

# Rapport du projet d'architecture des ordinateurs

*Profilage et optimisation du code Nbody3D*

*Réalisé par :*

**TOIHIR Yoa IATIC4**

**MM. Hugo Bollore et Mohammed Salah Ibnamar : Encadrants**  
**Projet IATIC4 effectué du 04/02/2022 au 03/03/2022**

## **TABLE DES MATIÈRES**

<b>Introduction</b>	<b>3</b>
<b>Partie 1 : Fonction initiale</b>	<b>4</b>
1.1. Définition :	4
1.2. Performances de la fonction :	4
1.2.1 Compilation avec gcc :	4
1.2.2 Compilation avec icc :	5
1.2.3 Compilation avec icx :	6
1.3. Profilages de la fonction :	7
1.3.1 Compilation avec gcc :	7
1.3.2 Compilation avec icc :	9
1.3.3 Compilation avec icx :	11
<b>Partie 2 : Fonction optimisée</b>	<b>15</b>
2.1. Optimisations :	15
2.1.1. Optimisation 1ère boucle et boucle imbriquée :	15
2.1.2. Optimisation 2ème boucle :	15
2.2. Performances de la fonction :	16
2.2.1 Compilation avec gcc :	16
2.2.2 Compilation avec icc :	17
2.2.3 Compilation avec icx :	18
2.3. Profilages de la fonction :	19
2.3.1 Compilation avec gcc :	19
2.3.2 Compilation avec icc :	21
2.3.3 Compilation avec icx :	23
2.4. Comparaison des performances entre versions de code et compilateurs :	25
<b>Conclusion</b>	<b>28</b>
<b>Bibliographie</b>	<b>29</b>

## **Introduction**

Dans ce rapport, nous travaillerons sur un code implémentant un noyau de simulation des équations de Newton. Nous présenterons des optimisations et profilages de ce code à l'aide des nœuds de calculs KNL du cluster OB-1, mis à disposition.

Nous utiliserons l'interface de programmation OpenMP qui permet de réaliser du calcul parallèle sur architecture à mémoire partagée pour améliorer les performances de la fonction *move\_particles*.

# Partie 1 : Fonction initiale

## 1.1. Définition :

La fonction est composée d'une première boucle, d'une seconde imbriquée et d'une troisième isolée. Ces trois boucles font  $n$  itérations.

Nous pouvons remarquer qu'aucune boucle `for` ne fait appel à la directive `for` d'OpenMP. Il n'y a par conséquent aucune optimisation OpenMP liées aux boucles.

## 1.2. Performances de la fonction :

Nous allons calculer les performances de la fonction initiale en utilisant les compilateurs GNU gcc et Intel OneAPI icc sur 50 000 particules.

Toutes les mesures sont réalisées sur la machine knl05. La taille du cache est égale à 1024KB et le nombre de cœurs est égal à 72. Il n'y a qu'un seul processus.

### 1.2.1 Compilation avec gcc :

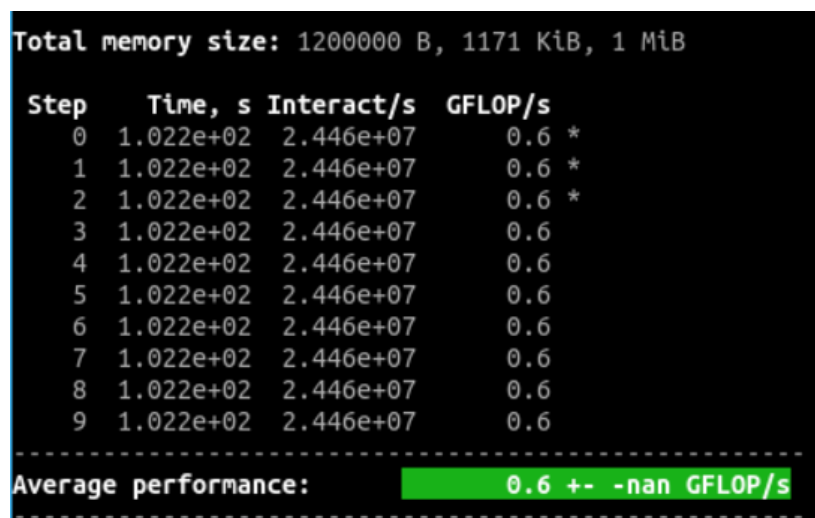
Pour compiler le programme avec gcc, nous utilisons les options d'optimisation suivantes : `-march=native`; `-mavx2`; `-Ofast`.

Nous pouvons voir les optimisations réalisées dans un fichier à l'aide de l'option : `-fopt-info-all`.

Nous obtenons la commande suivante :

```
gcc -march=native -mavx2 -Ofast -fopt-info-all=nbody.gcc.optprt nbody.c -o nbody.g -lm -fopenmp
```

Après exécution, nous obtenons les performances suivantes :



```

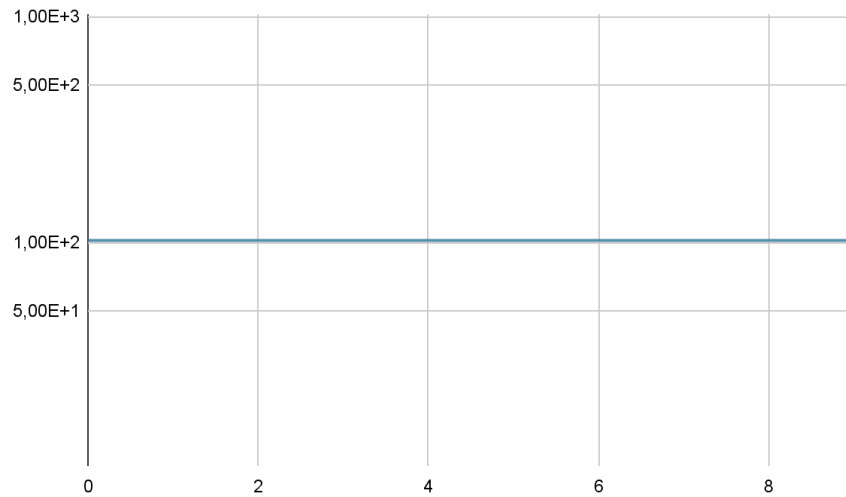
Total memory size: 1200000 B, 1171 KiB, 1 MiB

Step    Time, s  Interact/s  GFLOP/s
0    1.022e+02  2.446e+07    0.6 *
1    1.022e+02  2.446e+07    0.6 *
2    1.022e+02  2.446e+07    0.6 *
3    1.022e+02  2.446e+07    0.6
4    1.022e+02  2.446e+07    0.6
5    1.022e+02  2.446e+07    0.6
6    1.022e+02  2.446e+07    0.6
7    1.022e+02  2.446e+07    0.6
8    1.022e+02  2.446e+07    0.6
9    1.022e+02  2.446e+07    0.6

-----
Average performance:      0.6 +- -nan GFLOP/s
  
```

Figure 1 : Calcul des performances de la fonction initiale avec gcc

Temps d'exécution en moyenne (en s)



La moyenne de vitesse de calcul est de 0.6 GFLOP/s, ce qui est assez faible.

### 1.2.2 Compilation avec icc :

Pour compiler le programme avec icc, nous utilisons les options d'optimisation suivantes :

-xhost; -Ofast.

Nous pouvons voir les optimisations réalisées dans un fichier à l'aide de l'option :

-qot-report.

Nous obtenons la commande suivante :

***icc -xhost -Ofast -qot-report nbody.c -o nbody.i -qmkl***

Après exécution, nous obtenons les performances suivantes :

```
Total memory size: 1200000 B, 1171 KiB, 1 MiB
```

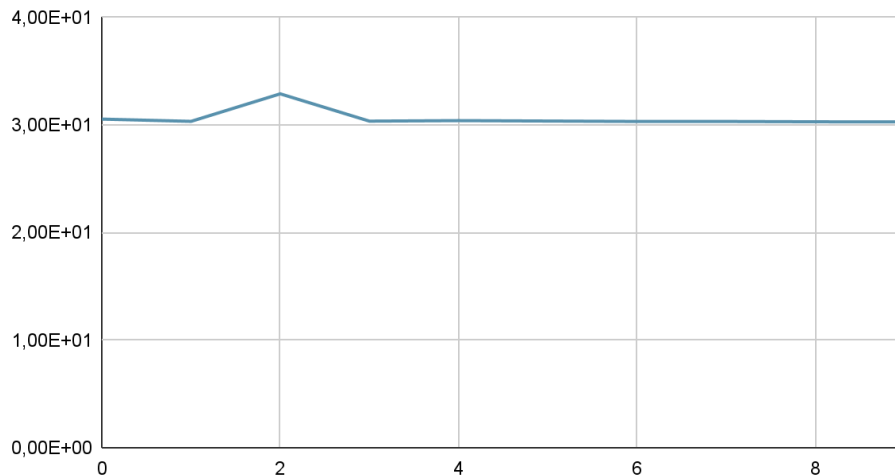
Step	Time, s	Interact/s	GFLOP/s
0	3.055e+01	8.183e+07	1.9 *
1	3.034e+01	8.240e+07	1.9 *
2	3.029e+01	8.254e+07	1.9 *
3	3.036e+01	8.235e+07	1.9
4	3.041e+01	8.221e+07	1.9
5	3.037e+01	8.232e+07	1.9
6	3.033e+01	8.243e+07	1.9
7	3.033e+01	8.244e+07	1.9
8	3.030e+01	8.251e+07	1.9
9	3.029e+01	8.254e+07	1.9

---

Average performance: **1.9 +- 0.0 GFLOP/s**

Figure 2 : Calcul des performances de la fonction initiale avec icc

Temps d'exécution en moyenne (en s)



La moyenne de vitesse de calcul est de 1.9 GFLOP/s, ce qui est assez faible.

