Rapport Master M1

Master Calcul Haute Performance Simulation (CHPS)

Amélioration des performances du problème "nbody3D"

Réalisé par: Aicha Maaoui

Janvier 2022

Institut des Sciences et Techniques des Yvelines (ISTY)

Abstract

Le but de ce rapport est d'améliorer les performances du code "**nbody.c**" proposé en TP de **l'Architecture parallèle**.

Ce rapport comporte les parties suivantes:

- Partie théorique du problème "nbody3D",
- Améliorations apportées au problème "nbody3D",

Contents

List of Figures

List of Tables

Chapter 1

Amélioration des performances

1.1 Partie théorique du problème "nbody3D"

On dispose de N particules (bodies) avec une position intiale x_i et vitesse v_i.

Soit G la constante gravitationnelle, alors la force f_{ij} de la particule i causée par l'attraction gravitationelle est, *"Improving performance of the N-Body problem"* 2016:

$$f_{ij} = G \times \frac{m_i \times m_j}{\|r_{ij}\|^2} \times \frac{r_{ij}}{\|r_{ij}\|} \tag{1.1}$$

avec:

$$\begin{cases} m_i,\ m_j & : \text{les masses des particules,} \\ r_{ij} = x_j - x_i & : \text{le vecteur distance du particule } i \text{ au particule } j. \end{cases}$$

La somme F_i des forces sur une particule i suite aux interactions aves les $(N-1)$ particules est donnée dans l'équation (1.2).

$$F_i = \sum_{\substack{1 \le j \le N \\ j \neq i}} f_{ij} = G \times m_i \times \sum_{\substack{1 \le j \le N \\ j \neq i}} \frac{m_i \times r_{ij}}{\|r_{ij}\|^3} \tag{1.2}$$

1.2 Code Initial du problème "nbody3D"

La fonction principale du programme "nbody.c" est donnée par la fonction *move_particles(particle_t *p, const f32 dt, u64 n)*, comme presenté dans listing (1.1).

```
1  void move_particles(particle_t *p, const f32 dt, u64 n){
2      % const f32 softening = 1e-20;  //Modif. du loi gravit. G at r < softening
3
4      for (u64 i = 0; i < n; i++){ //Boucle sur le nombre de particules n
5
6          //Initialisation des Net force fi
7          f32 fx = 0.0; //initialisation de fx
8          f32 fy = 0.0; //initialisation de fy
9          f32 fz = 0.0; //initialisation de fz
10
11         //23 floating-point operations
12         for (u64 j = 0; j < n; j++){
13             //Newton's law
14             const f32 dx = p[j].x - p[i].x; //1
15             const f32 dy = p[j].y - p[i].y; //2
16             const f32 dz = p[j].z - p[i].z; //3
17             const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //9
18             const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0); //11, (d_2)^(3/2)
19
20             //Net force fi
21             fx += dx / d_3_over_2; //13
22             fy += dy / d_3_over_2; //15
23             fz += dz / d_3_over_2; //17
24         }
25         p[i].vx += dt * fx; //19
26         p[i].vy += dt * fy; //21
27         p[i].vz += dt * fz; //23
28     }
29     //3 floating-point operations
30     for (u64 i = 0; i < n; i++){
31         p[i].x += dt * p[i].vx;
32         p[i].y += dt * p[i].vy;
33         p[i].z += dt * p[i].vz;    } }
```

Listing 1.1: Version initiale de la fonction de nbody3D *move_particles*.

La disposition de la mémoire dans la version initiale est: Array of Structures (AOS), comme presenté dans listings (1.1) et (1.2).

```
typedef struct particle_s { //structure
    f32 x, y, z; //f32 pour float
    f32 vx, vy, vz;
} particle_t;
```

Listing 1.2: Structure utilisée dans Array of Structures (AOS) de la version initiale.

Cette disposition dans la RAM est illustrée dans la figure (1.1).

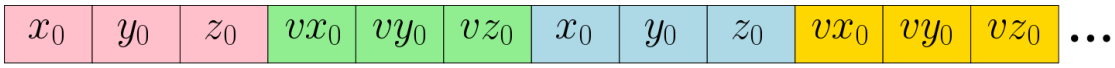

Figure 1.1: Array of structures (AOS) dans la RAM.

