POM : une machine virtuelle parallèle incorporant des mécanismes d'observation

Frédéric Guidec et Yves Mahéo

Irisa
Campus de Beaulieu
35042 Rennes Cedex
Tél. : +33 99 84 74 03
Fax : +33 99 84 71 71
E-mail : {guidec,maheo}@irisa.fr

RÉSUMÉ. *Nous décrivons dans cet article une machine parallèle virtuelle observable, la POM, qui offre une interface homogène au dessus des systèmes de communication des architectures parallèles. Elle a été conçue en vue d'un portage aisé et efficace sur de nombreuses plates-formes et incorpore des mécanismes élaborés d'observation des exécutions réparties.*

ABSTRACT. *We describe in this paper a Parallel Observable virtual Machine (POM). It offers 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.*

MOTS CLÉS : *architectures parallèles à mémoire distribuée, machine virtuelle, bibliothèque de communication, observation, traces, temps global.*
KEYWORDS : *Distributed memory parallel computers, virtual machine, communication library, observation, traces, global time.*

1 Introduction

POM (*Parallel Observable Machine*) désigne une machine parallèle virtuelle — familièrement référencée sous l'appellation «la POM» — dotée de mécanismes pour l'observation des applications réparties. Elle propose une interface homogène au dessus des multiples noyaux de communication disponibles sur les architectures parallèles actuelles, et peut être mise en œuvre très aisément et très efficacement sur ces architectures.

La POM n'a pas été conçue dans l'idée de concurrencer les interfaces et bibliothèques de communication telles que PVM [BEG 93], MPI [MPI 93] ou P4 [BUT 92].

Ces systèmes ont pour vocation première de faciliter la tâche du programmeur d'applications réparties, en lui offrant une grande variété de services de communication (empaquetage et dépaquetage de données typées dans les messages, communication de groupe, codage XDR permettant les échanges de données dans un réseau hétérogène, etc.) et de gestion dynamique de tâches. En règle générale, de tels services «de confort» ne sont pas fournis directement par les noyaux de communication associés aux systèmes d'exploitation des machines parallèles. Ces noyaux n'offrent le plus souvent que des primitives de communication de très bas niveau (échanges de données non typées, parfois limités aux seuls nœuds voisins dans la topologie physique de la machine).

La mise en œuvre de services plus confortables pour le programmeur d'application oblige à assembler en couches logicielles au dessus des noyaux de communication des mécanismes complexes qui, s'ils facilitent effectivement la tâche du programmeur d'applications réparties, sont en revanche difficiles à mettre en œuvre et d'un emploi coûteux. Par exemple, la gestion dynamique de tâches d'application entraîne des problèmes de nommage, souvent résolus par la mise en place de processus serveurs de noms. On peut également être amené à créer des processus dans le seul but de gérer dynamiquement les groupes de communication. Dans un autre registre, la mise en œuvre de services d'empaquetage et de dépaquetage de données typées dans les messages oblige à de nombreuses manipulations en mémoire afin d'assurer la gestion des messages émis et reçus.

Le coût de tels mécanismes peut s'avérer particulièrement prohibitif dans certains domaines d'application, tels que celui du calcul numérique intensif, où la recherche de performances l'emporte souvent sur le confort immédiat du programmeur d'application.

La POM n'a pas pour vocation première d'offrir une grande variété de services au programmeur d'application, mais de masquer sans dégradation significative de performances les spécificités des différents noyaux de communication des machines actuelles. En ce sens, notre approche est assez semblable à celle suivie dans les projets PICL [GEI 92] ou PARMACS [CAL 94]. Toutefois, l'une de nos priorités au cours de la spécification de la POM a été de définir un modèle de machine virtuelle et de spécifier une sémantique précise pour les communications dans cette machine. Nous nous sommes également efforcés de définir une machine aisément portable — *i.e.* dont le portage peut être réalisé dans un laps de temps réduit — et pouvant être mise en œuvre de manière efficace sur une grande variété de plates-formes parallèles.

Nous avons par ailleurs doté la POM de mécanismes d'observation élaborés. Nous estimons en effet que tout environnement de programmation parallèle doit inclure des outils d'aide à la mise au point des programmes répartis, que ce soit pour en contrôler la correction par détection et élimination des erreurs, ou pour en améliorer les performances. Partant du constat que la moindre perturbation d'une application répartie est susceptible d'enlever toute signification aux analyses d'exécution, nous avons décidé d'intégrer des mécanismes d'observation à bas niveau, au sein même de la POM. La technique d'observation privilégiée dans la POM est en outre fondée sur l'analyse de traces d'exécution, plutôt que sur une observation directe des applications réparties où le traitement consécutif à l'occurence d'un événement est effectué à l'intérieur même de l'application. Enfin, les différents mécanismes d'observation peuvent tous être activés ou désactivés individuellement. On limite ainsi au strict minimum l'intrusion dans les applications observées.

2 Machine virtuelle

2.1 Modèle de machine

La POM définit un modèle de machine virtuelle comprenant un nombre quelconque de *nœuds d'application* numérotés de 0 à $N - 1$. Ces nœuds communiquent via deux médiums distincts :

- le premier médium est un réseau complètement maillé dédié aux communications point-à-point. Les canaux de ce réseau sont FIFO et fiables (il n'y a ni perte ni déséquencement de messages).
- le deuxième médium permet la diffusion de messages. Il s'agit également d'un réseau complètement maillé de canaux fiables et FIFO. Avec ce médium, un nœud peut émettre un message simultanément sur tous les canaux sortants.

En définissant des réseaux complètement maillés, nous nous sommes volontairement affranchis des considérations de topologie. Ceci pour notamment tenir compte de l'évolution des machines parallèles dans lesquelles un routage de plus en plus efficace rend transparent la topologie sous-jacente.

