Université Catholique de Louvain-la-Neuve

Ecole Polytechnique de Louvain
LSINF1252 - Systèmes informatiques

- May 6, 2015-

Projet 2

Factorisation de nombres

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Monnoyer de Galland de Carnières Charles

Paris Antoine

1 Introduction

Dans le cadre de notre cursus en Sciences de l'Ingénieur à l'EPL, orientation Informatique, il nous est demandé de venir en aide à un mathématicien devant factoriser des nombres. Ce dernier doit déterminer parmi une liste d'entiers non-signés de 64 bits le seul facteur premier (les facteurs premiers sont des entiers non-signés de 32 bits) qui apparaît une et une seule fois.

Notre tâche consiste à élaborer un programme pouvant réaliser cela en profitant des possibilités d'un ordinateur multiprocesseurs : on utilisera des threads pour accélerer la recherche. De plus, le programme devra être en mesure de traiter plusieurs fichiers en même temps, pouvant provenir de l'entrée standard, du système de fichiers local ou des serveurs Internet, et d'identifier de quel fichier provient le facteur recherché. Enfin, une analyse de performances sera réalisée : le programme calculera le temps de son exécution à chaque utilisation.

Les nombres passés en arguments de notre programme seront écrits suivant la représentation *Big Endian*. L'utilisateur pourra faire varier une option *-maxthreads n* correspondant au nombre maximum de threads de calculs du programme.

Au final, notre programme retournera le facteur premier recherché, le fichier d'origine de ce facteur, ainsi que le temps pris par le programme lors de son exécution.

2 Algorithme de factorisation

L'élaboration de notre programme a d'abord exigé que nous trouvions un algorithme nous permettant de décomposer un nombre en ses facteurs premiers. Nous avons donc cherché les divers algorithmes existants et avons finalement retenu la méthode triale pour diverses raisons qui seront explicitées par la suite.

Nous avions d'avord envisagé d'employer un algorithme plus élaboré et performant. Cependant, il s'est avéré laborieux à mettre en place pour des nombres de 64 bits. De plus, les trop bonnes performances dudit algorithme ne permettaient pas d'observer clairement l'évolution des performances avec la variation du nombre de threads (notamment car les autres étapes indépendantes du nombre de threads ne retrouvaient à consommer beaucoup plus de temps que celles dépendantes, obstruant cette analyse). Nous avons donc opté pour la méthode triale.

L'algorithme trial est des plus simples : il s'agit de tester chaque diviseur possible, en commençant par 2 jusqu'à une valeur déterminée. Si le modulo du nombre et du diviseur potentiel est 0, alors le nombre n'est pas premier ; le diviseur est retenu comme facteur premier et le procédé est réitéré sur la division du nombre par le facteur. Remarquons que le facteur retenu est nécessairement premier, si ce n'était pas le cas, un diviseur de ce facteur aurait été intercépté avant.

Définir la borne d'arrêt des diviseurs potentiels n'est pas compliqué : un nombre ne sera jamais divisible par un nombre plus grand que sa racine carrée (excepté lui-même) ; dès lors, il suffit d'arrêter les

tests à sa racine. Si cette borne est atteinte, le nombre est premier.

3 Architecture générale

La logique que nous avons suivie pour réaliser notre programme est celle représentée sur la figure 1.
Premièrement, nous créons autant de threads que de fichiers à traiter : ce sont les *extractors*. Leur rôle est d'extraire les nombres un par un des fichiers, de les traduire du *Big Endian*, et de les mettre dans un premier buffer (*buffer1*). La taille de ce dernier est fixée par le nombre de threads *maxthreads n* : en effet, si le buffer est plus petit, les threads ne seront jamais utilisés de façon optimale, si il est plus grand, les threads n'auront pas vraiment de meilleurs performances et le buffer ne sera jamais rempli complètement.
Ensuite, les threads *factorizer* (dont le nombre est, pour rappel, fixé par la variable *maxthreads n*) se chargent d'extirper les nombres du buffer1 et de les décomposer en facteurs premiers. Pour ce faire, ils cherchent le plus petit diviseur premier du nombre et le mettent dans un second buffer de la même taille que le premier (*buffer2*). Ils réitèrent l'opération sur le quotient du nombre et du diviseur jusqu'à avoir entièrement décomposé le nombre (voir la section précédente pour plus de précisions sur l'algorithme de factorisation).
Enfin, un thread unique *save_data* se charge de prendre les facteurs premiers du second buffer et de les insérer dans la structure chainée *list* en mettant à jour la structure représentant un nombre afin de, au final, pouvoir identifier le facteur premier qui n'apparait qu'une seule fois.

