GIF-4104 - TP 2

Équipe 1 : Vincent Commin & Louis Leenart

1 Introduction

Pour ce deuxième TP, nous avons du reprendre notre approche de parallélisation de recherche de nombre premiers, en utilisant cette fois **OpenMP** au lieu de **pthread**. Nous utilisons cependant encore une fois l'algorithme de *Miller-Rabin*, et plus précisément l'implémentation de Stigen Larsen disponible sur *Github*. Nous avons modifié légèrement cette implémentation afin de permettre la parallélisation ainsi qu'améliorer les performances. Enfin, notre algorithme supporte la recherche sur les grands notre, via la librairie *GMP*.

2 Notre approche

Dans ce TP, nous avons fait de notre mieux pour paralléliser notre algorithme en simplifiant un maximum la lecture du code, tout en donnant des performances satisfaisantes. L'utilisation d'OpenMP a permis de rendre le code beaucoup plus concis. L'algorithme que nous avons mis en place est le suivant :

```
premiers : vecteur de grands nombres
intervalles : vecteur de pair de grands nombres
nb_threads : entier
tours : entier

// Section parallélisée sur nb_threads
POUR i ALLANT DE 0 A size(intervalles) FAIRE
    local_premiers : vecteur de grand nombres
    pair : pair de grands nombres = intervalles[i]
    aleatoire : générateur de nombre premier

    POUR j ALLANT DE pair.borne_basse A pair.borne_haute FAIRE
        est_premier : booléen = calculer_premier(j, tours, aleatoire)
        SI est_premier ALORS
            AJOUTER j DANS local_premiers
        FIN SI
    FIN POUR

    FUSIONNER local_premiers DANS premiers
FIN POUR
```

Aussi, afin de maximiser les performances, nous utilisons une fusion des intervalles pour limiter la duplication des calculs. En effet, si deux intervalles se chevauchent, alors ils sont réunis en un seul. Nous nous sommes inspirés de l'approche évoqué par l'équipe 3 dans leur rapport, et avons implémenté notre propre algorithme adapté à notre structure de données. Nous avons rapidement remarqué que la mise en place de cet algorithme a impacté grandement sur l'intérêt de la parallélisation de certains fichiers. En effet, notre parallélisation consiste à répartir les intervalles par threads, or la fusion d'intervalles a réduits certains fichiers à un seul interval. Nous évoquerons cette difficulté plus en détails dans l'analyse. Aussi, l'algorithme utilisé pour la fusion des intervalles est le suivant :

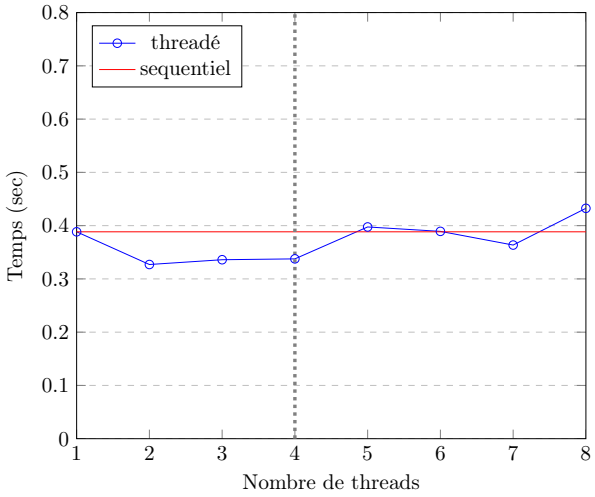
```
intervalles : vecteur de pair de grands nombres
resultat : vecteur de pair de grands nombres

TRI intervalles SELON (a.premier < b.premier)

POUR pair : pair de grands nombres DANS intervalles FAIRE
    SI (resultat EST VIDE) OU (resultat.dernier.second < pair.second) ALORS
        AJOUTER pair A resultat
    SINON
        resultat.dernier DEVIENT
            (resultat.dernier.premier,
            MAX(resultat.dernier.second, pair.second))
    FIN SI
FIN POUR
```