La distinction entre les deux réseaux est nécessaire car, sur bon nombre de plates-formes parallèles, il est difficile de garantir à faible coût le caractère FIFO de canaux virtuels de communication où circulent à la fois des messages diffusés et des messages émis en point-à-point. En effet, les communications en mode point-à-point et en diffusion font appel sur ces plates-formes à des protocoles différents, voire à des dispositifs matériels distincts. C'est par exemple le cas sur l'iPSC d'Intel : les messages émis en mode point-à-point et les messages diffusés n'empruntent pas nécessairement le même chemin physique pour rallier un nœud émetteur à un nœud récepteur. Le système d'exploitation associé n'assurant pas le séquencement entre les messages de différentes natures, l'ajout au niveau de la POM d'une couche logicielle comblant cette lacune entraînerait des surcoûts prohibitifs.

Outre les nœuds d'application, la machine virtuelle POM peut inclure un nœud supplémentaire, baptisé *nœud observateur*. En présence de cet observateur, on doit considérer un troisième médium de communication : un réseau comportant un ensemble de canaux fiables et FIFO reliant chaque nœud d'application à l'observateur. À travers ce médium d'observation, les communications ne se font que depuis les nœuds d'application vers l'observateur. Un nœud d'application peut émettre sur le canal sortant. Le nœud observateur peut quant à lui recevoir sur un canal entrant et tester les files de réception.

2.2 Modèle de communication

Le paradigme de communication retenu dans la machine virtuelle POM est celui des communications par messages asynchrones. La POM permet la plupart des variations sur ce type de communication :

- les communications peuvent se faire en point-à-point et en diffusion ;
- les émissions sont non-bloquantes, c'est-à-dire que le processus émetteur poursuit son exécution dès que le message à émettre a été pris en compte par le système ;
- les réceptions sont bloquantes : le processus récepteur ne poursuit son exécution qu'après avoir effectivement reçu le message attendu ;
- on peut néanmoins tester sans bloquer la présence de messages dans les files de réception.

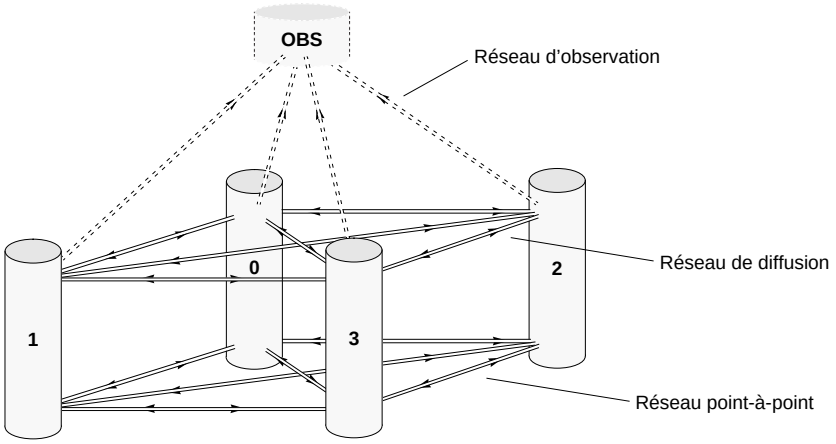

FIG. 1 - *Modèle de la machine virtuelle POM*

D'autres opérations de communication, telles que le rendez-vous, peuvent être construites aisément à l'aide des opérations de base énumérées ci-dessus.

2.3 Gestion de processus

La gestion des processus dans la POM est une gestion statique : il y a un seul processus par nœud de la machine virtuelle, et tous les processus sont créés au lancement de l'application. Ce choix est principalement justifié par le fait que l'intégration dans la POM de mécanismes permettant la gestion de processus multiples compliquerait fortement son interface. Cela rendrait en outre son portage beaucoup plus ardu, ne serait-ce que parce que toutes les plates-formes n'offrent pas un noyau d'exécution multi-tâches. D'autre part, dans le domaine du calcul numérique intensif pour lequel la POM a initialement été conçue, l'allocation dynamique de processus est souvent perçue comme trop coûteuse, ou tout simplement inutile. Toutefois, dans la perspective de pouvoir exprimer un parallélisme interne à chaque nœud d'application, il est envisagé d'offrir la gestion de processus légers dans une version future de la POM.

Il faut noter que, bien que la POM ne permette pas la création dynamique de processus au niveau d'un programme d'application, ceci n'empêche nullement certaines mises en œuvre de la POM d'utiliser les capacités multi-tâches des plates-formes parallèles. C'est notamment le cas lorsque la POM est mise en œuvre sur une station de travail afin de simuler un environnement d'exécution parallèle.

2.4 Entrées/sorties

Certaines recherches actuelles (*e.g.* [COR 94]) visent à définir des interfaces de haut niveau pour les entrées/sorties parallèles. Cependant, les capacités des architectures parallèles dans ce domaine sont encore très diverses. La plupart des plates-formes ne fournissent d'ailleurs pas de réels mécanismes d'entrées/sorties parallèles. Partant de ce constat, il nous a semblé prématuré d'essayer de spécifier au niveau de la POM des fonctions d'entrées/sorties parallèles portables. C'est la raison pour laquelle la POM n'offre pour l'instant aucune fonction de ce type.

3 Mécanismes d'observation

La POM permet d'associer aux nœuds d'application un nœud observateur ayant pour fonction de collecter et d'exploiter les informations de traces relatives au comportement de l'application. Le nœud observateur peut procéder à une analyse des informations reçues «à la volée», ou se contenter de les stocker pour une utilisation ultérieure (analyse *post–mortem*). L'observateur peut d'ailleurs n'être que le premier maillon d'un environnement de programmation incluant des outils de collecte de traces, de débogage de programmes parallèles, d'analyse de performances et de visualisation graphique. Le développement d'un tel environnement de trace est l'objectif des membres du projet inter-PRC Trace (Mesr/Cnrs).

3.1 Insertion de «points d'observation»

Il demeure à la charge du programmeur d'application de spécifier quels événements doivent être tracés. Pour ce faire, il doit insérer des *points d'observation* dans le code de l'application. Cette approche est assez semblable à celle qui consiste à insérer des points d'arrêt dans un programme séquentiel sur lequel on veut procéder à un débogage interactif. Lors de l'exécution, le passage sur un point d'observation va se traduire par l'envoi d'un *message de trace* vers le nœud observateur. Un message de trace contient typiquement des informations permettant d'identifier et de dater l'événement tracé. La POM offre plusieurs mécanismes de datation, dont la gestion demeure entièrement transparente pour le programmeur d'application. Lors du lancement de l'application répartie, il suffit à l'utilisateur de spécifier quel type de datation il désire voir employer. Les événements tracés peuvent ainsi être estampillés et/ou datés, la datation pouvant en outre être réalisée dans un référentiel de temps local ou global.

