Rapport TP1 — PVM

Bazire Houssin
Sylvain Vaglica
Stéphane Castelli

Mardi 29 Octobre 2013

Table des matières

1 Introduction

PVM est une bibliothèque C et Fortran permettant la communication entre des processus sur une machine parallèle ou un cluster de machines ; elle utilise pour cela un démon lancé sur chaque machine qui se charge du routage des messages, du contrôle des processus et des problèmes pouvant survenir. Créée en 1989, PVM a au fil des décennies perdu en popularité au profit de MPI, mais reste un moyen efficace d'apporter une solution à des problèmes dont la résolution en séquentiel mettrait trop de temps. Au travers de ce projet, il a été réalisé un système distribué de craquage de mot de passe en force brute, sur le modèle maître / esclaves, avec un processus maître créant et distribuant des tâches, et des processus esclaves les exécutant. Un des objectifs principaux du projet, en plus de nous initier à la communication inter-processus et plus particulièrement à PVM, est d'analyser et optimiser la création et la répartition des tâches afin d'obtenir les meilleures performances possibles. Il sera d'abord explicité les choix qui ont été effectués, puis ils seront analysés.

2 Représentation des données

L'ensemble des mots de passe possibles de longueur maximale p est un ensemble fini d'éléments énumérables de la manière suivante :

— a, b, c, ..., o,
— aa, ab, ac, ... ao, ba, bb, ... bo, ... oa, ob, ... oo,
— ...
— ..., w_p, le mot de longueur p ne contenant que des 'o'

Chacun est converti en un entier correspondant à sa position dans l'énumération. Il est alors possible de revenir ensuite au mot initial grâce à la fonction inverse. Grâce à cette conversion en entier, il est plus facile d'énumérer l'ensemble des possibilités car il suffit d'incrémenter un compteur puis de le convertir en chaîne de caractères. En effet, il suffit d'extraire le reste de la division par 15 (nombre de possibilités par caractère), pour obtenir la première lettre. On soustrait ce reste à l'entier, puis on divise par 15. On effectue une nouvelle division euclidienne et on déduit la lettre suivante du reste et ainsi de suite jusqu'à avoir déterminé toutes les lettres. Ceci revient donc à écrire en base 15 les mots précédents, en considérant que 'a' = 0, 'b'=1, ..., 'o'=14, tel que un mot de longueur p soit :

$$m_0 * 15^0 + m_1 * 15^1 + ... + m_p * 15^p$$

Le fait de convertir ce mot de passe en nombre est une manière simple de pouvoir répartir le travail de façon efficace : le maître ne travaille que sur des entiers et a donc plus de facilité à calculer une répartition des tâches, et l'esclave peut sans aucun problème convertir l'entier qu'il reçoit, en la chaîne de caractères initiale à tester.

2.1 Nombre de solutions

Un mot de passe peut se définir par sa complexité. Celle-ci peut être augmentée facilement par deux moyens : l'augmentation de la longueur maximale et l'augmentation du cardinal de l'alphabet utilisé. En effet, si l'on cherche un mot de passe sans le connaître, par force brute, il faut tester chacune des possibilités dans un ensemble donné, et plus il y a de mot de passes possibles dans cet ensemble, plus la tâche est longue à réaliser pour un ordinateur. Un mot de passe de longueur k sur un alphabet de n lettres fournit ainsi n^k possibilités à tester. Par conséquent, si l'on connaît l'alphabet et que l'on se fixe une longueur maximale de mot de passe à tester l, alors on a dans le pire des cas :

$$\sum_{k=0}^{l} n^k = \frac{n \cdot (n^l - 1)}{n - 1} = \mathcal{O}(n^l)$$

Dans le cas qui nous concerne, l'alphabet est les lettres minuscules comprises entre a et o, soit 15 lettres. On a donc :

$$\sum_{k=0}^{l} 15^k = \frac{15 \cdot (15^l - 1)}{14} = \mathcal{O}(15^l)$$

Même si cela limite la quantité de possibilités, le nombre de mots de passe croît de manière exponentielle avec la longueur. En conséquence, au-delà d'une certaine valeur, il devient impossible pour une machine actuelle de terminer la recherche en un temps raisonnable. Toutefois, la parallélisation du code et l'utilisation de PVM permettent de repousser cette valeur.

