Programmation parallèle

Michel Quercia, avril 1999
quercia@cal.enst.fr

1. Aspects théoriques du calcul parallèle

1.1. Efficacité

Une machine parallèle est une machine comportant plusieurs processeurs travaillant en même temps. On classe les machines parallèles en différentes catégories selon que ces processeurs exécutent ou non les mêmes instructions au même moment, travaillent sur les mêmes données ou sur des données différentes, ont accès à la même mémoire ou ont des mémoires indépendantes.

machine SIMD : *Single Instruction stream, Multiple Data stream.* Tous les processeurs exécutent une même instruction, chacun sur des données différentes. Une telle machine est aussi appelée machine vectorielle.

machine MISD : *Multiple Instruction stream, Single Data stream.* Chaque processeur exécute une instruction différente, mais le flot des données est commun à tous les processeurs. C'est l'organisation en pipeline.

machine MIMD-SM : *Multiple Instruction stream, Multiple Data stream, Shared Memory.* Les processeurs exécutent des instructions indépendantes sur des données indépendantes, ils ont accès à la même mémoire physique. Il s'agit des machines multiprocesseurs. Selon qu'une cellule de la mémoire partagée peut être accédée simultanément en lecture ou en écriture par un seul ou plusieurs processeurs, on parle de machines EREW (Exclusive Read/Exclusive Write), CREW (Concurrent Read/Exclusive Write) ou CRCW (Concurrent Read/Concurrent Write) ce dernier modèle étant en fait théorique avec des conventions diverses sur l'effet d'une écriture multiple.

machine MIMD-DM : *Multiple Instruction stream, Multiple Data stream, Distributed Memory.* Les instructions les données et les mémoires sont indépendantes. C'est le type de machine parallèle le plus facile à constituer, il suffit de se procurer plusieurs ordinateurs et de les relier entre eux par un réseau local.

En théorie, si un programme s'exécute sur une machine séquentielle en un temps T_1, on peut espérer exécuter une version parallèle de ce programme sur p processeurs en un temps $T_p \approx T_1/p$. En pratique les choses sont moins simples. D'une part l'utilisation de plusieurs machines induit des délais de communication (en première approximation proportionnels à la taille des données échangées entre les machines) et d'autre part le programme peut comporter des blocs non parallélisables où chaque instruction dépend des résultats des instructions précédentes et donc ne peut pas être exécutée avant la terminaison de ces instructions précédentes (sauf à changer d'algorithme). On appelle *facteur d'accélération* le rapport $f = T_1/T_p$, *travail* le produit $w = pT_p$ (somme des temps d'utilisation de l'ensemble des machines) et *efficacité* la quantité $e = T_1/pT_p$. Ces définitions sont ambiguës car elles sont censées se rapporter à l'exécution d'un même algorithme sur une ou p machines, mais le fait de paralléliser les calculs conduit automatiquement à modifier l'algorithme séquentiel initial.

Loi d'Amdahl : On suppose que chaque instruction s'exécute en un temps constant indépendant de la machine, que le programme exécuté comporte une proportion $\alpha > 0$ d'instructions séquentielles et $1-\alpha$ d'instructions parallélisables. Alors :

$$\begin{aligned}
T_p &\geqslant \alpha T_1 + \frac{1-\alpha}{p} T_1 \\
f &\leqslant \frac{p}{1+(p-1)\alpha} \xrightarrow[p\to\infty]{} \frac{1}{\alpha} \\
e &\leqslant \frac{1}{1+(p-1)\alpha} \xrightarrow[p\to\infty]{} 0
\end{aligned}$$

En particulier l'efficacité est toujours inférieure à 1 et décroît avec le nombre de processeurs utilisés et le facteur d'accélération ne dépassera pas la limite $1/\alpha$ quelque soit le nombre de processeurs utilisés. Les hypothèses de la loi d'Amdahl ne sont pas toujours vérifiées, la vitesse d'exécution d'une instruction peut varier dans des proportions notables sur une même machine selon que le code de l'instruction et les données référencées résident dans une mémoire à accès rapide (cache) ou dans une mémoire plus lente, voire en mémoire secondaire. On peut donc, dans certains cas observer une efficacité supérieure à 1.

Efficacité en travail : Lorsqu'un algorithme dépend d'un paramètre n représentant la taille des données, les quantités T_1, p et T_p dépendent aussi de n. On dit qu'une implémentation de l'algorithme sur $p(n)$ processeurs est *efficace en travail* si $w(n) = O(T_1(n))$. Grossièrement cela signifie que les $p(n)$ processeurs sont occupés la pendant plupart du temps de calcul. Si l'implémentation parallèle n'est pas efficace en travail, elle peut parfois l'être rendue en réduisant le nombre de processeurs. Considérons par exemple le problème de l'addition de n entiers : une implémentation séquentielle consiste à accumuler les termes par une simple boucle ce qui donne un temps de calcul séquentiel $T_1(n) = O(n)$. En utilisant n 1 processeurs disposés en arbre binaire on peut calculer la somme des entiers en $O(\log n)$ unités de temps d'où un travail $w(n) = O(n \log n)$. L'implémentation n'est pas efficace en travail et ceci est dû au fait qu'une proportion de plus en plus faible des processeurs est utilisée au cours du calcul (la moitié puis le quart, le huitième, etc). Puisqu'on sous-utilise les processeurs dans cette implémentation, il est envisageable d'en utiliser moins tout en conservant asymptotiquement le même temps de calcul. Dans le cas présent on obtient une implémentation parallèle efficace en travail en divisant les n éléments à additionner en $n{=}\log n$ groupes de taille $\log n$; dans une première étape on calcule les $n{=}\log n$ sommes des groupes par addition séquentielle ce qui peut être réalisé en $O(\log n)$ unités de temps avec $n{=}\log n$ processeurs, puis on agrège ces sommes en une seule en $O(\log(n{=}\log n)) = O(\log n)$ unités de temps à l'aide d'un arbre binaire à $n{=}\log n$ processeurs. Le temps de calcul reste un $O(\log n)$ et le travail devient un $O(n)$.

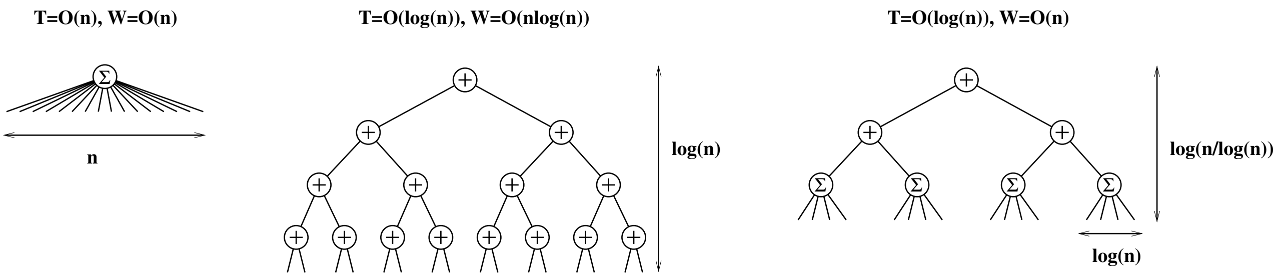

figure 1 : addition parallèle

Répartition du travail : Lors de l'implémentation d'un calcul sur une machine parallèle on peut décider à l'avance quelle part du calcul est attribuée à chaque processeur, on parle alors de *répartition statique du travail*, ou attribuer en cours d'exécution les tâches aux processeurs au fur et à mesure qu'ils sont disponibles (*répartition dynamique du travail*). Dans le second cas on tient à jour une liste des tâches à exécuter et chaque processeur libre consulte cette liste pour en retirer une tâche qu'il prend en charge et éventuellement pour en insérer de nouvelles. L'avantage de la répartition dynamique est qu'elle permet d'adapter automatiquement la distribution des travaux en fonction du nombre de processeurs présents au moment de l'exécution, on peut même ajouter de nouveaux processeurs en cours de calcul ou retirer des processeurs en replaçant les tâches non effectuées dans la liste. De plus, si la division du calcul complet en tâches est suffisamment fine, on obtient automatiquement un équilibrage du travail effectué par chaque processeur. En contrepartie il faut gérer les accès concurrents à la liste des tâches. Une méthode simple consiste à affecter la gestion de cette liste à un processeur appelé *maître*, les autres processeurs, *esclaves*, communiquent avec le maître pour obtenir un nouveau travail et éventuellement lui transmettre leurs résultats.

1.2. Quelques algorithmes parallèles

Préfixe parallèle

Le problème du *préfixe parallèle* consiste à calculer en parallèle sur n processeurs la liste des sommes partielles $(a_1; (a_1 + a_2); \dots; (a_1 + \dots + a_n))$ associée à une liste de nombres $(a_1; \dots; a_n)$. L'implémentation suivante peut être utilisée avec des listes de type quelconque et une opération binaire associative à la place de l'addition. On «dispose» les processeurs en arbre binaire comme pour une addition parallèle et on procède à deux phases.

– Remontée : chaque processeur calcule et transmet à son père la somme des nombres figurant dans son sous-arbre.

