Intitulé du projet :

Code génétique, distance de Hamming et matrices stochastiques

Réalisé par:

Katia CHIKHI

Année académique 2022-2023

Table de matières

1 Résumé de l'article

L'article propose une méthode pour étudier les propriétés des séquences génétiques, en utilisant la représentation en code Gray à 2 bits C = 00, U = 10, G = 11 et A = 01 pour générer des matrices basées sur le code génétique et en appliquant la distance de Hamming sur ces matrices générées pour créer des matrices numériques.

Trois tableaux ont été introduits: le premier présente les codons disposés selon un modèle carré de 8x8 avec le nombre de liaisons hydrogène. Les lignes et colonnes sont étiquetées en code Gray standard, le deuxième est un tableau bi-périodique et le troisième est généré sur la base d'un arbre 4-aire. La stochasticité de ces trois tableaux est examinée, les fréquences des distances de Hamming et des configurations matricielles sont déterminés.

En outre, il est mentionné que la distance de Levenshtein est une méthode plus sophistiquée pour mesurer les différences entre les chaînes de caractères.

Mots-clés : - Code génétique, code Gray, distance de Hamming, matrices stochastique, Distance de Levenshtein.

2 Analyse et optimisation du code

2.1 Introduction

Dans ce document, nous allons analyser et optimiser un code qui consiste à calculer la distance de Hamming de deux séquences données. Tous les résultats de performance seront en Nanoseconde et en GiB/s, avec les compilateurs gcc, icc et icx.

2.2 Optimisations

La figure ci-dessous montre les performances du code principal avec les différents compilateurs gcc, icc et icx.

