Travail Pratique 1

Pthread

Équipe 7:
Diego Silva Pires
Hervé Simard

10 février 2018

Introduction

Le travail consistait à prendre une version séquentielle du Crible d'Ératosthène écrite en C pas optimisé servant à trouver les chiffres primes jusqu'au chiffre fourni et l'adapter pour que cela fonctionne de façon parallèle en utilisant la librairie *pthread* de la *language C*.

Comme critères d'acceptation de ce travail, le programme doit accepter sur une ligne de commande un premier paramètre indiquant la limite de nombres à analyser ainsi qu'un second paramètre indiquant le nombre de *threads* que le programme utilisera.

Nous devons ainsi afficher le résultat du temps d'exécutions de la version parallèle et de la version séquentiel, ce qui nous permettra de déterminer si après la parallélisation du code nous avons été en mesure d'obtenir un speedup de notre application.

Procédé

Ce document présente 6 algorithmes différents utilisant *pthread* pour réaliser la parallélisation de l'algorithme séquentiel fourni en plus d'une nouvelle version séquentielle optimisée pour rendre les comparaisons plus semblables, lorsqu'une des solutions parallélisées ici présentes a été aussi optimisée.

Toutes les solutions ont comme principe général de déléguer le calcul du *Crible* complètement au fil d'exécution, c'est-à-dire que la fonction principale du programme s'occupe seulement de déclarer et initialiser les *threads*, sans aucun traitement supplémentaire. Toute la logique est rendue à la fonction de la *thread*.

Ainsi, la différence entre chaque solution réside entre autres dans la manière que les itérations sont implémentées à l'intérieur de la *thread*.

Tous les fils d'exécution reçoivent une structure de données semblable, afin de ne pas devoir créer un code écrit pour chacun d'entre eux.Toutefois, ce n'est pas toutes les propriétés de cette structure qui sont utilisées dans chaque implémentation :

```
struct ThreadInput{
  long id; // Identifiant du fil
  long begin; // où commencer à traiter
  long end;; // où terminer le traitement
  long wheelFactor; // facteur de répétition du chiffre
};
```

Dans le cas du *Crible*, nous avons réalisé qu'Il n'est pas évident du premier coup d'oeil de déléguer une tranche de chiffres à être traités dans chaque *thread*, lorsque sans optimisation chaque élément à traiter implique une validation sur toutes les autres éléments de notre *array*.

Parallèle 1:

Le principe est que chaque fil d'exécution reçoit une quantité égale d'éléments à traiter, par exemple, si nous avons 1000 éléments à traiter sur 4 *threads*, chaque *thread* doit traiter 250 éléments.

Les paramètres d'initialisation de la *thread* sont les suivants:

```
lInput->begin = 2 ;
lInput->end = (i + 1) * lItemPourThread ;
```

où "*lItemPourThread"* représente le morceau égal à être traité.

Le code dans le *thread* est le suivant:

```
for (unsigned long p=2; p < end; p++) {
  if (flagsParallel[p] == 0) {
    for (unsigned long i=2; i*p < end; i++) {
      flagsParallel[i*p]++;
    }
```

Chaque *thread* initie toujours par 2, pour supprimer les primes de deux, ce qui n'est pas une bonne idée puisque la première *thread* qui commence à exécuter ferait le calcul au complet du *crible*, et les autres ne feraient rien.

Sauf pour la valeur de "*end"* on peut déduire que c'est quasiment la version séquentielle exécutée plusieurs fois.

Juste un peu moins de traitement dans la première thread lors qu'on ne valide pas jusqu'au dernier élément de la liste (une chose qu'on s'est rendu compte que pourrait être optimisé dans toutes les cas)

Parallèle 2:

Nous avons essayé d'exécuter le *Crible* avec l'utilisation de la technique *Wheel Factorization*[1] dans les fils d'exécution, mais le calcul ne se faisait pas bien et il y avait des erreurs. Nous ne tenons donc pas compte des résultats de cette implémentation dans cette étude.

Cela explique la présence de la propriété "*wheelFactory*" dans la structure des fils d'exécution.

Parallèle 3:

La solution est presque identique que le Parallèle 1, mais au lieu de tous commencer à 2, les fils d'exécution commencent à un index précis. Par exemple, si on a 4 fils, sur une

[1] https://en.wikipedia.org/wiki/Wheel_factorization

limite de 1000 nombres, le premier commence à 2 et termine à 250, le second débute à 251 et termine à 500, ainsi de suite.

Il reste toujours que le premier fil d'exécution va faire la plus grande partie des traitements.

```
    unsigned int begin = (*lThreadInput).begin;
    unsigned int end = (*lThreadInput).end;

    if (begin == 1) begin = 2;

    for (unsigned long p=begin; p < end; p++) {
        if (flagsParallel3[p] == 0) {
            // invalider tous les multiples
            for (unsigned long i=begin; i*p < end; i++) {
                flagsParallel3[i*p]++;
            }
        }
    }
```

Si on ignore le code optimisé, c'est la solution la plus efficace en répartissant un peu de traitement aux autres fil. On voit un gain d'optimisation meilleur avec le plus de donnés que nous demandons à traiter.

Parallèle 4

Ce principe est le même que pour le procédé Parallèle 3, à l'exception que nous introduisons un mutex avant d'incrémenter la variable qui fait l'exclusion des chiffres qui ne sont pas primés. Cela a été fait afin d'évaluer la performance et de tester le fonctionnement de la mutex.

```
for (unsigned long i=begin; i*p < end; i++) {
   pthread_mutex_lock(&lock_x);
   flagsParallel4[i*p]++;
   pthread_mutex_unlock(&lock_x);
}
```

Le temps de réponse de cette solution, avec l'ajout de mutex, est beaucoup plus grand (on parle parfois de l'ordre de presque 10 fois plus lent lorsqu'on doit, par exemple, calculer les nombres premiers de 10 à la puissance 8 et plus).