2.2 GMP

Comme expliqué dans la section précédente, le nombre de mots de passe possibles est exponentiel en la taille maximale du mot de passe. Or la PVM ne permet l'envoi de grands entiers que jusqu'à ceux pouvant être stockés dans une variable de type `unsigned long` (le type `unsigned long long` n'étant pas supporté par PVM), c'est à dire ayant pour valeur maximale 2^{32}. Ainsi pour des mots de passe de longueur supérieure à 8, la limite de stockage des entiers est déjà dépassée ($15^9 > 2^{32}$).

Nous avons donc décidé d'utiliser une bibliothèque spécifique pour le stockage des grands entiers. Plus précisément, la "GNU Multiple Precision Arithmetic Library" (`http://gmplib.org/`) aussi abrégée en GMP qui permet de s'affranchir des limitations de PVM et même de la bibliothèque standard C. Ainsi tous les calculs sont effectués sur des variables de type `mpz_t`. Pour les envois de données, la valeur de ces variables est extraite et écrite en base 15 dans un tableau. Ce dernier est ensuite envoyé via PVM, le processus destinataire reçoit un ensemble d'octets (dont le nombre a été communiqué précédemment), qui est ensuite retransformé en type `mpz_t` pour pouvoir effectuer les opérations.

3 L'implémentation

3.1 Le maître

Il s'occupe de démarrer les processus esclaves et de leur assigner les tâches. Le processus maître commence par envoyer à chaque esclave le mot de passe en clair, puis transforme l'ensemble des mots de passe possibles en un intervalle d'entiers (fonction bijective décrite précédemment). Chaque fois qu'un esclave demande du travail, il lui assigne un sous intervalle de l'intervalle initial, en lui donnant le début de l'intervalle et un pas (nombres d'entiers à tester). Une gestion dynamique du pas est essentielle pour équilibrer le travail des esclaves et garantir une exécution la plus rapide possible. En effet, il est primordial que l'ensemble des esclaves termine dans un laps de temps court, afin d'éviter une famine de données, et donc une inutilisation de toute la puissance de calcul disponible.

3.2 Les esclaves

Chacun esclave reçoit à sa demande un intervalle d'entiers (sous forme d'un début et d'un pas) correspondant à un certain nombre de mots de passe possibles à tester. En effet, l'esclave va dans un premier temps convertir l'entier de début d'intervalle en entier, le tester, incrémenter la chaîne de caractère de 1 et recommencer le test. Il est beaucoup plus rapide et moins coûteux d'incrémenter la chaîne de caractères, en la considérant comme un nombre en base 15 (avec retenue, etc), que de réeffectuer une conversion d'un grand entier à l'aide de divisions et modulos.

Afin de respecter le fait que l'esclave n'est pas censé connaître le mot de passe, seule la fonction `strcmp` est utilisée à chaque itération pour comparer le mot de passe recherché, au mot courant. Si l'esclave trouve la solution, il l'envoie au maître, sinon il continue jusqu'à épuisement du travail, et en redemande au maître.

3.3 La répartition des tâches

L'intervalle de travail est aisé à construire : le calcul du maximum de possibilités évoqué précédemment est effectué, il détermine la valeur maximale de notre exécution. Cette valeur maximale est divisé en fonction de la longueur du mot de passe, et du nombre de processeurs (dans notre cas, en fonction du produit). Ce nombre de pas est volontairement petit, afin que le nombre d'échanges de données soit au début le plus réduit possible. Par la suite, quand un certain pourcentage des données a été consommé (50%, 75%, ...), ce pas est à nouveau divisé, afin d'éviter des problèmes de famine, qui feraient que des processus seraient inactifs, alors que d'autres seraient tout juste resservis. Ce procédé est itéré (dans notre cas, 2 fois). Dans l'idéal, le nombre d'itérations devraient dépendre de la longueur du mot de passe (plus le mot de passe est long, plus les intervalles le sont aussi), ainsi que du nombre de processus (plus il y a de processus, plus il y aura de demande).