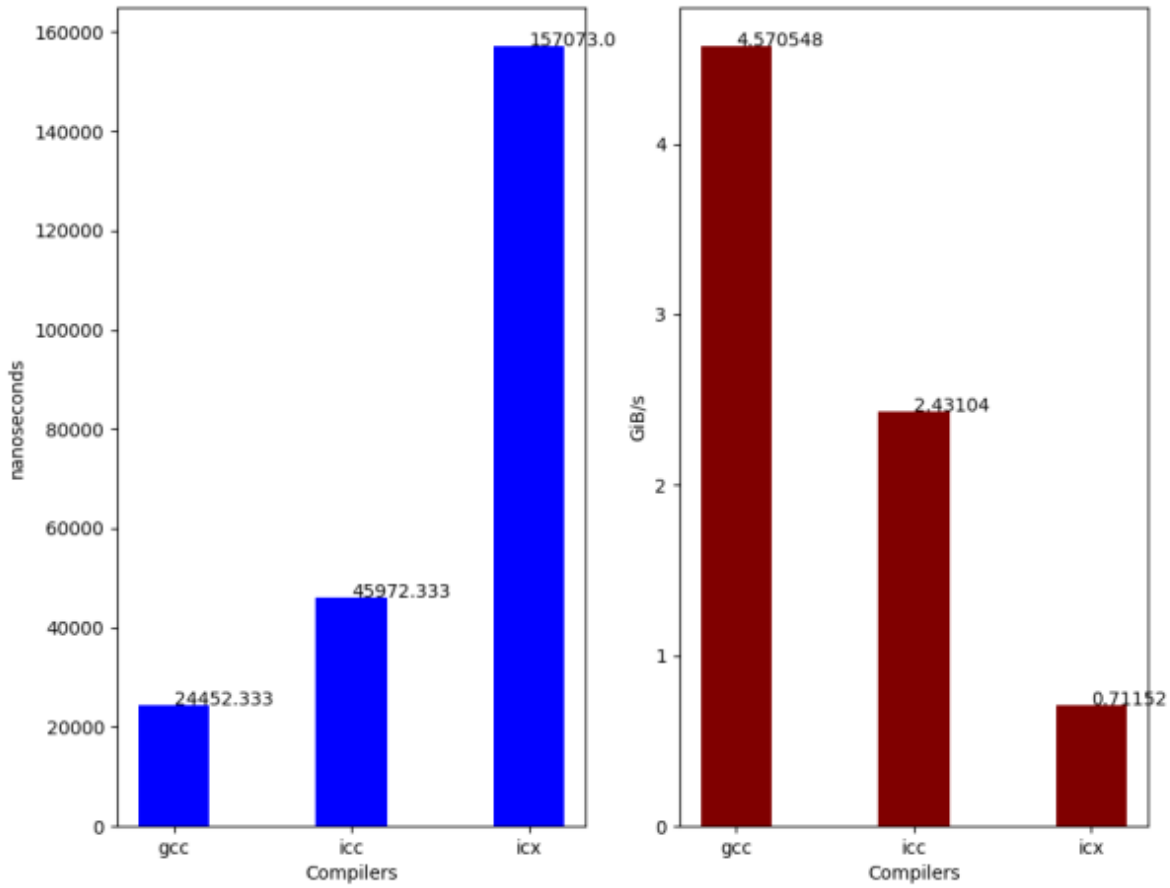

Figure 1: Performances du code principal

2.2.1 Alignement de la mémoire

L'alignement de la mémoire est la manière dont les données sont organisées et accessibles dans la mémoire, cela peut améliorer les performances en réduisant les coûts d'accès à la mémoire. L'alignement est fait en ajoutant les différents flags pour les différents compilateurs:

- **gcc** -falign-functions=16 -falign-loops=16 -falign-jumps -falign-labels
- **icc** -align -16

- **icx** -align

```
OFLAGS=-march=native -falign-functions=16 -falign-loops=16 -falign-jumps -falign-labels -O1 -fopt-info-all=dist.gcc.optrpt
INTEL_FLAGS=-xhost -align -16 -O1 -qopt-report
INTELx_FLAGS=-xhost -align -O1 -qopt-report
```

Figure 2: Alignement de la mémoire pour les différents compilateurs

La figure ci-dessous montre les performances du code avec l'alignement de la mémoire.

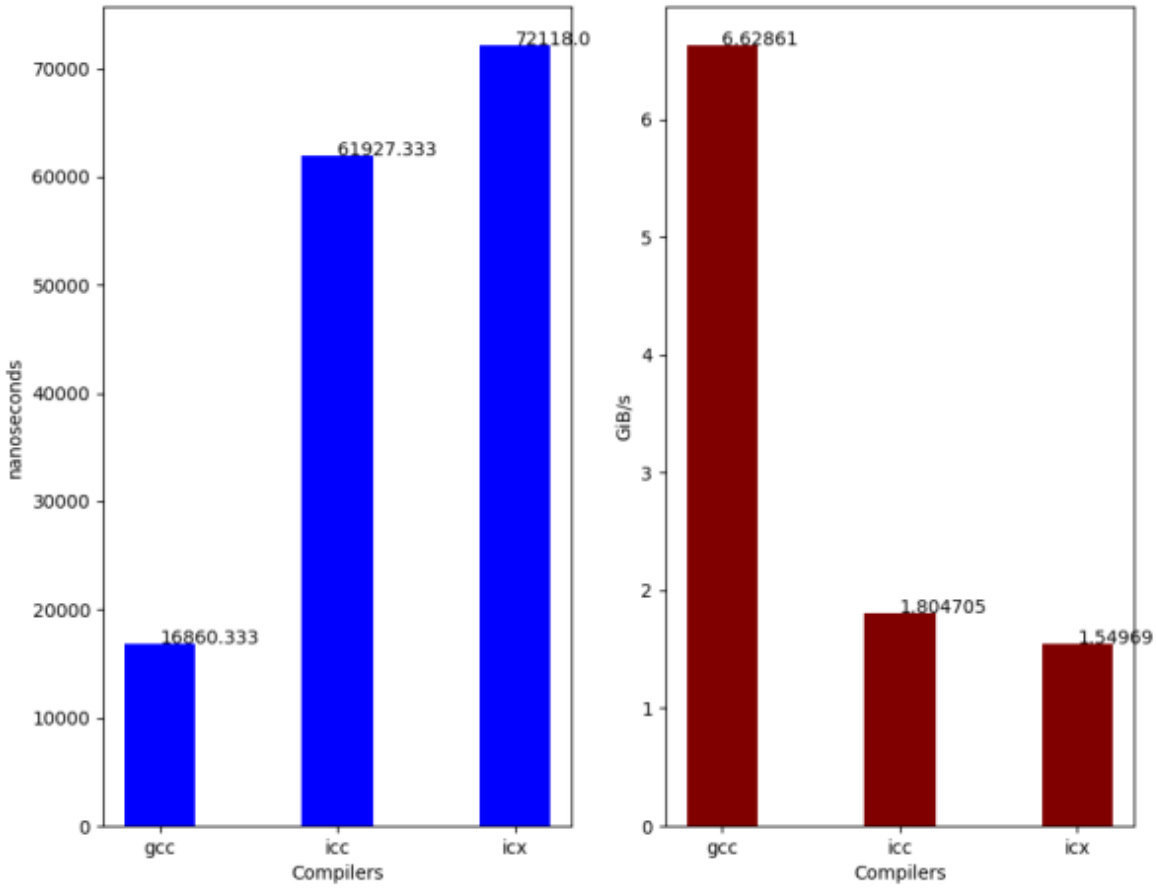

Figure 3: Alignement de la mémoire

2.2.2 Popcount

Pour savoir si le popcount est optimal, le popcount original est comparé avec un popcount qui utilise une table de correspondance car utiliser cette table est considéré comme le moyen le plus rapide pour faire un calcul car chercher un espace mémoire est plus rapide que de faire un calcul.
La figure ci-dessous montre les performances avec un popcount qui utilise une table de correspondance, on remarque que le popcount original a donné de meilleurs résultats donc on peut dire qu'il est **optimal**.