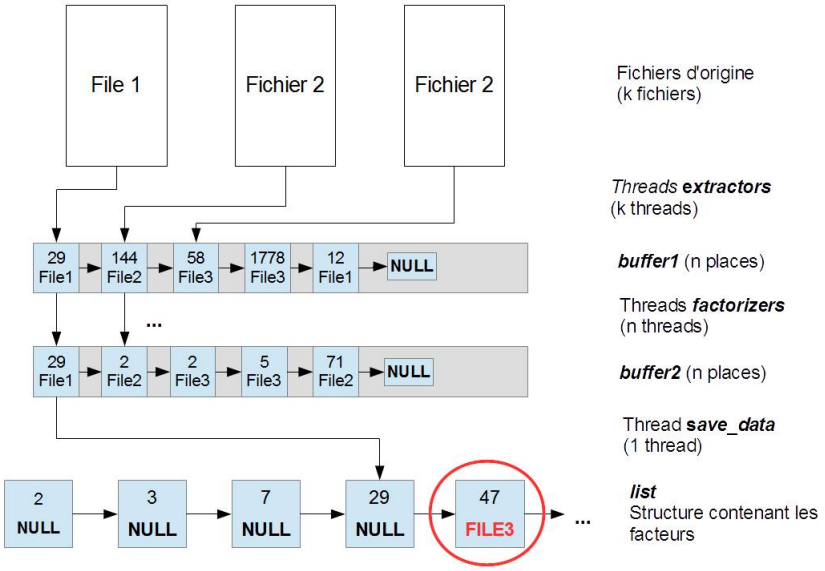

Figure 1: L'architecture de notre programme.

Comme dit précédemment, nous avons défini une structure permettant de représenter un nombre *number* définie par son contenu *n* (de type uint64_t) et d'une chaîne de caractère *origin* contenant le nom

de son fichier d'origine.
Pour déterminer le facteur premier n'apparaissant qu'une seule fois dans la liste chaînée *list*, nous avons procédé comme suit : lorsque une number est inséré dans la liste (ordonnée), on vérifie d'abord si un number possédant le même n ne s'y trouve pas déjà. Si ce n'est pas le cas, on se contente de l'insérer normalement, sinon, on change la variable origin du node en NULL. Ainsi, si un facteur n'apparaît qu'une seule fois, sa variable origin sera différente de NULL.

Notre programme est divisé en plusieurs fichiers (vous trouverez parmi eux un README pour de plus amples détails). Le corps principal du programme se trouve dans upfact.c : c'est ce fichier qui sera compilé et exécuté. Dans core.c, nous avons mis l'ensemble des fonctions dépendantes des threads (l'extraction de fichiers, l'insertion dans la liste ainsi que la factorisation). L'algorithme de factorisation proprement dit se trouve quant à lui dans trial.c. D'autres fichiers plus modestes mais malgré tout importants sont aussi présents : perf.c contient la fonction relative à l'analyse de performances, io.c une fonction gérant diverses erreurs et stack.c les fonctions de gestion de la structure chaînée. Enfin, fopen.c contient les fonctions relatives à l'extraction de fichiers *via* la librairie *libcurl*. Ce fichier n'a pas été initialement écrit par nous mais est basé sur le code d'exemple conseillé dans les consignes de ce projet.

Pour utiliser les threads à bon ascient, nous avons considéré la situation comme un problème de producteurs-consommateurs double. Les premiers producteurs sont les *extractors* qui stockent les nombres dans le *buffer1* qui sont consommés par les *factorizers*. Ensuite, ces mêmes threads font office de producteurs du *buffer2* où ils stockent les facteurs premiers consommés par le thread *save_data*.
Afin d'éviter les aléas du non-déterminisme, un tel problème doit être protégé comme suit : chaque buffer doit être protégé par un mutex (pour éviter les problèmes de section critique) et leur accès est réglé par deux sémaphores chacun : un pour leurs consommateurs, l'autre pour leurs producteurs.

4 Performances

Nous avons mesuré les performances de notre programme pour différents nombres de threads de calculs et pour des fichiers contenant de très grands nombres. Les résultats de ces mesures sont indiqués dans la figure 2. On constate que la figure 2 ressemble fortement à la loi d'Amdahl.