3.2 Estampillage des événements

L'estampillage permet l'analyse des synchronisations qui se produisent entre les nœuds d'application durant l'exécution [JAR 94]. Ces synchronisations sont capturées par la notion de dépendance causale, introduite par Lamport en 1978, et qui fait abstraction du temps physique. Chaque nœud d'application gère une estampille locale, mise à jour à chaque émission ou réception d'un message d'application. Lorsqu'un nœud d'application adresse un message à un autre nœud d'application, la valeur de l'estampille du nœud émetteur est automatiquement adjointe au message émis. L'émission de l'estampille ainsi que sa réception et son traitement au niveau du nœud destinataire du message sont réalisés directement par les mécanismes de la POM. Cela assure une complète transparence pour le programmeur de l'application. Dans sa version actuelle, la POM permet à l'utilisateur de choisir entre deux types d'estampilles : des estampilles vectorielles [FID 91] dont la taille est constante au cours d'une exécution et déterminée par le nombre de nœuds d'application, ou des estampilles dites «adaptatives» dont la taille peut varier en cours d'exécution [JAR 95]. La POM a en outre été conçue de manière à pouvoir aisément intégrer de nouveaux types d'estampilles dans un but d'expérimentation.

3.3 Datation physique des événements

La POM offre également des services de datation physique des événements tracés. Le mécanisme utilisé par défaut réalise une datation des événements en fonction du temps local à chaque nœud d'application (valeur retournée par l'horloge locale de chaque processeur). La POM intègre également un mécanisme de datation globale des événements. Nous avons opté pour une approche fondée sur une méthode statistique d'estimation des décalages et des dérives des différentes horloge locales des nœuds d'application par rapport à une horloge de référence. L'idée de base est

celle exposée dans [HAD 88]; elle a été pour la première fois implantée sur machine parallèle pour Echidna [JEZ 89] et une variante, développée dans l'équipe Apache à Grenoble [MAI 95], a ensuite été intégrée à la POM. Une fois déterminées les caractéristiques de dérive des horloges des différents nœuds d'application, on est en mesure de ramener les dates prises sur les horloges des différents nœuds d'application à celle du nœud de référence qui est alors la référence de temps pour l'ensemble des nœuds d'application. La précision du temps global ainsi obtenu suffit en général pour garantir une datation cohérente des événements. Cette approche présente l'avantage d'être totalement non intrusive. Les mesures nécessaires à l'évaluation des décalages et des dérives des horloges étant effectuées seulement avant et après l'exécution de l'application proprement dite, celle-ci ne subit aucune perturbation due au mécanisme de datation globale. En contrepartie, il est nécessaire d'attendre la fin de l'exécution répartie pour que les dates globales puissent être calculées. Le mécanisme de datation globale mis en œuvre dans la POM n'est donc approprié que pour l'analyse de traces *post–mortem*.

3.4 Observateurs génériques

La POM offre au programmeur un ensemble de primitives permettant le développement des programmes d'observation. Ces primitives permettent aux programmes observateurs de réceptionner les messages de trace de manière déterministe ou non-déterministe, et d'en extraire les différentes composantes significatives, telles que le nom identifiant l'événement tracé, sa date physique (en temps local ou global), la valeur de l'estampille associée, etc.

Il n'est nullement nécessaire de concevoir un programme observateur spécialement adapté à chaque nouvelle application répartie. En fait, les primitives de la POM permettent de construire des programmes d'observation génériques capables de réaliser les fonctions d'observation les plus simples, telles que la collecte, le filtrage et le stockage des informations de trace. Des programmes d'observation de ce type pourront d'ailleurs aisément s'interfacer avec les outils d'analyse développés dans le cadre du projet Trace. Quelques outils de ce type ont déjà été conçus dans l'équipe Pampa, qui permettent la visualisation du graphe de dépendance ou de l'espace des états globaux («treillis des idéaux»), le calcul de mesures de concurrence ainsi que le calcul de prédicats (pour isoler des motifs dans l'exécution) [DIE 93, JAR 94, BAR 95]. La figure 2 illustre le type de visualisation que nous pouvons effectuer. Le graphe reproduit dans cette figure est le graphe associé à l'exécution d'un algorithme de calcul d'un Jacobien sur la machine parallèle Paragon. On voit nettement la régularité du calcul et l'existence de resserrements, indice d'une mauvaise exploitation possible du parallélisme.

3.5 Observation sans nœud observateur

Il n'est pas toujours nécessaire — ni même souhaitable — d'allouer un nœud observateur lorsque l'on désire simplement tracer de manière grossière le comportement d'une application répartie. La POM permet donc de tracer une exécution répartie sans que les messages de trace soient adressés à un nœud observateur. Dans ce cas, les informations relatives aux événements tracés sont simplement affichées sur la sortie standard des nœuds d'application concernés. Cette approche peut présenter un intérêt, par exemple lorsque l'on désire consacrer tous les nœuds disponibles sur une plate-forme donnée à l'exécution de l'application. Elle peut également s'avérer utile lorsque les événements tracés sont en très petit nombre, ou lorsque l'application observée utilise au maximum de leur capacité les canaux de communication de la plate-forme utilisée.

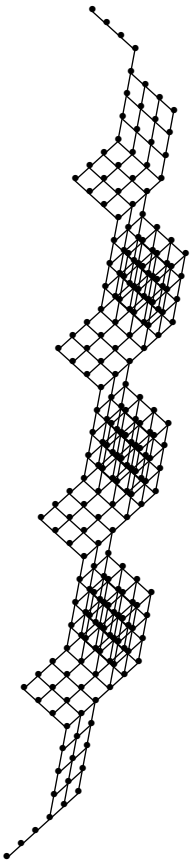

FIG. 2 - *Diagramme de Hasse du treillis généré à partir des traces d'exécution d'un algorithme de calcul d'un Jacobien*

3.6 Exécution sans observation

La POM permet de spécifier lors du lancement d'une application répartie quel type d'observation doit être réalisé, c'est-à-dire quelles informations de trace doivent être produites lors du passage sur un point d'observation et quel programme d'observation doit être utilisé pour collecter et traiter ces informations.

