

UNIVERSITÉ CATHOLIQUE DE LOUVAIN-LA-NEUVE

ECOLE POLYTECHNIQUE DE LOUVAIN
LSINF1252 - SYSTÈMES INFORMATIQUES

- 6 MAI 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 accélérer 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 avons 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 intercédé 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 au 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.

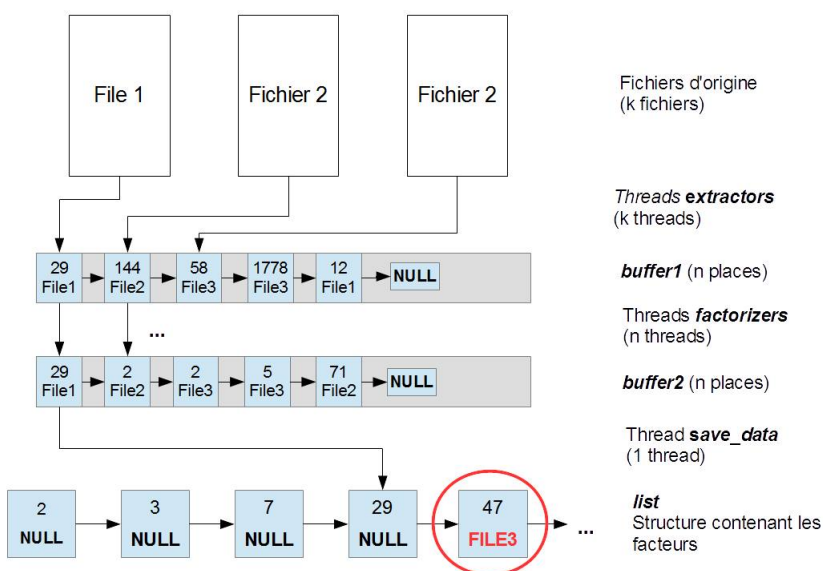


FIGURE 1 – L'architecture de notre programme.

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).

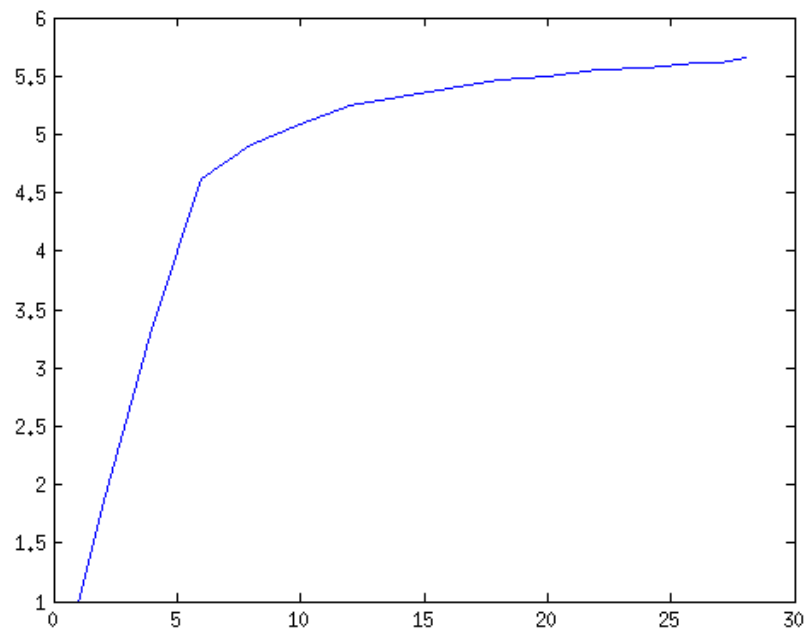


FIGURE 2 – Accélération du programme en fonction du nombre de threads sur un processeur i7 3770k (12 coeurs) avec 8GB de RAM.

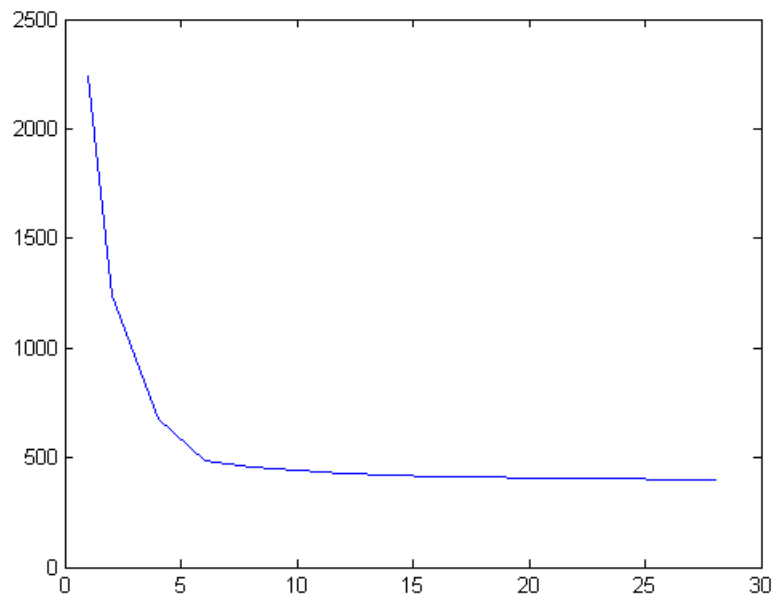


FIGURE 3 – Temps d'exécution en fonction du nombre de threads sur un processeur i7 3770k (12 coeurs) avec 8GB de RAM.