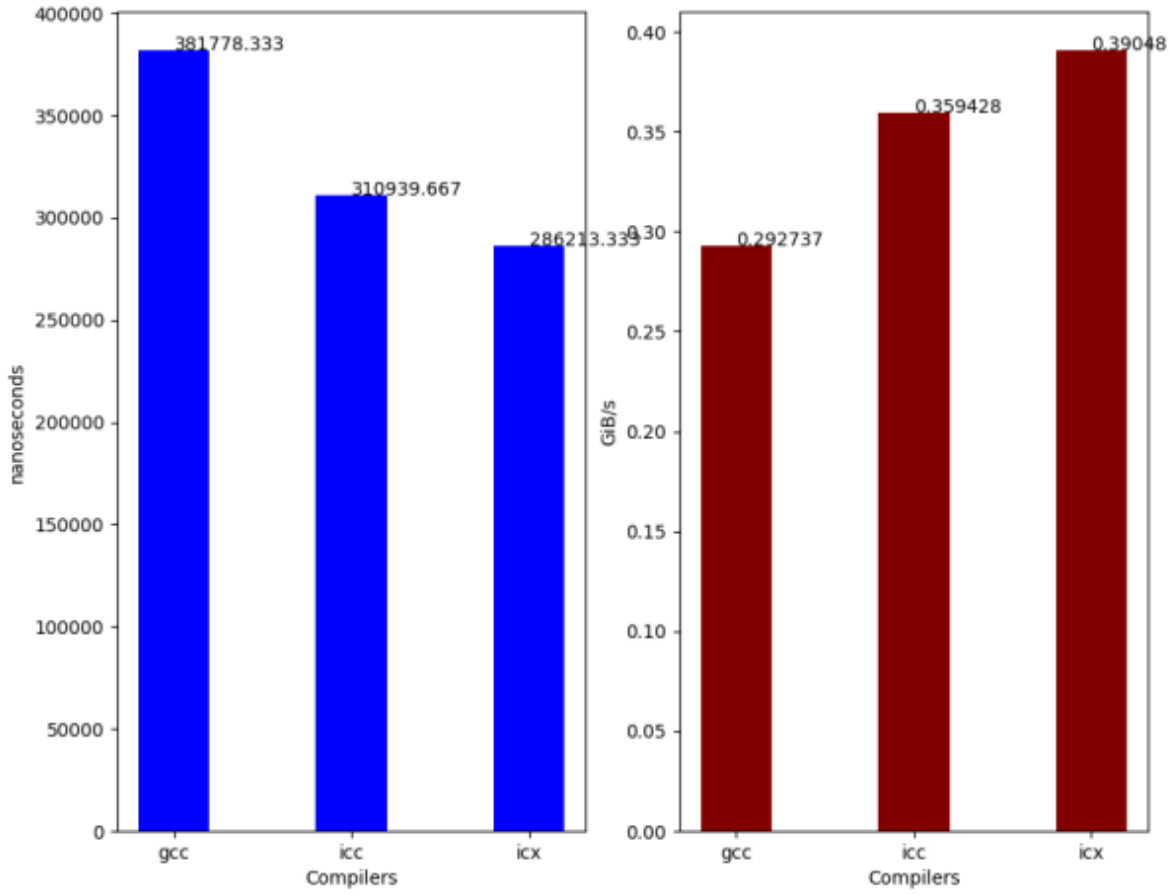

Figure 4: Popcount

2.2.3 Déroulage de boucle (Unrolling)

L'unrolling est une technique de transformation de boucle qui permet d'optimiser le temps d'exécution d'un programme, en réduisant le nombre de déclenchements de la boucle.
Chaque compilateur a ses propres directives ou pragmas pour spécifier l'unrolling de boucles, donc j'ai fait deux copies du fichier dist.c une pour gcc et l'autre pour icc, puis j'ai ajouté les pragma.

- **gcc**

```
#pragma GCC unroll 4
```

Figure 5

- **icc**

```
#pragma unroll 4
```

Figure 6

La figure ci-dessous montre les performances du code avec l'unrolling.