L'utilisateur spécifie le type d'observation désiré en passant des paramètres *ad hoc* au *lanceur* de la POM, décrit en détails au paragraphe 4.2. Il peut ainsi observer le comportement d'une application répartie en s'aidant des traces d'exécution puis, une fois assuré du bon fonctionnement de l'application, inhiber les mécanismes d'observation afin d'obtenir de meilleures performances.

Bien que les mécanismes d'observation de la POM soient mis en œuvre sur chaque plate-forme de manière à être aussi peu intrusifs que possible, la POM est mise à la disposition du programmeur sous la forme de deux bibliothèques distinctes. La première bibliothèque intègre tous les mécanismes de communication et d'observation alors que la seconde, qui ne fournit que les mécanismes de communication, permet éventuellement d'obtenir de meilleures performances sur certaines plates-formes.

La POM est donc proposée sur chaque plate-forme en deux versions totalement interchangeables du point de vue de l'application. Le choix de l'une ou l'autre version s'effectue lors de la compilation d'un programme d'application. Il suffit au programmeur désirant compiler et expérimenter un nouveau code d'application de spécifier laquelle des deux bibliothèques doit être utilisée, selon que l'exécution de l'application doit être tracée ou non.

Il est important de noter que les deux versions offrent exactement les mêmes services aux nœuds d'application. Un programmeur peut donc décider de générer d'abord une application répartie observable à l'aide de la première version, avant de produire un code d'application plus efficace — mais non observable — en employant cette fois la seconde version. Il lui suffit pour cela de recompiler l'application, sans toutefois apporter le moindre changement au code source.

4 Interface

4.1 Modules APS et OBS

Les services offerts par la POM sont mis à la disposition du programmeur d'application sous la forme d'une quarantaine de primitives regroupées de manière à constituer deux *modules* distincts. Le module APS (*APplication Services*) permet le développement de programmes d'application, alors que le module OBS (*OBservation Services*) est dédié à la mise en œuvre du programme d'observation.

Le nombre de fonctions offertes a été volontairement réduit pour maintenir une interface simple et pouvant être mise en œuvre efficacement. On ne trouve par exemple pas de notion de communication de groupe. De même, la POM n'offre pas de primitive de réception non déterministe car une telle fonction peut facilement être obtenue en combinant quelques unes des primitives existantes. Dans une même optique, il a été décidé de ne pas typer les messages, l'interprétation du contenu des messages est donc laissée au programmeur d'application et il n'est donc pas nécessaire d'utiliser de services d'empaquetage et de dépaquetage des messages.

4.1.1 *Primitives du module APS*

Les primitives du module APS sont pour la plupart des primitives de communication. Dans la figure 3, on montre comment ces primitives peuvent être utilisées pour construire une application SPMD faisant circuler deux «jetons» sur un anneau bidirectionnel (un jeton dans chaque sens).

On dispose de deux primitives permettant l'émission non bloquante en mode point-à-point (APS_send) et en diffusion (APS_bcast). Les primitives de réception correspondantes (respectivement APS_recv_from et APS_recv_bcast_from) sont bloquantes et nécessitent que le canal de réception soit nommé explicitement. Elles sont donc dédiées à la réception déterministe des messages. La réception non déterministe peut être réalisée à l'aide des primitives de test APS_probe_from et APS_probe_bcast_from, qui sont non bloquantes et servent à tester un canal d'entrée donné appartenant soit au réseau de communication point-à-point, soit au réseau de diffusion. Les primitives APS_probe et APS_probe_bcast permettent de même de détecter l'arrivée d'un message sur n'importe lequel des canaux entrants pour chacun des deux réseaux de communication. Lorsque le test d'un canal entrant a permis de détecter l'arrivée d'un message, les primitives APS_info_pid et APS_info_length peuvent être utilisées pour en connaître l'origine et la longueur.

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

#define MSG1 "sens des aiguilles d'une montre"
#define MSG2 "sens trigonométrique"

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

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

 — Le noeud 0 initie la communication
 if (moi == 0) {
     APS_send(1, strlen(MSG1), MSG1);
     APS_send(N−1, strlen(MSG2), MSG2);
 }

 — Transmission dans le premier sens :
 —— déterminer la source du premier message
 while (! APS_probe());
 lg = APS_info_length();
 pid1 = APS_info_pid();
 —— recevoir de cette source
 APS_recv_from(pid1, lg, msg);
 APS_trace("premiere reception", sizeof(int), &pid1);
 —— envoyer au voisin opposé
 pid2 = (pid1==((moi+1)%N)?((moi−1+N)%N) :((moi+1)%N));
 if (moi != 0) APS_send(pid2, lg, msg);

 — Transmission dans le sens opposé :
 —— recevoir du voisin
 while (! APS_probe_from(pid2));
 lg = APS_info_length();
 APS_recv_from(pid2, lg, msg);
 APS_trace("deuxieme reception", sizeof(int), &pid2);
 —— envoyer au voisin opposé
 if (moi != 0) APS_send(pid1, lg, msg);