L'attribution du travail est effectué de manière très simple : l'esclave contacte le maître, et lui demande des données. Si le maître possède encore du travail, il en redonne à l'esclave, sinon il l'ajoute dans son nombre d'esclaves ayant terminé leur exécution. Quand un esclave trouve la solution, il contacte le maître, et lui donne la solution, alors le maître fait stopper tous les esclaves.

4 Changements en version multi-threadée

Les changements en version multi-threadées sont beaucoup plus importants sur l'esclave que sur le maître. En effet, pour le maître, les changements sont presque inexistants : mis à part les changements sur les arguments à donner aux esclaves, et quelques modifications mineures comme le renvoi d'un message en cas de famine (toutes les données consommées), rien d'autre ne change.

En revanche, pour l'esclave les modifications sont importantes : il subdivise à nouveau l'intervalle reçu (par exemple en fonction de la longueur du mot de passe et du nombre de threads, afin de limiter la famine de données), et fait tester ces morceaux par des threads indépendants. Chacun de ses threads effectue les tests et incrémente sur son intervalle, comme dans la version précédente. Cependant, si un des threads, en piochant des données, réalise que la moitié des données a été consommée et qu'il n'y en a pas "en réserve", il exécute le thread de communication avec le maître, pour demander de nouvelles données, stockées dans des variables. Aussi, si un thread réalise que les données courantes sont vides, il va les remplacer par le nouvel intervalle en réserve, ou terminer si la réserve est vide.

Il est important d'appliquer un verrou sur les données suivantes, afin d'éviter des lectures/écritures concurrentes :

- la condition de fin
- l'intervalle de travail actuel
- le thread de communication avec le maître (PVM n'est pas "thread-safe", donc un seul à la fois)
- la variable permettant de savoir si il y a des données en réserve, ou si au moins elles ont été demandées.

De même, le coût de prise de ces verrous (`pthread_mutex_t`) est élevé, il est donc important de minimiser le nombre de prises, en effectuant des lectures en deux temps dans le cas de conditionnelles par exemple : le test est effectué une première fois sans le verrou, si le test est positif, le verrou sur la variable est pris, le test est effectué à nouveau, le code correspondant est effectué, et la variable déverrouillée.

En pratique, dans notre implémentation nous rencontrons des problèmes (erreurs de segmentation au niveau de l'esclave), que nous pensons dûs à des appels de fonctions PVM (en particulier `PVM_initsend(PVMDataDefault)`), en particulier quand les échanges de données ne sont pas totalement recouvert (c'est-à-dire quand le mot de passe est petit, et que les processus et threads sont très nombreux). Ainsi des exécutions comme `./craquage_multithread 1 1 6 p`, ou `./craquage_multithread 4 4 9 p` fonctionne parfaitement (entre autres, de nombreuses autres fonctionnent), de même que lorsque l'on remplace p par un mot de passe possible à trouver.

Enfin, sur les exécutions réussies, nous avons observé que les performances étaient décevantes : bien que les échanges de données soient totalement recouverts par les calculs, il semblerait que les prises de verrous ralentissent considérablement l'exécution, pour un résultat équivalent à celui de la version précédente. En effet, sur les exécutions sur des mots de passe de longueur 8 ou 9, avec un nombre de processus de l'ancienne version à 8 ou 16, la version multithreadée (peu importe la configuration : 8 threads 2 procs, 8 procs 2 threads, 4 threads 4 procs), les performances sont identiques : pour un mot de passe de longueur 8, environ 8s, et pour un de longueur 9 environ 125s. Cependant, peut être est-ce simplement parce que nous avons cherché à résoudre les problèmes ci-dessous, et que nous n'avons pas évalué suffisamment de configuration (nombre de processus, nombre de threads, taille du pas...). Il est raisonnable de penser que les performances soient meilleures pour des mots de passe plus long, car le nombre de verrouillages sera normalement moins important par rapport au nombre de calculs à effectuer.