Les désavantages de cette disposition sont les suivantes, *"Improving performance of the N-Body problem"* 2016:

- Exécution des opérations SIMD sur le format AOS peut nécessiter plus de calculs,
- Optimisation difficile par le compilateur.

1.2.1 Exécution du code initial

Suite à l'exécution du code avec le noeud de calcul *knl03* (cluster), on obtient une moyenne de performance égale à $(0.6 + -0.0 GFLOP/s)$ avec le compilateur *gcc* et une moyenne de performance égale à $(9.7 + -0.0 GFLOP/s)$ avec le compilateur *icc*, comme illustré dans les figures (1.2) et (1.3).

```
user2232@knl03:~/0.0$ make clean
rm -Rf *~ nbody.g nbody0.g nbody.i nbody0.i *.optrpt
user2232@knl03:~/0.0$ make nbody0.g
gcc -o -o3 -mavx2 -Ofast -ffast-math -fopt-info-all=nbody.gcc.optrpt nbody0.c -o nbody0.g -lm -fopenmp
user2232@knl03:~/0.0$ ./nbody0.g

Total memory size: 393216 B, 384 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  9.583e+00  2.801e+07      0.6 *
    1  9.591e+00  2.799e+07      0.6 *
    2  9.579e+00  2.802e+07      0.6 *
    3  9.592e+00  2.798e+07      0.6
    4  9.580e+00  2.802e+07      0.6
    5  9.584e+00  2.801e+07      0.6
    6  9.592e+00  2.798e+07      0.6
    7  9.579e+00  2.802e+07      0.6
    8  9.595e+00  2.798e+07      0.6
    9  9.582e+00  2.801e+07      0.6
-----------------------------------------------------
Average performance:          0.6 +- 0.0 GFLOP/s
-----------------------------------------------------
```

Figure 1.2: Compilation de "nbody.c" avec le compilateur *gcc*.

```
user2232@knl03:~/0.0$ make nbody0.i
icc -xhost -Ofast -qopt-report nbody0.c -o nbody0.i -qmkl -qopenmp
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
user2232@knl03:~/0.0$ ./nbody0.i

Total memory size: 393216 B, 384 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  6.484e-01  4.140e+08      9.5 *
    1  6.334e-01  4.238e+08      9.7 *
    2  6.326e-01  4.243e+08      9.8 *
    3  6.344e-01  4.231e+08      9.7
    4  6.336e-01  4.236e+08      9.7
    5  6.387e-01  4.203e+08      9.7
    6  6.422e-01  4.180e+08      9.6
    7  6.354e-01  4.224e+08      9.7
    8  6.324e-01  4.244e+08      9.8
    9  6.329e-01  4.241e+08      9.8
-----------------------------------------------------
Average performance:          9.7 +- 0.0 GFLOP/s
-----------------------------------------------------
```

Figure 1.3: Compilation de "nbody.c" avec le compilateur *icc*.

1.2.2 Etude de performance et optimisations possibles avec MAQAO

Pour pouvoir optimiser le code initial, on a fait recours à "MAQAO" qui était installé dans mon propre laptop (*"MAQAO (2004-2021)"* 2021). La génération du rapport à l'aide de ce dernier est possible à l'aide du commande: "*maqao oneview –create-report=one –binary=./executable*".

Les figure (1.4), (1.5) et (1.6) illustrent les indices de performance du code donnés avec MAQAO.

Global Metrics

Total Time (s)		8.26
Profiled Time (s)		8.26
Time in analyzed loops (%)		100
Time in analyzed innermost loops (%)		100
Time in user code (%)		100
Compilation Options		Not Available
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		75.0
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%	1
FP Vectorised	Potential Speedup	5.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	5.33
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.11
	Nb Loops to get 80%	1

Figure 1.4: "Global Metrics" du code initial obtenus avec "MAQAO".