3 Machine utilisée pour les tests de performance

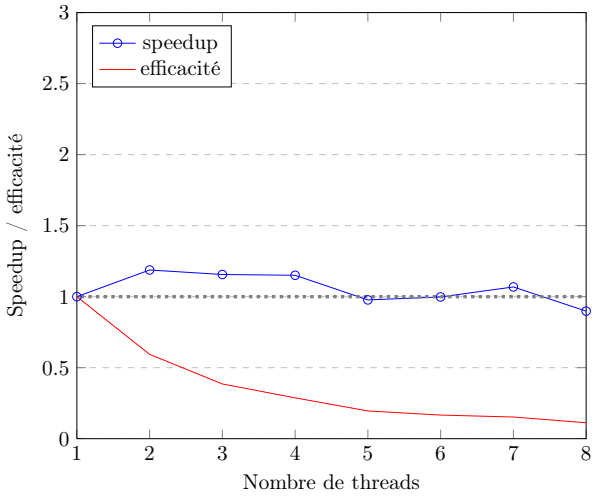
Modèle	intel i7-8550U
Architecture	x86_64
OS	Archlinux
Fréquence CPU	3.4GHz
Cœurs physiques	4
Cœurs logiques	8
Ram	16 Go, 2400 MT/s

4 Résultats obtenus

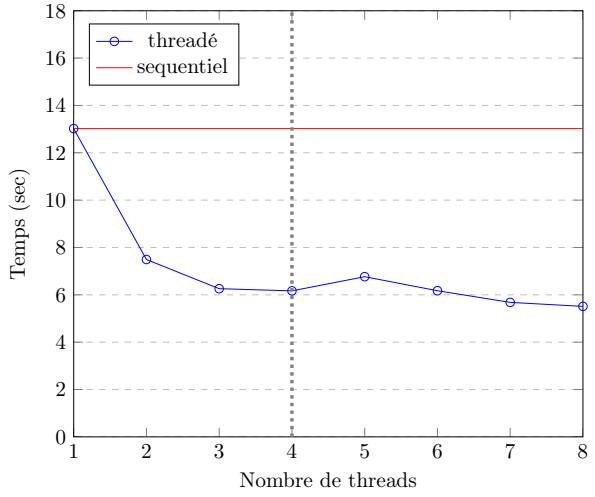
1_simple, temps vs nombre de threads



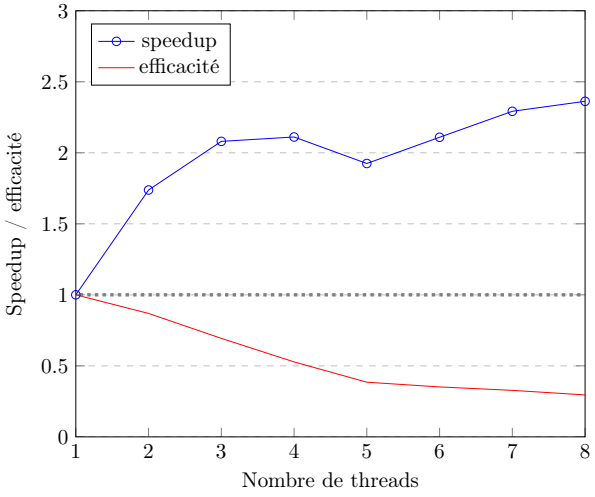
1_simple speedup & efficacité vs nombre de threads



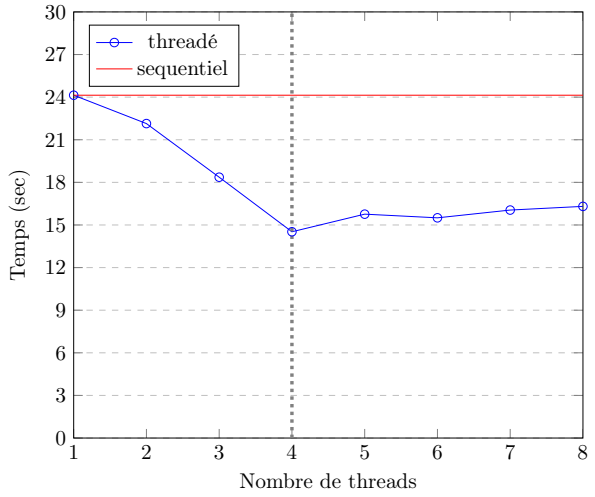
2_intervalles_courts, temps vs nombre de threads



