Rapport du TP3 : Systèmes concurrents

Parallèles

Myriam ROBBANA

Decembre 2024

Table des matières

Table des figures

1 Calcul du maximum d'un tableau.

Le but est de trouver l'élément maximal du tableau.

1.1 Comparaison des performances des solutions.

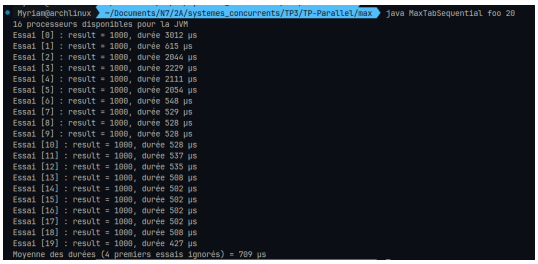


FIGURE 1 – java MaxTabSequential foo 20

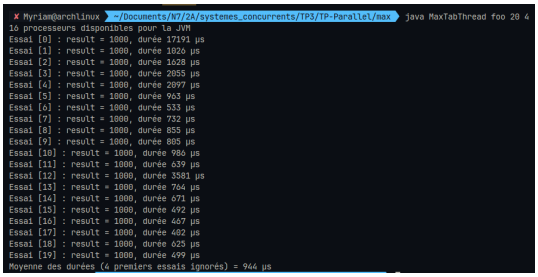


FIGURE 2 – java MaxTabThread foo 20 4

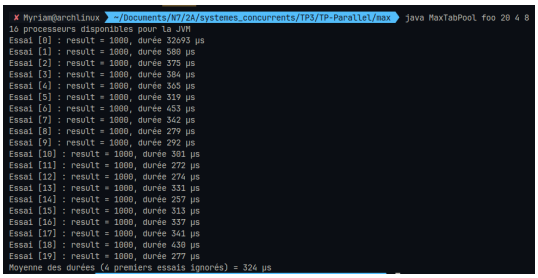


FIGURE 3 – java MaxTabPool foo 20 4 8

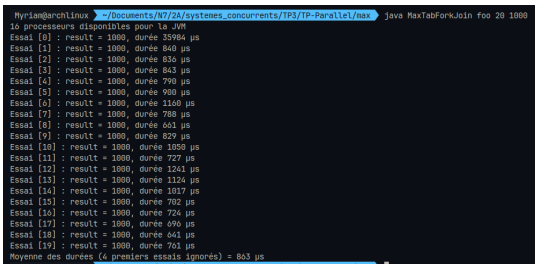


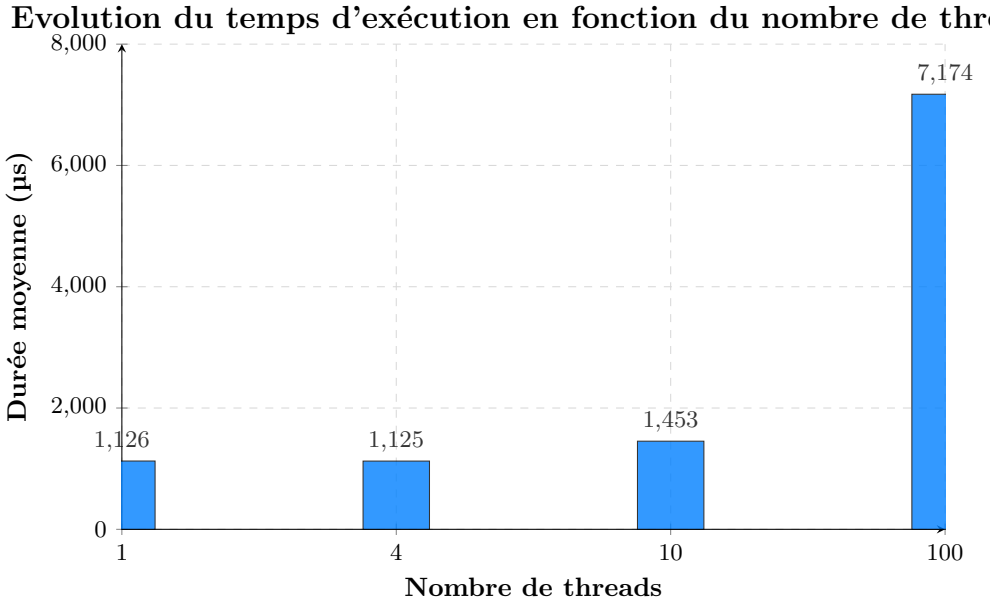
FIGURE 4 – java MaxTabForJoin foo 20 1000