### 1.2.3 Compilation avec icx :

Pour compiler le programme avec icx, nous utilisons les options d'optimisation suivantes :  
-xhost; -Ofast.

Nous pouvons voir les optimisations réalisées dans un fichier à l'aide de l'option :  
-fsave-optimization-record.

Nous obtenons la commande suivante :

**`icx -xhost -Ofast -fsave-optimization-record nbody.c -o nbody.i -qmk1 -qopenmp`**

Après exécution, nous obtenons les performances suivantes :

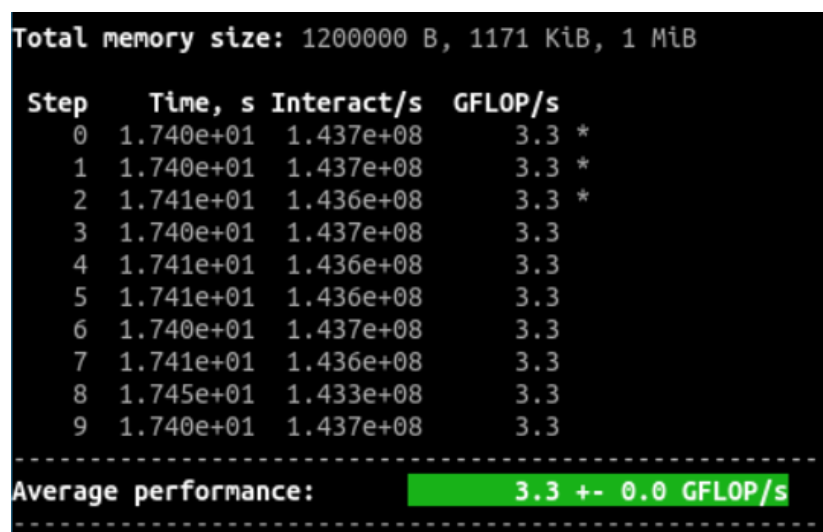
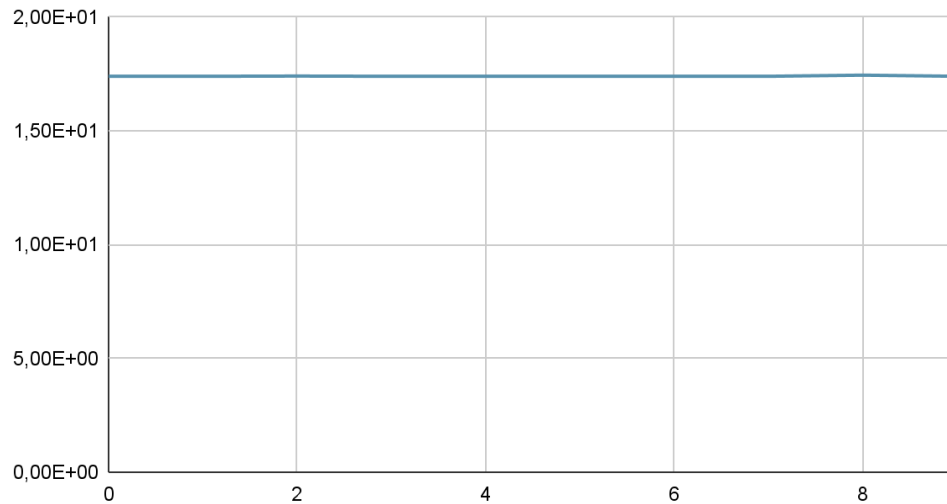


Figure 3 : Calcul des performances de la fonction initiale avec icx

### Vitesse d'exécution en moyenne (en s)



La moyenne de vitesse de calcul est de 3.3 GFLOP/s, ce qui est assez faible.

Nous pouvons voir que les performances de la fonction initiale compilée avec icx sont meilleures que celles avec gcc et icc.

### **1.3. Profilages de la fonction :**

Les différents profilages de la fonction ont été réalisés à l'aide de l'outil MAQAO.

#### **1.3.1 Compilation avec gcc :**

Lorsque la compilation est réalisé avec gcc, nous obtenons le profilage suivant :

Global Metrics <span>?</span>	
Total Time (s)	1.02 E3
Profiled Time (s)	1.02 E3
Time in analyzed loops (%)	100.0
Time in analyzed innermost loops (%)	100.0
Time in user code (%)	100.0
Compilation Options	Not Available
Perfect Flow Complexity	Not Available
Array Access Efficiency (%)	Not Available
Perfect OpenMP + MPI + Pthread	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.00

*Figure 4 : Profilage de la fonction initiale en utilisant gcc*

Nous pouvons observer que le temps total pour profiler les performances de la fonction est égal à 1.02E3 (1020) secondes, tout comme son temps d'exécution.

Les boucles et boucles imbriquées représentent 100% du temps d'exécution de la fonction. Le traitement des ces boucles prend 1020.561 secondes, soit la totalité du temps d'exécution.

Nous pouvons en déduire que le temps d'exécution de la fonction est fortement influencé par ses boucles. Pour l'améliorer nous devons très certainement modifier ces dernières.

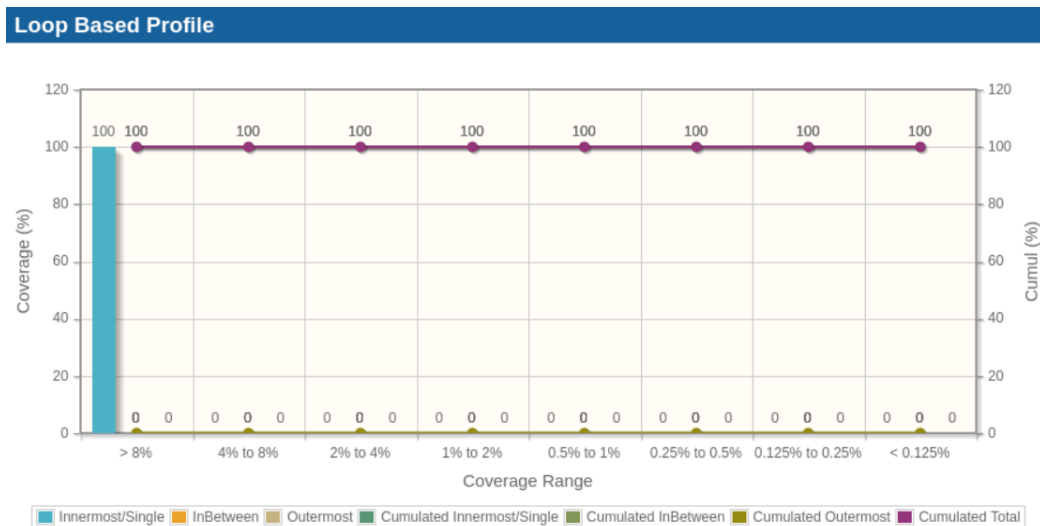


Figure 5 : Profilage lié aux boucles

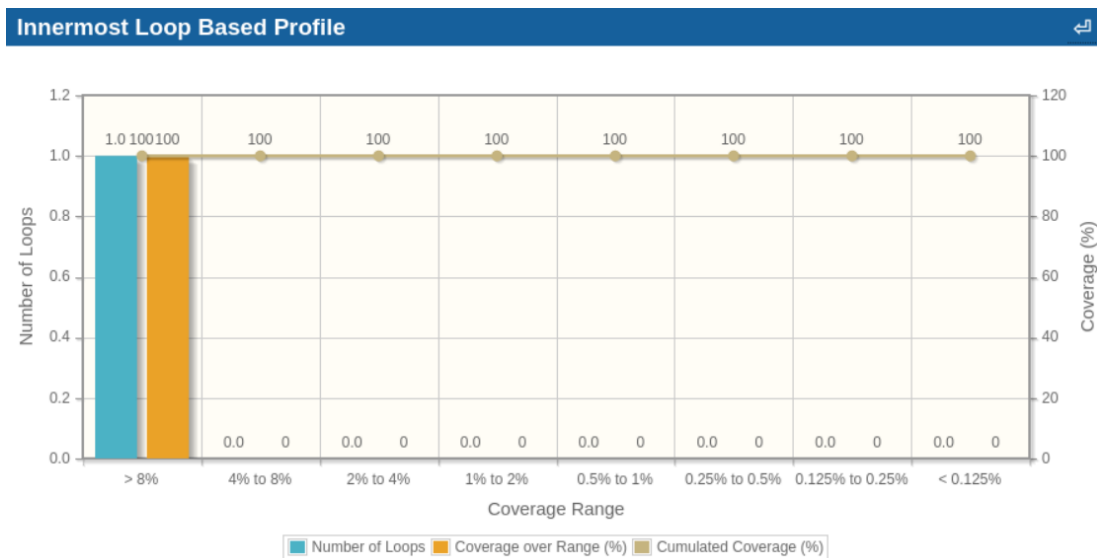


Figure 6 : Profilage lié aux boucles imbriquées

La fonction a passé 100% du temps dans les fichiers binaires et 0% dans les autres fichiers utilisateurs.