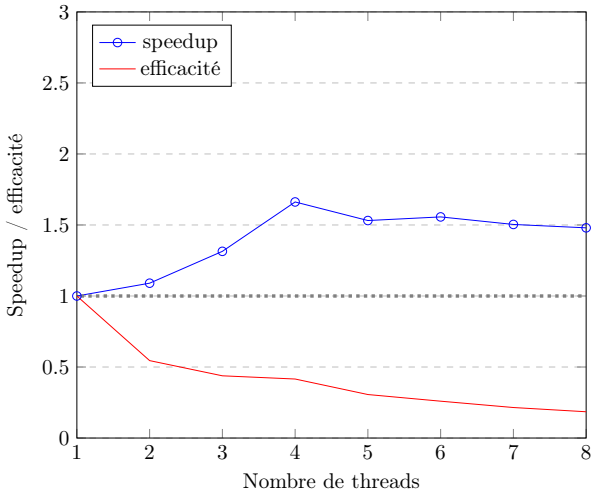
2_intervalles_courts, speedup & efficacité vs nombre de threads

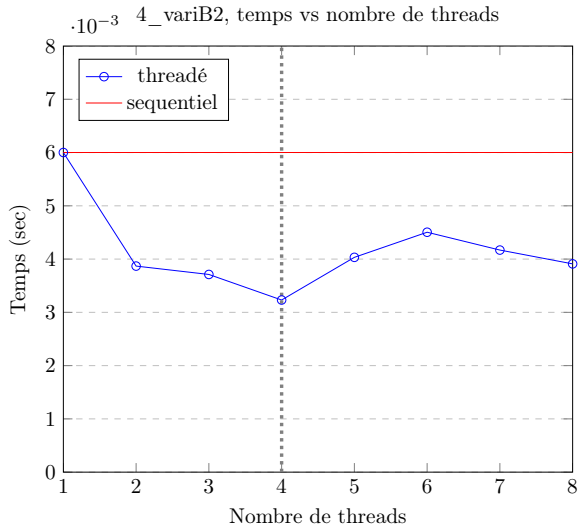


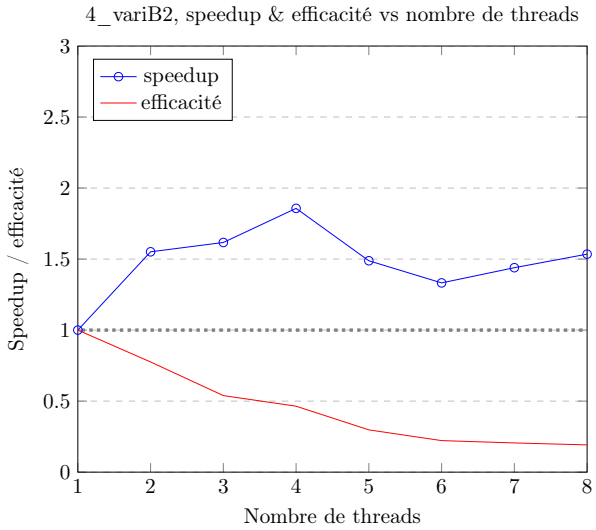
3_variB1, temps vs nombre de threads

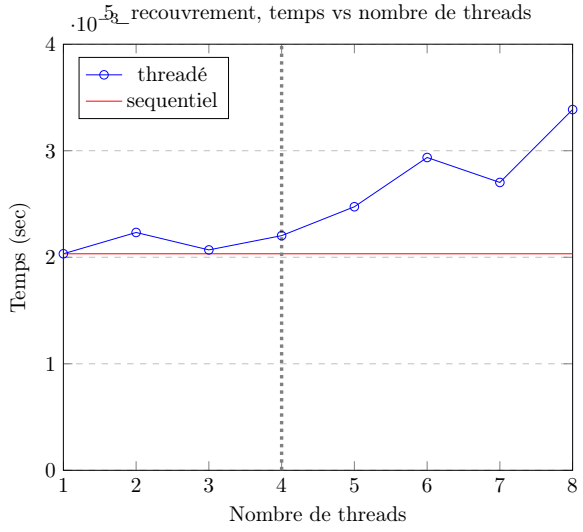