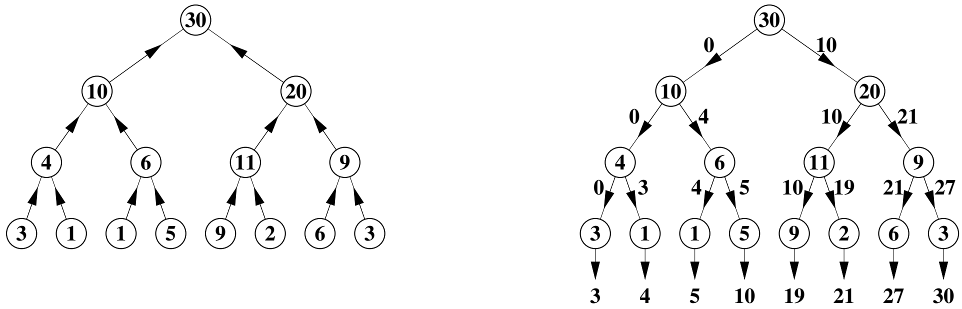

figure 2 : préfixe parallèle

– Descente : chaque processeur reçoit de son père la somme de tous les nombres situés à gauche de son sous-arbre, il transmet cette somme à son fils gauche et cette somme plus la somme du sous-arbre gauche à son fils droit.

Le temps de calcul est $O(\ln n)$ d'où un travail en $O(n \ln n)$. Comme pour l'addition parallèle, on peut améliorer l'efficacité en travail en regroupant les nombres en sous-listes de taille $\log n$.

L'algorithme du préfixe parallèle peut être utilisé pour contracter un vecteur v à partir d'un masque m indiquant pour chaque élément $v(i)$ s'il doit être conservé ($m(i) = 1$) ou supprimé ($m(i) = 0$) : dans un premier temps on calcule le vecteur m' des préfixes de m ce qui donne pour chaque élément $v(i)$ tel que $m(i) = 1$ la position où il faut placer cet élément dans le vecteur final, ensuite on procède à ces recopies en parallèle.

Une autre application, citée par Gengler, consiste à calculer pour une image bitmap la matrice des distances donnant pour chaque pixel la distance de Manhattan à la zone blanche. Soit M une matrice bicolore $n \times n$. On calcule dans un premier temps pour chaque pixel (i,j) la distance au pixel blanc le plus proche vers la gauche (ou la distance au bord gauche de l'image) par une opération de préfixe parallèle ligne par ligne où chaque processeur transmet à son père lors de la phase de remontée le nombre de pixels noirs consécutifs adjacents au bord droit du sous arbre controlé par ce processeur et l'information booléenne comme quoi tous les pixels du sous-arbre sont noirs ou non. Dans la phase de descente un processeur transmet à chacun de ses fils le nombre de pixels noirs consécutifs situés à gauche du sous-arbre controlé par le fils et l'information booléenne comme quoi il existe un pixel blanc vers la gauche. Formellement il s'agit d'un préfixe parallèle avec l'opération associative :

$$\begin{pmatrix} x \\ b \end{pmatrix} \star \begin{pmatrix} x' \\ b' \end{pmatrix} = \begin{pmatrix} \texttt{if } b' \texttt{ then } x + x' \texttt{ else } x' \\ b \,\&\, b' \end{pmatrix}$$

où x, x' désignent des nombres de pixels et b, b' des booléens. A la fin de cette première étape chaque pixel « connaît » le nombre de pixels noirs consécutifs situés à sa gauche avant le premier pixel blanc. On procède alors à une deuxième phase similaire pour calculer pour chaque pixel la distance au prochain pixel blanc vers la droite ce qui fournit l'information $\delta(i,j)$ donnant la distance éventuellement infinie au pixel blanc le plus proche lors d'un déplacement horizontal.

Troisième phase : on détermine pour chaque pixel la distance à la zone blanche en effectuant un déplacement vers le haut puis horizontal. Cette distance s'exprime par :

$$d(i,j) = \min\{i - k + \delta(k,j),\ 1 \leqslant k \leqslant i\} = i + \min\{-k + \delta(k,j),\ 1 \leqslant k \leqslant i\}$$

où les minimums sont encore calculés par préfixe parallèle en colonne cette fois (min est une opération associative). On détermine de même la distance à la zone blanche en effectuant un déplacement vers le bas d'abord puis horizontalement et on retient le minimum des deux distances obtenues. Maintenant on connaît la distance à la zone blanche pour un déplacement vertical puis horizontal.

On détermine de même la distance à la zone blanche pour un déplacement horizontal d'abord puis vertical et c'est terminé. L'opération complète peut donc être conduite en temps $O(\ln n)$ avec n^2 processeurs, ou éventuellement $n^2/\ln(n)$ processeurs en regroupant les pixels $\ln(n)$ par $\ln(n)$ pour assurer une efficacité en travail…

Tri parallèle

On peut trier n nombres sur une machine séquentielle en $O(n \ln n)$ unités de temps, il est donc raisonnable d'espérer trier ces nombres sur une machine comportant n processeurs en un temps $O(\ln n)$. La littérature mentionne l'existence de tels algorithmes en indiquant seulement qu'il s'agit d'une opération compliquée. Par contre, on trouve facilement des algorithmes de tri parallèle avec n processeurs et un temps de calcul en $O(\ln^2 n)$.

Par exemple le tri par fusion : pour fusionner deux vecteurs triés a et b de longueur n dans un vecteur c de longueur $2n$ il suffit, comme pour la contraction d'un vecteur, de calculer pour chaque élément de a et b à quelle position dans c il faut le placer ce qui revient pour un élément $a(i)$ à déterminer combien d'éléments dans c doivent le précéder. Le nombre de prédécesseurs de $a(i)$ dans a est $i-1$ connu en temps constant, le nombre de prédécesseurs de $a(i)$ dans b s'obtient en cherchant $a(i)$ dans b par dichotomie donc en temps $O(\ln n)$. Ceci permet donc d'effectuer une fusion parallèle avec $2n$ processeurs en $O(\ln n)$ unités de temps, d'où un tri en $O(\ln^2 n)$ unités de temps. Cet algorithme exige que la mémoire soit partagée entre les processeurs et supporte des lectures simultanées, donc une machine CREW.

Une autre technique de fusion pouvant être implémentée sur une machine EREW est la fusion *pair-impair* due à Batcher : on fusionne dans a' les éléments de rang pair de a et b et dans b' les éléments de rang impair de a et b. On a alors :

$$c(2k+1) = \min(a'(k+1); b'(k))$$
$$c(2k+2) = \max(a'(k+1); b'(k))$$

où les indices commencent à 0, $c(0) = a'(0)$ et $c(2n) = b'(n-1)$. Pour prouver la validité de ces formules il suffit de remarquer que si $a'(k+1) = a(2p+2)$ alors $b(2k-2p-2) \leqslant a(2p+2) \leqslant b(2k-2p)$ donc les prédécesseurs de $a'(k)$ dans c sont $a(0)\dots a(2p+1)$, $b(0)\dots b(2k-2p-2)$, et peut-être $b(2k-2p-1)$ ce qui prouve que $a'(k+1)$ doit aller en position $2k+1$ ou $2k+2$. Un raisonnement analogue prouve que $b'(k)$ doit aussi aller dans l'une de ces positions. Le calcul de c à partir de a' et b' se fait donc en temps constant avec $2n$ processeurs, les calculs de a' et b' peuvent être conduits de la même manière récursivement d'où un temps de fusion en $O(\ln n)$ et un temps de tri en $O(\ln^2 n)$.

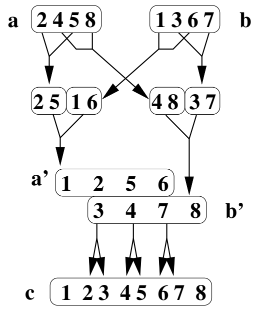

figure 3 : fusion pair-impair

Après dérécursification on obtient le code un peu cryptique suivant :

```
(* fichier batcher.ml : tri-fusion de Batcher *)

(* echange les elements distants de v dans les sous-tableaux de longueur 2u *)
(* u et v sont des puissances de 2 et v <= u *)
(* tous ces echanges peuvent être effectues en parallele *)
let echanges(t,u,v) =
  for b=0 to vect_length(t)-1 do
    if (u = v or b mod (2*v) < v) & b mod (2*u) >= v then begin
      let x = t.(b-v) and y = t.(b) in t.(b-v) <- min x y; t.(b) <- max x y
    end
  done;;

let tri(t) =
  let u = ref(1) in
  while !u < vect_length(t) do
    (* ici tous les sous-tableaux jointifs de longueur !u sont tries *)
    (* on les fusionne deux par deux *)
    let v = ref(!u) in
    while !v > 0 do echanges(t,!u,!v); v := !v/2 done;
    u := !u*2
  done;;
```

où la boucle sur b dans la fonction `echanges` est exécutable en parallèle sur n = `vect_length(t)` processeurs. Knuth donne une version légèrement différente (vol. 3, algorithme 5.2.2-M) où les sous-tableaux fusionnés sont entrelacés au lieu d'être jointifs.