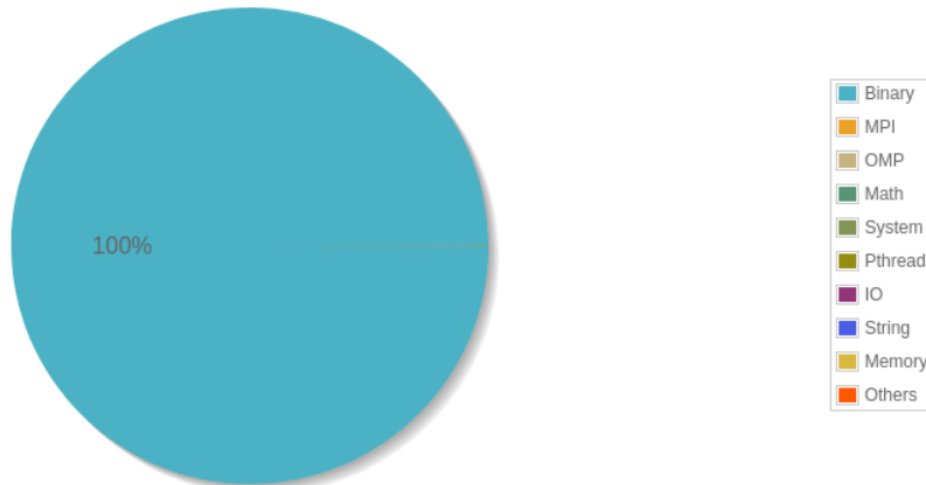


Figure 7 : Catégorisation de la fonction

L'accélération optimale que l'on obtiendrait si les outils OpenMP, MPI et Pthread sont correctement utilisés est égale à 1, ce qui est un résultat idéal. Cette accélération est calculée en divisant le temps d'exécution maximal obtenu en utilisant les outils cités par le temps maximal sans les utiliser.

Nous disposons d'un nouveau calcul de l'accélération optimale également égal à 1, ce qui est également un bon résultat. Cette accélération est calculée en divisant le temps maximal, en utilisant les mêmes outils cités plus tôt et en ayant une bonne répartition des charges aux différents threads, par la moyenne du temps sans utiliser les outils cités et en n'ayant par conséquent pas à répartir de tâches aux threads.

Les calculs d'accélération sont bons lorsque le résultat se rapproche de 1.0.

Le nombre de processus et de threads observés est de 1. La fonction est par conséquent exécutée en séquentiel.

### 1.3.2 Compilation avec icc :

Lorsque la compilation est réalisée avec icc, nous obtenons le profilage suivant :

Global Metrics <span>?</span>	
Total Time (s)	302.66
Profiled Time (s)	302.66
Time in analyzed loops (%)	100.0
Time in analyzed innermost loops (%)	99.9
Time in user code (%)	100.0
Compilation Options	Not Available
Perfect Flow Complexity	Not Available
Array Access Efficiency (%)	Not Available
Perfect OpenMP + MPI + Pthread	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.00

Figure 8 : Profilage de la fonction initiale en utilisant icc

Nous pouvons observer que le temps total pour profiler les performances de la fonction est égal à 302.66 secondes, tout comme son temps d'exécution.

Les boucles représentent 100% du temps d'exécution de la fonction. Le traitement des ces boucles prend 302.50867 secondes à la fonction, soit la totalité du temps d'analyse.

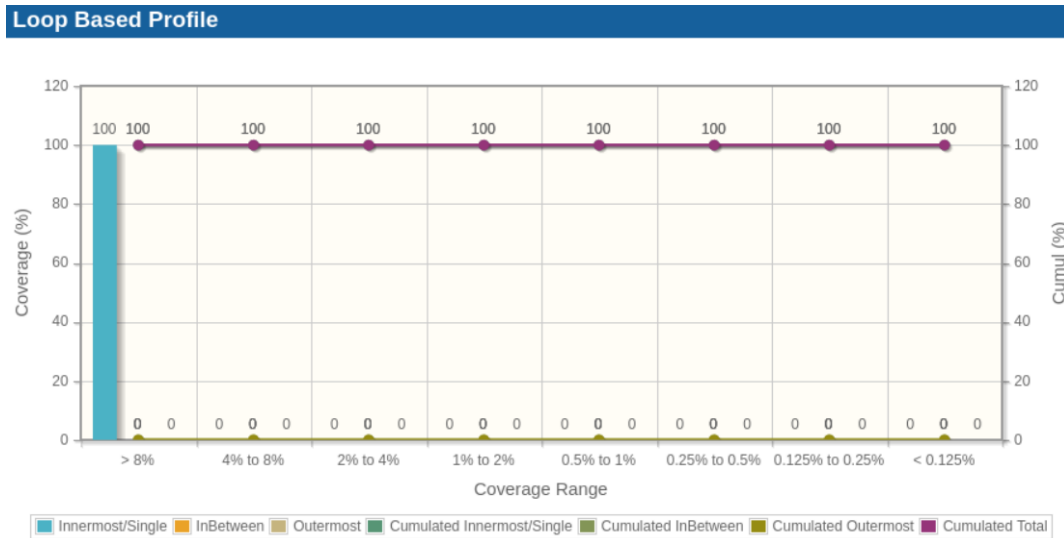


Figure 8 : Profilage lié aux boucles

Les boucles imbriquées, quant à elles, représentent 99% du temps d'exécution de la fonction. Le traitement des ces boucles prend 302.448138 secondes à la fonction.

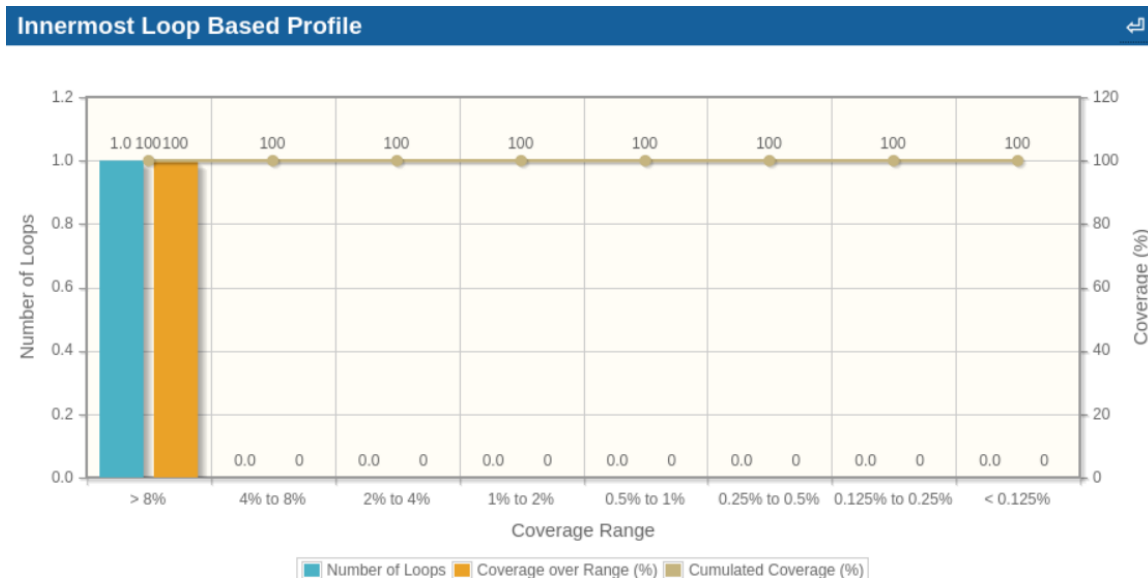
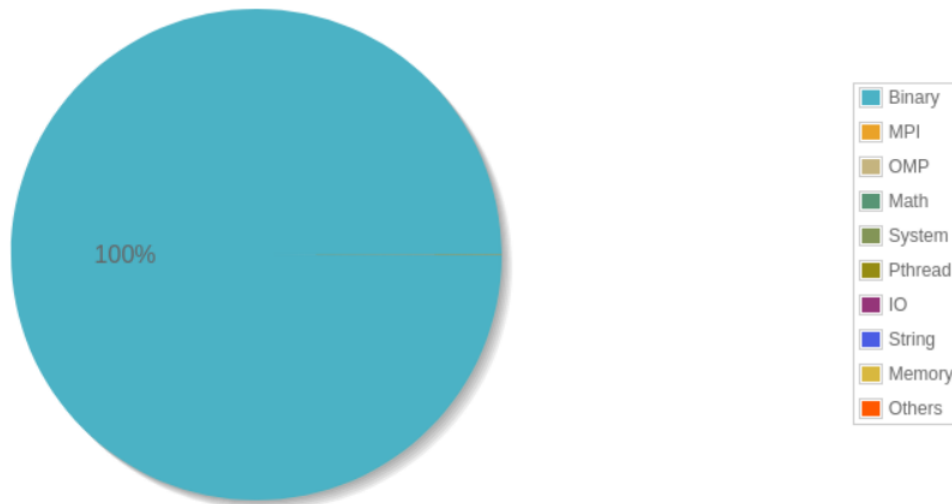


Figure 9 : Profilage lié aux boucles imbriquées

Nous pouvons en déduire que le temps d'exécution de la fonction est fortement influencé par ses boucles. Pour l'améliorer nous devons très certainement modifier ces dernières.

La fonction a passé 100% du temps dans les fichiers binaires et 0% dans les autres fichiers utilisateurs.



*Figure 10 : Catégorisation de la fonction*

L'accélération optimale que l'on obtiendrait si les outils OpenMP, MPI et Pthread sont correctement utilisés est égale à 1.

L'accélération optimale obtenue en utilisant parfaitement les outils cités précédemment et en ayant une bonne répartition des charges aux différents threads est également égale à 1.

Le nombre de processus et de threads observés est de 1. La fonction est par conséquent exécutée en séquentiel.

### **1.3.3 Compilation avec icx :**

Lorsque la compilation est réalisé avec icx, nous obtenons le profilage suivant :

Global Metrics		?
Total Time (s)	182.49	
Profiled Time (s)	182.49	
Time in analyzed loops (%)	100.0	
Time in analyzed innermost loops (%)	99.9	
Time in user code (%)	0	
Compilation Options	Not Available	
Perfect Flow Complexity	1.00	
Array Access Efficiency (%)	0.07	
Perfect OpenMP + MPI + Pthread	1.00	
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.00	
No Scalar Integer	Potential Speedup	1.00
	Nb Loops to get 80%	2
FP Vectorised	Potential Speedup	1.02
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.08
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.68
	Nb Loops to get 80%	1

Figure 11 : Profilage de la fonction initiale en utilisant icx

Nous pouvons observer que le temps total pour profiler les performances de la fonction est égal à 182.49 secondes, tout comme son temps d'exécution.