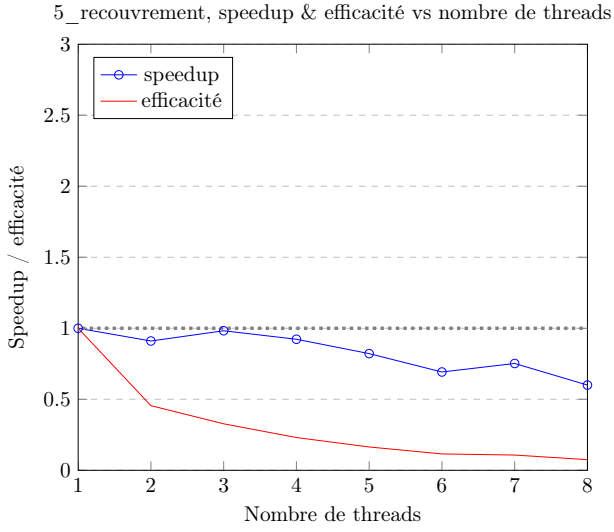
3_variB1, speedup & efficacité vs nombre de threads

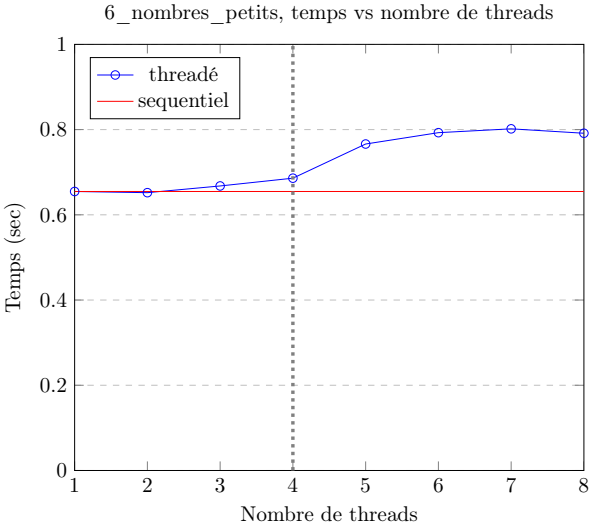


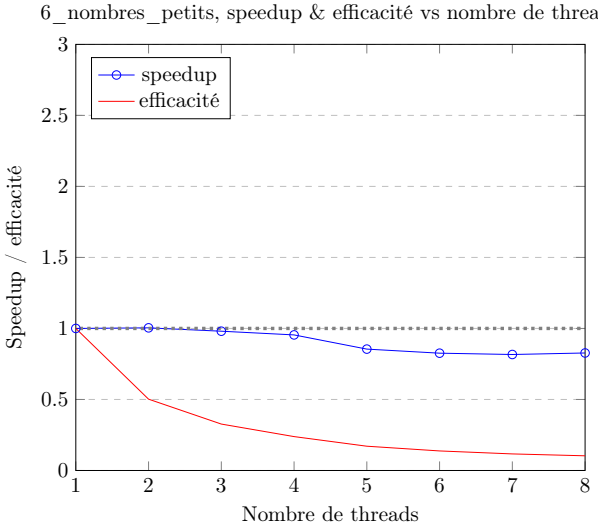

4_variB2, temps vs nombre de threads
$\cdot 10^{-3}$
threadé
sequentiel
Temps (sec)
Nombre de threads