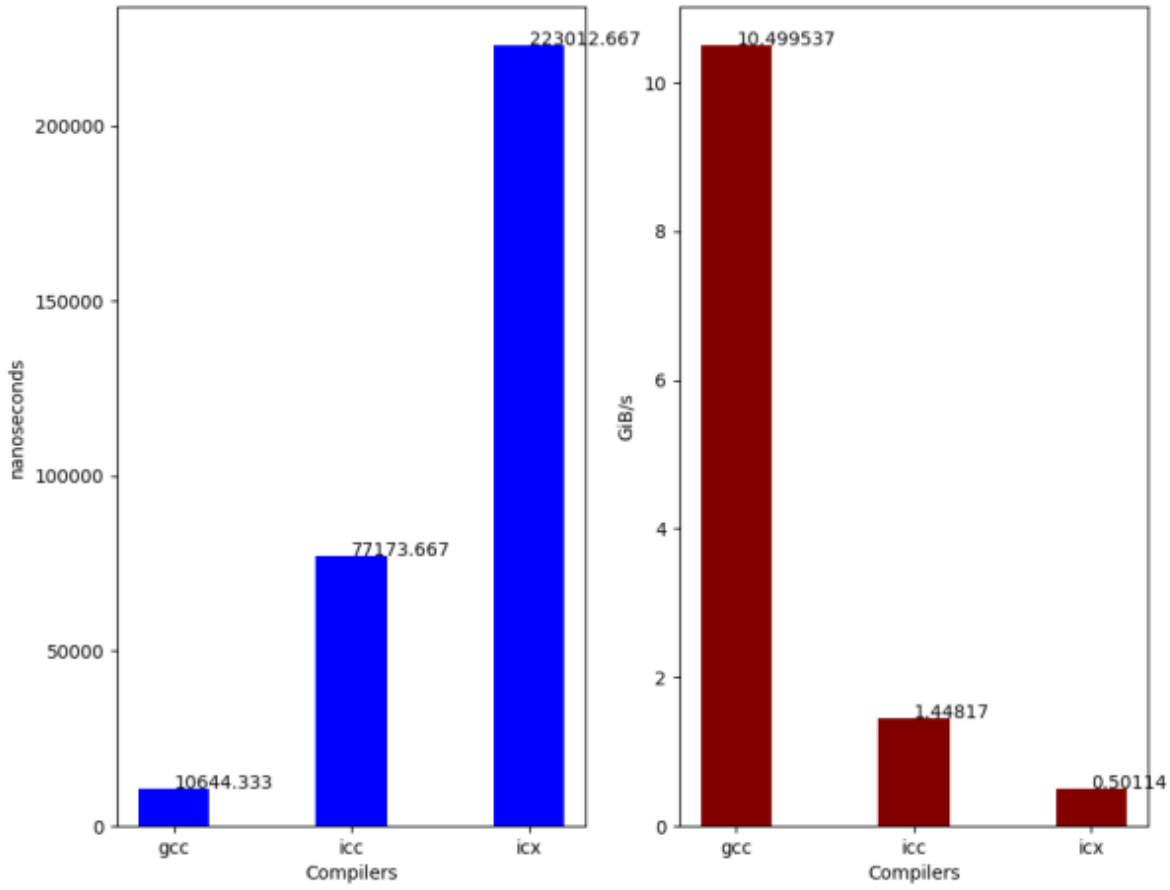

Figure 7: Déroulage de boucle

2.2.4 Vectorisation

- **Auto-vectorisation du compilateur**
  L'auto-vectorisation du compilateur est faite en ajoutant des flags de vectorisation pour chaque compilateur.

  - **gcc** -ftree-vectorize -fopt-info-vec-optimized -O3
  - **icc** -xAVX -qopt-report=5 -qopt-report-phase=vec -O3
  - **icx** -o3

```
OFLAGS=-march=native -ftree-vectorize -fopt-info-vec-optimized -O3 -fopt-info-all=dist.gcc.optrpt
INTEL_FLAGS=-xhost  -xAVX -qopt-report=5 -qopt-report-phase=vec -O3 -qopt-report
INTELx_FLAGS=-xhost -O3 -qopt-report
```

Figure 8

La figure ci-dessous montre les performances du code avec l'auto vectorisation.

