Rapport

1. **MaxTabSequential:**
    - Cette version est séquentielle, elle ne fait pas usage du parallélisme.
    - La moyenne des durées d'exécution est d'environ 671 µs.
2. **MaxTabThread:**
    - Cette version utilise le multithreading avec 4 threads.
    - La moyenne des durées d'exécution est d'environ 536 µs.
3. **MaxTabPool:**
    - Cette version utilise un pool de threads avec 4 threads.
    - La moyenne des durées d'exécution est d'environ 415 µs.
4. **MaxTabForkJoin:**
    - Cette version utilise Fork-Join avec un seuil de 1000 (c'est-à-dire que la tâche est décomposée en sous-tâches jusqu'à ce que la taille de la tâche soit inférieure à 1000).
    - La moyenne des durées d'exécution est d'environ 1509 µs.

Analyse:

- La version séquentielle est la plus lente en termes de durée d'exécution.
- Les versions utilisant le multithreading (MaxTab Thread) et le pool de threads (MaxTabPool) montrent des améliorations significatives en termes de rapidité par rapport à la version séquentielle.
- La version utilisant Fork-Join (MaxTabForkJoin) est plus lente que les versions multithreading et pool de threads dans cet exemple particulier.

Facilité d'écriture:

- En général, les versions utilisant le multithreading et le pool de threads sont souvent plus simples à écrire et à comprendre que les versions utilisant Fork-Join, en raison de la nature du modèle de programmation.

Efficacité:

- L'efficacité dépend de plusieurs facteurs tels que la taille du tableau, le nombre de cœurs disponibles, la complexité de la tâche, etc.
- Dans cet exemple, le multithreading avec un pool de threads semble être la solution la plus efficace en termes de durée d'exécution.

Remarques:

- Il est important de noter que les performances peuvent varier en fonction de la machine sur laquelle le programme est exécuté et des conditions d'exécution.
- Le fait que le résultat soit toujours "1000" dans les sorties semble être une anomalie. Il serait bon de vérifier si l'algorithme retourne réellement le résultat correct dans toutes les versions.

En conclusion, choisir la meilleure approche dépend souvent du contexte spécifique de l'application, des caractéristiques du problème, et des ressources matérielles disponibles.

Count :

Les résultats montrent que CountSequential prend plus de temps par essai par rapport aux autres versions, et ce temps ne diminue pas de manière significative lorsque le nombre d'essais augmente. Cela indique que cette version séquentielle ne bénéficie pas du parallélisme.

En revanche, CountThread, CountPool, et CountForkJoin montrent des réductions significatives du temps d'exécution par essai. Parmi ces trois versions parallèles, CountPool semble être la plus efficace en termes de temps d'exécution moyen par essai.

En ce qui concerne la facilité d'écriture, cela peut dépendre des préférences personnelles. Les approches parallèles nécessitent généralement une gestion plus complexe des threads ou des tâches, ce qui peut rendre le code plus difficile à comprendre. Cependant, l'utilisation de bibliothèques comme Fork-Join ou Executors dans Java facilite grandement la mise en œuvre de la parallélisation.

En résumé, si l'efficacité est la priorité et que la complexité supplémentaire de la gestion des threads n'est pas un problème, les versions parallèles (CountThread, CountPool, et CountForkJoin) sont à privilégier. Si la simplicité du code est plus importante, la version séquentielle peut être préférable malgré ses performances moins bonnes.