OVIPAR : Une Plate-forme d'Initiation au Parallélisme sous UNIX

Antonio ALMEIDA
Pierre MOUKELI

Institut Africain d'Informatique
(IAI) B.P. 2263 Libreville
GABON

Tel. (241) 72 00 05
Fax. (241) 72 00 11

Résumé. OVIPAR (Ordinateur Virtuel Parallèle) est un outil d'aide à l'apprentissage du parallélisme développé sous UNIX en langage C. Il s'adresse particulièrement aux étudiants et aux chercheurs désireux de découvrir ce domaine de l'informatique. OVIPAR se présente comme une bibliothèque de fonctions à intégrer dans un programme C classique. Ces fonctions mettent en oeuvre les instructions CSP. OVIPAR est en quelque sorte un modèle réduit de PVM, avec des ambitions très modestes.

Mots-clés. Outils de programmation, parallélisme.

Summary. Ovipar is a training platform for learning parallelism. It is based on Unix an C. It is well suited for students and researchers interested in this domain of computer science. Ovipar is designed as a library of functions to be inserted in usual C programs. These functions borrow the CSP model. Ovipar is a kind of reduced PVM, with modest ambitions.

Key words. Programming tools, parallelism.

Introduction.

OVIPAR (Ordinateur Virtuel Parallèle) est une plate-forme de programmation destinée à l'initiation au parallélisme [Alm2] [Mou]. Cette plate-forme est réalisée en langage C, sous le système d'exploitation UNIX. OVIPAR a été mis en oeuvre suivant le modèle MIMD à mémoire distribuée. L'architecture est basée sur des processeurs à quatre liens inspirés des transputers de INMOS [INM].

OVIPAR offre à l'utilisateur la possibilité de créer un ordinateur virtuel multiprocesseur sur lequel il peut exécuter des programmes parallèles écrits en langage C suivant le modèle CSP. Les processus communiquent à travers des canaux

internes à chaque noeud ou reliant des noeuds entre eux. Les processus d'un même noeud disposent d'une zone de mémoire partagée.

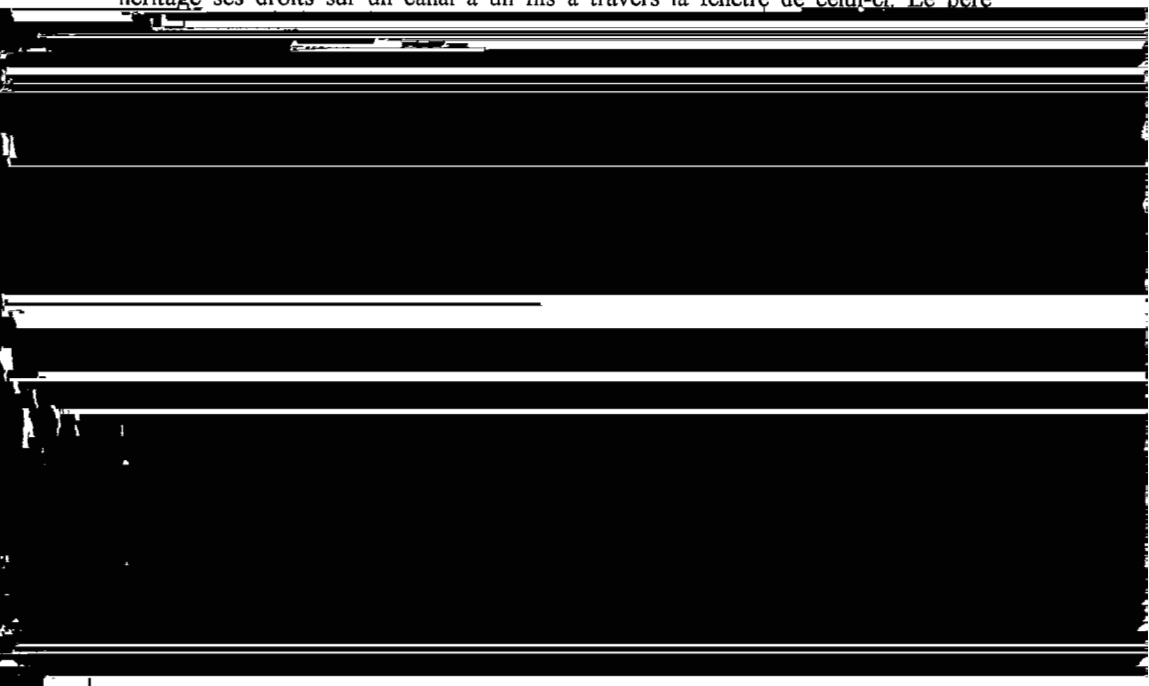
OVIPAR a des ambitions plus modestes que PVM [Che] [Gei] et MPI [Wal]. D'abord il ne fonctionne que sur une seule machine UNIX. Il n'offre qu'une bibliothèque de fonctions mettant en oeuvre les principales instructions CSP, alors que