Ces tests nous ont fait prendre conscience du temps que ça prenait pour verrouiller et déverrouiller le mutex dans un traitement simple.

Parallèle 5

Modification du premier algorithme parallèle afin d'inclure dans le calcul la racine carrée, ce qui évite ainsi de valider des nombres superflus:

```
for (unsigned long p=2; p < sqrt(end); p++) {
        if (flagsParallel5[p] == 0) {
            // invalider tous les multiples
            for (unsigned long i=2; i*p < end; i++) {
                if (flagsParallel5[i*p] == 0){
                    pthread_mutex_lock(&lock_x2);
                    flagsParallel5[i*p]++;
                    pthread_mutex_unlock(&lock_x2);
                }
            }
        }
    }
```

Toujours pas une bonne solution, lorsqu'on parcourt toujours toutes les élements de notre liste, cela fait en sorte que le premier fil d'exécution exécute tous les traitements.

Parallèle 6

L'algorithme le plus performant, c'est une implémentation semblable à l'idée du Parallèle 3, mais en plus le code est optimisé pour ne pas passer à travers de toutes les élements.

```
    if (begin == 1) {
        begin = 2;
    }

    for (unsigned long p=begin; p*p <= maxValue; p++) {
        if (flagsParallel6[p] == 0) {
            // invalider tous les multiples
            for (unsigned long i=p*p; i <= maxValue; i+=p) {
                flagsParallel6[i]++;
            }
        }
    }
```

En faisant la première boucle "*for*" jusqu'à l'exponentiel de la valeur maximum, ça réduit beaucoup le nombre de validations.

Une tentative à été faite pour donner une quantité progressive d'éléments à traiter dans cette algorithme aussi, mais ça n'a pas changé le temps moyen du traitement.

Sequentielle optimisée

Nous avons appliqué le code optimisé de la tentative 6 et créé une fonction séquentielle effectuant le même traitement. Il est facile de remarquer que cela donne des résultats très probants.

Ordinateur

Les tests ont été effectués sur l'ordinateur personnel d'Hervé. Voici les caractéristiques principales:
Processeur: Intel(r) Core(™) i5-4670K CPU @ 3.40 GHZ (4 CPUs), ~3.4 GHZ
Mémoire : 8192 MB Ram
4 coeurs disponibles.

Graphique

Limite	1000	10000	100000	1000000	10000000	100000000	1000000000
Séq	0.000009	0.000081	0.000996	0.010419	0.221195	2.665315	34.525694
Séq opt.	0.000005	0.000039	0.000487	0.005112	0.103012	1.257651	16.049457
Par 1	0.000350	0.000587	0.002473	0.015554	0.307447	3.866378	46.996571
Par 3	0.000317	0.000425	0.001133	0.010417	0.193822	2.447031	28.814587é
Par 4	0.000282	0.001884	0.019241	0.209296	2.569681	28.400710	315.258208
Par 5	0.000320	0.000769	0.006196	0.064968	0.653455	6.875489	71.415606
Par 6	0.000265	0.000427	0.000637	0.005461	0.087334	1.321014	16.031414

Le graphique démontre l'impact des traitements parallèles versus le traitement séquentiel. Dès que le programme a dénombré une limite de 100 000 nombres, le programme parallèle 6 a démontré une amélioration constante. On peut voir ainsi que l'optimisation du code permet d'en accélérer grandement le fonctionnement.

Chaque cas a été testé avec 4 fils d'exécution. Les temps sont rendus en secondes.

Analyse

Notre programme effectue le calcul des nombres premiers de la même façon que le programme séquentiel, à l'exception de la 6è solution qui introduit ici l'optimisation de l'algorithme. La différence vient du fait que nous séparons les chiffres à calculer pour chaque fil d'exécution plutôt que de forcer chaque fil d'exécution à analyser le même tableau. Chaque fil d'exécution fonctionne de façon autonome puisqu'ils possèdent leurs propres index à explorer.

Dans 2 cas nous avons introduits les mutex afin d'en évaluer l'impact et, pour ce genre de cas, nous nous sommes vite rendus compte qu'il ne faisait que alourdir la performance. L'utilisation de mutex coûte cher, heureusement dans notre cas nous n'en n'avons pas besoin parce que notre algorithme n'a pas de collision de données.

Notre raisonnement voulait en effet que l'usage de mutex, conditions ou sémaphore pose problème s'il n'y avait qu'un seul traitement central à effectuer. Ainsi en séparant le travail par le nombre de fil d'exécution nous pouvions envisager une meilleure performance, l'optimisation étant faite principalement au niveau de l'algorithme.

Nous nous sommes rendus compte aussi de la limite des ordinateurs à traiter ce genre de cas, par exemple en y allant à 10000000000 données, cela faisait littéralement geler l'ordinateur et un redémarrage forcé était nécessaire.

Comme conclusion, la parallélisation avec des fils d'exécution est intéressante mais dans le cas qui nous concerne ce n'est peut-être pas l'outil idéal. Le problème ici est qu'initialiser chaque fil d'exécution coûte cher en terme de performance. En optimisant l'algorithme et en répartissant le travail entre chaque *thread*, nous pouvons obtenir une optimisation pertinente qui répondrait aux besoins.