Remarque importante Comme expliqué dans la section précédente, pour permettre de mieux observer les améliorations de performances en fonction du nombre de threads, nous avons utilisé un algorithme de factorisation naïf. Celui-ci rend le programme assez lent pour de très gros nombres (à titre d'exemple 2200 secondes pour 1 threads).
On constate une nette amélioration du temps d'exécution par rapport au nombre de threads jusqu'à environ 10 threads (figure 1). Par après, les performances s'améliorent toujours mais dans une moindre mesure. Ce résultat est également bien traduit par la figure 2, illustrant l'accélération du programme par rapport au nombre de threads. En définitive, ces résultats sont conformes à ceux escomtés et à la Loi de Amdahl : l'amélioration des performances est considérable jusqu'au seuil des threads physiques où elles s'atténuent.

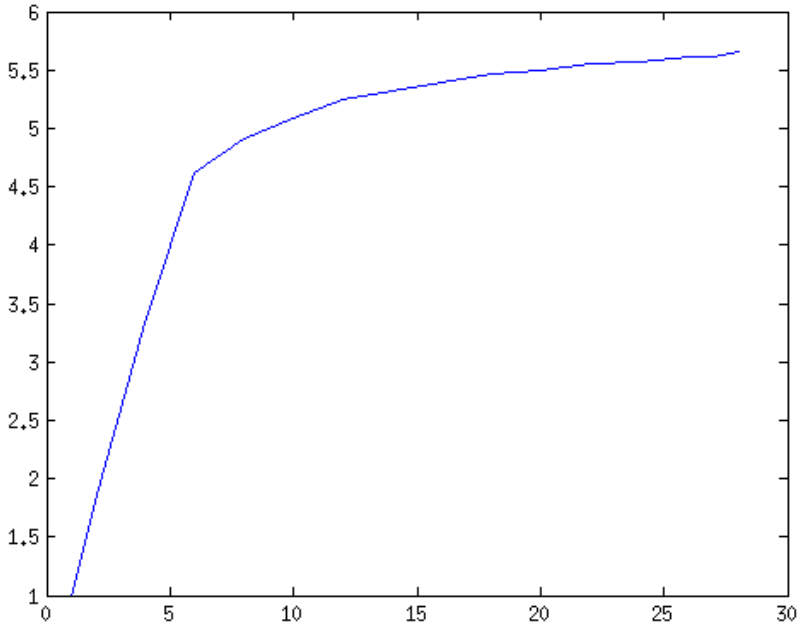

Figure 2: Accélération du programme en fonction du nombre de threads sur un processeur i7 3770k (12 coeurs) avec 8GB de RAM.

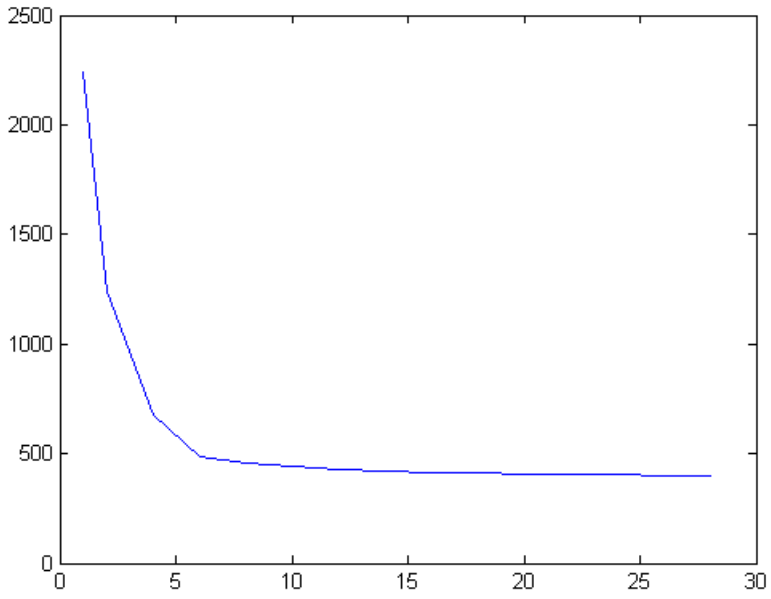

Figure 3: Temps d'éxécution en fonction du nombre de threads sur un processeur i7 3770k (12 coeurs) avec 8GB de RAM.