On remarque d'après la figure (1.4) que le temps total d'exécution du code est 8.26s. L'efficacité d'accès aux tableaux est 75%. Les cases colorées en orange indiquent la possibilité de la vectorisation du code pour améliorer ses performances. Ceci est confirmé également dans la

Loops Index

▶ Filters

▼ Colums Filter

☑ Level ☑ Coverage run_0 (%) ☑ Max Time Over Threads run_0 (s) ☑ Time w.r.t. Wall Time run_0 (s) ☑ Nb Threads run_0 ☑ Vectorization Ratio (%) ☑ Vectorization Efficiency (%) ☑ Speedup If No Scalar Integer
☑ Speedup If FP Vectorized ☑ Speedup If Fully Vectorized ☑ Speedup If Perfect Load Balancing run_0 ☑ Stride 0 ☑ Stride 1 ☑ Stride n ☑ Stride Unknown ☑ Stride Indirect Select none Select All Coverages
Select All Times

Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Max Time Over Threads run_0 (s)	Time w.r.t. Wall Time run_0 (s)	Nb Threads run_0	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	Speedup If Perfect Load Balancing run_0	Stride 0	Stride 1	Stride n	Stride Unknown	Stride Indirect
2	nbody0.g -	move_particles	Innermost	100	8.26	8.26	1	0	7.07	1	5	5.33	1	0	0	1	0	0

Figure 1.5: "Loops Index" du code initial obtenus avec "MAQAO".

Software Topology

Number processes: 1 Number nodes: 1 Run Command: <executable> MPI Command: Dataset: Run Directory: .

ID	Observed Processes	Observed Threads	Time(s)
▶ Node aicha-Vostro-3500	1	1	8.26

Figure 1.6: Array of structures (AOS) dans RAM.

figure (1.5) par le ratio nul de la vectorisation. De plus, si le code est vectorisé, alors on peut obtenir jusqu'à 5 fois de vitesse "Speed Up" (figure (1.5)).

Les processeurs et Threads observés sont égales à 1, comme montré dans la figure (1.6).

1.3 Changement de disposition de la mémoire de "AOS" à "SOA"

Dans un premier temps, on a changé la disposition de AOS à SOA (Structure of Array) dans listing (1.3) et listing (1.4), également illustré dans la figure (1.7), *"Improving performance of the N-Body problem"* 2016.

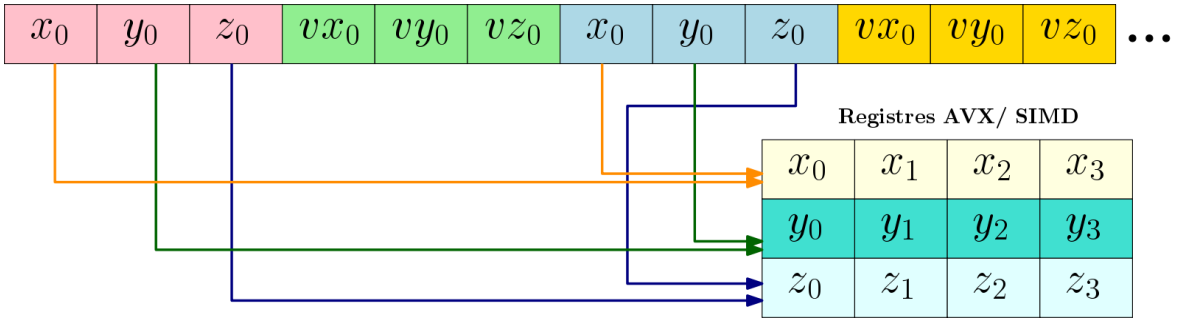

Figure 1.7: "Structure of Array (SOA)" dans la RAM.

```
1 typedef struct particle_s { //structure
2     f32 *x, *y, *z; //f32 pour float
3     f32 *vx, *vy, *vz;
4 } particle_t;
```