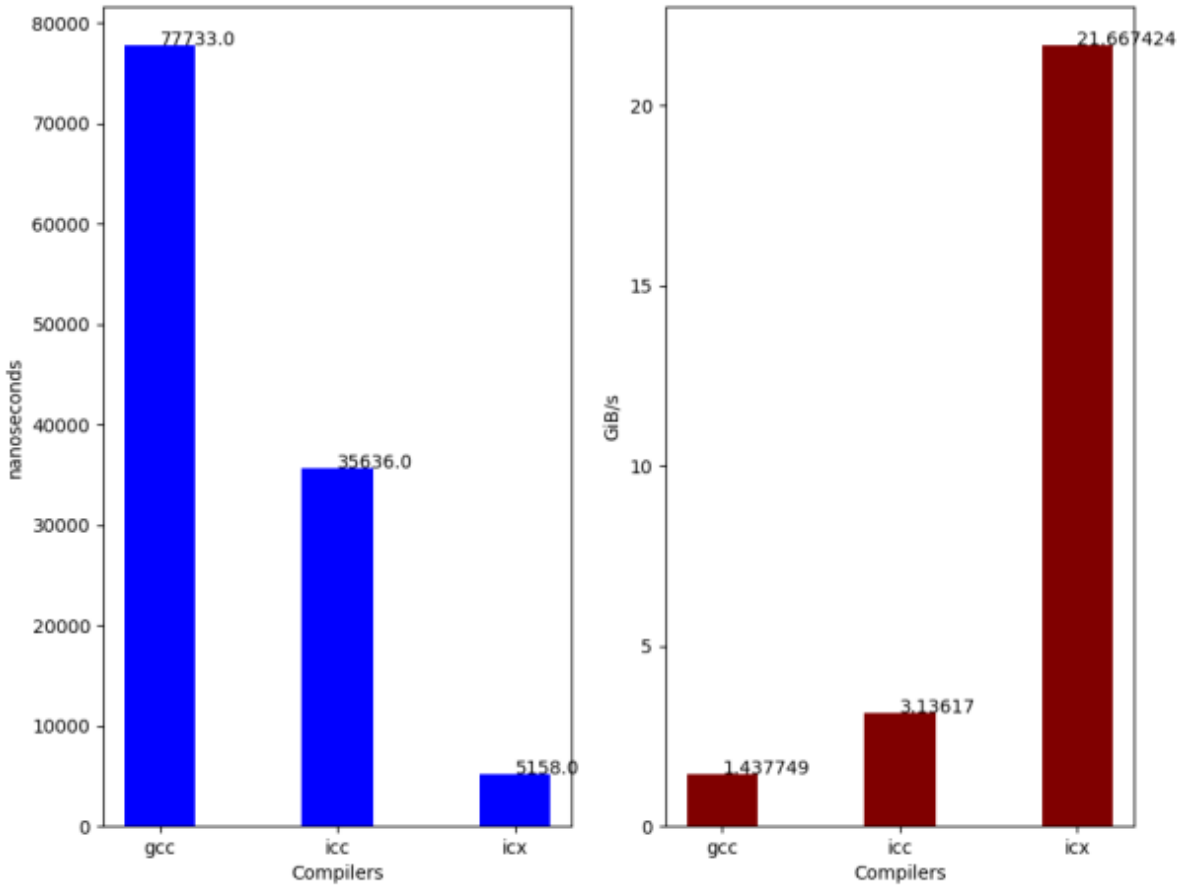

Figure 9: Auto-vectorisation du compilateur

- **Directive de vectorisation OpenMP**

```
#pragma omp simd
```

Figure 10

Cette directive permet de spécifier au compilateur que la boucle doit être exécutée de manière vectorielle.
La figure ci-dessous montre les performances du code avec la directive de vectorisation OpenMP.

