Xavier Lambein
54621300

Projet 2 : Rapport

Victor Lecomte
65531300

LSINF1252

Concept

Dans le cadre de ce projet, nous avons décidé de ne pas adopter une simple approche producteur-consommateur avec parallélisation externe. À la place, nous avons développé un algorithme qui utilise les caractéristiques de l'énoncé et s'adapte au mieux aux informations qu'il obtient au fur et à mesure.

Puisque l'énoncé mentionne que tous les facteurs premiers sauf un sont utilisés plus d'une fois, on peut tirer un avantage considérable des facteurs premiers déjà trouvés. Notre algorithme profite donc de cette propriété à chaque fois qu'il trouve un facteur premier, en le testant sur tous les autres nombres, et tant que la lecture n'est pas terminée, en essayant de diviser un nombre lu par chaque facteur premier déjà trouvé.

Cette approche permet une diminution très significative du travail de factorisation, mais comme nous nous y attendions nous avons trouvé des cas où il cette approche le ralentit. Nous allons discuter de ces problèmes dans la section d'analyse des performances. Toutefois, nous avons tout de même choisi de faire comme ça, pour le challenge de synchronisation et d'organisation qu'il représentait, et pour le temps économisé pour des cas typiques (grands facteurs premiers, 2 à 3 occurences de chaque premier).

Architecture du programme

Notre programme se décompose en trois parties différentes, qui possèdent leur fonction propre : la boucle principale, les *readers* et les *factorizers*. Les *readers* sont des threads servant à lire des entiers depuis les entrées pour les placer dans une liste interne ; les *factorizers* sont des threads parcourant un entier pour en trouver les facteurs ; enfin, la boucle principale sert à synchroniser les *factorizers* pour changer l'entier courant à factoriser `to_fact` et éventuellement terminer le programme.

Ces parties partagent deux listes d'entiers qui sont la *waiting list* et la *prime list*. La première contient tous les nombres à factoriser. Elle possède deux entrées : les *readers*, évidemment, mais aussi les *factorizers*, comme nous allons le voir plus loin.

La seconde contient tous les facteurs premiers trouvés, et n'est utilisée que temporairement, tant qu'il reste des *readers* actifs. En effet, tant que tous les nombres n'ont pas été lus, il est impossible d'être certain que ces facteurs ne sont pas solution, et il est donc nécessaire de les garder. Cette liste ne possède qu'une seule entrée, et il s'agit de la boucle principale.

Chaque nombre stocké dans *waiting list* et dans la *prime list* transporte également deux informations : le fichier d'où il provient et son nombre d'occurrences, c'est-à-dire, s'il s'agit d'un facteur (premier ou non), le nombre de fois qu'il apparaît (divise un nombre). Cela est nécessaire pour vérifier si le facteur est une solution.

Dans la suite, nous allons expliquer en détail le fonctionnement de chaque partie.

Les *factorizers*

Tous les *factorizers* partagent une même variable globale `to_fact`, le nombre qu'ils sont en train de factoriser. Ceux-ci parcourent tous les entiers de 2 à $\sqrt{\texttt{to\_fact}}$, en vérifiant à chaque fois

si l'entier en question divise `to_fact`. Si c'est le cas, un facteur (pas nécessairement premier) a été trouvé, et la variable globale `to_fact` est effectivement divisée par celui-ci, autant de fois que possible. Ensuite, le facteur est ajouté à la *waiting list*, en notant son nombre d'occurrences—ici le nombre de fois qu'il a divisé `to_fact`—, puis les *factorizers* continuent leur travail.

Bien évidemment, les différents threads ne parcourent pas chacun tous les nombres entre 2 à $\sqrt{\texttt{to\_fact}}$. Ceux-ci se répartissent le travail : le premier thread itérant sur $i = 2, 2+n, 2+2n, \dots$; le second thread itérant sur $i = 3, 3+n, 3+2n, \dots$; etc. Si ce n'était pas le cas, augmenter le nombre de threads n'aurait strictement aucun intérêt !