2. Implémentation du parallélisme avec des machines Unix

2.1. Processus, activités, ordonnancement

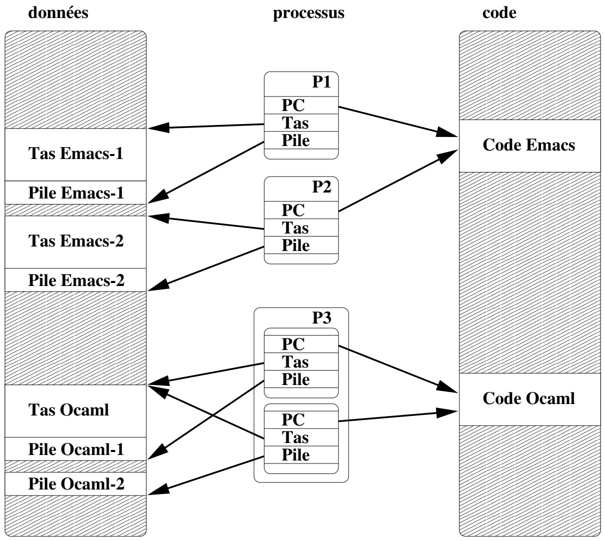

figure 4 : deux processus Emacs, un processus Ocaml multithread

Un programme est une liste d'instructions machine. Pour exécuter un programme sur un processeur on a besoin d'un *contexte d'exécution* qui regroupe les ressources utilisées par le programme :

– mémoire de code ;
– mémoire globale (tas) ;
– mémoire locale (pile) ;
– registres du processeur, notamment PC, le pointeur vers l'instruction courante ;
– canaux d'entrées/sorties ;
– données administratives.

On appelle *processus* l'association d'un programme et d'un contexte d'exécution. Un même programme peut à un instant donné être exécuté par plusieurs processus sur des données différentes, chaque processus disposant de son propre contexte d'exécution. En général le code exécutable (texte du programme) est *partagé* entre ces processus, ce qui signifie qu'il n'est chargé qu'une seule fois en mémoire, les différents processus accédant tous à cette mémoire de code commune (ceci suppose que le code n'est pas modifiable en cours d'exécution, et en pratique le système d'exploitation verrouille la mémoire de code en écriture). Un processus peut être constitué de plusieurs *activités* (*threads* en anglais) : chaque activité dispose d'une mémoire locale et de registres processeurs privés, le reste du contexte d'exécution en particulier la mémoire globale et les canaux d'entrées/sorties étant partagés aussi bien en lecture qu'en écriture avec les autres activités.

Le fonctionnement multitâche est obtenu par partage du temps processeur entre les processus concurrents. A un instant donné un seul processus est *actif* (devant le processeur) et son contexte d'exécution évolue en fonction des instructions du programme. Les autres processus peuvent être *activables* c'est à dire prêts à être exécutés ou *bloqués* par une opération d'entrée/sortie exécutée par un circuit spécialisé indépendant du processeur ou par une instruction d'attente. Périodiquement le système d'exploitation interrompt le processus actif, compare sa priorité courante avec celles des autres processus activables, et active un des processus de priorité maximale. Cette opération, appelée *ordonnancement* ou *échantillonage*, est également déclenchée lorsque le processus actif lance une opération d'entrée/sortie et passe dans l'état bloqué ou lorsqu'il se termine. Sur une machine multiprocesseurs l'ordonnancement se déroule de façon similaire à chaque fois qu'un processeur devient disponible.

En ce qui concerne la division d'un processus en activités, le partage du temps entre les différentes activités peut être réalisé soit directement par le système d'exploitation (threads de niveau noyau) ou par une bibliothèque spécialisée incluse dans le code exécutable du processus (threads de niveau utilisateur). Dans le premier cas, sur une machine multiprocesseurs, il est possible que deux activités d'un même processus s'exécutent en même temps

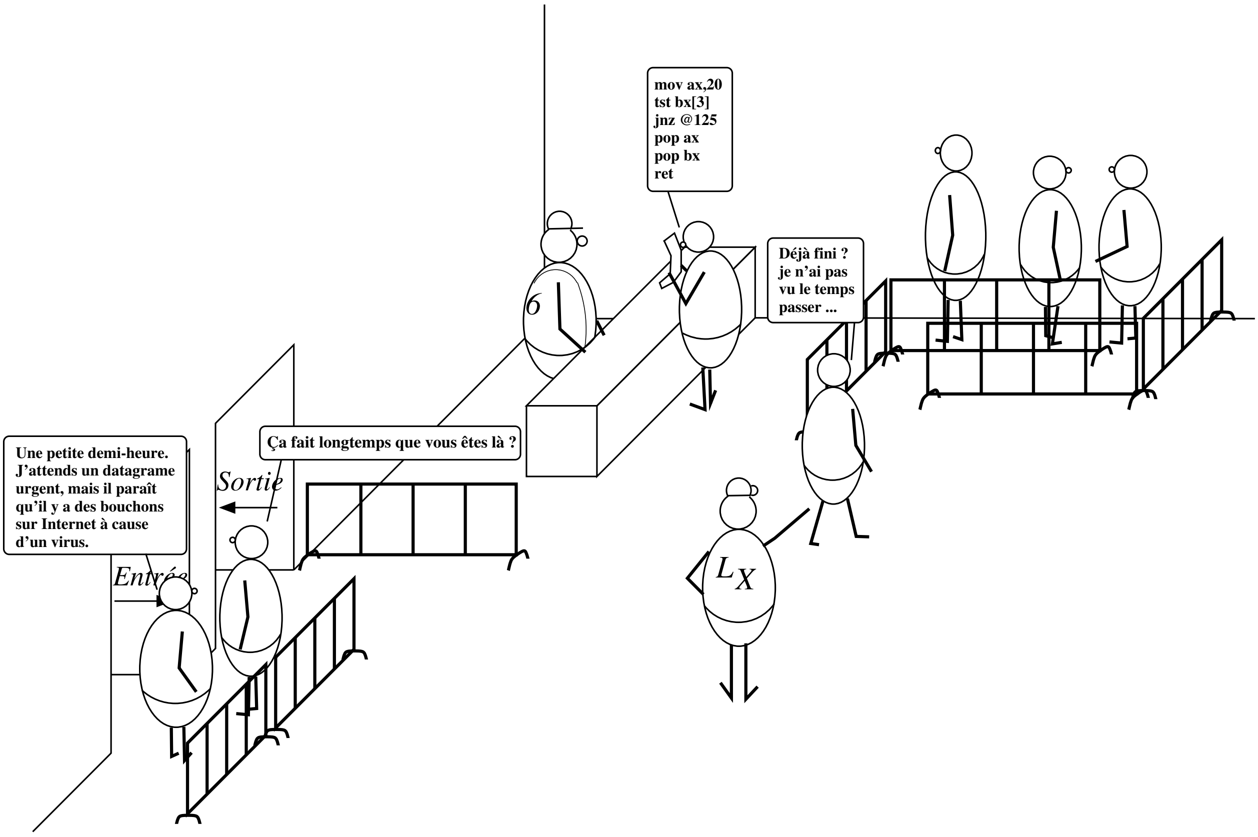

figure 5 : ordonnancement

sur des processeurs différents ; dans le second cas il ne peut y avoir qu'une seule activité en cours d'exécution à un instant donné, même s'il y a plusieurs processeurs disponibles.

2.2. Mécanismes de communication

Normalement les processus sont indépendants les uns des autres et l'exécution d'un processus n'interfère pas avec celle des autres si ce n'est par la captation périodique du processeur, mais il est possible d'organiser implicitement ou explicitement des échanges de données entre processus concurrents. Les systèmes Unix offrent plusieurs mécanismes de communication :

Mémoire partagée : une zone mémoire est rendue accessible en lecture/écriture à plusieurs processus simultanément. Pour des activités associées à un même processus il s'agit de toute la mémoire globale, pour des processus indépendants il s'agit d'une partie seulement de cette mémoire globale. Comme chaque processus ou activité est susceptible de modifier la mémoire partagée, il faut généralement implémenter des mécanismes de verrouillage total ou partiel de cette mémoire lors d'une opération de type lecture-modification-écriture de façon à ce que le contenu de la mémoire partagée reste cohérent, ainsi que des mécanismes d'avertissement des modifications effectuées.

Une ressource (zone mémoire, fichier, ...) partagée entre plusieurs processus ou activités peut être protégée contre des accès «simultanés» par utilisation d'un *verrou d'exclusion mutuelle* (mutex). Il s'agit d'une structure de données gérée par le noyau et constituée d'un entier indiquant si le verrou est libre (valeur = 1) ou occupé (valeur = 0) et d'une liste d'attente de processus. Lorsqu'un processus demande le verrou, si celui-ci est libre alors le noyau le lui accorde et le processus continue normalement son exécution tandis que si le verrou est déjà occupé alors le processus demandeur est bloqué et mis dans la liste d'attente. Lorsque le verrou sera libéré l'un des processus de la liste d'attente sera réactivé et obtiendra le verrou à son tour. On appelle *section critique* une portion de code comprise entre une instruction de verrouillage et l'instruction de déverrouillage associée. Une section critique n'est exécutable que par un seul processus ou une seule activité à la fois parmi les processus ou activités partageant le verrou ; de même si plusieurs sections critiques sont associées au même verrou alors une seule de ces sections peut être exécutée à un moment donné et par un seul processus ou une seule activité. Un *sémaphore* est un verrou généralisé (plus exactement un verrou est un sémaphore particulier) pour lequel l'entier associé peut recevoir des valeurs supérieures à 1. Les opérations autorisées sur le sémaphore sont l'incrémentation (opération `P`) et la décrémentation (opération `V`), la décrémentation pouvant conduire au blocage si la valeur courante du sémaphore est nulle. On peut utiliser un sémaphore pour délimiter une section de code ne pouvant être exécutée que par `k` au plus processus, `k` étant la valeur initiale du sémaphore.