Listing 1.3: Structure utilisée dans "Structure of Array (SOA)" de la version optimisée.

```
1 void move_particles(particle_t p, const f32 dt, u64 n){
2     const f32 softening = 1e-20;
```

```
3
4       #pragma omp parallel for schedule(dynamic) // OpenMP assigns one iteration to
         each thread, Ref: http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/
        OpenMP_Dynamic_Scheduling.pdf
5
6       for (u64 i = 0; i < n; i++){
7           f32 fx = 0.0; //initialisation du fx
8           f32 fy = 0.0; //initialisation du fy
9           f32 fz = 0.0; //initialisation du fz
10
11      #pragma vector aligned //All memory-based instructions in the coprocessor
        should be aligned to avoid instruction faults, data access is aligned by 64
        bytes
12
13      #pragma omp simd // Ignore all dependencies inside a loop ==> for data
        vectorization
14
15      //23 floating-point operations
16      for (u64 j = 0; j < n; j++)     {
17          //Newton's law
18          const f32 dx = p.x[j] - p.x[i]; //1
19          const f32 dy = p.y[j] - p.y[i]; //2
20          const f32 dz = p.z[j] - p.z[i]; //3
21          const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //9
22          //const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0); //11
23          const f32 d_3_over_2 = d_2 * sqrt(d_2); //11
24
25          //Net force
26          fx += dx * (1/ d_3_over_2); //13
27          fy += dy * (1/ d_3_over_2); //15
28          fz += dz * (1/ d_3_over_2); //17
29      }
30
31      p.vx[i] += dt * fx; //19
32      p.vy[i] += dt * fy; //21
33      p.vz[i] += dt * fz; //23
34      }
35      //3 floating-point operations
36      for (u64 i = 0; i < n; i++){
```

```
37          p.x[i] += dt * p.vx[i];
38          p.y[i] += dt * p.vy[i];
39          p.z[i] += dt * p.vz[i]; } }
```

Listing 1.4: Fonction "*move_particles*" utilisée dans la version optimisée "SOA".

Les avantages d'utilisation de "Structure of Array (SOA)" sont comme suit, *"Improving performance of the N-Body problem"* 2016:

- Accès mémoire contigus lors de la vectorisation,
- Utilisation plus efficace du parallélisme du SIMD (données calculées de manière plus optimale et de manière verticale),
- Utilisation plus efficace des caches et de la bande passante.

1.4 Changement des expressions utilisées dans le code

Dans listing (1.4), on a aussi effectué les modifications suivantes à part l'arrangement mémoire "SOA":

- **Changement de l'expression de "**$d_3\_over\_2$**" de "**$pow(d_2, 3.0/2.0)$**" à "**$d_2 \times sqrt(d_2)$**"** (listing(1.4), ligne 23). En effet, d'après (*"Difference between sqrt(x) and pow(x,0.5)"* 2014), la lecture de "$sqrt(x)$" est plus facile que "$pow(x, nb)$" et par conséquent a plus de performance et moins de temps de calcul,
- **Changement de l'expression de f_i, avec {$i = x, y, z$} de "**$f_i = di/d_3\_over\_2$**" à "**$f_i = di \times (1/d_3\_over\_2)$**"** (listing(1.4), lignes 26, 27 et 28). Ceci permet d'optimiser le code et de conserver sa clarité. D'après (*"Devrais-je utiliser la multiplication ou la division?"* 2008), le compilateur fera la division au moment de la compilation, et on obtientiendra une multiplication au moment de l'exécution.

1.5 Vectorisation du Code avec OpenMP

La vectorisation, ou la parallélisation des données, consiste à modifier l'exécution d'une seule opération sur un seul thread à la fois à plusieurs opérations qui s'effectuent de manière simultanée, *"Vectorisation"* 2021. A titre d'exemple, on considère l'operation de l'addition ($c[i] = a[i] + b[i],\ \ i \in [1, 4]$) dans la figure (1.8). Ainsi, ces additions s'effectueront simultanément.

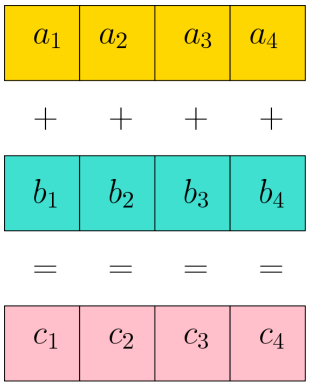

Figure 1.8: Vectorisation de l'addition des données ($c[i] = a[i] + b[i]$).