Le nombre de noeuds d'OVIPAR et le nombre de processus par noeud ne sont limités que par les capacités de la machine UNIX utilisée (e.g. nombre de processus qu'un utilisateur peut créer).

1.3. PROCESSUS OVIPAR.

Un processus OVIPAR est un processus UNIX disposant d'une fenêtre de communication avec d'autres processus. Cette fenêtre est constituée de canaux de communication. Elle peut être constituée de liens, ou de canaux internes au noeud. La fenêtre d'un processus est toujours créée par le processus père avec des canaux ou des liens que ce dernier a hérité ou de canaux créés par le père lui même pour ses fils.

Un processus peut créer autant de canaux qu'il veut. Ces canaux sont internes au noeud ... ne peuvent être utilisés que par les fils du processus et ses descendants qui en ont hérités. A tout moment un canal ne peut être partagé que par deux processus. Comme le père se bloque en attendant la fin de ses fils, le père peut transmettre par héritage ses droits sur un canal à un fils à travers la fenêtre de celui-ci. Le père



Les communications sont synchrones et binaires. Si le nombre n1 d'octets envoyés par un processus et différent du nombre n2 qu'attend le destinataire, alors le nombre d'octets effectivement transférés sera égal au minimum de n1 et n2. Les fonctions de communication retournent ce minimum.

Un processus peut créer plusieurs fils. Il spécifie pour chacun d'eux une fenêtre de communication. Le père se bloque après la création des fils, jusqu'à la terminaison du dernier fils. A tout moment un processus peut connaître l'identité de processeur sur lequel il s'exécute.

1.4. PROGRAMME OVIPAR.

Un programme OVIPAR est un programme C faisant référence aux fonctions de la

Le programme parallèle doit être compilé en faisant référence à la librairie d'OVIPAR. Comme tout environnement basé sur l'appel de fonctions, OVIPAR n'effectue aucun contrôle syntaxique et sémantique, et se limite à ceux qu'offre le langage C. C'est pourquoi le programmeur doit se conformer à la syntaxe du manuel d'utilisation d'OVIPAR.

1.5. UTILISATION D'OVIPAR.

Le programme OVIPAR doit être exécuté comme toute commande UNIX. OVIPAR démarre quand la fonction *ovipar* est exécutée et s'arrête automatiquement dès que tous les processus qu'il supporte sont terminés.

Mettant en oeuvre des techniques de programmation avancées, le programme OVIPAR ne doit pas être arrêté de manière inopinée. Dans sa version actuelle OVIPAR n'offre pas de possibilités de terminaison de programmes autres que celles de C et UNIX. Cela peut sembler être une limitation sévère, surtout dans le contexte d'apprentissage dans lequel il est censé être exploité. En fait dans un tel contexte, des applications lourdes qui peuvent exiger de tels soins ne sont pas conçues. OVIPAR n'est pas un outil de développement.

2. ELÉMENTS DE MISE EN OEUVRE D'OVIPAR.

OVIPAR a été mis en oeuvre en s'appuyant sur les mécanismes de programmation avancée du système UNIX version 5 (i.e. gestion des processus, mémoire partagée, sémaphores), et sur les techniques offertes par le langage C (fonctions à arguments variables, vecteurs de paramètres du style *argv* de *main*). Les choix de programmation ont été faits pour tirer au mieux partie de ces mécanismes [Alm2].

2.1. LES PROCESSEURS.

Un processeur est mis en oeuvre sous la forme d'un processus dont les descendants sont les processus du noeud. Ce processus initialise une ensemble de variables qui seront héritées par duplication de l'espace des données (e.g. numéro de processeur). L'arrêt de la machine correspond à la fin de tous ces processus.