Lorsque tous les *factorizers* ont dépassé $\sqrt{\texttt{to\_fact}}$, ceux-ci attendent que la boucle principale choisisse un nouvel entier à placer dans `to_fact`, puis recommencent leur travail.

La boucle principale

Il est important de noter que la valeur de `to_fact`, une fois que les *factorizers* ont terminé leur travail, est un facteur premier du nombre initial.

Le rôle de la boucle principal est double : tout d'abord, elle va récupérer ce facteur premier et le traiter ; ensuite, si la solution n'a pas encore été trouvée, elle va choisir un nouvel entier de la *waiting list* et le placer dans la variable `to_fact` avant de relancer les *factorizers*.

Traitement du facteur Si celui-ci n'est pas 1, alors la boucle principale va parcourir chaque entier de la *waiting list* et diviser celui-ci par le facteur. Ensuite, deux situations sont possibles :

- les *readers* n'ont pas fini de lire les fichiers ; dans ce cas, le facteur est simplement ajouté à la *prime list*, en prenant soin de noter le nombre de fois qu'il a divisé un nombre de la *waiting list* dans son nombre d'occurrences ;
- les *readers* ont terminé leur lecture ; dans ce cas, si le facteur n'a divisé aucun nombre de la *waiting list*, il s'agit de la solution, et le programme termine.

Choix d'un nouvel entier Si la solution n'a pas été trouvée, un nouvel entier est choisi pour `to_fact` : il s'agit du plus petit entier de la *waiting list*, trouvé en faisant une simple recherche en $\mathcal{O}(n)$; en effet, la liste n'est pas ordonnée.

Les *readers*

Les *readers* sont des threads génériques qui lisent des entiers au format BigEndian depuis *file descriptor* et placent ceux-ci dans la waiting list. Leur fonctionnement exact est en vérité un peu plus complexe que cela.

Pour chaque nombre lu, un *reader* va parcourir la *prime list* et diviser le nombre par chaque facteur premier autant de fois que possible, en mettant à jour le nombre de fois que ce facteur a divisé. Cela est nécessaire car les nombres lus n'ont pas eu l'occasion d'être divisés par la boucle principale lors du traitement du facteur.

C'est là la seule raison d'existence de la *prime list*—et celle-ci est d'ailleurs détruite lorsque le dernier *reader* a terminé son travail. Avant de la désallouer, cependant, le dernier *reader* la parcours une dernière fois : en effet, si l'un des facteurs a un nombre d'occurrences de 1, alors il s'agit de la solution et le programme doit se terminer.

Mécanismes de synchronisation

Les *readers* et la boucle principale

Ces deux parties doivent être synchronisées à la manière d'un producteur-consommateur où il n'y aurait qu'un seul consommateur : la boucle principale. Celle-ci attend qu'un *reader* lise un entier et le place dans la *waiting list*, afin de pouvoir le mettre dans `to_fact`.

Nous avons donc un sémaphore `sem_full`, dont la valeur est égale au nombre d'entiers présents dans la *waiting list*. À chaque fois qu'un *reader* ou un *factorizer* ajoute un entier dans celle-ci, le sémaphore est posté, et à chaque fois que les *factorizers* ont besoin d'un nouvel entier à factoriser, la boucle principale effectue un wait sur ce sémaphore pour s'assurer qu'il peut en effet récupérer un entier de la *waiting list*.

La modification de l'état global

L'état global, constitué notamment de la *waiting list*, de la *prime list* et de `to_fact`, peut être sans risque accédé en lecture mais doit toujours être protégé par une unique mutex lors d'un accès en écriture. Il s'agit de la même mutex pour toutes ces données, car la modification de d'une partie de l'état global a systématiquement un impact sur le reste.