Méthode	**Temps moyen (µs)**	**Remarque**
MaxTabSequential	709	Séquentiel simple
MaxTabThread (4 threads)	944	Threads multiples
MaxTabPool (4 threads, 8 tâches)	324	Pool de threads
MaxTabForkJoin (1000 tâches)	863	Fork-Join

TABLE 1 – Comparaison des performances des différentes méthodes

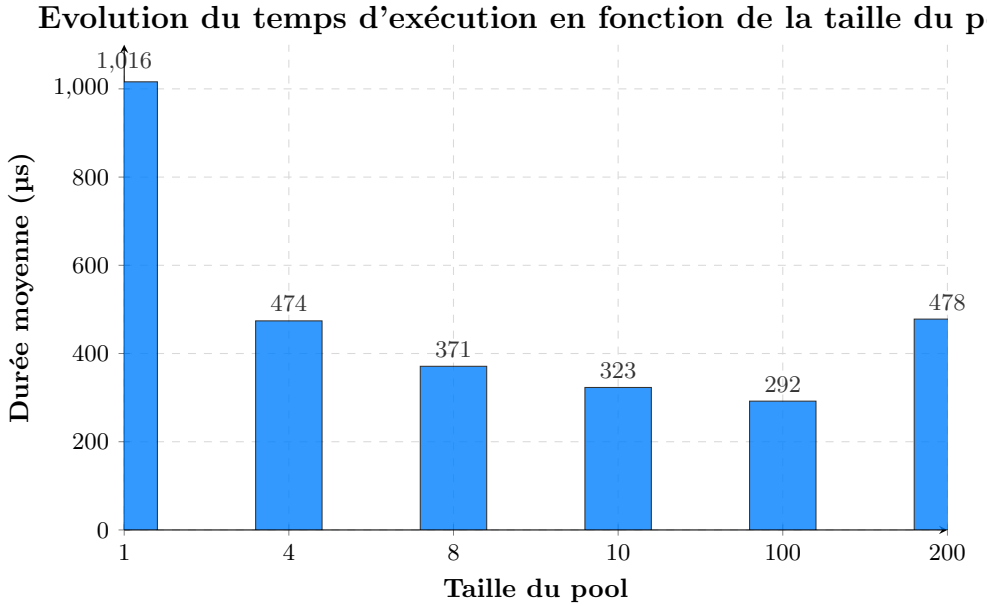
On remarque que le résultat est bien cohérent avec le tableau crée, et que le maximum donné pour chacune des versions est bien égale à 1000. Cependant, la méthode $MaxTabPool$, avec un temps moyen de 324 µs, est la plus performante, tandis que la méthode $MaxTabThread$, avec un temps moyen de 944 µs, est moins efficace.

1.2 Etude de la variation du nombre de threads, de lataille du pool et du seuil d'arrêt.

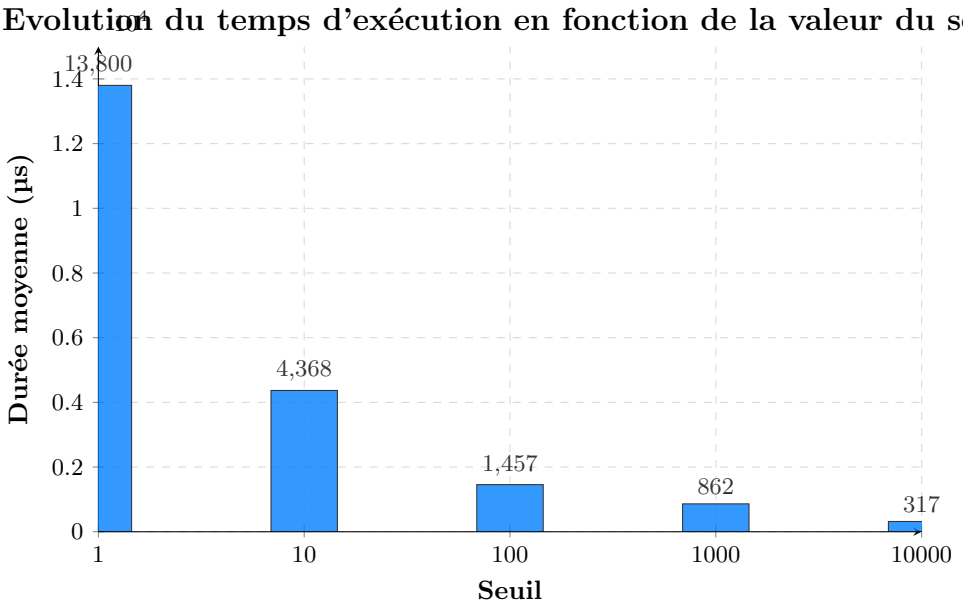


Logiquement, plus on augmente le nombre de threads, plus la durée d'exécution devrait diminuer. Cependant, au-delà d'un certain seuil, créer un trop grand nombre de threads devient contre-productif. En effet, le temps et les ressources nécessaires pour gérer ces threads dépassent les gains de parallélisme, rendant l'exécution plus longue que si l'on avait utilisé un nombre réduit de threads. Le graphe ci-dessus permet de mettre ce phénomène en exergue.