Par conséquent, on obtient un programme vectorisé à partir d'un programme scalaire, *"Vectorisation"* 2021. Dans la suite, on considère la la vectorisation SIMD et la planification des boucles avec *schedule*.

1.5.1 Planification des boucles (Loop Scheduling) dans OpenMP

OpenMP est utilisé pour spécifier que les instructions des boucles sont exécutées en parallèle par des threads, *"Loop Scheduling in OpenMP"* 2017. Ainsi, on utilise *OpenMP* comme montré dans listing (1.5).

```
1  #pragma omp parallel for [clause[ [,] clause] ... ]
2      for (...) //Boucle (Loop)
3          { ... }
```

Listing 1.5: Utilisation de *OpenMP*.

Dans cette section, on considère *schedule(...)* comme clause (listing (1.5)). Schedule dans *OpenMP* décrit la façon dont les itérations des boucles sont divisées en sous-ensembles contigus non vides, *"Loop Scheduling in OpenMP"* 2017. Par suite, il faut spécifier le type du *scheduling* utilisé, comme montré dans listing (1.6). Ce dernier indique la façon dont les itérations de la boucle OpenMP doivent être affecté à des threads, *"Loop Scheduling in OpenMP"* 2017.

```
1  #pragma omp parallel for schedule(Scheduling Type) //Scheduling
2      for (...) //Boucle (Loop)
3          { ... }
```

Listing 1.6: Type du *Scheduling* dans *OpenMP*.

Dans ce rapport, on a testé trois types de *scheduling*, qui sont:

- Static,
- Dynamic,
- Guided.

D'après *"OpenMP Scheduling"* n.d. et *"OpenMP"* n.d., le tableau (1.1) regroupe les trois types de *Scheduling* considérés. Soit les commandes suivantes utilisée:

- Static: "*pragma omp parallel for schedule(static)*",
- Dynamic: "*pragma omp parallel for schedule(dynamic)*",
- Guided: "*pragma omp parallel for schedule(guided)*".

Type du Scheduling	**Description**	**Perf. avec compilateur icc (Sans ajout de flags de compilation)**
Static	Les itérations sont divisées au temps de la compilation (défaut)	$897.4 + -8.2 GFLOP/s$
Dynamic	*OpenMP* attribue une itération à chaque thread. Quand le thread se termine, il reçoit la prochaine itération non exécutée.	$921.0 + -2.5 GFLOP/s$
Guided	Similaire au *Dynamic Scheduling*. La différence est que la taille du bloc change au cours de l'exécution du programme (puis, s'ajuste à des tailles de blocs plus petites dans le cas où la charge de travail est déséquilibrée).	$891.7 + -2.0 GFLOP/s$

Table 1.1: Types de *Scheduling* utilisés.

Donc, dans notre cas, *Dynamic Scheduling* est le plus adapté, car le travail sera réparti d'une manière plus équitable entre les threads.

Ceci est montré dans la ligne (4) du listing (1.4): "*pragma omp parallel for schedule(dynamic)*". Le but de cette ligne est d'atribuer un travail distribué aux instructions au lieu de partagé. Ceci permet d'éviter que les instructions attendent l'arrivée du dernier thread et réduire le temps d'attente et de calcul, *"D. Clause schedule"* 2021.

La planification dynamique (*schedule(dynamic)*) est utilisée pour une construction "*for*", avec les itérations utilisants des volumes variables de travail (*"D. Clause schedule"* 2021). Elle est caractérisée par le fait qu'un thread utilise un autre thread pour pouvoir exécuter sa dernière itération au lieu d'attendre au cloisonnement, *"D. Clause schedule"* 2021.

1.5.2 Vectorisation SIMD et alignement des données

Vectorisation SIMD

On s'intéresse dans cette section, dans un premier temps, au ligne (13) ("*pragma omp simd*") dans listing (1.4). Il s'agit d'un impératif qui oblige le compilateur à tout essayer pour vectoriser, *"Improving performance of the N-Body problem"* 2016. Ainsi, l'addition de ("*pragma omp simd*") au-dessus d'une boucle se trouve une directive OpenMP SIMD qui indique au compilateur qu'il doit vectoriser cette boucle, *"How to vectorise? (omp simd)"* n.d.