Commande des *factorizers*

Puisque la parallélisation de la factorisation est interne, il faut que les *factorizers* se synchronisent avec la boucle principale qui va choisir les nombres à factoriser un par un. La boucle principale va envoyer un signal aux factorizers pour qu'ils commencent à factoriser, ce qu'ils vont faire, puis ils vont attendre qu'eux tous aient fini avant d'envoyer un signal à la boucle principale et d'attendre le prochain nombre à factoriser.

La solution que nous avons développée est une extension du problème du rendez-vous, et se présente ainsi (son implémentation se trouve dans le fichier `factorizer.c`) :

```
1  while true:
2      wait start
3      post start
4
5      factorisation...
6
7      lock mutex
8          num_waiting++
9          if num_waiting == num_factorizers:
10             wait start
11             post handshake
12     unlock mutex
13
14     wait handshake
15     num_waiting--
16     if num_waiting == 0:
17         post finish
18     else:
19         post handshake
```

On retrouve tout à fait ci-dessus l'algorithme classique du problème du rendez-vous, géré par la mutex `mutex` et le sémaphore `handshake`, mais des ajouts ont été faits pour gérér l'échange entre la boucle principale et les *factorizers*. Nous allons commenter les lignes que nous avons ajoutées :

— Lignes 2 et 3 : avant de commencer à factoriser, le *factorizer* attend le message envoyé par la boucle principale via le sémaphore `start` et le transmet au *factorizer* suivant. De cette manière, la boucle principale ne doit envoyer ce message qu'une seule fois.

— Ligne 10 : quand tous les *factorizers* sont arrivés au bout de leur factorisation, on en profite pour décrémenter le sémaphore `start` que le dernier *factorizer* à entrer avait posté en ligne 3 sans que personne ne l'attende. Il est important que cette opération se fasse à cet endroit précis pour éviter qu'un factorizer ne soit libéré par l'incrémentation du sémaphore `handshake` et ne commence sans attendre le signal de la boucle principale.

— Lignes 15 à 17 : tous les *factorizers* passant la barrière vont avoir le même comportement que pour le problème de rendez-vous classique sauf le dernier qui à la place d'incrémenter `handshake` va incrémenter `finish` pour signaler à la boucle principale que tous les *factorizers* ont fini et sont prêts à recevoir un nouveau incrément de `start` en ligne 2.

Il suffit ensuite à la boucle principale de poster `start` puis d'attendre `finish` dès qu'elle veut lancer les threads sur un nombre.

Analyse des performances

Comme mentionné dans la section « Concept », notre algorithme marche particulièrement bien pour certains cas, et malheureusement il marche particulièrement mal pour d'autres. Pour mettre ces effets en évidence, nous mesuré ses performances sur trois types de tests en particulier, et nous allons commenter les résultats. [1] Commençons par le pire cas qui puisse arriver.

1. Les tests ont été générés par notre générateur `gen.c` et les temps ont été obtenus en chronométrant au moins trois fois chaque test avec un nombre de threads et en faisant la moyenne.

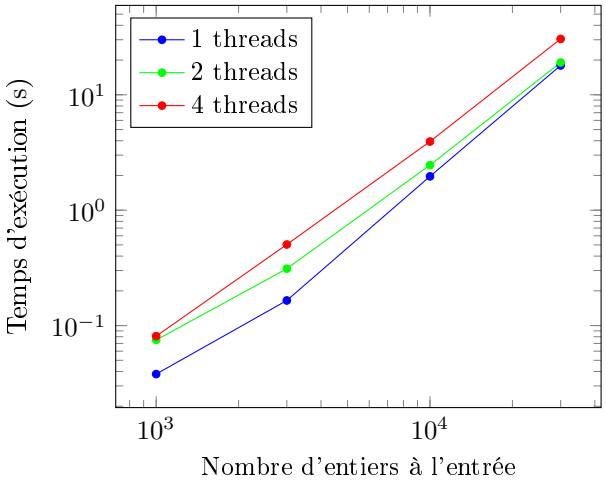

FIGURE 1 – Avec beaucoup de nombres faits de petits facteurs premiers, la complexité $O(n^2)$ des divisions par les premiers trouvés prédomine, et ajouter des threads n'améliore pas les choses.