Tuyaux : un tuyau (*pipe* en anglais) est une structure FIFO gérée directement par le noyau Unix. Des données (en fait des suites d'octets) peuvent être écrites dans un tuyau et elles peuvent être lues depuis le tuyau à l'aide des fonctions standard d'entrées/sorties (`read/write` en C, `input_xxx/output_xxx` en Caml). L'avantage d'un tuyau par rapport à un fichier disque est que les données qui y circulent résident en mémoire donc les accès à un tuyau sont rapides, l'inconvénient est que ces données ne sont pas adressables, on ne peut pas forcer la lecture ou l'écriture à une position quelconque. En pratique le noyau associe un tampon circulaire de stockage au tuyau, une opération d'écriture se résume à une copie dans ce tampon et une opération de lecture à une copie depuis ce tampon. Sous Linux la taille d'un tampon (définie dans `/usr/include/linux/limits.h`) est de 4 Ko ; si un processus tente d'écrire plus de données que la capacité courante du tampon ne le permet il passe dans l'état bloqué jusqu'à ce qu'il y ait assez de place ; si un processus tente de lire des données depuis un tampon vide il passe dans l'état bloqué jusqu'à ce qu'il y ait au moins un octet disponible.

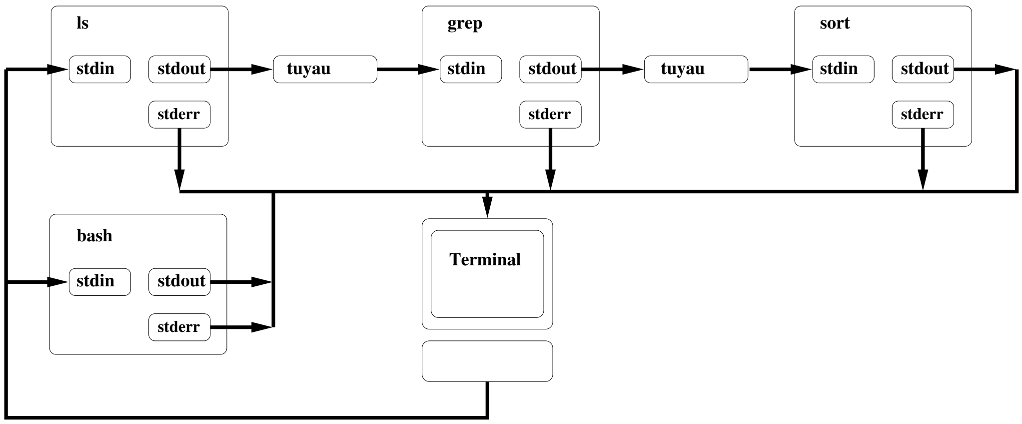

figure 6 : tuyaux

La distraction favorite du bidouilleur Unix consiste à réaliser des traitements complexes par enchaînement en ligne de commande de commandes élémentaires connectées entre elles par tuyaux :

```
bash$ ls -lR ~/Mail | grep "$(date +'%b[ ]*%-d')" | sort +9
```

liste par ordre alphabétique tous les fichiers situés sous le répertoire `~/Mail` et créés ce jour (avec une interface graphique et quelques clics de souris bien sentis on pourrait peut-être parvenir au même résultat, mais ce serait bien moins amusant). Concrètement, l'interpréteur de commandes lance trois processus, `ls`, `grep` et `sort`, crée deux tuyaux qu'il « intercale » entre le canal d'entrée standard d'un processus et le canal de sortie standard du processus précédent puis laisse l'ensemble tourner. L'entrée standard du premier processus est reliée à l'entrée standard de l'interpréteur de commandes, en principe le terminal de contrôle, la sortie standard du dernier processus est reliée à la sortie standard de l'interpréteur et tous les canaux d'erreur sont reliés à celui de l'interpréteur. Il y a un risque de mauvais aiguillage des caractères entrés au terminal qui peuvent être lus indifféremment par `bash` et `ls`, en fait `bash` ne tente aucune lecture sur son entrée standard tant que la chaîne de commandes n'est pas terminée. Rien n'empêche par contre les messages d'erreur des trois processus et les messages de sortie de `sort` de se mélanger à l'écran. Enfin, les tuyaux sont désalloués automatiquement par le noyau à la terminaison des trois processus.

Prises : une prise est une paire de tuyaux permettant une communication bidirectionnelle entre deux processus qui s'exécutent sur deux machines reliées par réseau, distinctes ou non. Chacun des processus doit créer une prise sur sa machine, et l'un d'entre eux doit demander la connexion de sa prise avec celle de l'autre processus. Une fois la connexion établie les deux processus peuvent se transmettre des données par lecture/écriture ordinaire sur les canaux associés à la connexion. Le processus qui demande la connexion est appelé *processus client*, celui qui attend la connexion est appelé *processus serveur*. Les prises sont identifiées sur le réseau par le numéro IP de la machine sur laquelle la prise est créée et un numéro interne à cette machine appelé *numéro de port*. Le serveur doit choisir un numéro de port et l'affecter à sa prise après création, le client doit connaître les références de la prise du serveur (numéro IP, numéro de port) pour demander la connexion. C'est le noyau de la machine cliente qui choisit automatiquement le numéro de port de la prise cliente. Comme pour les tuyaux ordinaires, des tampons sont alloués par les noyaux pour stocker les données en attente de transmission ou qui ont déjà été transmises mais pour lesquelles la machine distante n'a pas encore envoyé d'accusé de réception, et donc un processus est susceptible de passer à l'état bloqué s'il demande une lecture ou une écriture sur sa prise.

Remarque : La description ci-dessus est volontairement simplifiée, elle correspond au type de prises le plus couramment employées à savoir les prises TCP/IP (transmit control protocol/Internet protocol), il existe un protocole UDP (user datagram protocol) qui ne tamponne pas les données ni ne vérifie leur bon acheminement et il existe aussi des prises dont l'adresse est de type Unix (chemin d'accès dans l'arborescence Unix d'une machine) et qui ne sont référençables que sur la machine où elles sont créées.

2.3. Support du parallélisme en Caml

La bibliothèque `unix` de Caml-light/Ocaml fournit des primitives permettant de créer des processus, des tuyaux, des prises, et de connecter tout cela à sa guise. Elle ne permet pas le partage de mémoire avec un autre processus.

Ocaml dispose en plus d'une bibliothèque `threads` permettant de créer plusieurs activités à l'intérieur d'un processus, ce qui fournit une forme de partage mémoire. Cependant, compte tenu des contraintes du gestionnaire de mémoire intégré à Ocaml, une seule activité peut être exécutée à la fois donc on ne bénéficie d'aucune accélération en divisant un processus en activités concurrentes, même sur une machine multiprocesseurs. Ceci concerne la version actuelle, 2.02, peut-être un jour y aura-t-il une version équipée d'un gestionnaire mémoire réentrant, permettant ainsi une exécution réellement concurrente des activités ? Pour l'instant, le seul intérêt des activités est de faciliter la gestion des entrées/sorties *asynchrones* (les données arrivent ou partent à des moments imprévisibles) puisqu'une opération d'entrée/sortie bloque l'activité qui l'a déclenchée et seulement celle-là donc les autres activités peuvent continuer leur exécution.

Enfin il existe une version expérimentale d'un langage dérivé de Ocaml, Join-caml (`http://join.inria.fr/`), qui intègre le parallélisme directement dans les constructions syntaxiques du langage.

3. Communication par tuyaux

3.1. Interaction Caml – Maple

Le programme suivant met en œuvre une communication bidirectionnelle entre deux processus concurrents : un processus Caml lance en concurrence un processus Maple, lui soumet des calculs et récupère les résultats.

```
(* fichier lance-maple.ml : communication avec un processus concurrent *)

#open "unix";;
let (chin,chout) = open_process "maple";;

(* envoie une commande et affiche la reponse *)
let commande(s) =

  let cmd = s ^ "lprint(`OK`);\n" in
  print_string ("-----------\nenvoi: " ^ cmd);
  output_string chout cmd; flush chout;

  print_string "recu :\n";
  while
    let l = input_line chin in
    print_string(l); print_newline();
    l <> "OK"
  do () done;;

begin
  (* saute la baniere *)
  commande "";
```

```
  (* envoie quelques calculs *)
  commande "factor(x^20-1);";
  commande "lprint(factors(x^20-1));";
  commande "plotsetup(x11);plot(sin);"; sleep 15;

  (* termine *)
  let _ = close_process (chin,chout) in ()
end;;
```