5 Performances

Les résultats obtenus sur la plateforme PLAFRIM permettent de se rendre compte de l'influence de PVM et du nombre de processus sur la rapidité d'exécution (cf. fig. 1 et fig. 2).

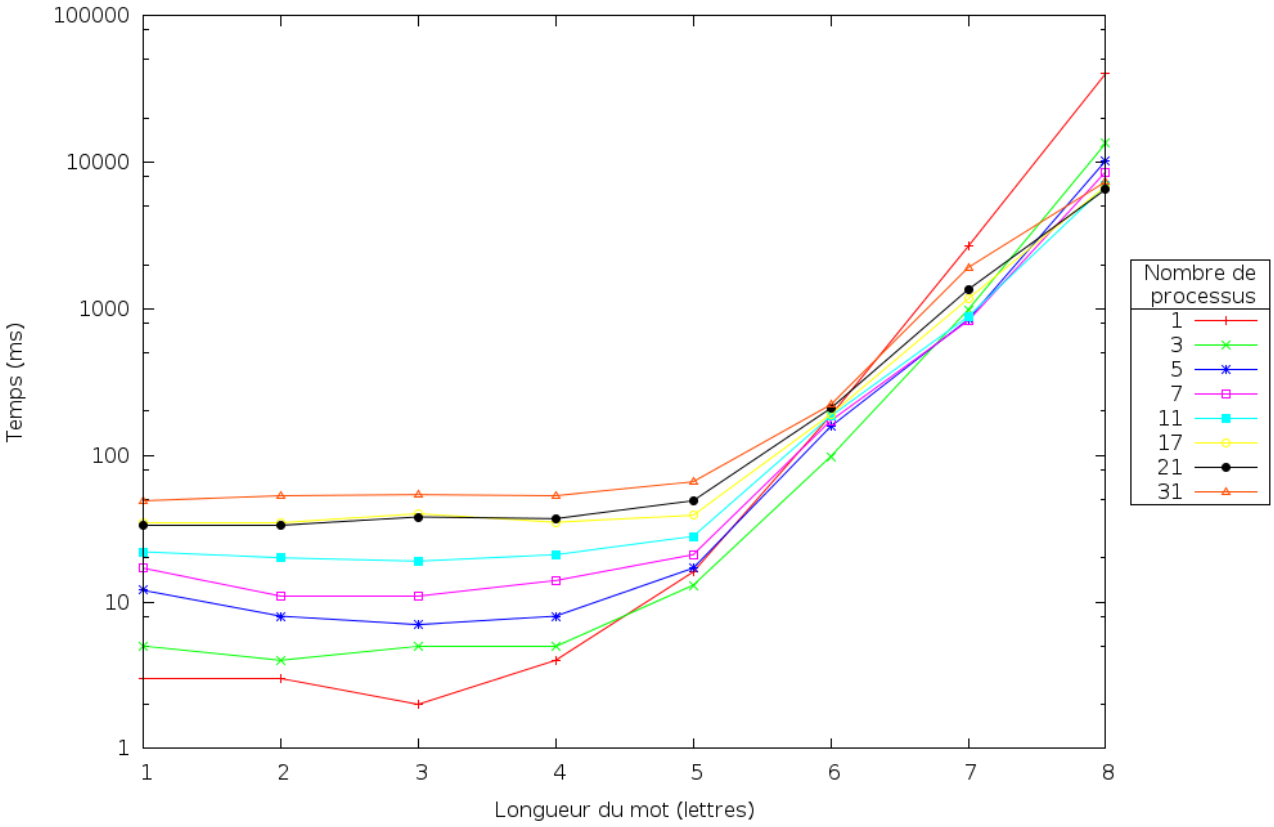

FIGURE 1 – Tests de performance, représentés sur une échelle logarithmique

6 Conclusion

PVM permet de faire communiquer entre eux différents processus afin que les opérations s'effectuent sur plusieurs unités de calcul, et donc plus rapidement. Toutefois, l'archaïsme de la bibliothèque, avec par exemple la nécessité d'empaqueter les données avant chaque envoi et les multiples fonctions devant être appelées au préalable, ainsi que la difficulté supplémentaire lors de la recherche des bogues