Les boucles représentent 100% du temps d'exécution de la fonction. Le traitement des ces boucles prend 182.41 secondes à la fonction, soit la totalité du temps d'analyse.

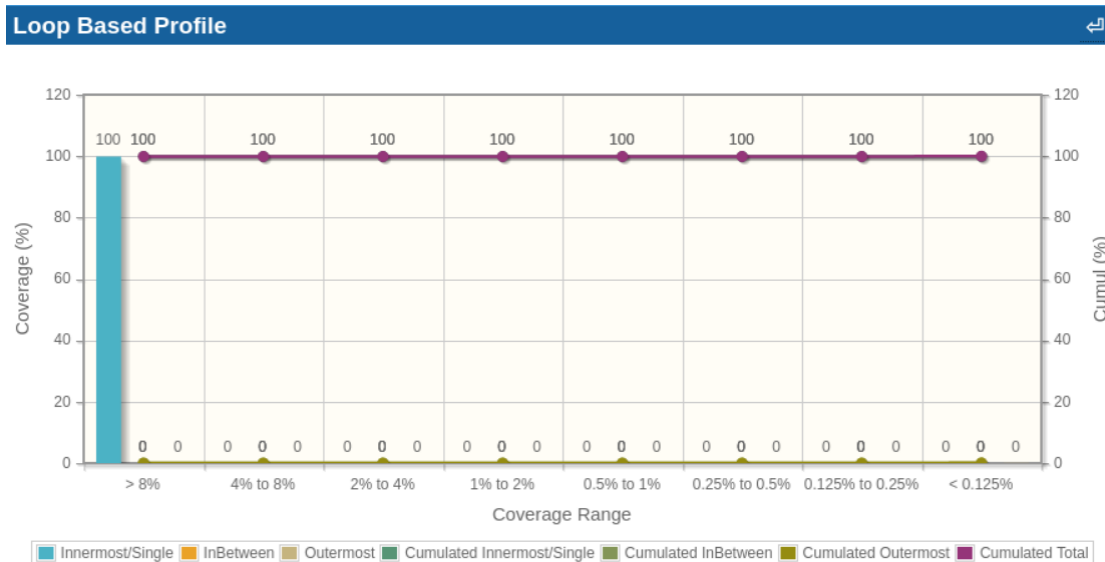


Figure 12 : Profilage lié aux boucles

Les boucles imbriquées, quant à elles, représentent 99,9% du temps d'exécution de la fonction. Le traitement des ces boucles prend 182.28 secondes à la fonction.

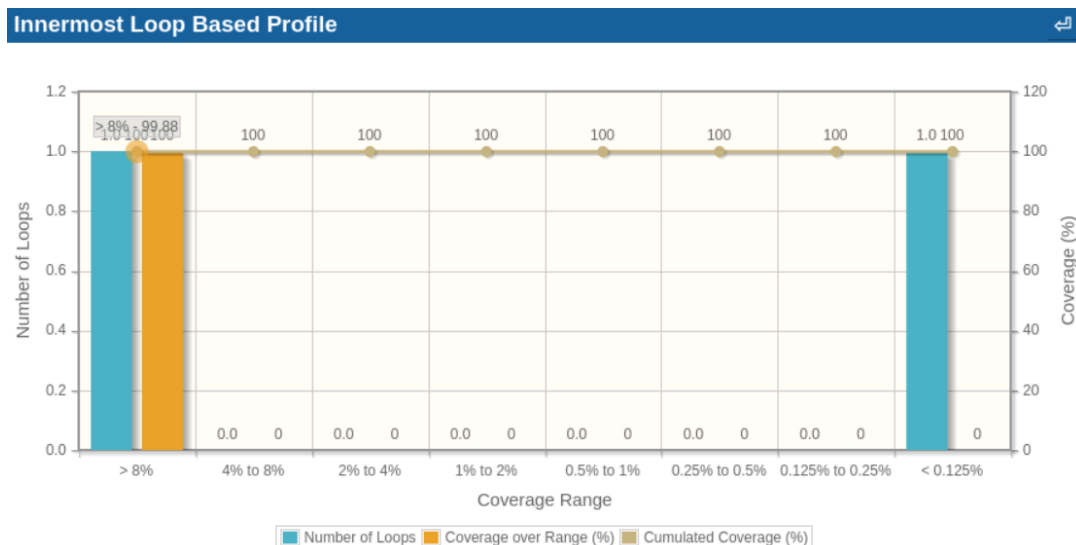


Figure 13 : Profilage lié aux boucles imbriquées

Nous pouvons en déduire que le temps d'exécution de la fonction est fortement influencé par ses boucles. Pour l'améliorer nous devons très certainement modifier ces dernières.

La fonction a passé 100% du temps dans les fichiers String et 0% dans les autres fichiers utilisateurs.

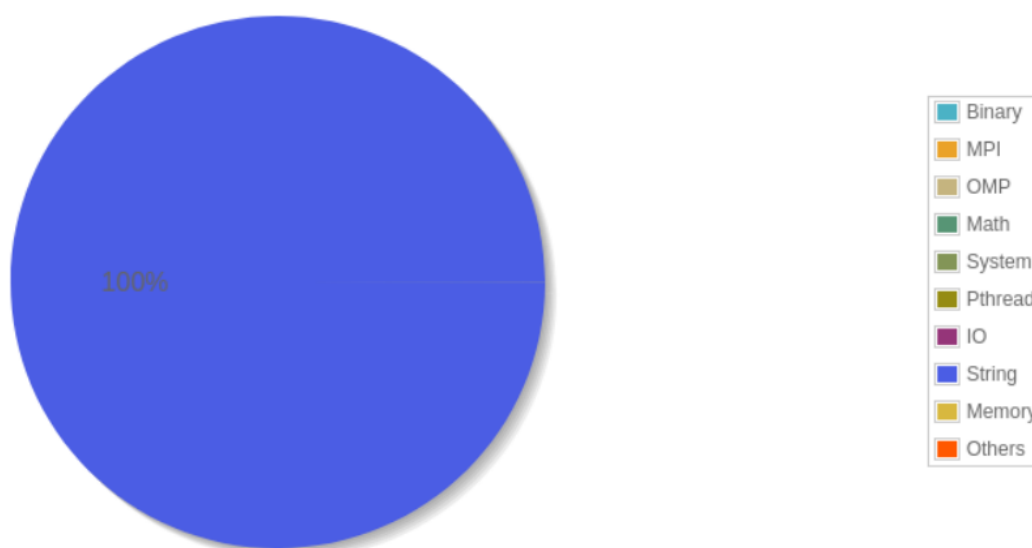


Figure 14 : Catégorisation de la fonction

L'accélération optimale que l'on obtiendrait en réduisant le nombre de chemins dans les boucles et en favorisant par conséquent la vectorisation du code est égale à 1, ce qui est un bon résultat.

L'efficacité d'accès aux tableaux est très mauvaise, elle est égale à 0,07.

L'accélération optimale que l'on obtiendrait si les outils OpenMP, MPI et Pthread sont correctement utilisés est égale à 1.

L'accélération optimale obtenue en utilisant parfaitement les outils cités précédemment et en ayant une bonne répartition des charges aux différents threads est également égale à 1.

L'accélération optimale que l'on obtiendrait si toutes les instructions réalisant des calculs d'adresses et calculs d'entiers scalaires ont été supprimés est égale à 1. On sait également que le nombre de boucles à optimiser pour obtenir 80% de cette accélération s'élève au nombre de 2.

L'accélération optimale que l'on obtiendrait si toutes les instructions en virgule flottante sont vectorisées est égale à 1.02. On sait également qu'il faut optimiser une boucle pour obtenir 80% de cette accélération.

L'accélération optimale que l'on obtiendrait si toutes les instructions sont vectorisées est égale à 1.08. On sait également qu'il faut optimiser une boucle pour obtenir 80% de cette accélération.

L'accélération optimale que l'on obtiendrait si seulement les instructions virgules flottantes arithmétiques sont conservées est égale à 1.68, ce qui est un bon résultat mais il peut toujours être amélioré. On sait également qu'il faut optimiser une boucle pour obtenir 80% de cette accélération.

Le nombre de processus et de threads observés est de 1. La fonction est par conséquent exécutée en séquentiel.

## **Partie 2 : Fonction optimisée**

### **2.1. Optimisations :**

Après avoir analysé les résultats des différents profilages nous pouvons proposer plusieurs optimisations au code et à la compilation.

#### **2.1.1. Optimisation 1ère boucle et boucle imbriquée :**

La première boucle contient une boucle imbriquée dont le contenu dépend de la boucle externe.

Nous pouvons remarquer que nous travaillons dans ces deux boucles avec un tableau de structures (Array of structures ou AoS) nommé *p*, sur lequel nous réalisons des modifications et de la récupération de contenu dans les cases de ce tableau.

Ces accès et modifications n'ont aucune dépendance avec d'autres itérations, nous pouvons par conséquent paralléliser cette boucle en utilisant la clause *#pragma omp for* et la directive *schedule(dynamic, taille paquet)* d'OpenMp. Grâce à cela, les threads se partageront le travail de manière dynamique, les itérations seront divisées par OpenMp en fonction de la taille de paquet renseignée et ces paquets seront distribués aux threads en fonction de leurs disponibilités. Ici, la taille des paquets sera égale à 1.