 APS_end();
}
```

FIG. 3 - *Exemple de code SPMD pour les nœuds d'application*

Le module APS offre également quelques routines supplémentaires, qui fournissent divers renseignements sur le contexte d'exécution du programme courant (*e.g.* nombre de nœuds d'application, identité du nœud local, heure locale retournée par l'horloge physique du processeur).

Le module APS offre enfin une primitive APS_trace permettant au programmeur d'insérer dans les programmes d'application les points d'observation évoqués au chapitre 3. Lors de l'exécution de cette primitive, un message de trace est généré et adressé automatiquement au nœud observateur. Ce message contient les informations passées en paramètres à la primitive APS_trace par le programmeur, à savoir une chaîne de caractères identifiant l'événement observé, ainsi que des données facultatives de type quelconque dont l'interprétation est laissée libre au programmeur. À ces informations de base, la POM adjoint automatiquement des données de datation (date et/ou estampille), dont la nature exacte dépend des options d'observation spécifiées par l'utilisateur lors du lancement de l'application (voir à ce sujet le paragraphe 4.2).

4.1.2 *Primitives du module OBS*

Le module OBS permet la réception des messages de trace grâce à une primitive de réception bloquante sur un canal donné (OBS_recv_trace_from). La réception indéterministe peut être obtenue en utilisant l'une des primitives de test OBS_probe_trace et OBS_probe_trace_from, ainsi que les primitives OBS_info_pid et OBS_info_length. L'observation devant être aussi peu intrusive que possible, le module OBS n'offre aucune primitive d'émission. Le nœud observateur doit se contenter de collecter et de traiter localement les messages de trace reçus. Pour ce faire, il dispose de plusieurs fonctions pour extraire les champs d'un message de trace. Ces fonctions permettent d'accéder aux données utilisateur ainsi qu'à la valeur de l'estampille et du temps local. L'obtention de la valeur du temps global ne peut se faire qu'après que l'application répartie observée ait terminé. La fonction OBS_convert_to_gclock permet alors de transformer la date locale d'un événement en date globale.

Des mécanismes de détection de terminaison des nœuds d'application sont également disponibles. Les fonctions OBS_node_ended et OBS_application_ended permettent à l'observateur de détecter la terminaison d'un nœud donné et de l'ensemble de l'application observée.

4.2 Le «lanceur» de la POM

La marche à suivre pour charger et exécuter une application répartie sur une plate-forme donnée est, la plupart du temps, étroitement dépendante des caractéristiques de cette plate-forme. Chaque architecture impose ses exigences propres lorsqu'il s'agit d'allouer une partition de processeurs et d'y charger des programmes exécutables. En spécifiant la POM, nous avons défini une syntaxe générique grâce à laquelle le chargement et le lancement d'une application répartie peuvent s'effectuer de manière homogène sur n'importe quelle plate-forme. La POM est dotée d'un «lanceur» baptisé pom_load dont la mise en œuvre peut dépendre de la plate-forme considérée, mais dont l'interface demeure homogène sur l'ensemble des plates-formes. Ainsi, le chargement de l'application SPMD constituée de six noeuds d'application exécutant le programme ring de la figure 3 peut être effectué en entrant simplement la commande suivante :

```
> pom_load -s 6 -on all ring
```

Pour observer le comportement de cette application à l'aide du programme obs_ring de la figure 4, il suffit d'enrichir la ligne de commande précédente en demandant

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

main()
{
 int pid,lg;
 char trace[200];

 OBS_init(0,NULL);
 while (!OBS_application_ended()) {
     while (!OBS_probe_trace());
     pid = OBS_info_pid();
     lg = OBS_info_length();
     if (OBS_recv_trace_from(pid,lg,trace) != 0)
       printf("noeud %d : la %s se fait depuis le noeud %d\n",
             pid,
             OBS_name_field(trace),
             *(int*)OBS_data_field(trace),
 }
 OBS_end();
}
```

FIG. 4 - *Exemple de code pour le nœud observateur*

l'allocation d'un nœud observateur :

```
> pom_load -s 6 -on all ring -obs obs_ring
```

La syntaxe admise par le lanceur est beaucoup plus riche que ne le montre ces simples exemples. On peut par exemple charger un programme exécutable différent sur chacun des nœuds d'application (ou sur un sous-ensemble des nœuds d'application), passer des paramètres aux différents programmes exécutables (y compris au programme d'observation), spécifier à quel type d'observation on désire se livrer pendant l'exécution, etc. Dans l'exemple suivant, on montre comment il est possible de charger et exécuter une application répartie fonctionnant selon le modèle maître-esclaves, avec une seule tâche maître sur laquelle est chargée l'exécutable master et six tâches esclaves exécutant le même programme slave. On passe en paramètre au programme maître le nombre de tâches esclaves. On souhaite en outre observer le comportement de cette application répartie à l'aide du programme d'observation my_obs. On veut enfin que les informations de trace incluent des estampilles vectorielles (option -stm VECT) et que les événements soient datés dans un référentiel de temps global (option -gtm).

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

Les numéros des nœuds d'application utilisés dans les exemples précédents sont des numéros logiques, qu'il est nécessaire de mettre en correspondance avec les nœuds physiques de la plate-forme cible. Sur certaines machines telles que l'iPSC ou la Paragon XP/S, le système d'exploitation associé par défaut à chaque nœud physique

d'une partition de taille N un numéro logique compris entre 0 et $N-1$. L'utilisateur de la POM n'est donc pas contraint de décrire explicitement le placement des nœuds d'application. Il peut cependant le faire grâce à l'option -map du lanceur pom_load. Le placement explicite demeure d'ailleurs indispensable lorsque la plate-forme considérée est constituée d'un ensemble de stations de travail. Il faut alors nommer explicitement chacune des stations devant participer à l'exécution de l'application. Ainsi, pour charger et exécuter sur deux stations de travail nommées excalibur et durandal une application répartie constituée de deux programmes ping et pong, il suffit d'entrer une ligne de commande de la forme :

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

5 Portages effectués

À ce jour, la POM a été portée sur les machines parallèles à mémoire distribuée de l'Irisa, à savoir les machines Intel iPSC/2 et Paragon XP/S. Elle utilise sur ces machines les noyaux de communication NX/2, NX-OSF/1 et SunMOS. Une version de la POM a également été mise en œuvre, qui permet l'exécution d'applications réparties sur réseau de stations de travail (*e.g.* stations Sun). Cette version utilise les *sockets* TCP-IP et UDP. Une autre version permet la simulation du parallélisme sur une seule station de travail. Ces deux dernières versions de la POM se révèlent particulièrement utiles car elles permettent le développement et la mise au point de nouvelles applications réparties, sans qu'il faille monopoliser inutilement l'iPSC ou la Paragon. Les applications peuvent ainsi n'être chargées et exécutées sur les machines parallèles qu'après avoir été suffisamment testées et corrigées. Nous avons également mis en œuvre la POM au dessus de PVM [BEG 93]. Cette version s'avère intéressante pour exécuter des applications réparties sur réseau de stations de travail, mais elle ne permet en revanche pas d'atteindre des performances satisfaisantes sur les machines massivement parallèles de l'Irisa : les mécanismes internes à PVM sont trop coûteux pour que l'on puisse concurrencer avec PVM les performances obtenues via les autres mises en œuvre plus «directes» de la POM.