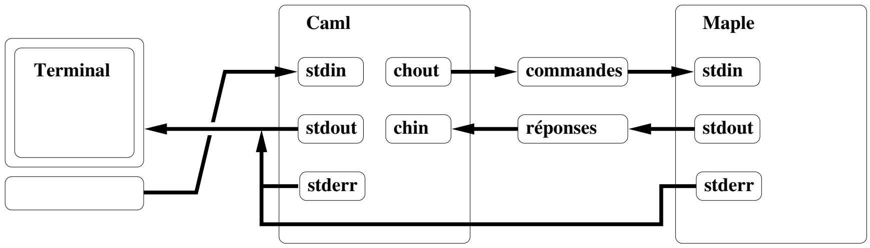

figure 7 : interaction Caml — Maple

exécution :

```
bash$ lance-maple
-----------
envoi: lprint(`OK`);
recu :
    |\^/|     Maple V Release 5.1 (Linux Single User)
._|\|   |/|_. Copyright (c) 1981-1998 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> lprint(`OK`);
OK
-----------
envoi: factor(x^20-1);lprint(`OK`);
recu :
> factor(x^20-1);lprint(`OK`);
          4    3    2                            2    3    4        2
(x - 1) (x  + x  + x  + x + 1) (x + 1) (1 - x + x  - x  + x ) (1 + x )

      8    6    4    2
    (x  - x  + x  - x  + 1)

OK
-----------
envoi: lprint(factors(x^20-1));lprint(`OK`);
recu :
> lprint(factors(x^20-1));lprint(`OK`);
[1, [[x+1, 1], [1+x^2, 1], [x^4+x^3+x^2+x+1, 1], [x^8-x^6+x^4-x^2+1, 1], [1-x+x
^2-x^3+x^4, 1], [x-1, 1]]]
OK
-----------
envoi: plotsetup(x11);plot(sin);lprint(`OK`);
recu :
> plotsetup(x11);plot(sin);lprint(`OK`);
OK
```

`open_process` est une fonction de la bibliothèque `unix` de Caml-light qui crée un processus concurrent exécutant la commande passée en argument et deux tuyaux reliés d'un côté à l'entrée et à la sortie standards du processus créé et de l'autre aux canaux de lecture et d'écriture `chin` et `chout` retournés en résultat.

Pour envoyer une commande à Maple, il suffit d'envoyer le texte de la commande sur `chout` puis de vider le tampon associé par l'instruction `flush`. Il s'agit ici du tampon que Caml met systématiquement en place lors de l'ouverture d'un canal d'entrée/sortie et non du tampon associé au tuyau ; sans cette instruction `flush` les textes des commandes s'accumuleraient dans le tampon jusqu'à ce que celui-ci soit rempli et ce n'est qu'à ce moment que les données partiraient effectivement dans le tuyau.

Pour recevoir la réponse fournie par Maple, il suffit en théorie de lire les caractères provenant du canal `chin`. En pratique il faut en plus implémenter un mécanisme permettant de détecter la fin d'une réponse : l'appel `input_line chin` retourne une ligne complète envoyée par Maple s'il y en a une présente dans le tuyau, sinon il provoque le blocage du processus Caml jusqu'à ce qu'une ligne complète puisse être lue. Si l'on connaît à l'avance le nombre de lignes que Maple fournira en réponse il suffit d'appeler `input_line` ce nombre de fois ; dans le cas présent on ne peut pas le connaître donc on force Maple à signaler la fin de sa réponse par une ligne supplémentaire contenant le texte `OK` (les tentatives de détection de l'invite > par `input_char` échouent de façon inexplicable).

3.2. Transmission de données structurées

Dans l'exemple précédent les données circulent entre les deux processus `lance-maple` et `maple` sous forme textuelle ce qui impose d'implémenter une fonction de codage des données internes en chaînes de caractères et une fonction de décodage des chaînes reçues, aussi bien du côté Caml que du côté Maple. Du côté Maple, au moins dans les cas simples, il n'y a rien de spécial à faire puisqu'un analyseur d'entrée et un imprimeur sont intégrés au programme. Du côté Caml, il faut extraire des lignes reçues celles qui correspondent effectivement à la réponse que l'on attend et ⟪désenjoliver⟫ le résultat pour en analyser la structure ... Le mode de communication textuel présente deux avantages et un inconvénient :

– on peut facilement tracer les communications, voire simuler à la main l'un des deux processus pour analyser le comportement de l'autre ;
– on peut utiliser tel quel tout programme prévu pour fonctionner avec un terminal caractère ;
– il faut décoder les réponses fournies par le programme piloté.

Lorsqu'on veut faire communiquer deux processus Caml on peut facilement transmettre des valeurs structurées (vecteurs, listes, arbres,...) sans programmer explicitement des fonctions de conversion :

`output_value chout v` encode la valeur `v` et transmet sur le canal `chout` la chaîne binaire correspondante ;
`input_value chin` lit sur le canal `chin` une chaîne binaire et reconstruit la valeur associée.

La seule restriction au couple `input_value/output_value` est que les valeurs à transmettre ne contiennent ni valeurs fonctionnelles ni données externes telles que descripteurs de fichiers, `nums` de la bibliothèque `bignum`, images de la bibliothèque `graphics` entre autres. Par ailleurs il n'y a pas d'information de typage transmise avec la valeur encodée, donc le type d'une valeur reçue par `input_value` ne peut pas être contrôlé ni à la compilation ni à l'exécution.

Voici un exemple de communication interprocessus utilisant `input_value/output_value` : le but du programme est de trier une liste d'entiers par tri rapide. Après segmentation de la liste le processus maître lance un processus esclave chargé de trier un des segments pendant que le maître trie l'autre puis les résultats sont agrégés dans un arbre binaire de recherche. Le maître et son esclave procèdent chacun à une deuxième segmentation et confient l'un des segments produits à un nouveau processus esclave et ainsi de suite.

```
(* fichier bqsort.ml : transmission de donnees structurees *)

#open "printf";;
type arbre = Vide | Noeud of arbre * int * arbre;;

(* segmentation *)
let rec decoupe p = function
  | [] -> [],[]
  | h::t -> let (u,v) = decoupe p t in if h < p then (h::u,v) else (u,h::v)
;;

(* tri recursif *)
let rec tri = function
  | [] -> Vide
  | p::suite -> match decoupe p suite with
      | [],v -> Noeud(Vide, p, tri v)
      | u,[] -> Noeud(tri u, p, Vide)
```

```
        | u,v  -> let (chin,chout) = unix__open_process "bqsort -s" in
                  output_value chout v; flush chout;
                  let u' = tri(u) in
                  let v' = input_value(chin) in
                  let _  = unix__close_process (chin,chout) in
                  Noeud(u', p, v')
;;

(* affichage ...  non liste *)
(* programme principal *)
begin
  match sys__command_line.(1) with
    | "-s" -> (* esclave *)
        let l = input_value stdin     in
        let a = tri_trace "esclave" l in
        output_value stdout a;
        flush stdout
    | ch -> (* ma^tre *)
        let n = int_of_string(ch)    in
        let l = random_list(n)       in
        let a = tri_trace "ma^tre" l in ()
end;;
```

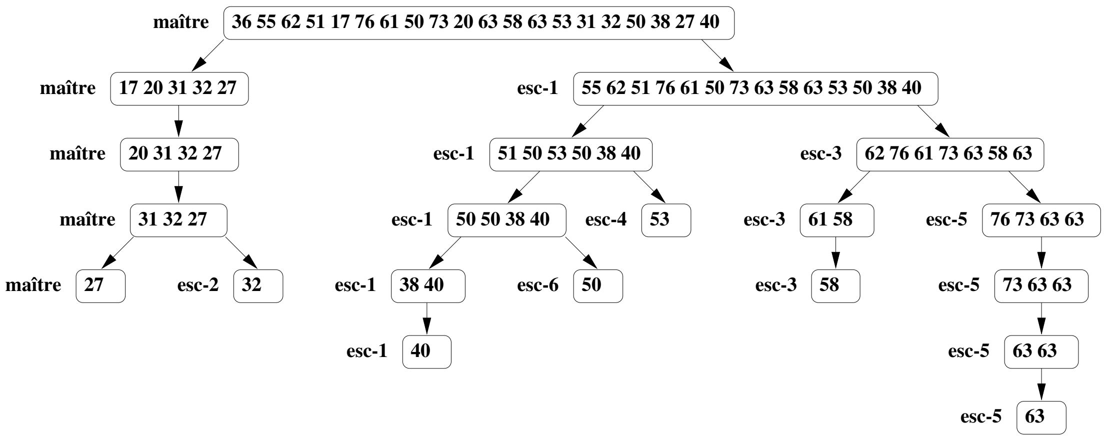

figure 8 : tri parallèle

exécution (les numéros de la première colonne sont les numéros d'identification des processus) :

```
bash$ bqsort 20
ma^tre(11684) [36 55 62 51 17 76 61 50 73 20 63 58 63 53 31 32 50 38 27 40]
esclave(11685) [55 62 51 76 61 50 73 63 58 63 53 50 38 40]
esclave(11686) [32]
esclave(11686) 32
esclave(11687) [62 76 61 73 63 58 63]
esclave(11690) [76 73 63 63]
esclave(11690) (((. 63 63) 73 .) 76 .)
esclave(11687) ((58 61 .) 62 (((. 63 63) 73 .) 76 .))
esclave(11688) [53]
esclave(11688) 53
esclave(11689) [50]
esclave(11689) 50
esclave(11685) ((((. 38 40) 50 50) 51 53) 55 ((58 61 .) 62 (((. 63 63) 73 .) 76 .)))
ma^tre(11684) ((. 17 (. 20 (27 31 32))) 36 ((((. 38 40) 50 50) 51 53) 55 ((58 61 .) 62 (((. 63 63) 73 .) 76 .))))
```

Sur une machine ayant suffisamment de processeurs disponibles tous les processus à un même niveau de profondeur s'exécuteront simultanément et l'on peut espérer un temps de tri en O (n) avec de l'ordre de n=2 processeurs impliqués dans le tri ; mais la charge de travail est inégalement répartie entre les processeurs puisque chaque esclave travaille environ deux fois moins que son maître (ce qui est ethiquement discutable).