L'objectif est donc de trouver un équilibre : le nombre optimal de threads qui minimise la durée moyenne. Dans les exemples que j'ai réalisés, 4 threads semblent offrir un bon compromis !



On remarque que plus le taille de pool augmente et plus la durée d'exécution diminue jusqu'à un certain point. En effet, au-delà d'une taille de pool égale à 100, les performances diminuent légèrement, comme on peut le voir avec une augmentation du temps moyen pour une taille de pool égale à 200. Cela s'explique par le fait qu'augmenter la taille du pool au-delà d'une certaine limite n'apporte plus d'amélioration et peut même entraîner une légère perte d'efficacité. Le meilleur résultat dans notre exemple, est obtenu avec une taille de pool autour de 100, où le temps moyen d'exécution est le plus court.



On observe que plus le seuil est grand et plus le temps d'exécution diminue. Et pour un seuil assez élevé on atteint une durée moyenne d'exécution relativement rapide.

1.3 Analyser les résultats

L'optimisation du nombre de threads et de la taille du pool est importante pour améliorer les performances et diminuer la durée moyenne d'exécution. Bien qu'un nombre croissant de threads et une taille de pool plus grande réduisent le temps d'exécution, il existe un seuil au-delà duquel les gains deviennent négligeables, voire contre-productifs. Dans notre exemple, une taille de pool d'environ 100 et un nombre optimal de 4 threads offrent les meilleures performances, avec un temps d'exécution réduit de manière significative. QUand au seuil, plus ce dernier est élevé et plus l'exécution est rapide.

La version la plus efficace semble donc être celle qui prend en compte ces 3 paramètres dont le seuil pour minimiser un maximum la durée d'exécution. Cette version est `MaxTabForkJoin.java`

Les différentes comparaisons réalisées montrent les performance

Les fichiers fournis pour cette partie sont : `MaxTabThread.java, MaxTabPool.java, MaxTabForkJoin.java`.

2 Comptage des éléments dans un intervalle.

Le but est de compter le nombre de valeurs dans le tableau entre 0 et 9 (constantes VMIN/VMAX arbitrairement fixées).