6 Performances

Nous avons procédé à une série d'expériences visant à estimer les performances relatives des différentes versions de la POM, mises en œuvre d'une part pour l'exécution sur réseau de stations de travail, et d'autre part pour la machine Intel Paragon. Nous rapportons dans ce paragraphe les résultats de ces expériences.

Nous avons dans un premier temps cherché à évaluer la bande passante maximale pouvant être obtenue avec chaque version de la POM. Pour ce faire, nous avons développé une application distribuée très simple dans laquelle deux nœuds d'application s'échangent des messages de taille croissante selon un protocole de communication de type «ping-pong».

Pour chacune des expériences réalisées, nous avons tout d'abord construit des programmes d'application invoquant *directement* les primitives de communication du noyau utilisé (TCP-IP, UDP, PVM, NX) afin de comparer les performances ainsi obtenues avec celles de la version correspondante de la POM. Il s'avère que le surcoût introduit par la POM demeure négligeable dans tous les cas. C'est la raison pour laquelle n'apparaissent dans les figures qui suivent que les performances de la POM.

Nous avons par ailleurs mesuré l'impact des services d'observation sur les performances des primitives de communication de la POM. La technique de calcul du temps global ne perturbant pas les échanges de données réalisés par l'application, seule l'utilisation du service d'estampillage peut entraîner une dégradation des performances.

Toutefois, les mesures montrent que le coût de l'estampillage demeure raisonnable sur l'ensemble des plates-formes.

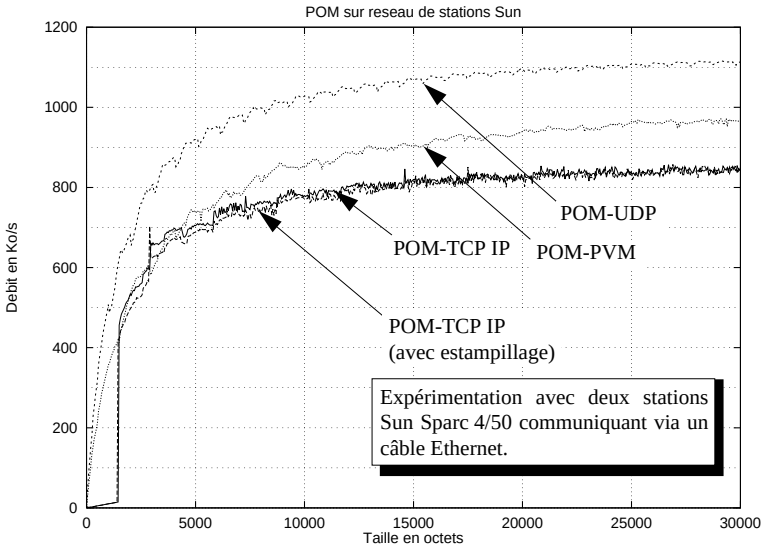

FIG. 5 - *Débits observés lors d'échanges de messages courts entre deux stations Sun Sparc 4/50 IPX reliées par un câble Ethernet.*

La figure 5 montre les bandes passantes maximales observées en faisant communiquer deux stations de travail de type Sun Sparc 4/50 IPX appartenant à un même segment Ethernet. La bibliothèque POM-UDP — mise en œuvre en utilisant des sockets UDP — permet d'atteindre une bande passante maximale légèrement supérieure à 1100 Ko/s, ce qui est très proche de la bande passante nominale du support Ethernet (environ 1250 Ko/s). Cependant, cette version de la POM doit être utilisée avec précaution car les échanges de données n'y sont pas fiabilisés.

Dans la même figure, on voit que la bibliothèque POM-PVM — autre version de la POM mise en œuvre cette fois au dessus de PVM 3.3.6— permet d'obtenir des performances intéressantes (environ 950 Ko/s) pour les messages de petite taille, alors que POM-TCP — version utilisant des sockets TCP-IP — présente des performances moindres (environ 850 Ko/s), voire médiocres lorsque les messages echangés sont de longueur inférieure à 1460 octets.

Sur réseau de stations de travail, les performances de POM-PVM sont donc bonnes, ce qui s'explique par le fait que les communications entre stations s'appuient sur des sockets UDP. En outre, pour mettre en œuvre la POM au dessus de PVM, nous avons utilisé les primitives `pvm_psend()` et `pvm_precv()`, qui ne sont apparues que dans la version 3.3 de PVM et permettent de réaliser des communications «directes» (c'est-à-dire sans empaquetage/dépaquetage des messages).

On peut observer sur la figure 6 que lorsque les messages échangés sont de très grande taille (certaines applications de calcul numérique amènent à des transferts de messages dépassant 1 Mo), les performances de POM-PVM se dégradent sensiblement alors que celles de POM-TCP se maintiennent.