Dans la boucle externe, nous réalisons des incrémentations sur le tableau de structures *p*; dans la boucle interne, nous accédons à des cases de ce tableau.

Avoir un tableau de structure signifie qu'on doit stocker plusieurs structures dans un tableau.

Dans notre cas, avoir de multiples structures n'est pas intéressant car si notre but est de stocker plusieurs flottants *x*, *y*, *z*, *vx*, *vy* et *vz*, créer plusieurs structures permettant d'en stocker un seul pour les variables que nous venons de lister représente du stockage inutile.

Toutes les données des structures créées peuvent être réunies en transformant le tableau de structure en structure de tableaux. Les variables *x*, *y*, *z*, *vx*, *vy* et *vz* deviendront par conséquent des tableaux de taille *n*.

Pour éviter de réaliser plusieurs des accès mémoires au même endroit lors d'une même itération, on stockera dans des variables les valeurs de ces accès.

Nous allons également remplacer toutes les divisions par des multiplications par l'inverse, ce qui nous permettra de gagner un peu de temps d'exécution.

La dernière optimisation consiste à améliorer le calcul de la fonction *pow* en retirant la division permettant d'obtenir la valeur de la puissance et en remplaçant cette division par une constante.

#### **2.1.2. Optimisation 2ème boucle :**

Nous pouvons voir dans la troisième boucle, que nous accédons aux tableaux *x*, *y*, *z*, *vx*, *vy* et *vz* de la structure *p*, puis, nous redéfinissons le contenu des cases de ces tableaux à

chaque itération. Ces accès et modifications ne nécessitent aucune dépendance avec d'autres itérations, nous pouvons par conséquent paralléliser et vectoriser cette boucle en utilisant la clause `#pragma omp for` et la directive `simd` d'OpenMp. Grâce à cela, les itérations des deux boucles seront distribuées entre les threads existants et grâce l'instruction `simd`, les threads sont autorisés à exécuter plusieurs itérations simultanément. Le compilateur ignorera les dépendances vectorielles.

## **2.2. Performances de la fonction :**

Nous allons calculer les performances de la fonction initiale en utilisant les compilateurs GNU gcc et Intel OneAPI icc sur 50 000 particules.

Toutes les mesures sont réalisées sur la machine knl05. La taille du cache est égale à 1024KB et le nombre de cœurs est égal à 72. Le nombre de threads est égal à 256.

### **2.2.1 Compilation avec gcc :**

Pour compiler le programme avec gcc, nous utilisons les options d'optimisation suivantes : `-march=native; -mavx2; -Ofast`.

Nous pouvons voir les optimisations réalisées dans un fichier à l'aide de l'option : `-fopt-info-all`.

Nous obtenons la commande suivante :

```
gcc -march=native -mavx2 -Ofast -fopt-info-all=nbody.gcc.optprt nbody.c -o nbody.g -lm -fopenmp
```

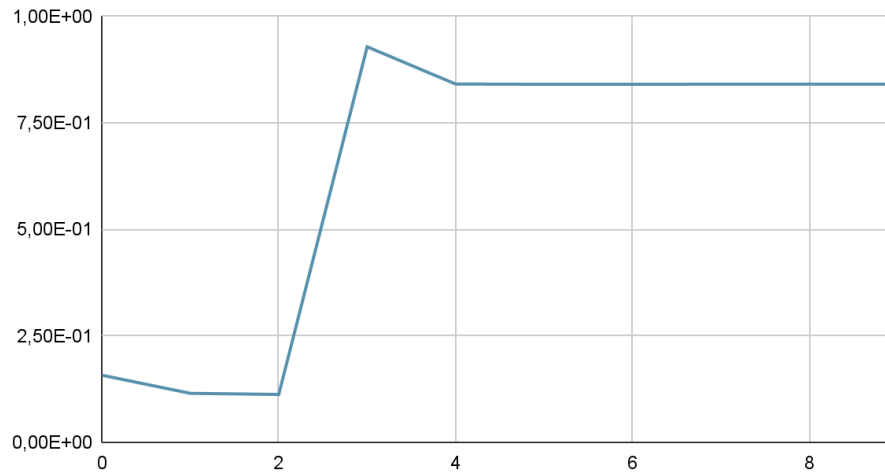
Après exécution, nous obtenons les performances suivantes :

Total memory size: 2400000 B, 2343 KiB, 2 MiB			
Step	Time, s	Interact/s	GFLOP/s
0	1.576e-01	1.587e+10	364.9 *
1	1.150e-01	2.175e+10	500.2 *
2	1.124e-01	2.224e+10	511.5 *
3	9.288e-02	2.692e+10	619.1
4	8.413e-02	2.972e+10	683.5
5	8.407e-02	2.974e+10	683.9
6	8.408e-02	2.973e+10	683.8
7	8.409e-02	2.973e+10	683.8
8	8.411e-02	2.972e+10	683.6
9	8.409e-02	2.973e+10	683.8
Average performance:			674.5 +- 22.6 GFLOP/s

Figure 15 : Calcul des performances de la fonction optimisée avec gcc



### Temps d'exécution en moyenne (en s)



La moyenne de vitesse de calcul est de 674.5 GFLOP/s, ce qui est beaucoup plus élevé que le résultat précédemment obtenu avec la version initiale du code.

### 2.2.2 Compilation avec icc :

Pour compiler le programme avec icc, nous utilisons les options d'optimisation suivantes :

-xhost; -Ofast.

Nous pouvons voir les optimisations réalisées dans un fichier à l'aide de l'option :

-qot-report.

Nous obtenons la commande suivante :

**`icc -xhost -Ofast -qopt-report nbody.c -o nbody.i -qmkl -qopenmp`**

Après exécution, nous obtenons les performances suivantes :

```
Total memory size: 2400000 B, 2343 KiB, 2 MiB
```

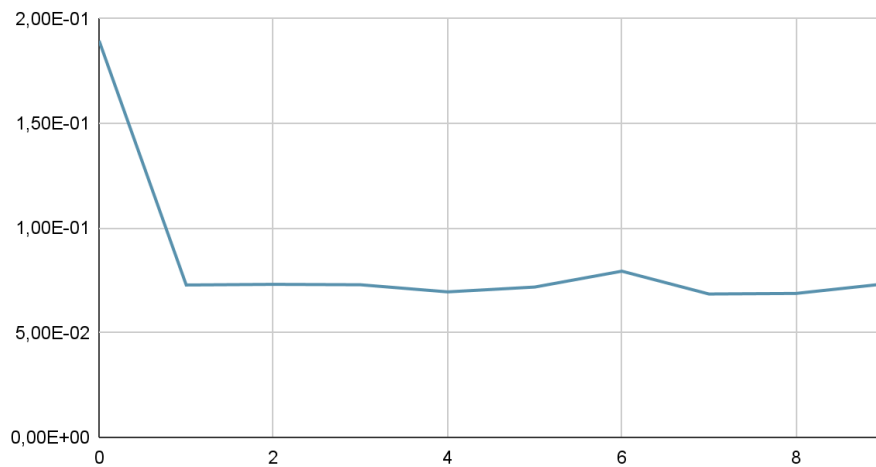
Step	Time, s	Interact/s	GFLOP/s
0	1.895e-01	1.320e+10	303.5 *
1	7.280e-02	3.434e+10	789.9 *
2	7.311e-02	3.419e+10	786.5 *
3	7.291e-02	3.429e+10	788.6
4	6.951e-02	3.596e+10	827.2
5	7.182e-02	3.481e+10	800.6
6	7.940e-02	3.149e+10	724.2
7	6.849e-02	3.650e+10	839.6
8	6.878e-02	3.634e+10	835.9
9	7.325e-02	3.413e+10	784.9

---

Average performance: **800.1 +- 37.2 GFLOP/s**

Figure 16 : Calcul des performances de la fonction optimisée avec icc

Temps d'exécution en moyenne (en s)



La moyenne de vitesse de calcul est de 800.1 GFLOP/s, ce qui est beaucoup plus élevé que le résultat précédemment obtenu avec la version initiale du code.

### 2.2.3 Compilation avec icx :

Pour compiler le programme avec icx, nous utilisons les options d'optimisation suivantes :  
-xhost; -Ofast.

Nous pouvons voir les optimisations réalisées dans un fichier à l'aide de l'option :  
-fsave-optimization-record.

Nous obtenons la commande suivante :

**`icx -xhost -Ofast -fsave-optimization-record nbody.c -o nbody -qmk1 -qopenmp`**

Après exécution, nous obtenons les performances suivantes :

```
Total memory size: 2400000 B, 2343 KiB, 2 MiB
```

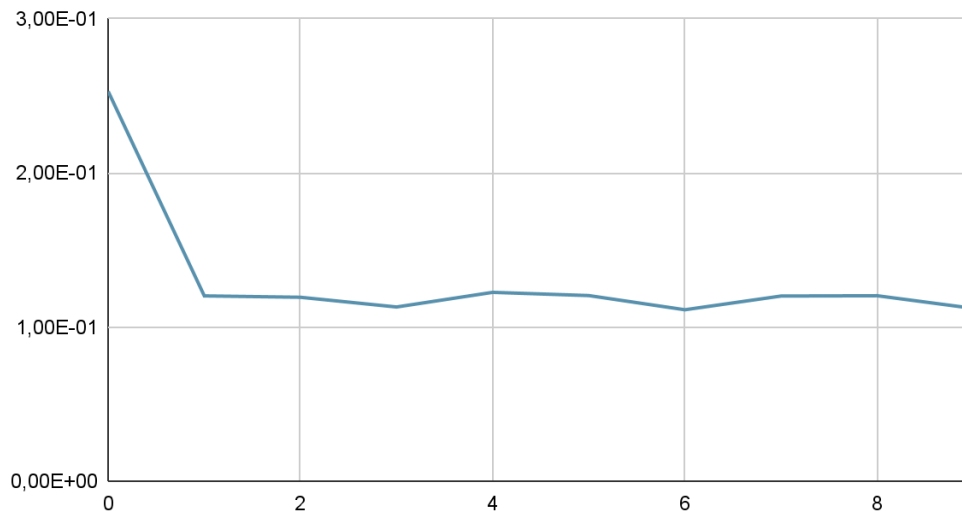
Step	Time, s	Interact/s	GFLOP/s
0	2.529e-01	9.885e+09	227.4 *
1	1.202e-01	2.079e+10	478.3 *
2	1.193e-01	2.096e+10	482.1 *
3	1.130e-01	2.212e+10	508.8
4	1.225e-01	2.040e+10	469.3
5	1.204e-01	2.076e+10	477.6
6	1.112e-01	2.249e+10	517.2
7	1.201e-01	2.082e+10	478.9
8	1.203e-01	2.079e+10	478.1
9	1.121e-01	2.231e+10	513.1

---

Average performance: **491.8 +- 18.7 GFLOP/s**

Figure 17 : Calcul des performances de la fonction optimisée avec icx

### Temps d'exécution en moyenne (en s)



La moyenne de vitesse de calcul est de 491.8 GFLOP/s, ce qui est beaucoup plus élevé que le résultat précédemment obtenu avec la version initiale du code.

