POM: a Parallel Observable Machine

Frédéric Guidec and Yves Mahéo
IRISA, Campus de Beaulieu
35042 Rennes, France
E-mail: {guidec,maheo}@irisa.fr

1 Introduction

POM is a Parallel Observable Machine featuring mechanisms for building and observing distributed applications. It comes in the form of a library built upon the many communication kernels available on current parallel architectures and a loader that provides the user with a homogeneous syntax for launching parallel applications on any parallel platform.

The prior goal of POM is not to offer numerous services to the application programmer as it is done in PVM [2], MPI [9] or P4 [3]. It mostly aims at masking the specificities of the various communication kernels of today's machines with no significant degradation of performances. In that sense, our approach is quite similar to that of projects PICL [5] and PARMACS [4]. Yet, 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.

POM is also provided with sophisticated observation mechanisms that are incorporated at a low level so as to keep perturbations at a minimum. 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). Besides, all the observation facilities can be enabled or disabled separately.

2 Machine model

POM defines a model of virtual machine that consists of a set of *application nodes* numbered 0 to N-1. Two distinct media are used for communications between these nodes. 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.

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.

The communication paradigm implemented in POM is that of asynchronous message passing. POM allows 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: 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 (that is, 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.

3 Observation mechanisms

The role of an observation program, if present, is to collect and handle trace information relative to the behaviour of the application. The observer 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 [10].

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 observers that can perform the most simple observation functions, such as collecting, filtering and storing trace information. Generic observers can easily interface with analysis tools such as those designed in our laboratory [1]. These tools make it possible to visualize dependency graphs as well as graphs of global states, to measure the concurrency in distributed executions, to evaluate predicates, etc. Figure 1 shows some of the graphical views that can be obtained using generic observers coupled with visualization tools.

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

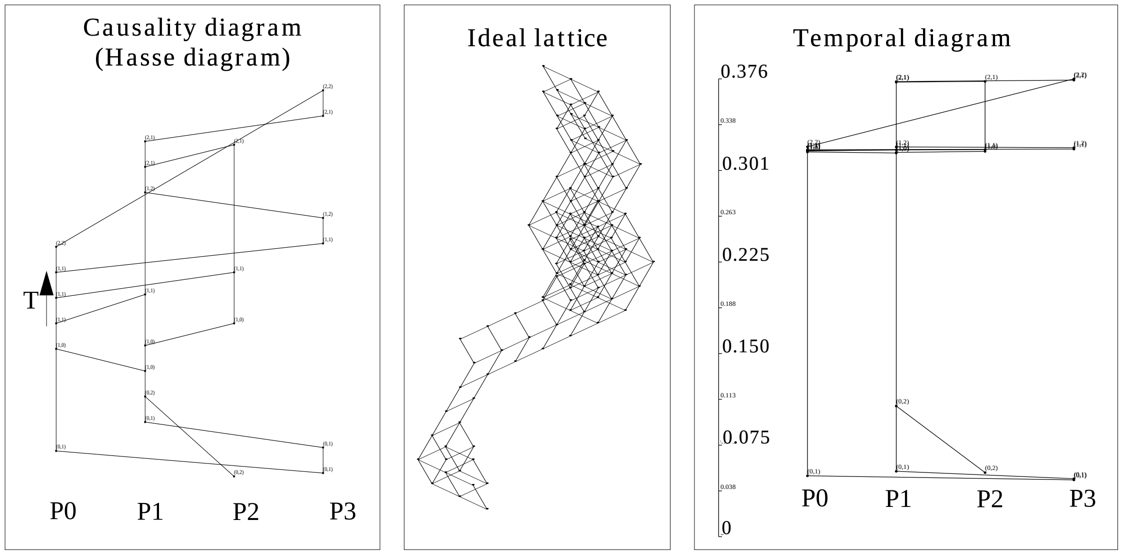


Figure 1: Examples of graphical views produced by generic observers

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 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 [7]). POM was designed so that it can easily incorporate new kinds of stamps.

- 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 [8]. 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. Hence, the execution is not altered by the global dating mechanism. The drawback of this approach is that it is necessary to wait till the end of the distributed execution before global dates can be computed. The global dating mechanism implemented in POM is thus only appropriate for a *post-mortem* analysis.

4 The POM interface

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. Figure 2 shows how these primitives can be used to build a SPMD application that passes two tokens around a bidirectional ring (one token in each direction).

The primitives of module APS are mostly communication primitives for sending and receiving messages in point-to-point or in broadcast mode. Non-deterministic receive can be realized thanks to the primitives that permit to test any incoming channel. 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. The primitive APS_trace (see Figure 2) 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, as well as complementary dating data whose nature depends on the observation options specified by the user when loading the application (see below for more details).

The primitives of module OBS allow the programmer to develop observers. Module OBS offers no send primitive. The observation node can only collect trace messages and extract data fields from these messages. Access is given 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, using a function that converts the local date of an event into the corresponding global date.