Premier cas : beaucoup de petits facteurs premiers

À cause des vérifications que nous effectuons pour voir si un facteur premier est utilisé dans d'autres nombres, notre complexité par rapport au nombre d'entiers n dans les fichiers d'entrée est environ $O(n^2)$ (si l'on néglige le temps de factorisation). Par conséquent, pour des fichiers contenant beaucoup de nombres avec de petits facteurs premiers, notre programme marche plutôt lentement : en effet, les facteurs premiers étant petits, le temps passé à factoriser est négligeable par rapport au temps passé à tenter de diviser tous les autres nombres par un facteur premier donné. Cela mène à deux inconvénients :

— La complexité en $O(n^2)$ fait que le temps d'exécution augmente rapidement avec la taille de l'input, et l'algorithme a du mal à factoriser plus de quelques dizaines de milliers de nombres.

— Les tentatives de division par les facteurs premiers trouvés n'étant pas parallélisables, l'ajout de threads n'améliore pas les performances et a même tendance à les empirer.

Les résultats des tests sont donnés à la figure 1. La tendance quadratique du temps d'exécution se montre clairement sur le graphe logarithmique, ce qui signifie que le temps de factorisation est ici négligeable. On voit aussi comme conséquence logique que l'ajout de threads ne régle pas le problème, et l'empire même.

Toutefois, nous considérons que ce genre de cas n'est pas représentatif d'un usage normal d'un programme de factorisation. Il nous semble clair que la factorisation devrait être la partie demandant le plus de calculs, ce qui nous a été confirmé à la séance d'architecture. Pour régler efficacement ce genre de cas, il conviendrait de d'abord précalculer les petits nombres premiers puis factoriser les nombres dans des threads différents en utilisant les premiers précalculés, en producteur consommateur.

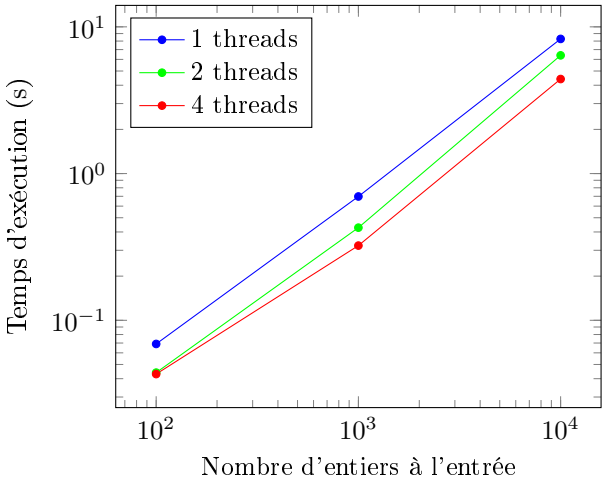

FIGURE 2 – Avec des nombres formés de grands facteurs premiers placés aléatoirement, le temps d'exécution reste très court pour de grandes entrées.

Deuxième cas : beaucoup de grands facteurs premiers répartis aléatoirement

Montrons maintenant un type de cas où notre algorithme se montre particulièrement efficace. Il s'agit d'un cas avec de grands facteurs premiers (choisis aléatoirement entre 1 et 2^{32}), chacun avec 2 à 3 occurences (sauf la solution), et répartis aléatoirement dans les nombres de l'entrée, qui peuvent aller jusqu'à 2^{64}. Il s'agit donc d'un cas d'utilisation typique très raisonnable.

Sur ce genre de cas, notre programme va d'abord tenter de factoriser un nombre dans la *waiting list*, ce qui va lui prendre environ 2^{32} essais au pire cas. Ensuite, il aura trouvé un facteur premier, qu'il pourra tester sur d'autres nombres en attente. Ceux-ci vont très probablement à leur tour donner de nouveaux facteurs premiers, qui vont permettre de factoriser facilement d'autres nombres, etc. Au final, la découverte du premier facteur premier va mener à une réaction en chaîne qui va permettre de factoriser les autres nombres très facilement.