2.1 Comparaison des performances des solutions.

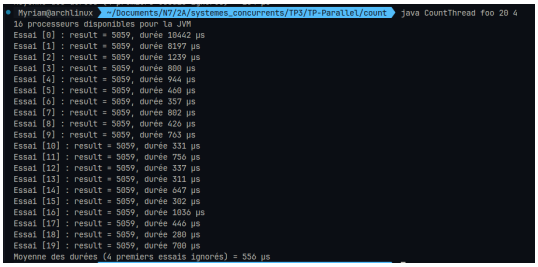


FIGURE 5 – java CountSequential foo 20

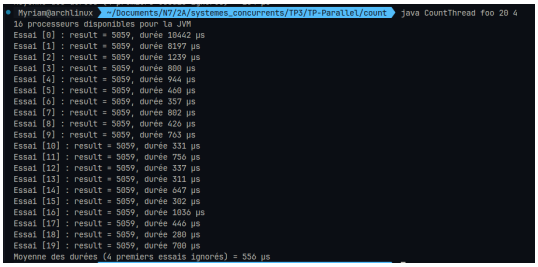


FIGURE 6 – java CountThread foo 20 4

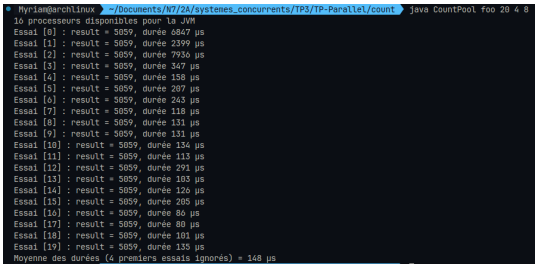


FIGURE 7 – java CountPool foo 20 4 8

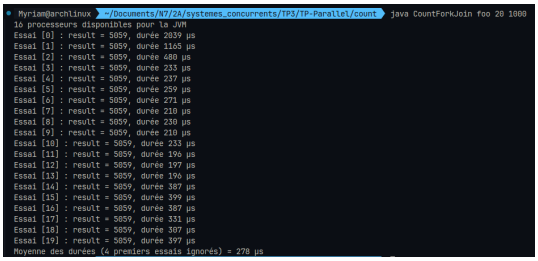


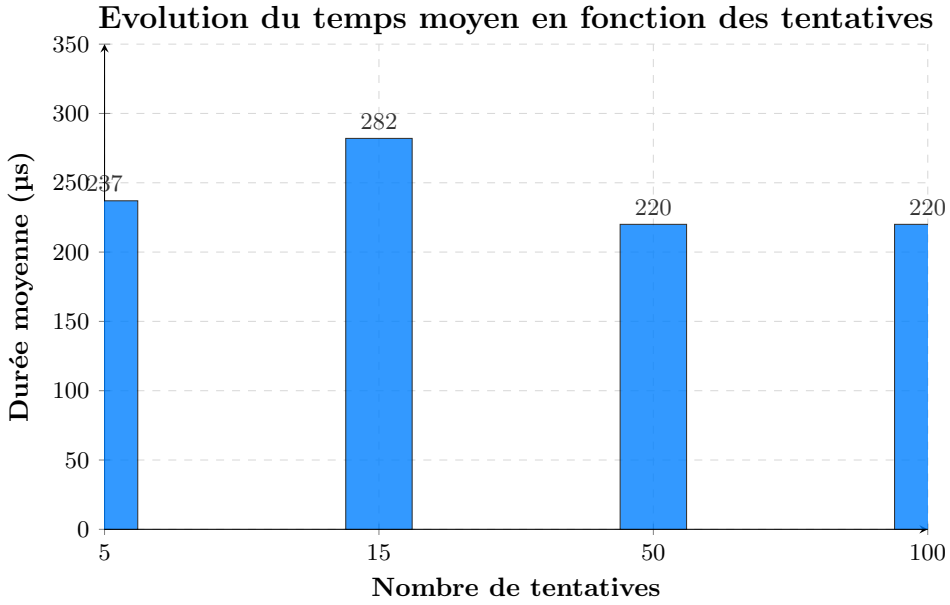
FIGURE 8 – java CountForkJoin foo 20 1000

Méthode	**Temps moyen (µs)**	**Remarque**
CountSequential	234	Exécution séquentielle simple
CountThread (4 threads)	556	Threads multiples
CountPool (4 threads, 8 tâches)	148	Pool de threads
CountForkJoin (1000 tâches)	278	Fork-Join

TABLE 2 – Comparaison des performances des différentes méthodes "Count" en fonction du temps moyen

On remarque que le résultat est bien cohérent avec le tableau crée, et que le nombre de valeur dans le tableau entre 0 et 9 est toujours de 5059. Cependant, tout comme dans la première partie, la méthode *CountPool*, avec un temps moyen de 148 µs, est la plus performante, tandis que la méthode *CountThread*, avec un temps moyen de 556 µs, est moins efficace.