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 whose implementation may depend on the platform considered but whose interface remains homogeneous on all platforms. When loading a distributed application, the user can specify what kind of trace information —if any— must be generated every time an observation point is reached, and which observation program must be used to

```
#include "aps.h"
#include <stdio.h>

#define MSG1 "clockwise"
#define MSG2 "anti-clockwise"

main()
{
 int me, pid1, pid2, lg, N;
 char msg[100];

 APS_init(0,NULL);
 me = APS_node_id();
 N = APS_nb_nodes();

 — Node 0 starts communication
 if (me == 0) {
    APS_send(1, strlen(MSG1), MSG1);
    APS_send(N−1, strlen(MSG2), MSG2);
— First direction:
—— determine the source of the first message
while (! APS_probe());
lg = APS_info_length();
pid1 = APS_info_pid();
—— receive from this source
APS_recv_from(pid1, lg, msg);
APS_trace("First receipt", sizeof(int), &pid1);
—— send to the opposite neighbour
pid2 = (pid1==((me+1)%N) ?((me−1+N)%N)
          :((me+1)%N));
if (me != 0) APS_send(pid2, lg, msg);

— Second direction:
—— receive from the neighbour
while (! APS_probe_from(pid2));
lg = APS_info_length();
APS_recv_from(pid2, lg, msg);
APS_trace("Second receipt", sizeof(int), &pid2);
—— send to the opposite neighbour
if (me != 0) APS_send(pid1, lg, msg);

APS_end();  }
```

Figure 2: Example of SPMD code for the application nodes

collect and deal with this information.

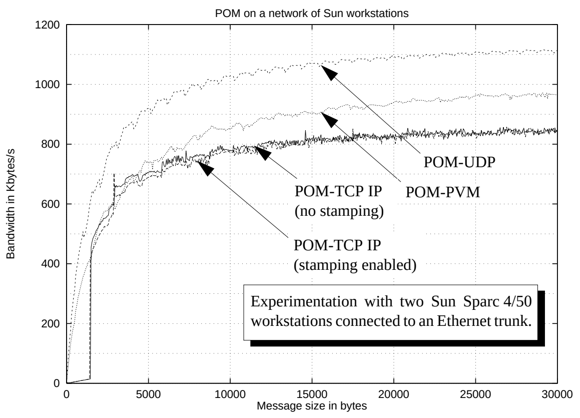
The following example shows how to load and start a distributed application based on the master-slave model, with a single master task running the executable master and six slave tasks running the same program slave. The master program takes as a parameter the number of slave tasks. Moreover, the behaviour 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
```

Other options may also be used for mapping logical node identifiers with the physical nodes of the target platform when necessary. More details on the syntax for loading an application can be found in [6].

5 Performances

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



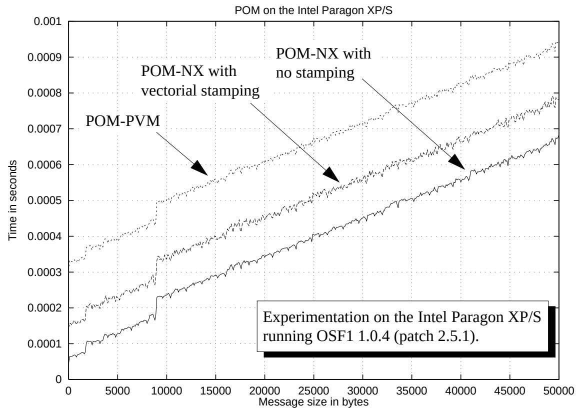


Figure 3: Performances observed for messages exchanged between two workstations (left) and on the Intel Paragon XP/S (right).

Sparc workstations), using TCP-IP and UDP sockets. Another version makes it possible to simulate parallelism on a single workstation (using Unix sockets). We also implemented POM above PVM [2].

We tested several versions of POM corresponding to alternative implementations on a network of Sun workstations and on the Intel Paragon XP/S.

The left part of Figure 3 shows the maximal bandwidths observed on two Sun Sparc 4/50 IPX workstations connected to the same Ethernet trunk. It also shows the bandwidth observed when the stamping service is enabled in POM-TCP. Measurements show that the alteration due to the stamping mechanisms remains negligible. As for the the computation of the global time, it has absolutely no effect upon the behaviour of the application, as explained in section 3.

The right part of Figure 3 shows the transmission times observed with several versions of the POM library developed for the Intel Paragon XP/S. Low latency can be obtained with POM-NX because it is implemented directly above the NX-OSF/1 kernel, whereas POM-PVM exhibits a much higher latency. Actually, the bandwidth observed with POM-NX corresponds exactly to that of the maximal bandwidth that can be obtained when calling directly the NX primitives. The figure also shows 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.

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 facility 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 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] F. Guidec and Y. Mahéo. *POM: a Virtual Parallel Machine Featuring Observation Mechanisms.* Technical report 902, IRISA, Rennes, France, January 1995.

[7] C. Jard and G.-V. Jourdan. *Dependency Tracking and Filtering in Distributed Computations.* Research Report 851, IRISA, Rennes, France, August 1994.

[8] E. Maillet and C. Tron. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 1994.

[9] Message Passing Interface Forum. *Document for a Standard Message–Passing Interface.* Technical Report CS-93-214, University of Tennessee, November 1993.

[10] M. van Riek, B. Tourancheau, and X.-F. Vigouroux. *Monitoring of Distributed Memory Multicomputer Programs.* Technical Report TR93441, Center for Research on Parallel Computation, Rice University, Houston, Texas, 1993.