3.3. Interblocage

Lorsqu'un processus lance une opération d'entrée/sortie il passe à l'état bloqué jusqu'à ce que l'opération soit terminée (en Caml, compte tenu des tampons associés aux canaux d'entrée/sortie, un blocage n'a lieu que si un tampon de lecture est vide ou incomplet, un tampon d'écriture est plein, ou si l'on provoque l'envoi d'un tampon d'écriture par `flush`). Ceci peut conduire à des situations où deux processus communiquant se bloquent mutuellement :

– interblocage par manque de données : chaque processus attend que l'autre lui envoie quelque chose ;
– interblocage par excès de données : chaque processus envoie à l'autre un bloc de données trop gros pour tenir dans les tampons.

Considérons par exemple un processus Caml en communication bidirectionnelle avec un processus Maple :

a) Le processus Caml envoie à Maple une commande non terminée par `;` ou `:` puis se met en attente de la réponse de Maple. Maple n'ayant pas reçu de caractère de terminaison attend la suite. Il y a interblocage par manque de données.

b) Le processus Caml envoie une commande à Maple correctement terminée mais « oublie » de forcer l'envoi par `flush` ; ensuite il se met en attente d'une réponse. La commande ne part pas, Maple reste en attente et il y a interblocage par manque de données.

c) Le processus Caml envoie d'affilée une grosse série de commandes à Maple, avec tous les points-virgules nécessaires et effectue un `flush` avant de se mettre en attente d'une réponse. Maple reçoit et exécute les premières commandes, puis génère une réponse volumineuse avant de réceptionner les commandes suivantes. Suivant la taille de la série de commandes et des premières réponses il se peut que le processus Caml soit bloqué au milieu de son envoi et le processus Maple soit bloqué au milieu d'une réponse. Dans ce cas il y a interblocage par excès de données.

Les cas **a** et **b** relèvent d'une erreur de programmation, il n'y a rien à dire. Par contre le cas **c** relève d'une limitation due au système (tampons insuffisants).

Mise en évidence : le programme Caml suivant lance un esclave, lui envoie une en rafale `n` blocs de données puis se met à l'écoute de l'esclave. Cet esclave se contente de retourner au maître les données qu'il reçoit. La taille d'un tampon d'entrée/sortie en Caml est de 4 Ko de même que la taille d'un tampon associé à un tuyau. les blocs de données (entier,chaîne) ont été calibrés expérimentalement pour avoir une taille de 1 Ko. A l'exécution, on observe un fonctionnement normal pour `n` = 12 et un interblocage à partir de `n` = 13.

```
(* fichier interblocage.ml : interblocage par exces de donnees *)

#open "printf";;
begin
  match sys__command_line.(1) with

    | "-s" -> begin (* esclave *)
        try while true do
        let (i,s) = input_value stdin in
        fprintf stderr ">%d " i;   flush stderr;
        output_value stdout (i,s); flush stdout
        done
      with End_of_file -> ()
      end

    | ch -> (* ma^tre *)
        let n = int_of_string(ch)   in
        let s = make_string 996 ` ` in
        let (chin,chout) = unix__open_process "interblocage -s" in
        for i = 1 to n do
          fprintf stderr "<%d " i;  flush stderr;
          output_value chout (i,s); flush chout;
        done;
        for i = 1 to n do
          let (j,_) = input_value chin in
          fprintf stderr ".%d " j;  flush stderr
        done;
```

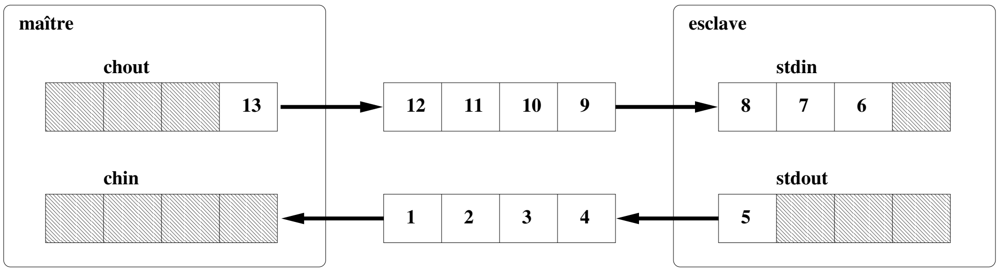

figure 9 : interblocage par excès de données

```
        fprintf stderr "\n"; flush stderr;
        let _ = unix__close_process (chin,chout) in ()
end;;

bash$ interblocage 12
<1 <2 <3 <4 <5 >1 >2 >3 >4 <6 <7 <8 <9 >5 <10 <11 <12 .1 .2 .3 .4 >6 >7 >8 >9 .5 .6 .7 .8 >10 >11 >12 .9 .10 .11 .12
bash$ interblocage 13
<1 <2 <3 <4 <5 >1 >2 >3 >4 <6 <7 <8 <9 >5 <10 <11 <12 <13 killed
```

Que faire en cas d'interblocage ? Arrêter les processus, il n'y a pas moyen de débloquer la situation en douceur. Comment éviter l'interblocage ? En communication unidirectionnelle il ne peut pas y avoir interblocage. En communication bidirectionnelle le plus simple est de n'envoyer que des blocs courts et d'attendre une réponse après chaque envoi. Avec Ocaml il est possible de faire exécuter les opérations de lecture et d'écriture par des activités séparées, le blocage d'une activité n'empêchant pas les autres de s'exécuter.

4. Communication par prises

Comme indiqué précédemment, pour établir une communication entre deux processus tournant sur des machines distinctes on lance un des processus en mode serveur (création d'une prise TCP et attente de connexion) et l'autre en mode client (création d'une prise TCP et connexion à celle du serveur). Dans l'optique d'un calcul distribué il y a deux possibilités de fonctionnement :

– les esclaves sont serveurs, le maître est client ;
– le maître est serveur, les esclaves sont clients.

L'avantage du premier mode est qu'il ressemble beaucoup au lancement de processus concurrents sur une même machine, on remplace les `open_process` par des demandes de connexion. L'avantage du second mode est que la liste des esclaves n'est pas figée, on peut à tout moment en ajouter de nouveaux.

4.1. Utilisation des prises en Caml

On crée une prise TCP par l'appel `socket PF_INET SOCK_STREAM 0`. les arguments ont la signification suivante :

– `PF_INET` : la prise recevra une adresse de type Internet (numéro IP, numéro de port). L'autre type d'adressage, `PF_UNIX`, est réservé aux communications à l'intérieur d'une seule machine.

– `SOCK_STREAM` : les données circuleront sous forme de flux. Les autres possibilités sont `SOCK_DGRAM` (datagrammes) `SOCK_RAW` (données brutes) et `SOCK_SEQPACKET` (paquets utilisés avec IPX).

– `0` : type de protocole à utiliser par le noyau. 0 représente le protocole par défaut pour le type de transfert désigné par le second argument, ici il s'agit du protocole TCP.

Le résultat de `socket` est un descripteur de fichier Unix par l'intermédiaire duquel les échanges de données se feront. Dans l'immédiat, ce descripteur n'est pas encore opérationnel, il faudra attribuer une adresse à la prise et établir une connexion avec la prise distante.

Pour définir une adresse de type Internet, on doit fournir un numéro IP et un numéro de port. Le numéro de port est un entier quelconque compris entre 0 et 65535, les numéros de 0 à 1023 étant réservés pour le super utilisateur. N'importe quel numéro de port non utilisé peut être choisi, dans les exemples qui suivent on prendra le port 1515. Les numéros IP peuvent être obtenus par l'appel `gethostbyname nom` où `nom` est une chaîne de caractères désignant la machine dont on veut le numéro IP. On peut spécifier au choix un des noms symboliques sous lesquels cette machine est connue ou directement son adresse IP, si on la connaît, sous la forme `"aa.bb.cc.dd"`. La machine courante est généralement désignée par `"localhost"`, si on veut son nom officiel on l'obtient par `gethostname()`. Le résultat de `gethostbyname` est une structure définie dans la bibliothèque `unix` comportant entre autres la liste des adresses IP de la machine désignée dans le vecteur `h_addr_list`. En général on utilise la première de ces adresses, soit `h_addr_list.(0)`. Exemples :

– `ADDR_INET((gethostbyname "localhost").h_addr_list.(0), 1515)` désigne le port 1515 sur la machine courante.
– `ADDR_INET((gethostbyname "125.192.68.2").h_addr_list.(0), 1515)` désigne le port 1515 sur la machine ayant le numéro IP 125.192.68.2.