Alignement des données

On s'intéresse dans cette section, dans un deuxième temps, au ligne (11) ("*pragma vector aligned*") dans listing (1.4). Une optimisation supplémentaire consiste au fait que la mémoire contenant les données soit également correctement alignée, *"omp simd limitations"* n.d.

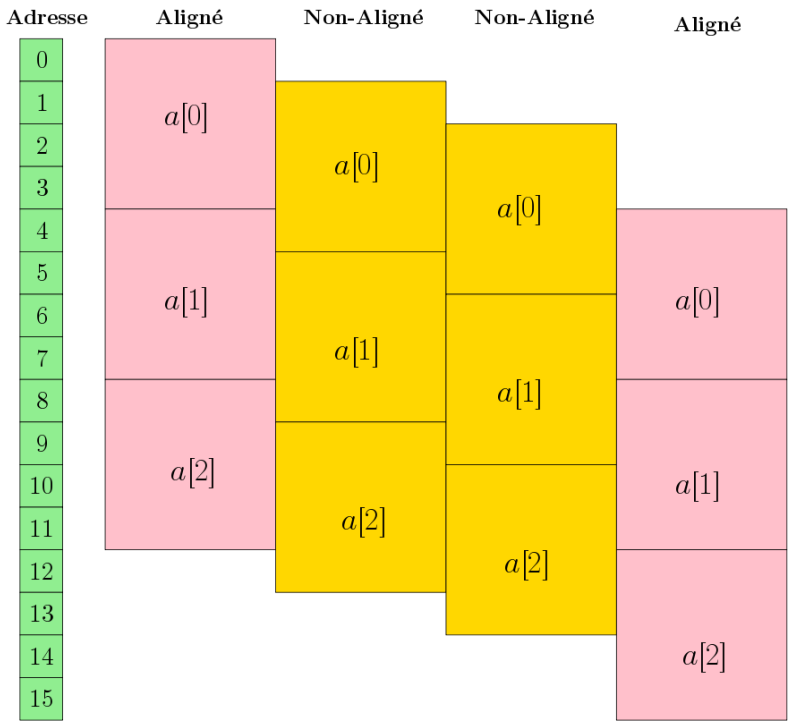

Figure 1.9: Données alignées ou non-alignées.

La figure (1.9) illustre un tableau de trois *floats*: $a[0]$, $a[1]$ et $a[2]$. Chaque *float* utilise 4 octets (32 bits) pour stocker les données. On a, d'après *"omp simd limitations"* n.d.:

- $a[0]$ occupe la mémoire de l'adresse d'octet 0 à 3,
- $a[1]$ occupe la mémoire de l'adresse d'octet 4 à 7,
- $a[2]$ occupe la mémoire de l'adresse d'octet 8 à 11.

Ainsi, le tableau est aligné sur 4 octets, car l'adresse du premier bit de chaque entrée (0, 4, 8) est divisible par 4.

Dans les cas de données non-alignées (cas 2 et 3), le tableau est alloué à des adresses d'octets décalées d'un octet. Soit:

- $a[0]$ est à l'octet 1 (cas 2 de données non-alignées),
- $a[0]$ est à l'octet 2 (cas 3 de données non-alignées).

Comme les adresses du premier octect de chaque entrée n'est pas divisible par 4, les tableaux ne sont pas alignés sur 4 octets.

Le processeur, les registres et les sous-système de mémoire d'un ordinateur fonctionne plus efficacement lorsque les données sont alignées d'une manière particulière, par exemple aligné sur 4 octets (32 bits) ou aligné sur 8 octets (64 bits), *"omp simd limitations"* n.d.

1.6 Passage du compilateur *gcc* au compilateur *icc*

Bien que les instructions utilisés précédemment permettent d'écrire un code vectorisable, les amélioration des performances dépendent de la qualité du compilateur et du processeur sur lequel le code sera exécuté.

On se propose dans cette section de comparer les performances de deux compilateurs *gcc* et *icc*, comme illustré dans les figures (1.10) et (1.11).