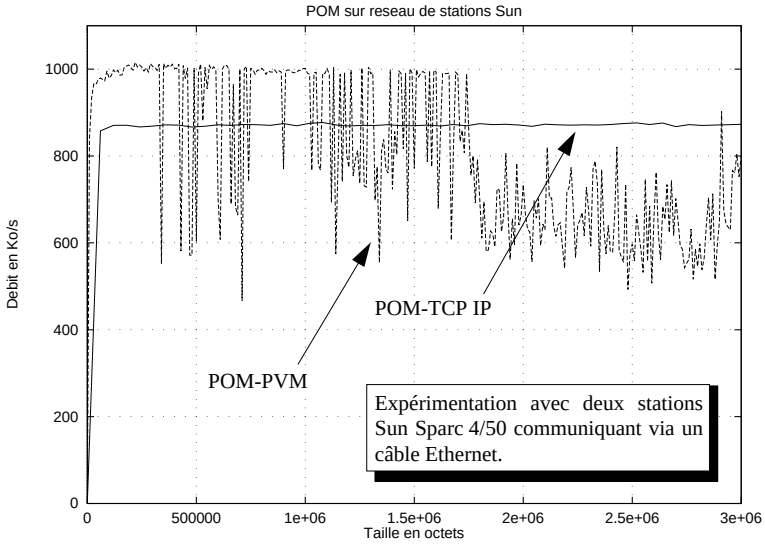

FIG. 6 - *Débits observés lors d'échanges de messages longs entre deux stations Sun Sparc 4/50 IPX reliées par un câble Ethernet.*

Nous avons également fait apparaître dans la figure 5 la bande passante observée lorsque le mécanisme d'estampillage est activé dans la version TCP-IP (à chaque message émis est alors associée une estampille vectorielle). On constate que les deux courbes (POM-TCP IP avec et sans estampillage) sont pratiquement confondues. L'impact de l'estampillage reste donc minime. Nous avons calculé que la bande passante est diminuée d'au plus 7 %, la différence devenant totalement négligeable pour les messages dont la taille dépasse 10 Ko (c'est la raison pour laquelle nous n'avons pas fait apparaître deux courbes distinctes pour POM-TCP IP dans la figure 6).

La figure 7 montre les temps de transmission mesurés avec les différentes versions de la POM développées pour la machine parallèle Intel Paragon XP/S de l'Irisa. On constate que les temps de latence de la version POM-PVM sont nettement supérieurs à ceux de la version POM-NX (mise en œuvre au dessus du noyau de communication NX-OSF/1). Nous avons également fait apparaître dans cette figure les temps de transmission observés avec la bibliothèque POM-NX lorsque le service d'estampillage est activé. Les mesures révèlent que, dans la version actuelle de la POM, l'utilisation du service d'estampillage dégrade de manière significative les temps de latence, qui se trouvent multipliés par un facteur 2 dans le cas de la bibliothèque POM-NX. Nous pensons cependant être capables de ramener ce coût à une valeur plus raisonnable en apportant quelques modifications à la POM. Quoi qu'il en soit, la figure 7 montre également que le coût de l'estampillage reste tout à fait supportable, puisque les performances observées avec la bibliothèque POM-NX lorsque l'estampillage est activé demeurent meilleures que celles de POM-PVM fonctionnant sans estampillage.

La figure 8 montre les débits maximaux obtenus sur la machine Paragon pour des messages de grande taille (1 Mo et au delà). Nous n'avons pas fait apparaître les courbes correspondant aux mesures avec estampillage, car elles se confondent totalement avec celles des mesures sans estampillage.

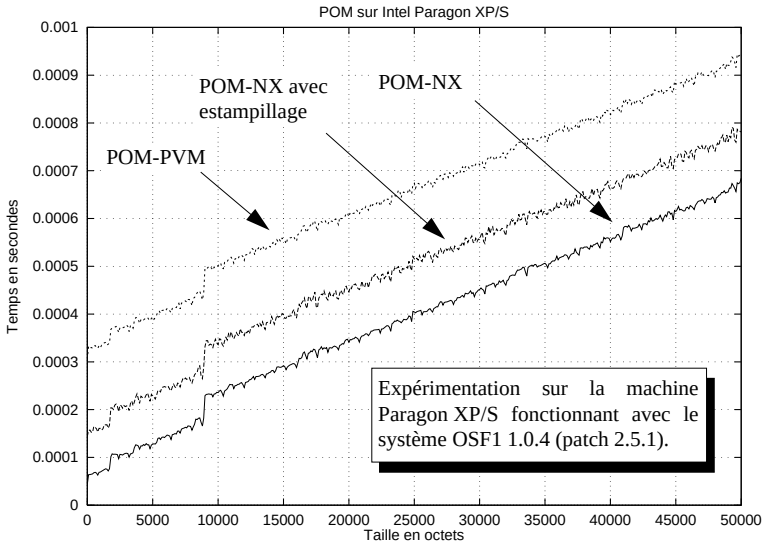

FIG. 7 - *Temps de transmission observés sur la machine Intel Paragon XP/S (système NX-OSF/1).*

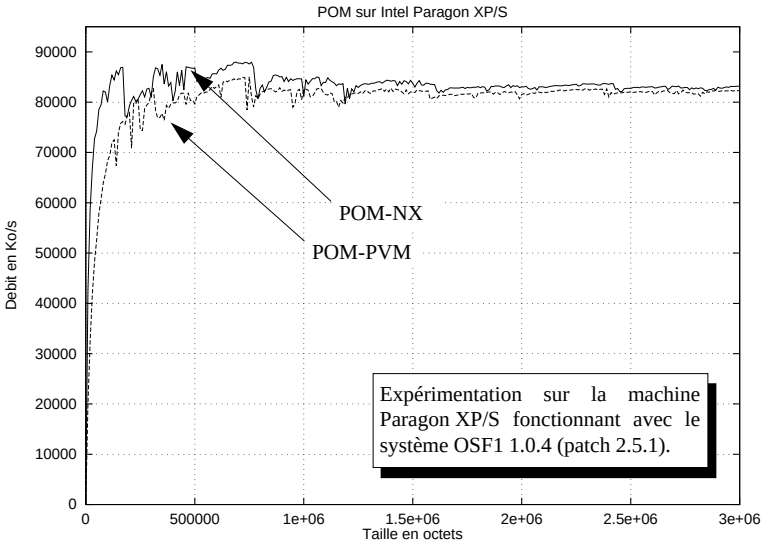

FIG. 8 - *Débits observés sur la machine Intel Paragon XP/S (système NX-OSF/1).*

On constate sur la figure 8 que la bande passante maximale obtenue se situe à environ 82 Mo/s, que ce soit avec POM-NX ou avec POM-PVM. Cette valeur coïncide avec la bande passante maximale pouvant être atteinte en utilisant directement les primitives NX sur la machine de l'Irisa dans sa configuration actuelle (système OSF/1 1.0.4, patch 2.5.1).