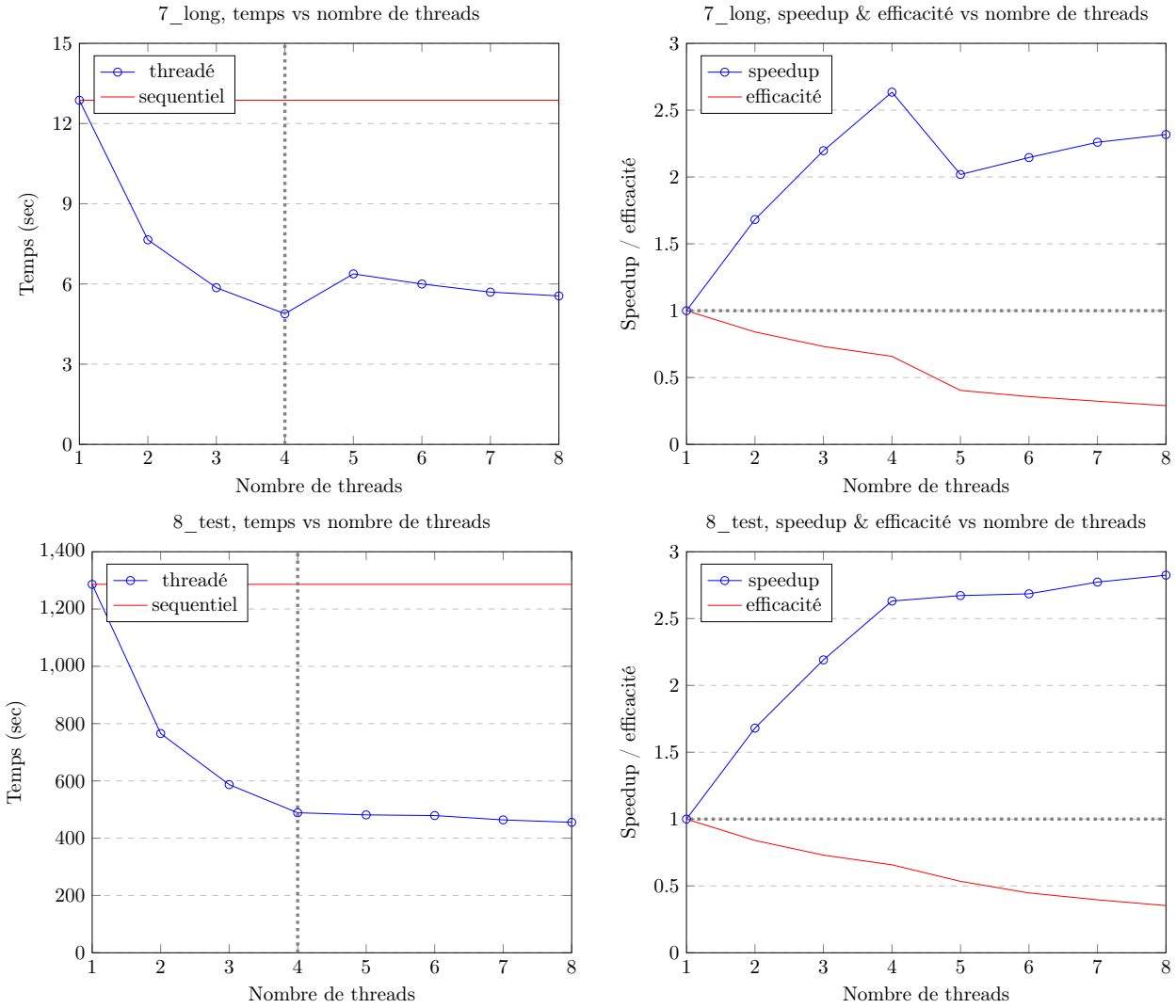

4_variB2, speedup & efficacité vs nombre de threads
speedup
efficacité
Speedup / efficacité
Nombre de threads


5_recouvrement, temps vs nombre de threads
$\cdot 10^{-3}$
threadé
sequentiel
Temps (sec)
Nombre de threads


5_recouvrement, speedup & efficacité vs nombre de threads
speedup
efficacité
Speedup / efficacité
Nombre de threads


6_nombres_petits, temps vs nombre de threads
threadé
sequentiel
Temps (sec)
Nombre de threads


6_nombres_petits, speedup & efficacité vs nombre de threads
speedup
efficacité
Speedup / efficacité
Nombre de threads



5 Analyse

Avec l'utilisation d'OpenMP, en comparaison avec pthread, nous avons remarqué que le développement et le debug d'une application multi-threadée est beaucoup plus facile. L'abstraction que propose OpenMP quant aux contraintes techniques qu'imposent la parallélisation permet d'avoir d'un côté du code performant (si correctement implémenté évidemment), mais aussi du code lisible, ce qui est plus compliqué avec pthread.