Nous pouvons voir que les performances de la fonction optimisée compilée avec icc sont meilleures que celles avec gcc et icx.

### **2.3. Profilages de la fonction :**

Les différents profilages de la fonction ont été réalisés à l'aide de l'outil MAQAO.

#### **2.3.1 Compilation avec gcc :**

Lorsque la compilation est réalisé avec gcc, nous obtenons le profilage suivant :

Global Metrics <span>?</span>	
Total Time (s)	0.92
Profiled Time (s)	0.92
Time in analyzed loops (%)	90.6
Time in analyzed innermost loops (%)	90.4
Time in user code (%)	90.6
Compilation Options	Not Available
Perfect Flow Complexity	Not Available
Array Access Efficiency (%)	Not Available
Perfect OpenMP + MPI + Pthread	1.07
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.12

Figure 18 : Profilage de la fonction optimisée en utilisant gcc

Nous pouvons observer que le temps total pour profiler les performances de la fonction est égal à 0.92 secondes, tout comme son temps d'exécution.

Les boucles représentent 90.6% du temps passé d'exécution de la fonction. Le traitement des ces boucles prend 0.8336 secondes.

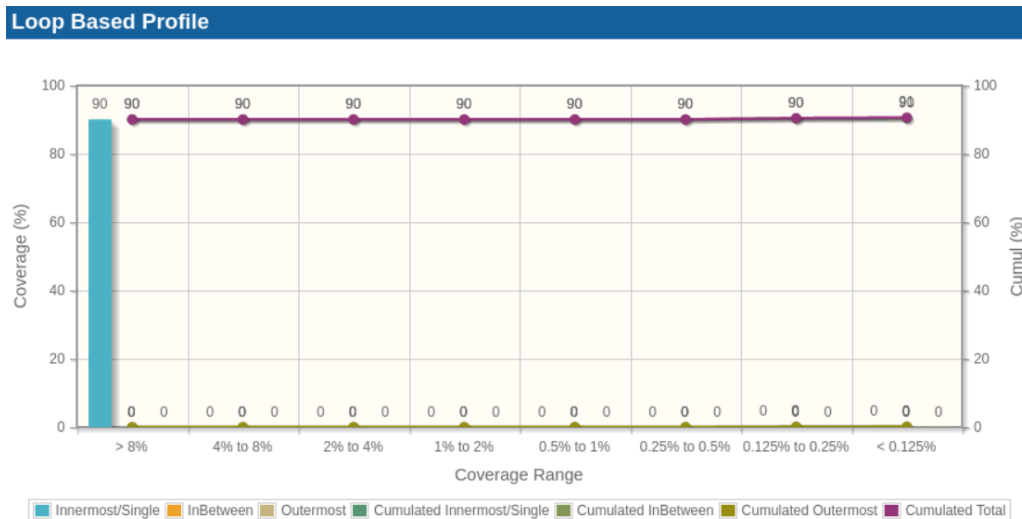


Figure 19 : Profilage lié aux boucles

Les boucles imbriquées, quant à elles, représentent 90.4% du temps d'exécution de la fonction. Le traitement des ces boucles prend 0.8316 secondes à la fonction.

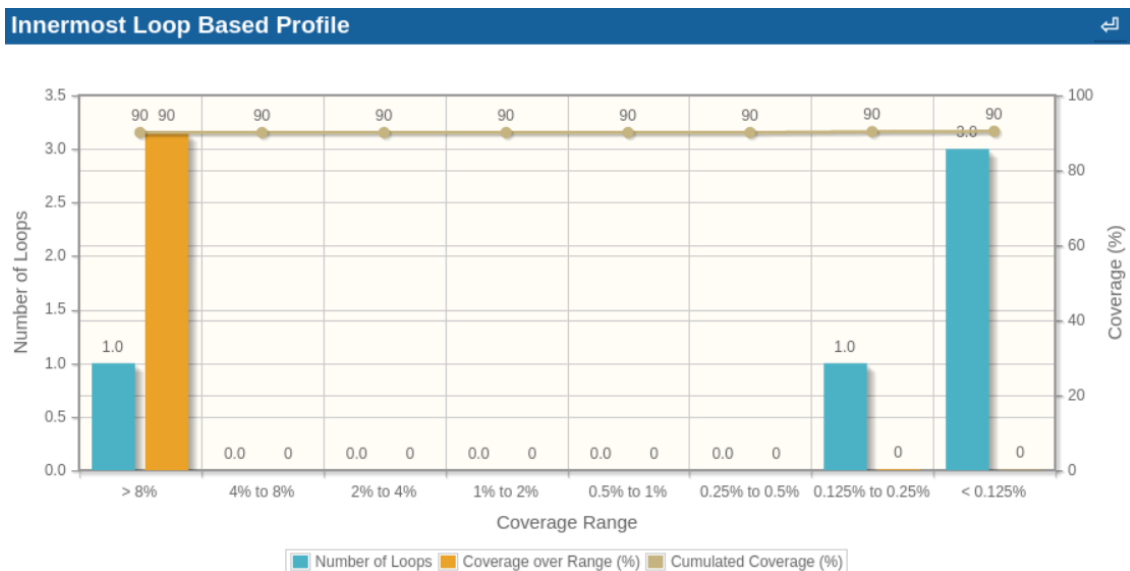


Figure 20 : Profilage lié aux boucles imbriquées

La fonction a passé 90,6% du temps dans les fichiers utilisateurs dont 91% concerne les fichiers binaires et 9% OMP.

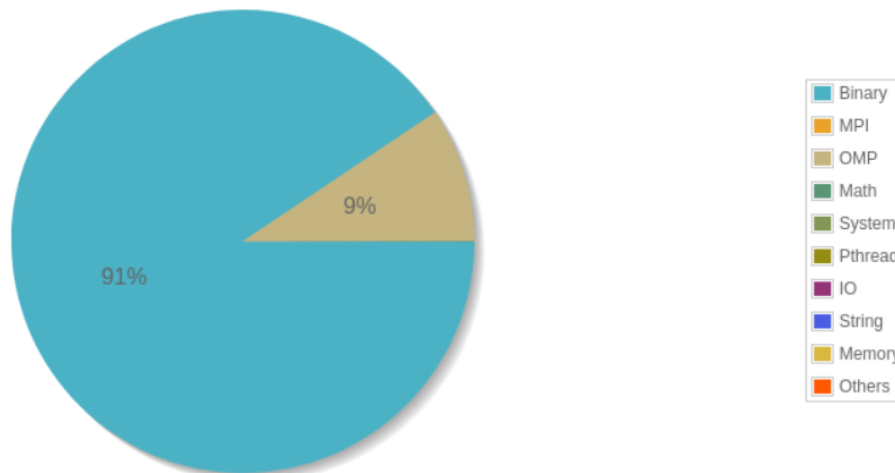


Figure 21 : Catégorisation de la fonction

L'accélération optimale que l'on obtiendrait si les outils OpenMP, MPI et Pthread sont correctement utilisés est égale à 1.07.

L'accélération optimale obtenue en utilisant parfaitement les outils cités précédemment et en ayant une bonne répartition des charges aux différents threads est également égale à 1.12, ce qui est correct mais peut être amélioré.

Le nombre de processus et de threads observés est de 256. La fonction est par conséquent exécutée en parallèle.

### 2.3.3 Compilation avec icc :

Lorsque la compilation est réalisé avec icc, nous obtenons le profilage suivant :

Global Metrics <span>?</span>	
Total Time (s)	0.82
Profiled Time (s)	0.82
Time in analyzed loops (%)	0.42
Time in analyzed innermost loops (%)	0.10
Time in user code (%)	92.5
Compilation Options	Not Available
Perfect Flow Complexity	Not Available
Array Access Efficiency (%)	Not Available
Perfect OpenMP + MPI + Pthread	1.17
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.30

Figure 22 : Profilage de la fonction optimisée en utilisant icc

Nous pouvons observer que le temps total pour profiler les performances de la fonction est égal à 0.82 secondes, tout comme son temps d'exécution.

Les boucles représentent 0.42% du temps passé d'exécution de la fonction. Le traitement des ces boucles prend 0.0034 secondes.

