St-Hilaire, Jimmy 111 020 393

Victorette, Giovanni 111 059 549

Equipe : 14

Travail Pratique 1

GIF-7104; Programmation parallèle et distribuée

Présenté à

Pr. Marc Parizeau

Le 31 jan 2014

INTRODUCTION

En utilisant le programme fourni par le professeur, nous avons débuté par optimiser le code séquentiel pour ensuite le paralléliser. Le programme devait aussi afficher à la console le total de nombres premiers trouvés, les nombres premiers trouvés ainsi que le temps d'exécution.

MÉTHODE

Notre première tentative d'optimisation était de séparer les nombres à exclure en autant de parties que de fils d'exécution. Cette optimisation ne s'est pas avéré efficace puisque certains threads effectuais beaucoup plus de travail que d'autres.

La première optimisation est basée sur la présentation des transparents du TP1 en classe. On a utilisé l'idée fourni dans les diapositives.

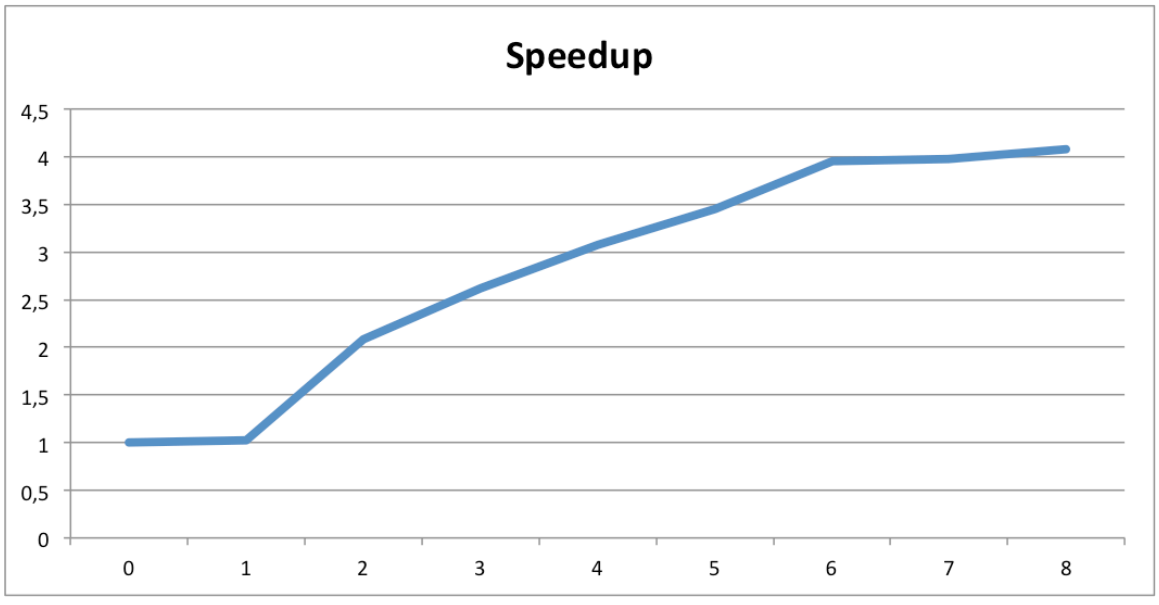
```
long lSquareRoot = sqrt(lMaxLimit);
for (int i = 3 ; i <= lSquareRoot ; i += 2) {
      if ((int)lArrayPrimes[i] == 0) {
            for (int j = i*i ; j <= lMaxLimit ; j += 2 * i) {
                  lArrayPrimes[j]++;
            }
      }

}
```