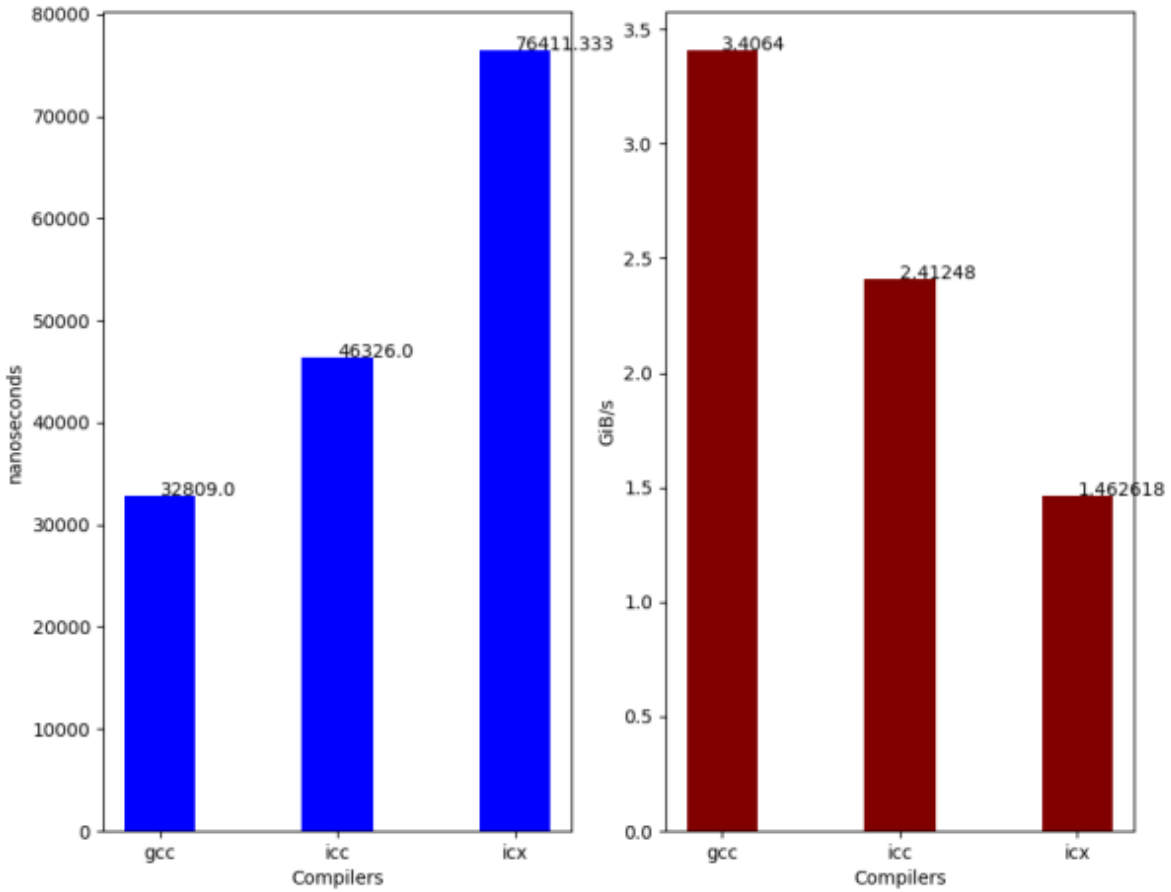

Figure 11: Directive de vectorisation OpenMP

2.2.5 Optimisations du compilateur

La figure ci-dessous montre les performances du code selon les différents flags d'optimisation.