2.2. GESTION DES PROCESSUS.

Les processus OVIPAR sont créés avec la fonction UNIX *fork*. Le processus ainsi créé exécute comme première action une fonction chargée d'initialiser la structure de données du processus. Pour créer plusieurs processus, il suffit de procéder par itération de ce principe.

Le processus père attend la fin des processus fils avec une itération sur la fonction *wait*; sachant que cette fonction retourne une valeur négative quand le processus n'a plus de fils actif. Chaque processus fils doit donc se terminer par la fonction *exit*.

L'appel à cette fonction est pris en charge par OVIPAR. La gestion d'un processus suit alors le schéma suivant :

```
if (fork() == 0) {
        init_process () ;
        call_process_function (argc, argv) ;
        exit (0) ;
        }
```

2.3. GESTION DES CANAUX ET LIENS.

Les canaux et les liens sont construits avec des segments de mémoire partagée. Le bout de chaque canal est marqué avec le numéro du processus qui en hérite à travers sa fenêtre. Ce marquage est realisé par la fonction *init_process* appelée juste après la fonction *fork* (voir §3.2). A chaque canal est associé un ensemble de sémaphores qui permettent d'assurer l'exclusion mutuelle. Les fonctions de communication exploitent ces informations pour garantir l'intégrité des canaux et liens. Pour reprendre ses droits sur les canaux et liens, le processus père restaure son propre marquage à la fin des fils.

2.4. MÉMOIRE PARTAGÉE.

Tout comme les canaux, la mémoire partagée sur chaque noeud est mise en oeuvre

sur la base de segments de mémoire partagée. OVIPAR ne garantit pas l'accès en

term of couples of links to be connected.

int (fct*)() ; Entry point of the parallel program. This function is

expected to load the nodes. It must be written in the format of C main function.

int argc ; Argument count of fct.

char* argv ; Argument vector of fct.

3.2. CRÉATION D'UN DESCRIPTEUR DE PROCESSUS : INITPROC.

Cette fonction permet de créer un descripteur de processus, c'est à dire une structure

machine parallèle. Le programme ayant appelé cette fonction se bloque jusqu'à la fin des processus qui ont été créés.

int **placedpar** (sz, pdesctab)

int	sz ;	Size of table pdesctab.
csp_t	pdesctab[] ;	Table in which each item is a process descriptor. These descriptors are created with the function *initproc*.

La fonction ***placedparl*** est une version de *placedpar* qui prend en paramètre une liste de descripteurs de processus terminée par 0.

3.4. CRÉATION DE PROCESSUS SUR UN NOEUD: PAR.

Cette fonction prend en paramètre une table de descripteurs de processus créés avec la fonction *initproc*, et crée les processus sur le noeud à partir duquel elle a été appelée. Le processus ayant appelé cette fonction se bloque jusqu'à la fin des processus qui ont été créés.

int **par** (sz, pdesctab)

int	sz ;	Size of table pdesctab.
csp_t	pdesctab[] ;	Table in which each item is a process descriptor. These descriptors are created with the function initpdesc.

La fonction ***parl*** est une version de *par* qui prend en paramètre une liste de descripteurs de processus terminée par 0.

3.5. EMISSION D'UN MESSAGE: SEND.

Cette fonction permet au processus qui l'appelle d'envoyer un message à un autre processus. La fonction prend en paramètre le canal par lequel le message doit être envoyé, ainsi que l'adresse du message et sa taille. Le processus qui l'a appelé reste bloqué jusqu'à ce que le message soit transmis. La fonction retourne le minimum entre le nombre d'octets envoyés et le nombre d'octets attendus par le destinataire.

int **send** (chn, msg, msgsz)

chn_t	chn ;	Channel descriptor.
char*	msg ;	Message storage buffer.
int	msgsz ;	Message size.

