not appear in this figure, because they superimpose with those obtained when the stamping service is disabled. The maximal bandwidth observed with POM-NX corresponds to that of the maximal bandwidth that can be obtained when calling directly the NX primitives on our Paragon in its current configuration (OSF/1 1.0.4, patch 2.5.1).

6 Conclusion

POM allows the programmer of a distributed application to disregard a given architecture or a given operating system to a large extent. The communication services it offers are basic services, but they can be easily and efficiently implemented on most parallel machines. POM thus fits especially well the design of applications for which performances are the primary concern. Moreover, the observation services it provides broaden its range of application, since they permit the generation, the collection and the exploitation of execution traces and incorporate mechanisms for stamping events and for computing global dates. POM can therefore be perceived as a convenient fa-

cility to interface a distributed application with many trace analysers and graphical viewers.

To date, POM has been ported on several platforms as different as the Intel Paragon XP/S and a network of workstations. We could thus check its effective portability and it is now part of the various parallel programming environments developed in our laboratory.

In the future, we may port POM on new platforms such as the Cray T3D and the IBM SP1. We would also like to experiment POM on a set of workstations connected via a FDDI network. We also consider designing an extended POM featuring parallel I/O mechanisms and allowing lightweight processing on each application node.

References

- [1] C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. Measuring Concurrency of Regular Distributed Computations. In *TAPSOFT'95, Theory and Practice of Software Development*, LNCS 915, Springer Verlag, Aarhus, May 1995.
- [2] A. Beguelin, G. A. Geist, W. Jiang, R. Manchek, K. Moore, and V. Sunderam. *The PVM Project*. Technical Report, Oak Ridge National Laboratory, February 1993.
- [3] R. Butler and E. Lusk. *User's Guide to the P4 Programming System*. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992.
- [4] R. Calkin, R. Hempel, H.-S. Hoppe, and P. Wypior. Portable Programming with the PARMACS Message-Passing Library. *Parallel Computing*, 1994.
- [5] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. *A User's Guide to PICL - A Portable Instrumented Communication Library*. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, May 1992.
- [6] C. Jard and G.-V. Jourdan. *Dependency Tracking and Filtering in Distributed Computations*. Research Report 851, Irista, Rennes, France, August 1994.
- [7] Message Passing Interface Forum. *Document for a Standard Message-Passing Interface*. Technical Report CS-93-214, University of Tennessee, November 1993.