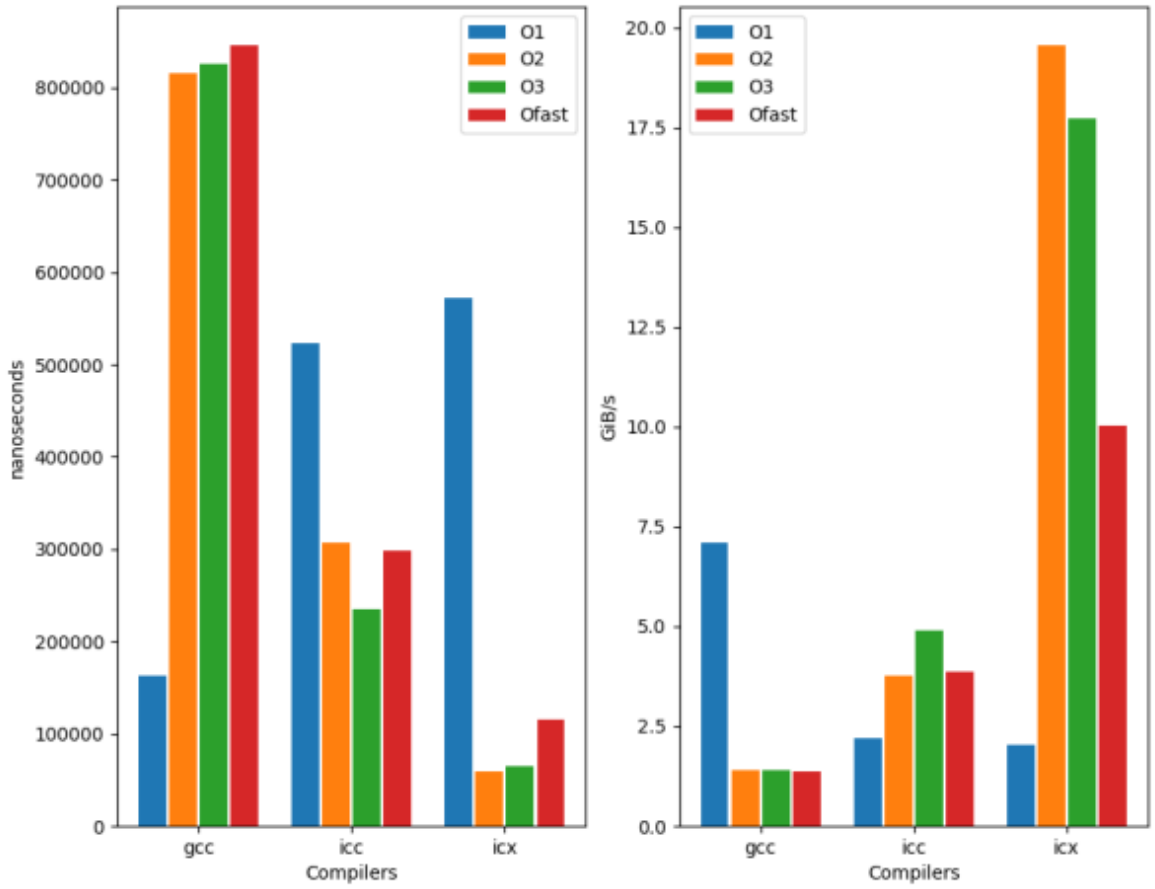

Figure 12: optimisations du compilateur

2.2.6 Tester le code en utilisant différentes longueurs de séquences qui correspondent à différents niveaux de cache.

Le code est testé pour chaque compilateur 3 fois chaque fois avec une taille de séquence différente:

- La première séquence avec 120000 bases (120kb), car L1 a une taille maximum de 128kb
- La deuxieme séquence avec 500000 bases (500kb), car L2 a une taille maximum de 512kb.
- La troisième avec 2800000 bases (2.8mb), car L3 a une taille maximum de 3mb.

La figure ci-dessous montre les performances du code selon les différentes tailles de séquences.