Comme expliqué précédemment, nous avons mis en place un algorithme de fusion des intervalles afin de ne pas dupliquer les calculs sur des nombres contenus dans deux intervalles différents. En l'appliquant avant le calcul de primalité sur les intervalles, nous avons remarqué que pour certains fichiers, les intervalles proposés se chevauchent tous, formant alors un seul intervalle final. Étant donné la structure de notre parallélisation (chaque intervalle est donné à un et un seul thread, jusqu'à ce que tous les intervalles soient traités), la parallélisation est bornée par le nombre d'intervalles. De ce fait, les-dits fichiers, (1, 5 et 6) ont des résultats assez peu intéressants étant donné qu'il sont traités de manière séquentielle. Leurs temps de calcul est presque constant par rapport au nombre de threads, comportant un peu de bruit. Leurs speedup tendent vers 1 et l'efficacité décroît rapidement pour tendre vers 0 à un rythme beaucoup plus important que les cas mieux threadés.

Notre objectif pour ce TP était de minimiser le temps de calcul pour le fichier 8, puisqu'il représente pour nous l'utilisation la plus propice de la parallélisation de calculs. En effet, les autres fichiers ayant un temps de traitement suffisamment faible, le temps brut gagné par parallélisation est presque négligeable. Concernant le fichier 8, le temps de traitement initial est d'environ 21 minutes, comporte de nombreux

intervalles, plus ou moins grands. On remarque que le temps de calcul passe à 8 minutes pour 4 threads (la machine utilisée possède 4 cœurs physiques), donc presque une division par 3. La qualité de ces résultats est reflétée par le speedup et l'efficacité. En effet, un speedup de 2.6 (pour 4 threads) indique que la parallélisation répond bien à ce problème. L'efficacité, valant 0.6 (pour 4 threads) laisse entendre le même résultat (l'idéal de l'efficacité étant 1).

Aussi, même si nous avons développé un algorithme pour réduire au plus le temps de traitement du fichier 8, nous pouvons aussi remarquer que celui-ci donne de bons résultats pour les autres fichiers possédant plusieurs intervalles ne se recouvrant pas. Par exemple, pour les fichiers 2 et 7, on remarque des résultats similaires à ceux obtenus pour le fichier 8.

6 Conclusion

Notre approche a ses limites, et pourrait être améliorée sur plusieurs points. Nous avons par ailleurs fait des choix qui sont contestables. Déjà, la mise en place de la fusion des intervalles est intéressant, cependant cela ne répond pas vraiment au problème demandé. Même si les performances sont évidemment améliorées, certains fichiers tests sont rendus inutilisables (aussi, cet algorithme de fusion n'est pas des plus performant, avec une complexité de $O(n)$, n le nombres d'intervalles). Un deuxième point que nous avons rencontré est notre approche pour la parallélisation. Étant donné le fonctionnement d'OpenMP, nous n'avons jusque là par trouvé de moyen de paralléliser la boucle d'itération sur chacun des éléments de chaque intervalle. Paralléliser cette section pourrait donner des performances bien plus importantes, notamment pour le traitement de peu d'intervalles contenant beaucoup d'éléments. De plus, nous n'avons pas sélectionné de stratégie d'ordonnancement via OpenMP, une étude des différentes possibilités ainsi qu'une comparaison des performances de ces derniers nous aurait sûrement permis d'optimiser un peu plus notre algorithme. Enfin, d'un point de vue plus technique, le conteneur que nous avons utilisé est `std::vector`, dont l'implémentation n'est pas particulièrement orientée vers la performance (et l'utilisation qu'on en fait n'aide pas). L'utilisation d'une autre structure, ou même l'implémentation d'un conteneur, serait aussi un moyen d'améliorer les performances.

Nous avons aussi remarqué que la parallélisation est entre autre limitée par le nombre de threads que peut gérer le CPU en même temps. Pour améliorer le temps de traitement, l'utilisation d'un protocole de communication pour distribuer les tâches à travers un réseau d'ordinateurs serait idéal. L'implémentation de cet algorithme avec MPI permettra sans doute diviser encore plus le temps de traitement du fichier 8.