2.2 Etude de la variation du nombre de threads, de lataille du pool et du seuil d'arrêt.

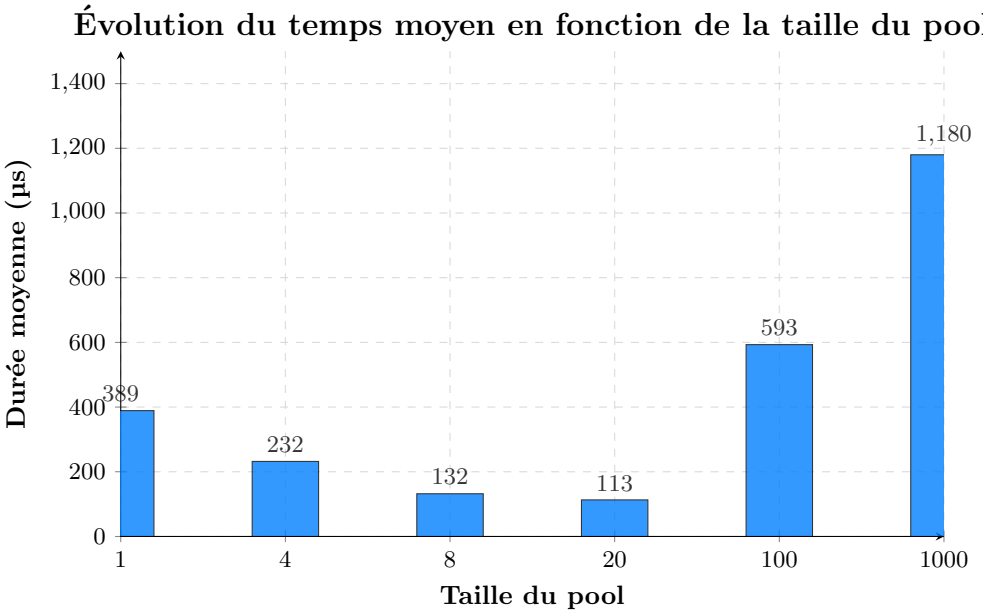


Les résultats montrent une tendance générale où la durée moyenne augmente légèrement au début, passant de 237 µs à 282 µs entre 5 et 15 tentatives. Cependant, à partir de 50 tentatives et au-delà, la durée reste stable autour de 220 µs. Cela suggère qu'après un certain nombre de tentatives, l'ajout de plus d'éléments dans le pool n'a plus d'impact significatif sur la durée d'exécution.

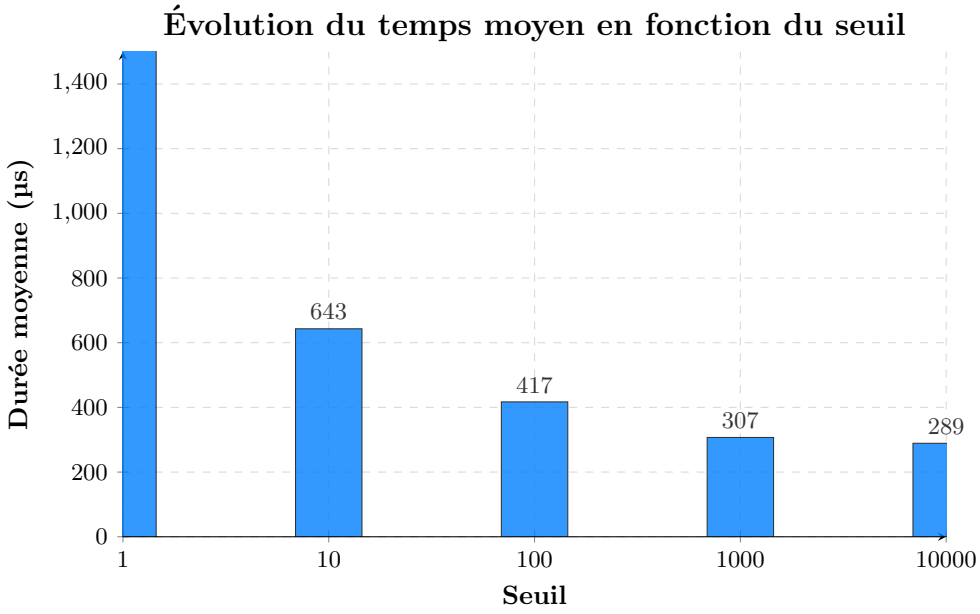
Nombre de threads	Durée moyenne (µs)
1	822
5	442
20	908
100	3663
1000	35682

TABLE 3 – Evolution du temps moyen en fonction du nombre de threads

On observe une augmentation significative du temps moyen en microsecondes avec l'augmentation du nombre de tentatives. En effet, à mesure que le nombre de tentatives passe de 1 à 1000, la durée moyenne augmente, passant de 822 µs à 35 682 µs, ce qui indique que le processus devient de plus en plus lent et coûteux en ressources avec le nombre de threads ou d'opérations simultanées.



Les résultats montrent une réduction significative des durées moyennes à mesure que le nombre de tentatives augmente, indiquant que le parallélisme des tâches améliore l'efficacité du calcul. Cependant, à partir d'une certaine taille de pool, la durée d'exécution auglente considérablement. Il existe donc une taille de pool optimale, de manière analogue à la première partie.



De manière identique à la première partie, plus on augmente le seuil et plus l'exécution est rapide.

2.3 Analyser les résultats

L'analyse des différentes méthodes pour le comptage des éléments montre que les solutions parallèles, comme l'utilisation de nombreux threads, offrent de meilleures performances que la méthode séquentielle. Toutefois, un trop grand nombre de threads peut entraîner une surcharge, réduisant les gains de performance. La solution avec un pool de threads optimisé, combinée à un seuil bien choisi, permet d'obtenir les meilleurs résultats. La méthode `CountForkJoin.java` s'est avérée la plus rapide, offrant un bon compromis entre performance et précision.

Le fichier fourni pour cette partie est : `CountSequential.java, CountThread.java, CountPool.java, CountForkJoin.java`.