D'après les figures ci-dessus, on remarque qu'en utilisant le compilateur *icc*, on obtient une optimisation de performance de $\frac{921.0}{248.8} \approx 4$ par rapport au compilateur *gcc*.
En effet, d'après *"Benchmark Comparison for Different Compilers"* n.d., le compilateur *icc* est utile lorsqu'on a des opérations en virgule flottante et de grande gestion de mémoire. Dans la suite, on va considérer les performances obtenues en utilisant le compilateur *icc*.

```
user2232@knl03:~/0.0$ make nbody.g
gcc -o -o3 -mavx2 -Ofast -ffast-math -fopt-info-all=nbody.gcc.optrpt nbody.c -o nbody.g -lm -fopenmp
user2232@knl03:~/0.0$ ./nbody.g

Total memory size: 4718592 B, 4608 KiB, 4 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  7.115e-02  3.773e+09     86.8 *
    1  3.963e-02  6.773e+09    155.8 *
    2  3.854e-02  6.965e+09    160.2 *
    3  4.810e-02  5.580e+09    128.3
    4  3.007e-02  8.927e+09    205.3
    5  2.815e-02  9.536e+09    219.3
    6  2.077e-02  1.292e+10    297.2
    7  2.077e-02  1.292e+10    297.2
    8  2.079e-02  1.291e+10    296.9
    9  2.076e-02  1.293e+10    297.3
-----------------------------------------------------
Average performance:       248.8 +- 61.7 GFLOP/s
-----------------------------------------------------
```

Figure 1.10: Performances en utilisant le compilateur *gcc*.

```
user2232@knl03:~/0.0$ make clean
rm -Rf *~ nbody.g nbody0.g nbody.i nbody0.i nbody.d *.optrpt
user2232@knl03:~/0.0$ make nbody.d
icc -xhost -Ofast -qopt-report nbody.c -o nbody.d -qmkl -qopenmp
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
^[[Auser2232@knl03:~/0.0$ ./nbody.d

Total memory size: 4718592 B, 4608 KiB, 4 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  1.245e-01  2.155e+09     49.6 *
    1  6.721e-03  3.994e+10    918.6 *
    2  6.714e-03  3.998e+10    919.5 *
    3  6.734e-03  3.986e+10    916.8
    4  6.685e-03  4.015e+10    923.5
    5  6.714e-03  3.998e+10    919.5
    6  6.721e-03  3.994e+10    918.6
    7  6.680e-03  4.018e+10    924.2
    8  6.695e-03  4.009e+10    922.1
    9  6.695e-03  4.009e+10    922.1
-----------------------------------------------------
Average performance:       921.0 +- 2.5 GFLOP/s
-----------------------------------------------------
```

Figure 1.11: Performances en utilisant le compilateur *icc*, sans flags additionnels de compilation.

1.7 Ajout des Flags de compilation au compilateur Intel "*icc*"

D'après *"Intel Compiler Flags"* 2022, il est recommandé d'utiliser les flags de compilations dans listing (1.6) pour une compatibilité maximale entre les noeuds de calcul et les meilleures performances.

```
1  icc -Ofast -msse4.2 -axAVX,CORE-AVX2
```

Listing 1.7: Flags de compilation recommandés pour le compilateur *icc*.

1.8 Résultats

En considérant toutes les options décrites dans les sections précédentes, la compilation du code (à partir du fichier *makefike*) donnant le maximum de performance à partir du programme *nbody.c* est donnée dans listing (1.8).

```
1 nbody.i: nbody.c
2     icc -xhost -Ofast -qopt-report $< -o $@ -qmkl -qopenmp -msse4.2 -axAVX,CORE-
      AVX2
```

Listing 1.8: Commande d'exécution dans le fichier *makefile* donnant le maximum de performance.