7 Conclusion

La POM permet au programmeur d'une application répartie de s'affranchir totalement de toute dépendance vis à vis d'une architecture donnée ou d'un système d'exploitation particulier. Les services de communication offerts sont des services de base, qui présentent l'avantage de pouvoir être mis en œuvre aisément et de manière très efficace sur la plupart des machines parallèles actuelles. La POM est donc particulièrement adaptée au développement d'applications pour lesquelles les performances sont primordiales. D'autre part, les services d'observation qui y sont intégrés élargissent son champ d'application, puisqu'ils permettent la génération, la collecte et l'exploitation de traces d'exécutions réparties et incluent des mécanismes d'estampillage et de calcul du temps global. La POM constitue ainsi un moyen privilégié d'interfacer une application répartie avec les outils développés dans le cadre du projet inter-PRC Trace (Mesr/Cnrs).

À ce jour, la POM a été portée sur des plates-formes parallèles aussi diverses que la machine Intel Paragon XP/S et un réseau de stations de travail. Nous avons ainsi pu vérifier sa réelle portabilité, et elle fait à présent partie intégrante des divers environnements de programmation pour machines massivement parallèles à mémoire distribuée développés dans l'équipe Pampa de l'Irisa : l'environnement Pandore [AND 95], compilateur-paralléliseur semi-automatique pour langage de type HPF [HPF 93], l'environnement de programmation parallèle par objets EPEE [JEZ 93], et Echidna, un environnement d'exécution parallèle pour le langage Estelle [JAR 92]. Ces trois environnements ont pour point commun le fait que les communications n'y sont pas gérées explicitement par le programmeur d'application, mais générées automatiquement par un compilateur (cas de Pandore et d'Echidna) ou encapsulées au sein d'une bibliothèque (cas de EPEE). Dans ce contexte d'utilisation, la POM se révèle parfaitement adaptée à nos besoins.

Dans l'avenir, nous projetons de porter la POM sur de nouvelles plates-formes, parmi lesquelles la machine Cray T3D et l'IBM SP1. Nous aimerions également tester ses performances sur un ensemble de stations de travail connectées par un réseau FDDI (une mise en œuvre sur ATM est en cours). Nous envisageons par ailleurs de définir une version étendue de la POM dotée de mécanismes d'entrées/sorties parallèles et autorisant l'utilisation de processus légers sur chaque nœud de la machine virtuelle.

Remerciements

La POM est le fruit d'un travail collectif. Nous tenons à remercier les membres et stagiaires du projet Pampa de l'Irisa qui ont participé à sa conception et sa réalisation.

Références

[AND 95] ANDRÉ F., LE FUR M., MAHÉO Y. et PAZAT J.-L. The Pandore Data Parallel Compiler and its Portable Runtime. In *HPCN Europe'95*, LNCS, Springer Verlag, Milan, Italy, May 1995.

[BAR 95] BAREAU C., CAILLAUD B., JARD C. et THORAVAL R. Measuring Concurrency of Regular Distributed Computations. In *TAPSOFT'95, Theory and Practice of Software Development*, LNCS, Springer Verlag, à paraître, Aarhus, May 1995. (Également disponible en rapport de recherche INRIA n^o 2394, octobre 1994).

[BEG 93] BEGUELIN A., GEIST G.A., JIANG W., MANCHEK R., MOORE K. et SUNDERAM S. *The PVM Project*. Technical Report, Oak Ridge National Laboratory, February 1993.

[BUT 92] BUTLER R. et LUSK E. *User's Guide to the P4 Programming System*. Technical Report TM-ANL-92/17, Argonne National Laboratory, 1992.

[CAL 94] CALKIN R., HEMPEL R., HOPPE H.-S. et WYPIOR P. Portable Programming with the PARMACS Message-Passing Library. *Parallel Computing*, 1994.

[DIE 93] DIEHL C., JARD C. et RAMPON J.-X. Reachability Analysis on Distributed Executions. In J.P. Jouannaud M.C. Gaudel, editor, *Proc. TAPSOFT'93 LNCS 668*, pages 629–643, Springer–Verlag, Orsay, Paris, April 1993.

[COR 94] CORBETT P. et al. *MPI-IO: A Parallel File I/O Interface for MPI*. Research Report 19841 (87784), IBM T.J. Watson Research Center and NASA Ames Research Center, November 1994.

[FID 91] FIDGE C.J. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[HPF 93] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Technical Report Version 1.0, Rice University, May 1993.

[GEI 92] GEIST G.A., HEATH M.T., PEYTON B.W. et WORLEY P.H. *A User's Guide to PICL - A Portable Instrumented Communication Library*. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, May 1992.

[HAD 88] HADDAD Y. Performance dans les systèmes répartis : des outils pour les mesures. Thèse de Doctorat, Univ. Paris-Sud, Centre Orsay Paris, septembre 1988.

[JAR 94] JARD C., JÉRON T., JOURDAN G.-V. et RAMPON J.-X. A General Approach to Trace-Checking in Distributed Computing Systems. In 14^{th} *International Conference on Distributed Computing Systems, Poznan, Pologne*, pages 396–403, IEEE Computer Society Press, June 1994.

[JAR 92] JARD C. et JÉZÉQUEL J.-M. ECHIDNA, an Estelle-Compiler to Prototype Protocols on Distributed Computers. *Concurrency Practice and Experience*, 4(5):377–397, August 1992.

[JAR 95] JARD C. et JOURDAN G.-V. Incremental Transitive Dependency Tracking in Distributed Computations. *Parallel Processing Letters*, à paraître, 1995. (Également disponible en rapport de recherche IRISA n^o 851, août 1994).

[JEZ 89] JÉZÉQUEL J.-M. Building a Global Time on Parallel Machines. In *Proc. of the* 3^{rd} *International Workshop on Distributed Algorithms*, pages 136–147, LNCS, Springer Verlag, 1989.

[JEZ 93] JÉZÉQUEL J.-M. EPEE: an Eiffel Environment to Program Distributed Memory Parallel Computers. *Journal of Object Oriented Programming*, 6(2):48–54, May 1993.

[MAI 95] MAILLET E. et TRON C. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 1995. À paraître.

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