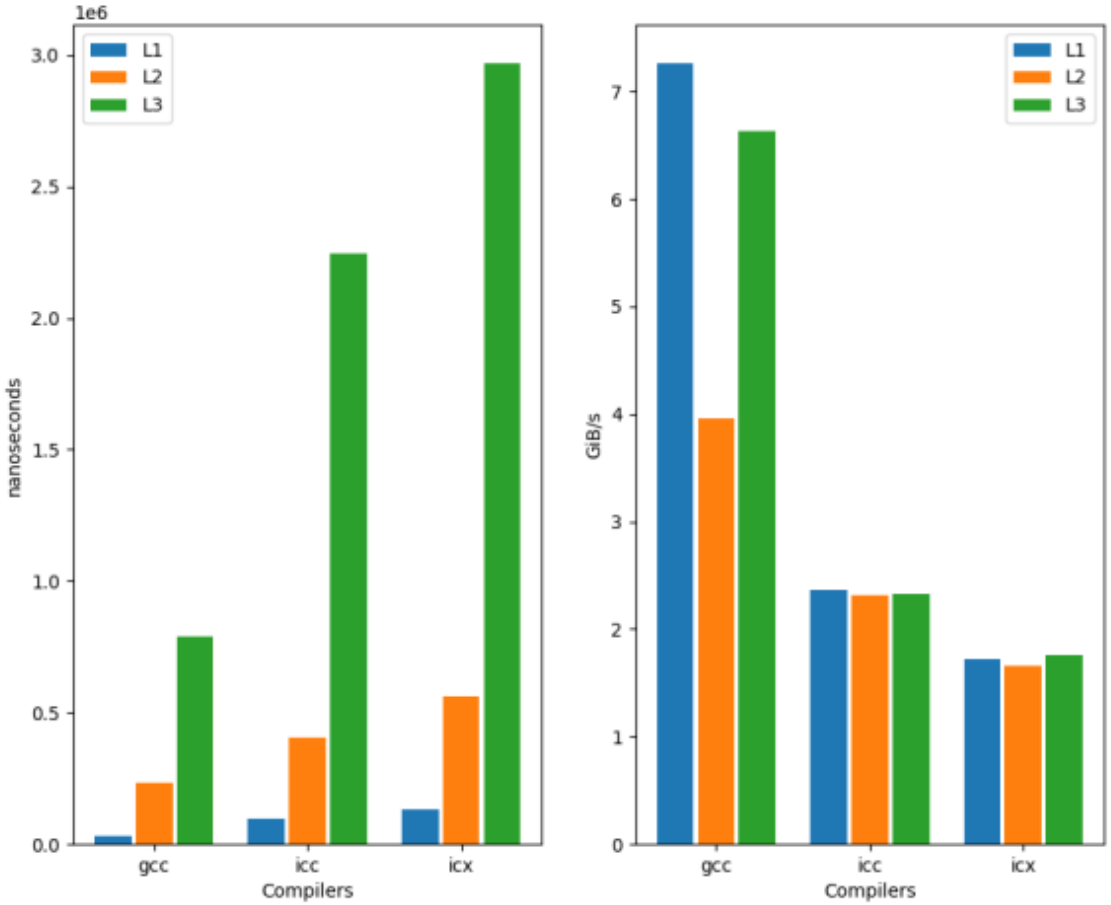

Figure 13: Tests sur différentes longueurs de séquences

2.2.7 Parallélisation à l'aide d'OpenMP sur la boucle principale

```
#pragma omp parallel for
```

Figure 14

Cette directive indique au compilateur de créer un ensemble de threads qui seront exécutés en parallèle.
La figure ci-dessous montre les performances du code avec parallélisation à l'aide d'OpenMP sur la boucle principale.

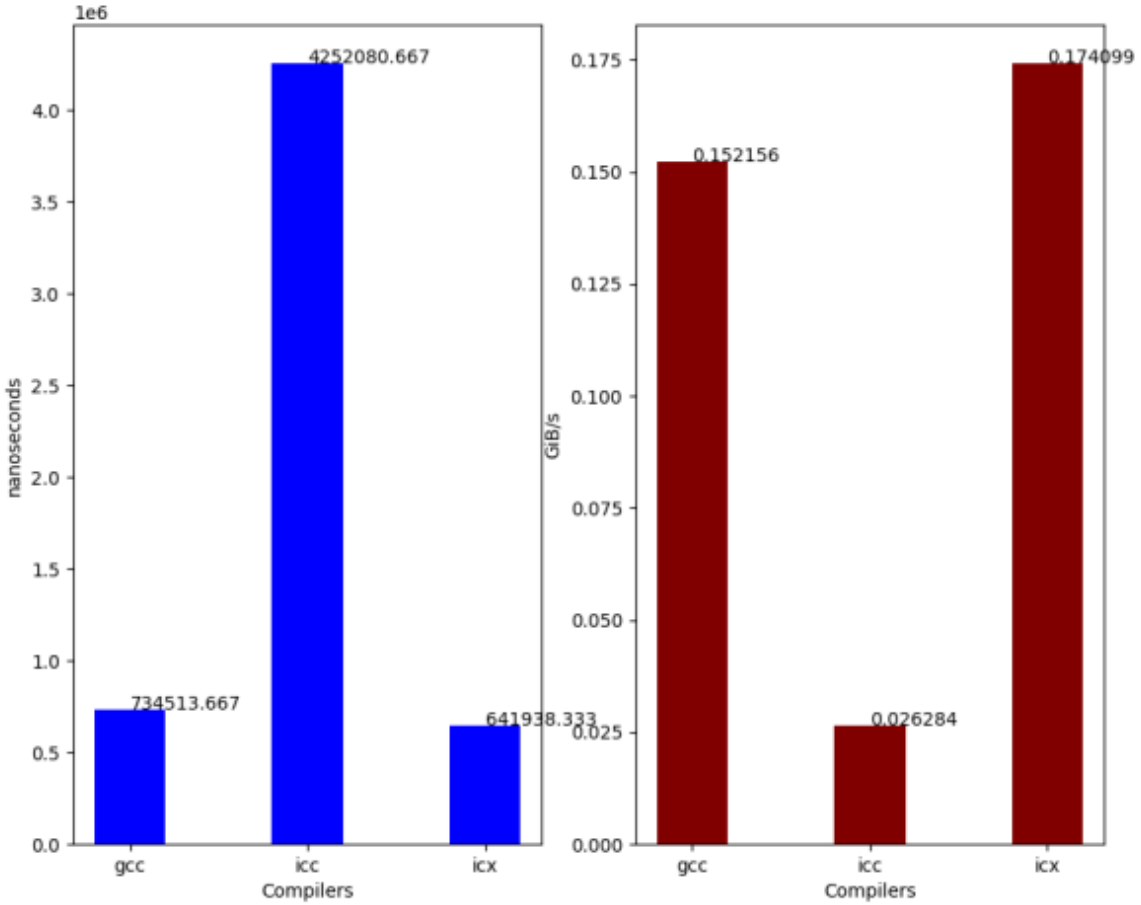

Figure 15: Parallélisation à l'aide d'OpenMP sur la boucle principale

2.2.8 Conclusion

Dans ce document, le code principal est analysé, ensuite plusieurs optimisations sont effectuées pour améliorer ses performances.