Rapport Labo 2: Implémentation du Bogosort déterministe avec threads en C++

Auteurs : Nathan Füllemann, Maxime Regenass

Table des matières

[Introduction au problème 1](#_Toc1961729631)

[Choix d'implémentation 1](#_Toc174027698)

[2.1 Création de la méthode Bogosort 1](#_Toc1038427418)

[2.2 Répartition des tâches entre les threads 2](#_Toc1432147959)

[Tests effectués 2](#_Toc1844783541)

[3.1 Tests unitaires 2](#_Toc1062533675)

[3.2 Problèmes rencontrés 2](#_Toc1623297573)

[Discussion des résultats 2](#_Toc296520520)

[4.1 Performance 3](#_Toc2101453559)

[4.2 Efficacité du parallélisme 3](#_Toc2136401622)

[Conclusion 3](#_Toc1970236540)

# Introduction au problème

Le problème auquel nous avons été confrontés durant ce laboratoire consistait à implémenter un **Bogosort déterministe** dans une application C++ partiellement développée. Le Bogosort, étant un algorithme de tri notoirement inefficace qui fonctionne en générant des permutations aléatoires jusqu'à ce que le tableau soit trié, devait être adapté ici pour utiliser des **threads**, permettant ainsi de répartir les permutations entre plusieurs threads exécutant simultanément le tri. Cela visait à accélérer la recherche de la permutation triée, en utilisant le parallélisme.

Les étapes principales de notre implémentation comprenaient :

* La création des threads et d'un vecteur de résultats,
* Le lancement des threads avec la fonction Bogosort,
* L’arrêt des threads et la récupération du tableau trié,
* Le retour du tableau une fois celui-ci trié.

# Choix d'implémentation

Pour réaliser le **Bogosort déterministe**, nous avons commencé par examiner et comprendre le pseudo-code des permutations fourni. La première étape consistait donc à écrire une fonction capable de générer toutes les permutations possibles d’un tableau, de manière à ce que l'algorithme de tri puisse les utiliser.

### **2.1 Création de la méthode Bogosort**

Une fois la fonction de permutation implémentée, nous avons conçu une méthode **Bogosort** qui fait appel à cette dernière. Cette méthode prend plusieurs paramètres :

* Le tableau à trier,
* Un objet partagé pour la gestion de l’état des threads,
* Un entier k pour identifier quelle permutation est en cours de traitement,
* Le nombre de threads à créer pour trier le tableau.

La méthode Bogosort contient une boucle while qui continue jusqu'à ce que le tableau soit trié. L'arrêt de cette boucle est géré par une **variable partagée** indiquant qu’un des threads a trouvé la permutation correcte. Grâce à cette approche, tous les threads s’arrêtent dès que l'un d'entre eux réussit à trier le tableau.

### **2.2 Répartition des tâches entre les threads**

Pour garantir que les différents threads explorent des permutations distinctes, nous avons introduit une **variable k de départ** pour chaque thread, initialisée en fonction de l’identifiant du thread. Ensuite, cette variable est incrémentée selon le nombre de threads, de sorte que chaque thread traite des permutations différentes. Cela permet de répartir la charge de travail de manière efficace et évite que deux threads ne testent la même permutation.

# Tests effectués

### **3.1 Tests unitaires**

Pour commencer, nous avons tester notre programme en rentrant des valeurs tests. Grâce à ces tests, nous avons pu détecter un problème notamment que suivant quelles données on entrait.

### **3.2 Problèmes rencontrés**

Lors des premiers tests, nous avons découvert un problème : le programme ne parvenait pas toujours à trouver une permutation triée, en particulier lorsque certaines données étaient entrées. Cela s'expliquait par le fait que notre implémentation ne vérifiait pas encore si une permutation avait déjà été testée. En conséquence, certaines permutations étaient testées plusieurs fois, ce qui ralentissait le processus et empêchait parfois d'explorer toutes les permutations possibles avant de trouver la bonne solution.

Après avoir identifié ce problème, nous avons introduit la vérification des permutations déjà testées, ce qui a grandement amélioré l’efficacité de l'algorithme. Cette modification a permis de garantir que toutes les permutations possibles étaient explorées une seule fois.

## **Discussion des résultats**

### **4.1 Performance**

L’utilisation des **threads** a permis de paralléliser la recherche de la permutation triée, ce qui a considérablement amélioré la rapidité de l'algorithme. Toutefois, la nature inefficace du Bogosort reste un facteur limitant, car même avec plusieurs threads, le tri des tableaux de grande taille devient rapidement impraticable. Le nombre de permutations croît exponentiellement avec la taille du tableau, rendant cet algorithme inadapté pour des données volumineuses.

### **4.2 Efficacité du parallélisme**

Le **parallélisme** a permis de réduire le temps de traitement, en répartissant la charge entre les threads. Cependant, le gain de performance dépendait du nombre de threads et de la taille du tableau. Dans certains cas, l'overhead lié à la gestion des threads et à la synchronisation a annulé une partie des bénéfices apportés par le parallélisme. Il est donc crucial de bien ajuster le nombre de threads en fonction de la taille du tableau pour optimiser les performances.

## **Conclusion**

En conclusion, ce laboratoire nous a permis de mettre en œuvre un **Bogosort déterministe** en utilisant des threads pour paralléliser la recherche de permutations. Bien que l’algorithme reste inefficace par nature, l’utilisation de threads a permis d’accélérer le processus et d'optimiser l’utilisation des ressources du processeur. Les tests effectués ont démontré que la gestion des permutations déjà testées était essentielle pour éviter les répétitions et améliorer l’efficacité globale de l'algorithme.

Ce projet nous a aussi permis de renforcer nos compétences en **programmation multithread** et en **synchronisation des threads**, des aspects cruciaux pour les applications nécessitant des performances élevées.