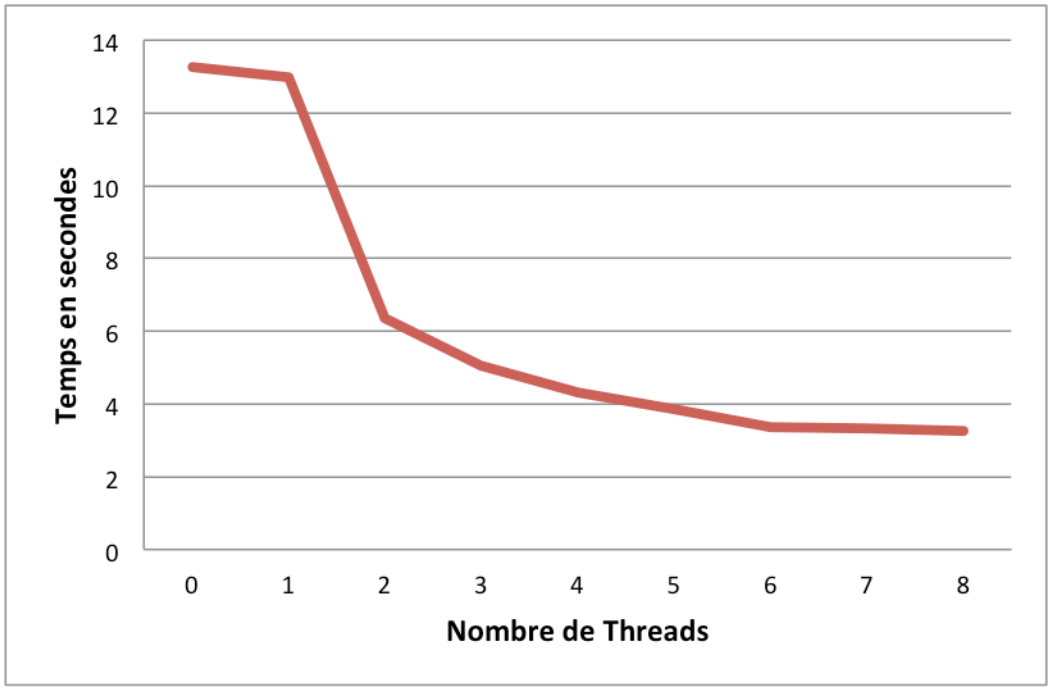
Pour la version finale avec fils d'exécution, nous avons quelque peu modifié le code. Les multiples de 2 sont un cas spécial qui ne sont pas de nombres premiers, nous avons donc exclus ces nombres avant même de séparer le travail en plusieurs fils d'exécution. Chaque thread utilise un concept de base de multiplication en commençant par 3, ce qui veut dire que le prochain nombre à exclure est le nombre premier actuel multiplié par lui même, donc «3 * 3». Pour faire l'incrémentation de base, un simple mutex a été utilisé, de cette façon, nous évitons d'effectuer la même exclusion sur plusieurs fils d'exécution.

```
do {
      pthread_mutex_lock(&nextbaselock);
      base = nextbase;
      nextbase += 2;
      pthread_mutex_unlock(&nextbaselock);
      if ((int)lArrayPrimes[base] == 0) {
            for (int i = base; i * base <= maxLimit; i += 2){
                  lArrayPrimes[i * base]++;
            }
      }
} while(base <= lSquareRoot);
```

RÉSULTATS

L'expérimentation a été faite sur colosse avec une limite maximale de 1 000 000 000 (1x10^9) nombres.





CONCLUSION

Malheureusement, le graphique de « SpeedUP » n'est pas une ligne droite, mais entre 1 et 6 threads on peut percevoir une bonne amélioration de performance. Nous n'avons pas pris le temps pour voir les contraintes de mémoire cache pour optimiser la performance et n'y profiter des options d'optimisation du compilateur. C'est une façon intéressante que peut nous aider à améliorer le code.

Afin de vérifier si le speedup pourrait être plus avantageux sur 8 coeurs, il faudrait pouvoir augmenter la taille du problême. Tenter le même exercise, mais pour 1 000 000 000 000 (1x10^12) nombres. Pour ce faire, il faudrait compresser la taille du tableau utilisé pour conserver les nombres qui sont premiers et utiliser des nombres de tailles arbitraires afin de pouvoir représenter des nombres plus grand que $2^{64}-1$.