Au final, ce qui prendra le plus de temps sera de vérifier que tous les facteurs trouvés sont premiers, ce qui prendra de l'ordre de 2^{16} opérations par nombre, bien mieux que les 2^{32} que nous aurions obtenus avec une simple factorisation parallèle de tous les nombres en producteur-consommateur. Puisque c'est cette partie qui est le facteur limitant, nous avons une complexité de l'ordre de $O(n)$ où n est le nombre d'entiers dans l'input (en éliminant le facteur constant 2^{16}.

Les résultats sont donnés à la figure 2. La tendance en $O(n^2)$ apparaît très clairement et on remarque que l'ajout de threads aide significativement à accélérer le programme, malgré la parallélisation interne. Pour $n = 10000$, il y a un facteur 1.29 de différence entre 1 threads et 2 threads, et un facteur 1.45 entre 2 threads et 4 threads. Il est normal que l'on ne s'approche pas très près du facteur 2 ici car la vérification de division des facteurs premiers sur 10000 nombres amène un terme constant non-négligeable.

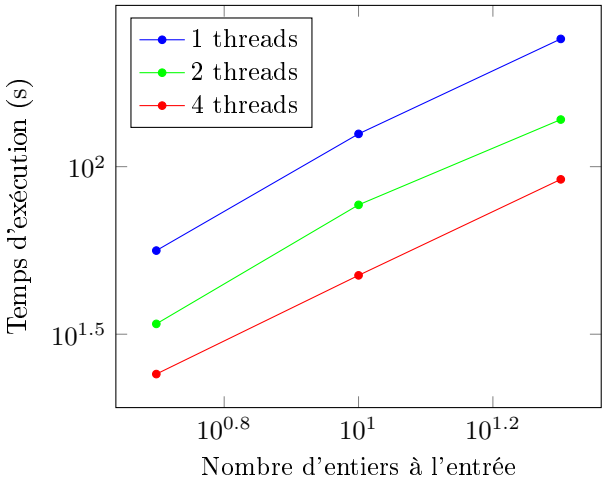

FIGURE 3 – Avec des carrés de grands facteurs, notre algorithme perd son avantage, mais la parallélisation accélère nettement le calcul.

Troisième cas : quelques carrés de grands premiers

Cette série de cas a été spécialement imaginée pour éliminer l'avantage que notre programme a grâce à sa réutilisation des facteurs premiers trouvés. Il consiste en un fichier contenant uniquement des carrés de grands premiers, sauf pour le dernier nombre, le produit du plus grand premier parmi les précédents et de la solution, encore plus grande.

Étant donné que notre programme utilise une heuristique qui consiste à toujours choisir le plus petit nombre de la liste d'attente, il va d'abord tenter de factoriser le plus petit carré, puis le suivant, etc. pour finalement arriver au dernier nombre, contenant la solution. Puisque chaque nombre premier n'est présent que dans un nombre (son carré), la réutilisation des facteurs premiers va être complètement inutile, en ne va donc pas permettre d'abaisser le temps de factorisation des autres nombres de 2^{32} à 2^{16} comme pour le cas précédent. Il s'agit donc d'un ralentissement de l'ordre de 2^{16} (on a observé un ralentissement relatif de 15000 environ, ce qui est raisonnablement dans l'ordre de grandeur de $2^{16} = 65536$).

Ce cas est évidemment très improbable sur des entrées aléatoires. Son intérêt principal est de mesurer précisément les performances de la partie factorisation. On voit bien ici que le temps est directement proportionnel à la taille de l'entrée, et l'augmentation du nombre de thread améliore la performance : d'un facteur 1.74 entre 1 et 2 threads et d'un facteur 1.51 entre 2 et 4 threads. On peut aisément attribuer la distance par rapport au facteur 2 à la parallélisation interne, et éventuellement au système d'exploitation.