### Loop Based Profile

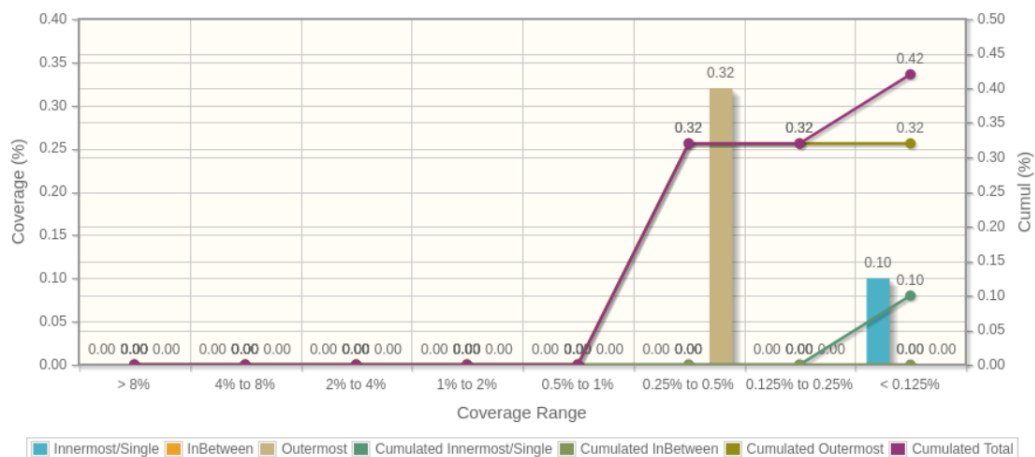


Figure 23 : Profilage lié aux boucles

Les boucles imbriquées, quant à elles, représentent 0.10% du temps d'exécution de la fonction. Le traitement de ces boucles prend 0.00082 secondes à la fonction.

### Innermost Loop Based Profile

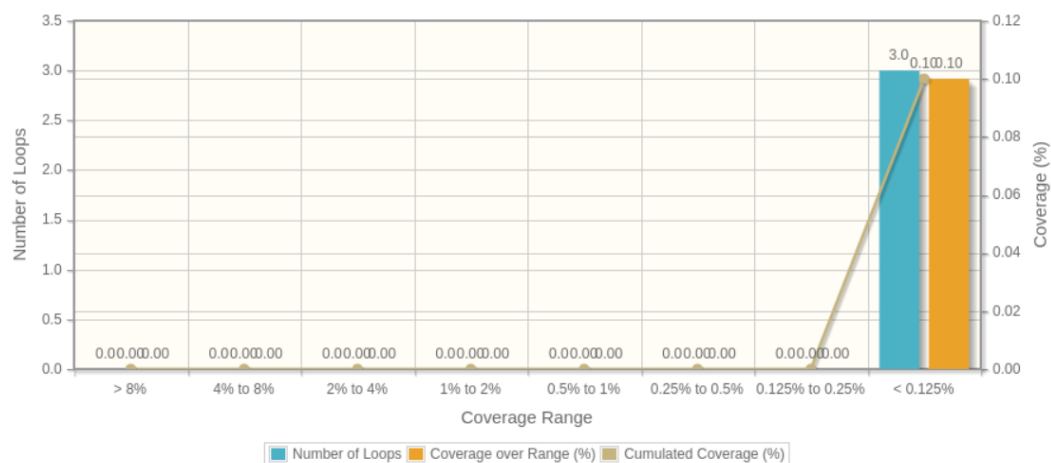


Figure 24 : Profilage lié aux boucles imbriquées

La fonction a passé 92,5% du temps dans les fichiers utilisateurs dont 92% concerne les fichiers binaires et 7% OMP.

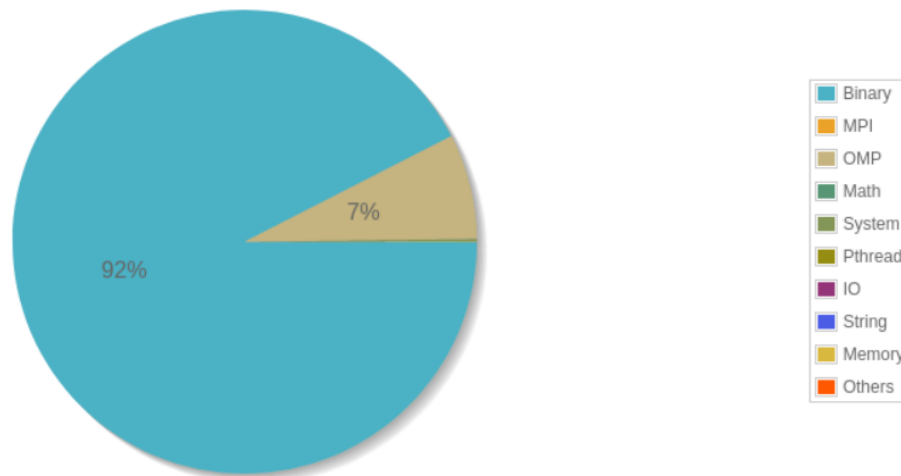


Figure 25 : Catégorisation de la fonction

L'accélération optimale que l'on obtiendrait si les outils OpenMP, MPI et Pthread sont correctement utilisés est égale à 1.17.

L'accélération optimale obtenue en utilisant parfaitement les outils cités précédemment et en ayant une bonne répartition des charges aux différents threads est également égale à 1.30.

Ces deux résultats sont corrects mais peuvent être améliorés.

Le nombre de processus et de threads observés est de 256. La fonction est par conséquent exécutée en parallèle.

### 2.3.3 Compilation avec icx :

Lorsque la compilation est réalisée avec icx, nous obtenons le profilage suivant :

Global Metrics <span>?</span>	
Total Time (s)	1.26
Profiled Time (s)	1.26
Time in analyzed loops (%)	93.0
Time in analyzed innermost loops (%)	92.8
Time in user code (%)	93.0
Compilation Options	Not Available
Perfect Flow Complexity	Not Available
Array Access Efficiency (%)	Not Available
Perfect OpenMP + MPI + Pthread	1.11
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.20

Figure 26 : Profilage de la fonction optimisée en utilisant icx

Nous pouvons observer que le temps total pour profiler les performances de la fonction est égal à 1.26 secondes, tout comme son temps d'exécution.

Les boucles représentent 93% du temps passé d'exécution de la fonction. Le traitement des ces boucles prend 1.172 secondes.

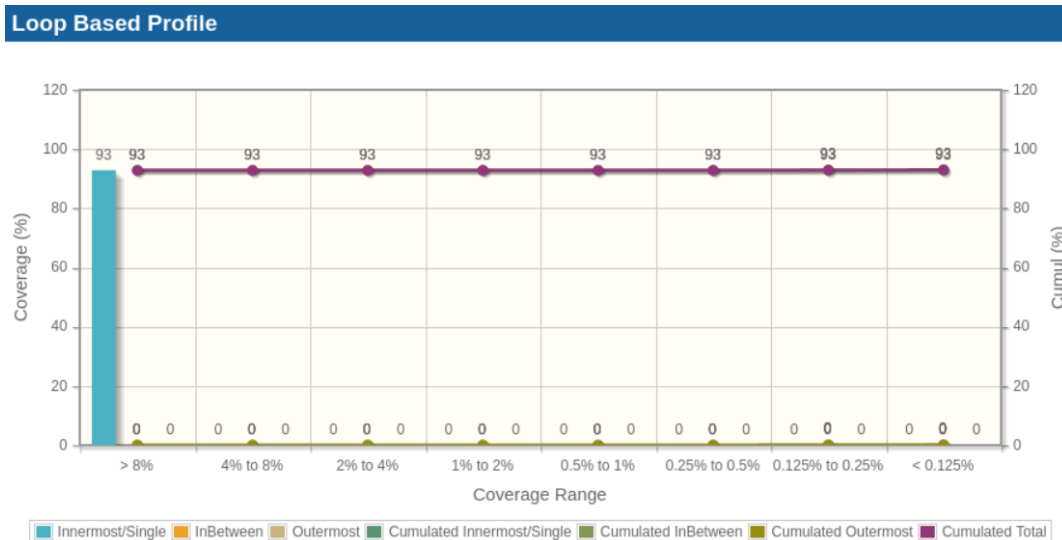


Figure 27 : Profilage lié aux boucles

Les boucles imbriquées, quant à elles, représentent 92.8% du temps d'exécution de la fonction. Le traitement des ces boucles prend 1.169 secondes à la fonction.

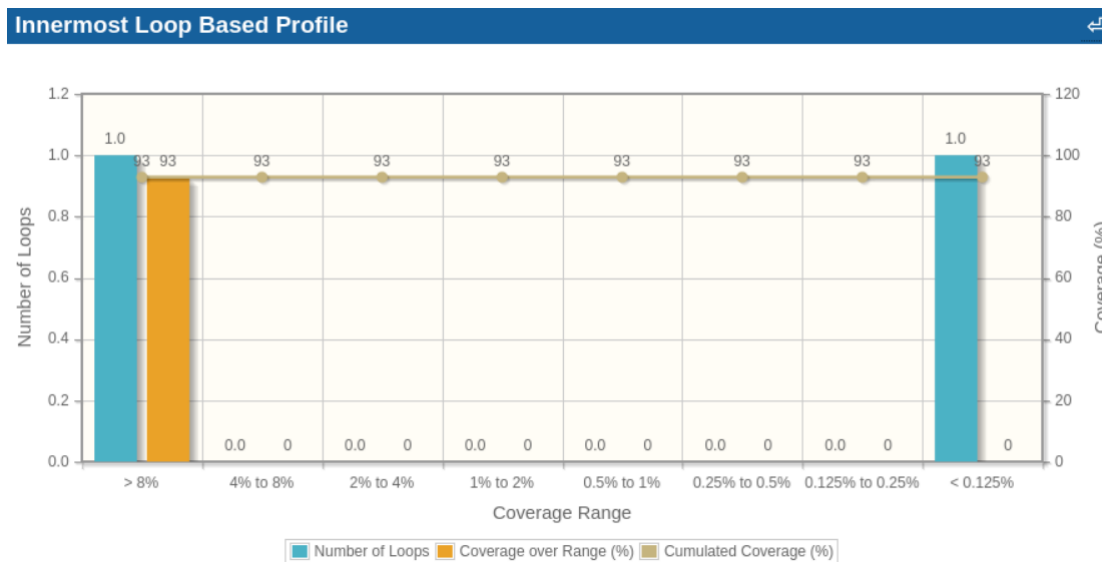


Figure 28 : Profilage lié aux boucles imbriquées

La fonction a passé 93% du temps dans les fichiers utilisateurs dont 93% concerne les fichiers binaires et 7% OMP.