3.6. RÉCEPTION D'UN MESSAGE: RCEV.

Cette fonction permet au processus qui l'appelle de recevoir un message émis par un autre processus. La fonction prend en paramètre le canal par lequel le message sera

reçu, ainsi que l'adresse du buffer où il sera stocké, et le nombre d'octets attendus. Le processus qui l'a appelé reste bloqué jusqu'à ce que le message soit reçu. La fonction retourne le minimum entre le nombre d'octets envoyés et le nombre d'octets attendus.

int **recv** (chn, msg, msgsz)

chn_t chn ;	Channel descriptor.
char* msg ;	Storage buffer.
int msgsz ;	Expected message size.

3.7. GESTION DE CANAUX INTERNES : CHANGET, CHANDEL.

La première de ces fonction permet de créer un canal sur un noeud, dont la taille donnée correspond au nombre maximal d'octets qui pourront transiter par ce canal. Si le nombre d'octets émis ou le nombre attendus dépassent cette taille, alors le nombre d'octets effectivement transmis sera égal à cette taille. La deuxième fonction permet de supprimer un canal.

chn_t **changet** (sz) ;

int sz ;	Maximum number of bytes which can transit by the channel.

int **chandel** (chn) ;

chn_t chn ;	Descriptor of the channel to be deleted.

3.8. CHOIX GARDÉ D'UNE COMMUNICATION : ALT.

La fonction *alt* permet d'attendre la présence d'une communication sur une liste de canaux. Si plusieurs communications sont présentes, la fonction choisit aléatoirement l'une d'elle et retourne le numéro dans la liste du canal associé. Sinon elle attend l'arrivée de la première communication et retourne le numéro du canal associé. Cette fonction ne reçoit pas le message. Le nombre de canaux passés en paramètre est variable. Le dernier paramètre doit être 0.

int **alt** (chn, ...)

chn_t chn ;	The first of a list of channels ending by 0.

3.9. ACCÈS À LA MÉMOIRE PARTAGÉE D'UN NOEUD : VMEM.

Cette fonction permet à un processus de lire ou écrire une valeur dans la mémoire partagée du noeud dans lequel il s'exécute. L'accès se fait à partir d'un emplacement donné relativement au début de la mémoire partagée. Cette mémoire a une taille d'un kilo octets.

int **vmem** (buf, sz, os, cmd)

char* buf ;	Buffer containing the message to be stored in the shared memory, or buffer in which the message extracted from the shared memory will be stored.

int	sz ;	Message size.
int	offset ;	Offset from the beginning of the shared memory.
int	cmd ;	0 to read and 1 to write.

CONCLUSION.

L'outil qui vient d'être présenté est disponible à l'IAI et est en phase de diffusion. Cet outil n'est utilisable que dans un cadre strictement académique. Son principal avantage est le nombre de fonctions réduit qui en facilite l'apprentissage.

Nous avons dit qu'OVIPAR doit être arrêté par les procédures habituelles de terminaisons de programmes C. Une version améliorée permettrait de terminer proprement OVIPAR à la suite d'un incident. Il s'agira en particulier de terminer les processus pendants, et de libérer les sémaphores et les segments de mémoire partagée. La mise en oeuvre d'une telle procédure ne pose pas de problème particulier en exploitant par exemple un fichier de trace système.

Une autre amélioration possible est de rendre la machine OVIPAR permanente. Cela permettrait d'exécuter plusieurs programmes consécutifs sans arrêter OVIPAR. Là aussi, il n'y a pas de problème majeur, étant donné que les techniques de communication entre processus et avec le shell d'UNIX sont bien maîtrisées.

Références.

[Alm1] Almeida Antonio
OVIPAR, Ordinateur Vituel Parallèle sous UNIX
Manuel d'utilisation, 1995, IAI BP 2263 Libreville GABON.
[Alm2] Almeida Antonio
OVIPAR, Conception et Mise en Oeuvre d'un Ordinateur Vituel Parallèle sous UNIX

African Regional Workshop on Parallel Processing and its Applications
31 July - 11 August 1995, Yaoundé Cameroon

[Wal] Walker D.W.
The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers
Parallel Computing, Vol. 20, 1994, PP. 657-673

ANNEXE: EXEMPLES DE PROGRAMMES OVIPAR.

Les programmes décrits dans cette section sont disponibles dans la bibliothèque des exemples OVIPAR et doivent être compilés avec la commande suivante (dans laquelle *demo.c* désigne le nom du programme) :

cc -o demo.c -lovipar

A.1. PROGRAMMATION D'UN ANNEAU.