Le client, une fois qu'il a créé sa prise, doit juste demander la connexion avec la prise du serveur par l'appel : `connect s addr` où `s` est sa prise et `addr` l'adresse de la prise du serveur. Si la connexion échoue une exception est générée. Si elle réussit le client peut envoyer et recevoir des données sur le descripteur de fichier associé à `s` par les primitives de bas niveau `read/write` de la bibliothèque `unix`, ou construire des canaux avec tampons par les fonctions `in_channel_of_descr/out_channel_of_descr` et utiliser ces canaux avec les fonctions `input_xxx/output_xxx` de la bibliothèque `io`. Voici un fragment de code type pour un client :

```
let s    = socket PF_INET SOCK_STREAM 0
and addr = (gethostbyname "nom du serveur").h_addr_list.(0);;
connect s (ADDR_INET(addr, 1515));;
let chin  = in_channel_of_descr s
and chout = out_channel_of_descr s;;

(*... travail avec chin et chout, ne pas oublier flush(chout) svp ... *)

close(s);;
```

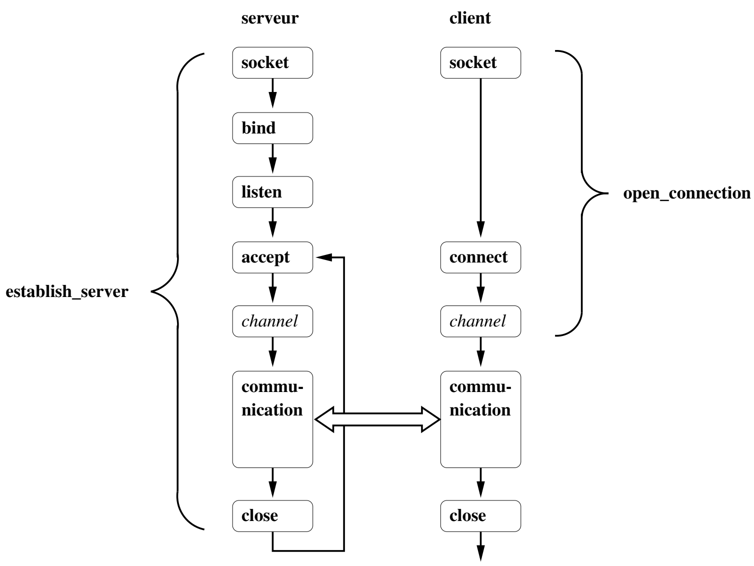

figure 10 : établissement d'une connexion

Ceci peut être simplifié avec les fonctions `open_connection` et `shutdown_connection` de la bibliothèque `unix`. `open_connection` prend en argument l'adresse de la prise serveur et s'occupe de créer la prise client, la connecter au serveur et construire les canaux `chin` et `chout` associés. `shutdown_connection` prend en argument le canal `chin` retourné par `open_connection` et met fin à la connexion. Le code précédent peut donc être remplacé par :

```
let addr = (gethostbyname "nom du serveur").h_addr_list.(0);;
```

```
let (chin,chout) = open_connection (ADDR_INET(addr, 1515));;
(*... travail avec chin et chout, ne pas oublier flush(chout) svp ... *)
shutdown_connection chin;;
```

Du côté serveur les choses sont un peu plus compliquées. Le processus serveur doit dans l'ordre : attribuer une adresse à sa prise (`bind`), activer la réception de connexions (`listen`) puis se mettre en attente d'une connexion (`accept`). `bind` prend en argument la prise et l'adresse à attribuer, `listen` la prise et le nombre maximal de connexions simultanées qui seront enregistrées, `accept` la prise seulement et retourne un descripteur de fichier associé à la connexion ainsi que l'adresse de la prise client (pour contrôle éventuel). Voici un fragment de code type pour un serveur :

```
let s    = socket PF_INET SOCK_STREAM 0
and addr = (gethostbyname "0.0.0.0").h_addr_list.(0);;
bind s (ADDR_INET(addr, 1515));;
listen s 5;;

while true do
  let (descr,addr_client) = accept s in
  let chin  = in_channel_of_descr descr
  and chout = out_channel_of_descr descr in

  (*... travail avec chin et chout, ne pas oublier flush(chout) svp ... *)

  close(descr)
done;;
```

`accept` provoque le blocage du processus serveur jusqu'à ce qu'une demande de connexion parvienne au noyau. Le processus serveur est alors débloqué et peut la traiter. Si d'autres demandes parviennent pendant ce traitement elles seront mises en attente par le noyau dans la limite indiquée par l'appel `listen`. On fournit en argument à `bind` le numéro IP `"0.0.0.0"` de façon à ce que la prise soit appelable sur n'importe quelle interface réseau de la machine serveur. `"localhost"` qui correspond normalement au numéro IP 127.0.0.1 n'est pas recommandé ici car seuls les processus tournant sur la même machine peuvent utiliser cette adresse, `gethostbyname(gethostname()).h_addr_list.(0)` pose des problèmes si la machine a plusieurs interfaces réseau car seules les demandes de connexion adressées à la première interface seraient acceptées.

Pour établir un service permanent, on dispose de deux méthodes :

– Le daemon `inetd` qui tourne normalement sur toute machine Unix peut être configuré pour accepter les connexions sur certains ports et lancer à chaque connexion un processus traitant cette connexion après avoir redirigé l'entrée et la sortie standards du processus vers la prise associée à la connexion. Pour mettre ce mécanisme en œuvre il faut disposer des droits du super utilisateur.

– la bibliothèque `unix` de Caml comporte une fonction `establish_server` qui prend en paramètre une fonction `f` de type `in_channel -> out_channel -> unit` et une adresse de prise. `establish_server` crée la prise, lui attribue l'adresse spécifiée et se met en attente de connexion. Pour chaque connexion reçue `establish_server` lance un processus indépendant qui exécute la fonction `f` avec comme arguments les canaux associés à la connexion.

4.2. Maître client, esclaves serveurs

On reprend l'exemple du tri « rapide » en parallèle vu précédemment. Ici le maître divise la liste à trier en quatre segments et transmet ces segments à quatre esclaves fonctionnant en mode serveur. Les esclaves peuvent être lancés par l'intermédiaire de la fonction `establish_server` (option `-s`) ou par `inetd` (option `-i`) auquel cas il faut au préalable avoir reconfiguré `inetd` (avec l'autorité du super utilisateur) en entrant la ligne :

```
parallel        1515/tcp                # calcul parallele
```

dans le fichier `/etc/services` et

```
parallel stream tcp nowait quercia /home/quercia/tmp/communication/sort-socket
         /home/quercia/tmp/communication/sort-socket -i
```

dans le fichier `/etc/inetd.conf` (sur une seule ligne). La modification de ces fichiers est prise en compte par `inetd` sur réception du signal HUP, lancer donc :

```
killall -HUP inetd
```

pour ce faire.

```
(* fichier sort-socket.ml : communication par prises TCP *)

#open "printf";;
#open "unix";;

(* decoupe une liste en 4 ...  non liste *)
(* machines esclaves *)
let machine = [|"Montchapet-1"; "Montchapet-2"; "Montchapet-3"; "Montchapet-4"|]
and port    = 1515;;

(* programme ma^tre *)
let maitre() =

  (* canaux associes aux esclaves *)
  let chin  = make_vect 4 io__stdin
  and chout = make_vect 4 io__stdout in

  (* contacte chaque esclave *)
  for i=0 to 3 do
    let addr = ADDR_INET((gethostbyname machine.(i)).h_addr_list.(0),port) in
    let (inch,outch) = open_connection addr in
    chin.(i)  <- inch;
    chout.(i) <- outch
  done;

  (* distribue les travaux *)
  let l = [36;55;62;51;17;76;61;50;73;20;63;58;63;53;31;32;50;38;27;40] in
  let liste = decoupe4 l in
  for i=0 to 3 do
    printf "%s <- " machine.(i); print_list liste.(i);
    output_value chout.(i) liste.(i);
    flush chout.(i)
  done;

  (* recupere et affiche les resultats *)
  let res = ref [] in
  for i=3 downto 0 do
    let l = input_value(chin.(i)) in
    printf "%s -> " machine.(i); print_list l;
    res := l @ !res;
    shutdown_connection chin.(i)
  done;
  print_list !res
;;

(* programme esclave *)
let esclave chin chout =
  let l = input_value(chin) in
  output_value chout (sort__sort (prefix <=) l);
  flush chout;
  exit(0)
;;

(* aiguillage *)
begin
  match sys__command_line.(1) with
    | "-m" -> maitre()
    | "-i" -> esclave io__stdin io__stdout
    | "-s" -> establish_server esclave (ADDR_INET(inet_addr_of_string "0.0.0.0", port))
    | _ -> failwith "option imprevue"
end;;
```

exécution :

```
bash$ sort-socket -m
Montchapet-1 <- [17]
Montchapet-2 <- [20 31 32 27 36]
Montchapet-3 <- [51 50 53 50 38 40 55]
Montchapet-4 <- [62 76 61 73 63 58 63]
Montchapet-4 -> [58 61 62 63 63 73 76]
Montchapet-3 -> [38 40 50 50 51 53 55]
Montchapet-2 -> [20 27 31 32 36]
Montchapet-1 -> [17]
[17 20 27 31 32 36 38 40 50 50 51 53 55 58 61 62 63 63 73 76]
```

4.3. Maître serveur, esclaves clients

Le programme listé ci-après recherche le premier nombre pseudo-premier fort à $e+1$ chiffres. Un nombre $n = 2^k q+1$ avec q impair est pseudo-premier fort en base a si la suite $(a^q; a^{2q}; a^{4q}; \dots; a^{2^k q})$ calculée modulo n ne contient que des 1 ou se termine par $1; 1; \dots; 1$. Lorsque n est premier alors il est pseudo-premier fort pour toute base a non multiple de n. Si n est pseudo-premier fort pour au moins 25 bases a tirées au hasard dans $[\![2; n \quad 2]\!]$ alors il est qualifié de « probablement premier » (test de Rabin-Miller, la probabilité d'une conclusion fausse est majorée par $4^{\ 25} \quad 10^{\ 15}$). Ici on se contentera de tester seulement 4 bases non aléatoires pour simplifier et ne pas attendre trop longtemps.