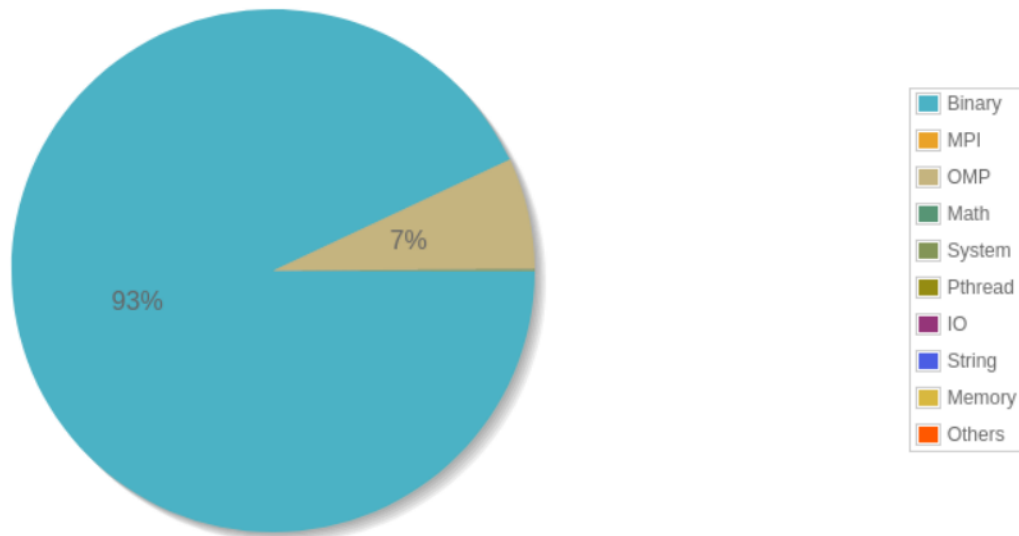


Figure 29 : Catégorisation de la fonction

L'accélération optimale que l'on obtiendrait si les outils OpenMP, MPI et Pthread sont correctement utilisés est égale à 1.11.

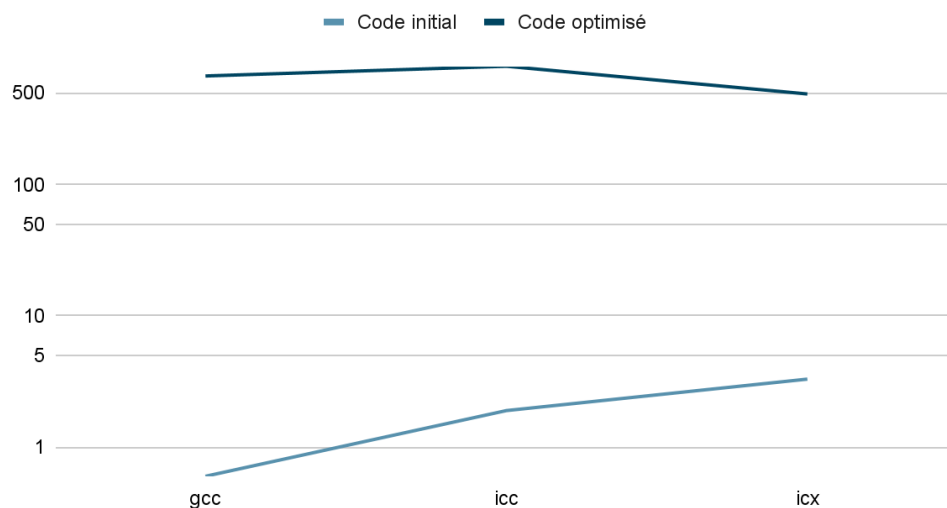
L'accélération optimale obtenue en utilisant parfaitement les outils cités précédemment et en ayant une bonne répartition des charges aux différents threads est également égale à 1.20. Ces deux résultats sont corrects mais peuvent être améliorés.

Le nombre de processus et de threads observés est de 256. La fonction est par conséquent exécutée en parallèle.

#### **2.4. Comparaison des performances entre versions de code et compilateurs :**

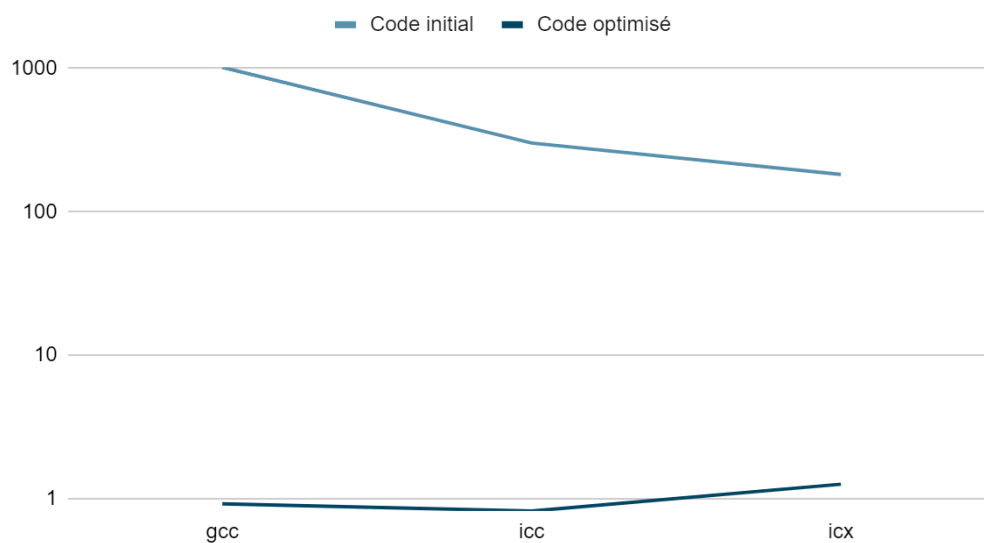
Vitesse de calcul du système (en gigaFlops)		
	Code initial	Code amélioré
gcc	0.6	674.5
icc	1.9	800.1
icx	3.3	491.8

## Vitesse de calcul du système (en gigaFlops)



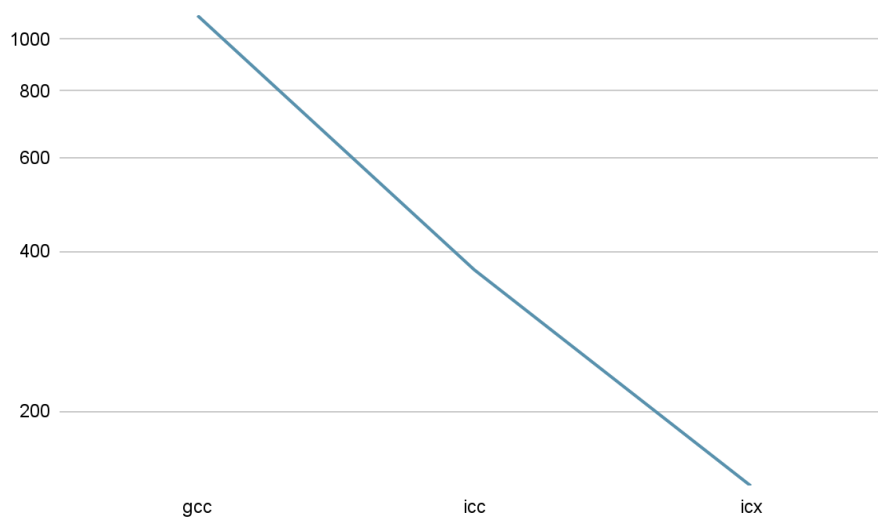
Temps d'exécution (en secondes)		
	Code initial	Code amélioré
gcc	1020	0.92
icc	302.66	0.82
icx	182.49	1.26

## Temps d'exécution (en secondes)



Compilateurs	Accélération (en s)
gcc	$(1020/0.92) = 1108.69$
icc	$(302.66/0.82) = 369.09$
icx	$(182.49/1.26) = 144.83$

## Accélération



## **Conclusion**

Nous pouvons observer que le code a globalement de meilleures performances lorsque son contenu est amélioré et parallélisé.

L'exécution du code initial donne une moyenne de 1.93 gigaFlop/s, l'exécution du code optimisé donne, quant à lui, une moyenne de 655.47 gigaFlop/s. On passe également d'une moyenne de temps d'exécution de 501.72 secondes à une moyenne d'une seconde. La vitesse de calcul est par conséquent beaucoup plus élevée avec le code optimisé et son temps d'exécution est moindre.

On obtient de meilleures performances avec le code initial, lorsqu'il est compilé avec icx. En ce qui concerne le code optimisé, les meilleures performances sont obtenues avec icc.

# **Bibliographie**

- Maqao : [http://www.maqao.org/release/MAQAO\\_QuickReferenceSheet\\_V12.pdf](http://www.maqao.org/release/MAQAO_QuickReferenceSheet_V12.pdf)
- OpenMP : <https://www.openmp.org/>