Soit une performance totale de $924.1 + -2.4 GFLOP/s$, comme montré dans la figure (1.12).

```
user2232@knl03:~/0.0$ make clean
rm -Rf *~ nbody.g nbody0.g nbody.i nbody0.i *.optrpt
user2232@knl03:~/0.0$ make nbody.i
icc -xhost -Ofast -qopt-report nbody.c -o nbody.i -qmkl -qopenmp -msse4.2 -axAVX,CORE-AVX2
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
user2232@knl03:~/0.0$ ./nbody.i

Total memory size: 4718592 B, 4608 KiB, 4 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  1.325e-01  2.026e+09     46.6 *
    1  6.677e-03  4.020e+10    924.6 *
    2  6.705e-03  4.003e+10    920.8 *
    3  6.703e-03  4.004e+10    921.0
    4  6.666e-03  4.027e+10    926.1
    5  6.694e-03  4.010e+10    922.3
    6  6.696e-03  4.009e+10    922.0
    7  6.654e-03  4.034e+10    927.8
    8  6.664e-03  4.028e+10    926.4
    9  6.689e-03  4.013e+10    923.0
-----------------------------------------------------
Average performance:          924.1 +- 2.4 GFLOP/s
-----------------------------------------------------
```

Figure 1.12: Performance finale avec le compilateur *icc*, avec flags de compilation.

On résume ainsi les performances obtenues dans le tableau (1.2).

Compilateur	Version initiale	Version optimisée	Speed Up
gcc	$0.6 + -0.0 GFLOP/s$	$248.8 + -61.7 GFLOP/s$	≈ 415
icc	$9.7 + -0.0 GFLOP/s$	$924.1 + -2.4 GFLOP/s$	≈ 96

Table 1.2: Résultats d'optimisation pour les compilateurs *gcc* et *icc*.

On conclut que la meilleure performance est celle obtenue en utilisant les optimisations et en compilant avec le compilateur *icc*, qui est évaluée à $924.1 + -2.4 GFLOP/s$.

References

"Benchmark Comparison for Different Compilers" (n.d.). Anurag Aggarwal. URL: https://iitd-plos.github.io/col729/labs/lab0/lab0_submissions/siy187504.pdf.

"D. Clause schedule" (2021). Microsoft. URL: https://docs.microsoft.com/fr-fr/cpp/parallel/openmp/d-using-the-schedule-clause?view=msvc-170.

"Devrais-je utiliser la multiplication ou la division?" (2008). it-swarm-fr. URL: https://www.it-swarm-fr.com/fr/performance/devrais-je-utiliser-la-multiplication-ou-la-division/958436621/.

"Difference between sqrt(x) and pow(x,0.5)" (2014). Stackoverflow. URL: https://stackoverflow.com/questions/17417490/difference-between-sqrtx-and-powx-0-5.

"How to vectorise? (omp simd)" (n.d.). Chryswoods. URL: https://chryswoods.com/vector_c++/simd.html.

"Improving performance of the N-Body problem" (2016). Efim Sergeev, Lab LLC. URL: https://www.singularis-lab.com/docs/materials/LSA_2016_05_N-Body-Problem.pdf.

"Intel Compiler Flags" (2022). TechWeb. URL: https://www.bu.edu/tech/support/research/software-and-programming/programming/compilers/intel-compiler-flags/.

"Loop Scheduling in OpenMP" (2017). Vivek Kale. URL: https://www.openmp.org/wp-content/uploads/SC17-Kale-LoopSchedforOMP_BoothTalk.pdf.

"MAQAO (2004-2021)" (2021). MAQAO Team. URL: http://www.maqao.org/.

"omp simd limitations" (n.d.). Chryswoods. URL: https://chryswoods.com/vector_c++/limitations.html.

"OpenMP Scheduling" (n.d.). Université fédérale de Santa Catarina. URL: http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf.

"OpenMP" (n.d.). SERC. URL: http://www.serc.iisc.ac.in/serc_web/wp-content/uploads/2020/02/Akhila-OPENMP_Part2_final.pdf.

"Vectorisation" (2021). Wikipedia. URL: https://fr.wikipedia.org/wiki/Vectorisation_(informatique).

Appendix A

Dépot Github

Les codes associé à ce rapport sont déposés dans le dépot git **"Nbody3D"**. Le code SSH de ce dépot est le suivant:

git@github.com:Chaichas/Nbody3D.git

Lien Github: `https://github.com/Chaichas/Nbody3D`