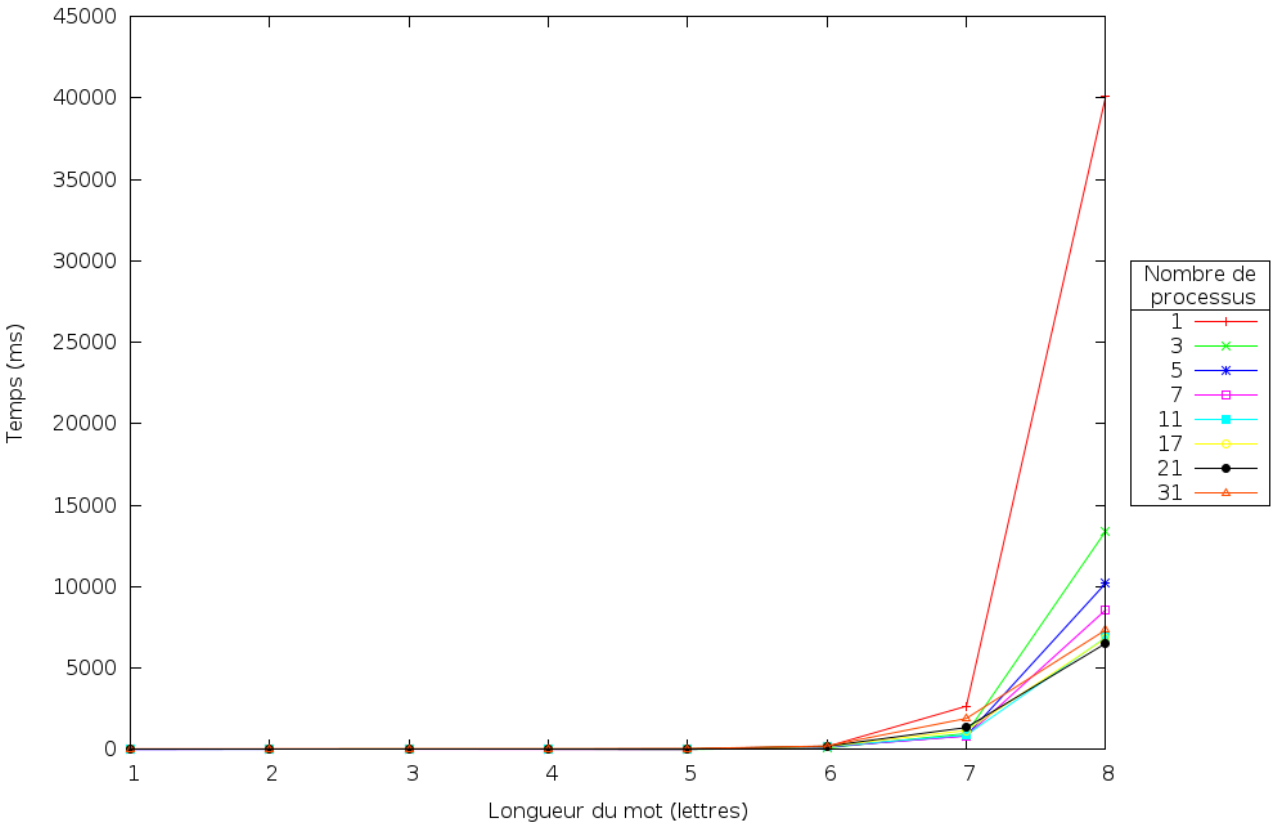

FIGURE 2 – Tests de performance

font qu'il est peu aisé de l'utiliser, notamment dans le cas où l'on souhaite multithreader les processus. En effet, PVM n'a pas été adapté pour un usage dans ce contexte, et par conséquent l'utilisateur doit prendre beaucoup de précautions. On peut toutefois remarquer le gain de performance appréciable apporté par la bibliothèque lorsque la taille de l'entrée devient importante.