La recherche se déroule de la manière suivante : le processus maître sélectionne en séquence les nombres $n = 10^e + k$ qui n'ont pas de diviseur premier inférieur ou égal à 1000 puis soumet aux esclaves les tests de Rabin pour la base $a = 2$ dans un premier temps, puis pour les bases $3;5;7$ si le test en base 2 est positif. La recherche est terminée lorsqu'un entier k a passé avec succès tous ces tests et que tous les entiers $k^0 < k$ ont produit une conclusion négative.

Le processus maître est divisé en plusieurs activités : l'activité principale, après avoir initialisé les structures de données associées au calcul, se met en attente de connexions et lance pour chaque esclave qui se présente une activité concurrente associée à cet esclave qui exécute la fonction `esclave`. Cette nouvelle activité transmet à l'esclave l'exposant e puis les couples $(k; a)$ correspondant aux tests à effectuer et réceptionne les résultats.

La division du processus en activités indépendantes permet de gérer simplement le problème des attentes multiples : chaque activité est bloquée sur une condition différente, attente du résultat de l'esclave associé à l'activité ou attente d'une connexion pour l'activité principale ce qui fait que le processus complet reste dans l'état bloqué tant qu'il n'y a pas de résultat ou de connexion à traiter. Lorsqu'un résultat ou une connexion survient le processus est débloqué et exécute automatiquement le code approprié.

En contrepartie il faut gérer les conflits d'accès à la structure `data` partagée entre toutes les activités et qui résume l'état du calcul. Il est théoriquement possible que deux esclaves communiquent leurs résultats au même moment et que les instructions des deux activités associées soient entrelacées dans le temps ce qui pourrait produire une structure `data` incohérente. Pour éviter cela, on associe un verrou d'exclusion mutuelle à cette structure et tous les accès à `data` sont protégés par l'acquisition de ce verrou.

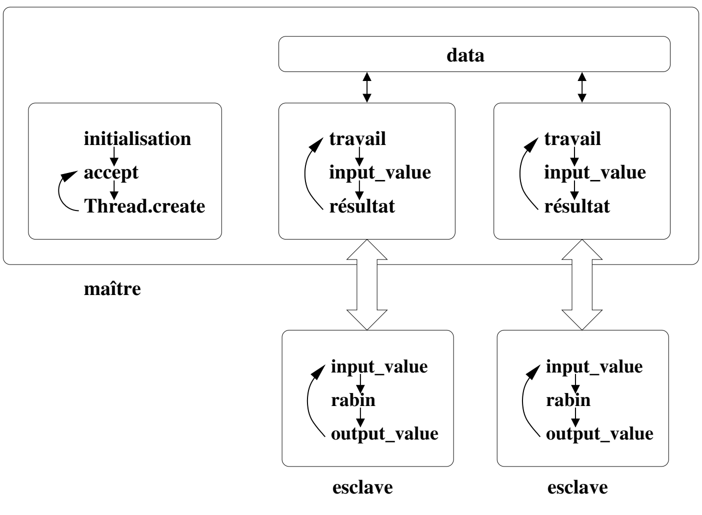

figure 11 : maître multi-thread

```
(* fichier rabin.ml : recherche parallele de nombres pseudo-premiers *)

open Printf
open Unix
open Num

module Calcul : sig (* implementation des operations arithmetiques *)
  val pprime  : int -> num               (* n -> produit des nb premiers impairs <= n   *)
  val suivant : num -> num -> int -> int  (* n p k -> k' >= k tq n+k' est premier avec p *)
  val rabin   : num -> int -> int -> bool (* n k a -> test de Rabin pour n+k en base a   *)
  val extrait : ('a -> bool) -> 'a list -> 'a list    (* elements verifiant un predicat  *)
end = struct (* ...  non liste *)

module Esclave = struct (* ...  non liste *)

module Ma^tre = struct

  type data = {
    e  : int;                          (* exposant                         *)
    n0 : num;                          (* nombre de base = 10^e            *)
    p  : num;                          (* produit des premiers impairs     *)
    bases : int list;                  (* bases de test pour Rabin-Miller  *)
    mutable knext    : int;            (* prochain k a tester              *)
    mutable en_cours : (int*int) list;  (* tests en cours                   *)
    mutable a_faire  : (int*int) list;  (* tests a faire en priorite        *)
    mutable kfound   : int;            (* plus petit k trouve              *)
    verrou : Mutex.t                   (* verrou d'acces concurrent        *)
  }

  let travail data = (*************************** donne un nouveau travail *)
    Mutex.lock data.verrou;
    let j = match data.a_faire with
      | j::t -> data.a_faire  <- t; j
      | []   -> let k = Calcul.suivant data.n0 data.p data.knext in data.knext <- k + 2; (k,2)
    in
    data.en_cours <- j :: data.en_cours;
    Mutex.unlock data.verrou;
    j

  let resultat data ((k,a) as j) res = (**************** recoit un resultat *)

    Mutex.lock data.verrou;
    if List.mem j data.en_cours then begin

      if res then begin (* test positif *)
        data.en_cours <- Calcul.extrait (fun x -> x <> j) data.en_cours;
        if a = 2 then data.a_faire <- data.a_faire @ List.map (fun b -> (k,b)) data.bases;
        if not(List.exists (fun (x,_) -> x=k) (data.a_faire @ data.en_cours))
          & (data.kfound = 0 or k < data.kfound)
        then data.kfound <- k
      end

      else begin (* test negatif *)
        data.en_cours <- Calcul.extrait (fun (x,_) -> x<>k) data.en_cours;
        data.a_faire  <- Calcul.extrait (fun (x,_) -> x<>k) data.a_faire
      end;

      if data.kfound > 0 (* a-t-on termine ? *)
       & not(List.exists (fun (x,_) -> x < data.kfound) (data.a_faire @ data.en_cours))
      then begin
        printf "k=%d\n" data.kfound;
        flush Pervasives.stdout;
        exit 0
      end

    end;
    Mutex.unlock data.verrou

  let esclave (data,chin,chout) = (****************** gestion d'un esclave *)
    output_value chout data.e; flush chout;
    while true do
```

```
      let j = travail data in
      output_value chout j; flush chout;
      resultat data j (input_value chin)
    done

  let go exposant port = (**************************** programme principal *)

    let data = {
      e  = exposant;
      n0 = (Int 10) **/ (Int exposant);
      p  = Calcul.pprime(1000);
      bases = [3;5;7];
      knext = 1;
      en_cours = [];
      a_faire  = [];
      kfound   = 0;
      verrou = Mutex.create()} in

    let s    = socket PF_INET SOCK_STREAM 0
    and addr = ADDR_INET((gethostbyname "0.0.0.0").h_addr_list.(0),port) in
    bind s addr;
    listen s 5;

    while true do
      let (descr,addr) = ThreadUnix.accept s in
      let chin  = in_channel_of_descr(descr)
      and chout = out_channel_of_descr(descr) in
      let _ = Thread.create esclave (data,chin,chout) in ()
    done

end;; (* module Ma^tre *)

(* aiguillage ...  non liste *)
```

exécution :

```
bash$ rabin -m 100 1515
k=267
```

```
bash$ rabin -s localhost 1515
(9,2) -> false
(37,2) -> false
(39,2) -> false
(61,2) -> false
(63,2) -> false
(99,2) -> false
(103,2) -> false
(111,2) -> false
(253,2) -> false
(261,2) -> false
(267,2) -> true
(267,3) -> true
```

```
bash$ rabin -s localhost 1515
(123,2) -> false
(133,2) -> false
(151,2) -> false
(159,2) -> false
(267,5) -> true
(267,7) -> true
(271,2) -> false
```

```
bash$ rabin -s localhost 1515
(177,2) -> false
(207,2) -> false
(211,2) -> false
(223,2) -> false
(229,2) -> false
(273,2) -> false
(277,2) -> false
(289,2) -> false
```

5. Bibliographie

Programmation du système Unix en Caml-Light
Xavier Leroy,
RT 147, Inria 1992

La communication sous Unix
Jean-Marie Rifflet,
Ediscience 1995

Initiation au parallélisme
Marc Gengler, Stéphane Ubéda, Frédéric Desprez,
Masson 1996

Parallel Programming
Barry Wilkinson, Michael Allen,
Prentice Hall 1999