Cet exemple montre comment programmer un anneau. Il crée cinq processeurs reliés entre eux par les liens est ouest. Sur chaque processeur s'exécute un processus dont le rôle est de recevoir un jeton du voisin ouest, de le modifier, et de le transmettre au voisin est. Le premier processus sert de frontal.

```
void front (int argc, char* argv[], chn_t chnv[]) {          /* executed by the front node */
        int     token ;
                        /* Read the token */
        printf (« Input an integer : ») ;
        scanf (« %d », &token) ;
                        /* Send the token to the front node */
        send (chn_east, (char*)token, sizeof (int)) ;
                        /* expect the token coming back */
        recv (chn_west, (char*)token, sizeof (int)) ;
        printf («New value of the token is : %d\n», token) ;
        }

void transit (int argc, char* argv[], chn_t chnv[]) {        /* executed by the intermediate
nodes */
        int     token ;
                        /* expect the token from west channel */
        recv (chn_west, (char*)token, sizeof (int)) ;
        token++ ;
                        /* Send the token to the east channel */
        send (chn_east, (char*)token, sizeof (int)) ;
        }
int tokenring () {                  /* Create process descriptors, and specify node entries */
        csp_t   p[5] ;
        initcomp (&p[0], front, 1, CHN_WEST + CHN_EAST, NULL, 0, NULL) ;
        initcomp(&p[1], transit, 2, CHN_WEST + CHN_EAST, NULL, 0, NULL) ;
        initcomp(&p[2], transit, 3, CHN_WEST + CHN_EAST, NULL, 0, NULL) ;
        initcomp(&p[3], transit, 4, CHN_WEST + CHN_EAST, NULL, 0, NULL)
```

```
initproc (&p[4], transit, 5, CHN_WEST + CHN_EAST, NULL, 0, NULL) ;
        return placedparl (5, p) ;
        }

int main () {                   /* Host entry point : computer creation and configuration */
                                /* E refers to EAST link and W refers to WEST link */
        ovipar (5, «1E2W 2E3W 3E4W 4E5W 5E1W», tokenring, 0 NULL)) ;
        }
```

A.2. PROGRAMMATION D'UN PRODUCTEUR CONSOMMATEUR.

L'objectif est de montrer comment deux processus peuvent accéder à la mémoire paratgée de leur noeud, et comment utiliser la fonction *par*. Cet exemple crée deux processus auxquels sont associés les fonctions *père* et *arbitre*. Le père crée deux fils: *lire*, *écrire*, l'un récupère dans la mémoire partagée une valeur que l'autre a incrite. Les deux fils sont synchronisés par le processus *arbitre*.

```
void arbitre (int argc, char* argv[], chn_t chnv[]) {
        int     c ;
        recv (chn_north, &c, 1) ;
        send (chn_south, &c, 1) ;
        }
void lire (int argc, char* argv[], chn_t chnv[])) {
        int     a = 0 ;
        int     c ;
        recv (chn_ south. &c. 1) ;
        vpmem (&a, sizeof(int). 0, VP_READ) ;
        printf (« Valeur reçue: %d\n ». a) ;
        }
void ecrire (int argc, char* argv[], chn_t chnv[]) {
        int     a ;
        printf (« Entrer une valeur entière: ») ;
        scanf (« %d ». &a) ;
        vpmem (&a, sizeof(int), 0, VP_WRITE) ;
        send ((chn_north, «*», 1) ;
        }
void père (int argc, char* argv[], chn_t chnv[])  {
        csp_t p[2] ;
        initproc (&p[0], écrire, CHN_NORTH, 0, NULL) ;
        initproc (&p[1], lire,   CHN_SOUTH. 0, NULL) ;
        par (2. p) ;
        }
void chargement () {
        csp_t p[2] ;
        initcomp (&p[0], arbitre, 1, CHN_NORTH + CHN_SOUTH, 0, NULL) ;
        initcomp (&p[1], pere,    2, CHN_NORTH + CHN_SOUTH, 0, NULL) ;
        placedpar (2, p) :
        }
main () {
        if (ovipar(2, « 1N2N 1S2S, chargement. 0. NULL))
                printf (« Erreur ovipar n° %d\n ». vp_errno) ;
        }
```