M1 CHPS - AP - TP nbody3D

Rapport d'analyse de performances

Candice Astier

22 janvier 2022

Table des matières

1 Introduction et son contexte

Le programme nbody3D (implémenté dans le fichier nbody0.c) permet de simuler des corps en mouvement dans un espace en 3 dimensions. Il est composé de trois fonctions principales :

— une fonction *init*() : permet d'initialiser toutes les variables et données du programme,
— une fonction *move_particles*() : permet de modifier à chaque pas de temps les positions de l'ensemble des corps stockés (selon les trois axes x, y et z),
— une fonction *main*() : permet de contrôler et d'étudier le programme.

L'objectif de ce TP est d'optimiser nbody3D et plus particulièrement la fonction *move_particles*() qui est le centre du programme. Pour cela, huit autres versions de ce code, de plus en plus performantes, ont été développées.
L'ensemble des calculs de performances de ces huit versions a été réalisé sur un noeud de calcul de type KNL (Intel Xeon Phi - Knights Landing) en mode performance.

2 Optimisations

Afin optimiser le programme, il a été effectué plusieurs modifications que l'on peut regrouper en quatre grandes catégories. Celles-ci ont été majoritairement produites dans la fonction *move_particles*().

2.1 Modifications lors de la compilation

L'utilisation du compilateur icc (Intel) au lieu de gcc (GNU) a permis d'augmenter la performance [1].
Le compilateur icc (Intel) est en effet très utile lorsque l'on cherche à avoir des meilleurs temps CPU [2] sur un processeur Intel (ce qui est notre cas), et d'autant plus lorsque le code comporte beaucoup de calculs.

Des flags [3] d'optimisations ont également été ajoutés lors de la compilation des versions, tels que :

— **Ofast** : permet d'accélérer les performances d'un programme du point de vue du temps d'exécution principalement ;
— **xhost** : permet notamment d'indiquer au compilateur le processeur utilisé pour exécuter le programme, afin qu'il puisse créer un exécutable plus adapté et plus performant ;
— **ipo** : permet d'activer un ensemble de flags d'optimisations généraux, comme par exemple celui permettant d'indiquer que les données sont bien alignées en mémoire ;
— **mavx2** : permet d'indiquer au compilateur quels types d'instructions assembleur utiliser (ici AVX2) ;
— **funroll-all-loops** : permet de réduire le nombre de tours de boucles en augmentant le nombre d'instructions à l'intérieur, afin de saturer les registres [4] ce qui permet d'augmenter les chances de vectorisation et de réduire le temps d'exécution du programme ;
— **finline-functions** : permet de réduire le nombre d'appels à des fonctions, en les remplaçant par le corps de la fonction en elle-même. Ce flag permet de réduire la complexité des programmes, et donc de potentiellement l'accélérer.

2.2 Modifications de la structure du code

Une des modifications majeures de la structure du programme est la transformation des structures de données. Chaque corps étant représenté par trois positions, le stockage initial est un tableau de sous-structures. Chaque structure est composée de trois nombre réels (appelés flottants), représentant les trois coordonnées spatiales d'un corps. Ce type de stockage est appelé "Array of Struct" (tableau de structures).
La transformation a alors été de passer de ce type de stockage vers un autre type appelé "Struct of Array" (Structure de tableaux) : une unique structure comprenant trois tableaux (X, Y et Z) permettant de stocker chacune des positions des corps. On a alors dans la première case du tableau X la position sur l'axe des x du premier corps, dans la première case du tableau Y la position sur l'axe des y du premier corps, et, dans la première case du tableau Z, la position sur l'axe des z du premier corps.
Ce nouveau type de stockage est beaucoup moins lourd, ce qui surcharge moins les fonctions et les appels mémoire pour récupérer leurs données. De plus, le fait de trier les données dans trois tableaux va permettre d'accélérer la récupération de celles-ci, car comme dans le programme elles subissent les mêmes modifications, elle pourront être récupérés (en mémoire) et traitées en mêmes temps grâce à de la vectorisation.

1. Voir Annexes 6.1.
2. Un cycle CPU correspond à la durée d'une opération de base. Plus un programme utilisera de cycles CPU, plus son execution sera longue.
3. Options passées lors de la compilation.
4. Petites mémoires internes aux processeurs, principalement utilisé pour le stockage de variables.

Une autre modification de la structure du code est la fusion de deux boucles consécutives de mêmes tailles (tours de boucles). Cela permet notamment d'augmenter le pourcentage de vectorisation potentielle, ainsi que de mieux exploiter la taille des registres. Cela a également comme conséquence indirecte de réduire le temps d'exécution générale.

2.3 Modifications de l'allocation en mémoire

Le passage de la fonction d'allocation mémoire dynamique *malloc*() à la fonction *aligned_alloc*() permet d'indiquer au compilateur d'allouer de la place dans des zones mémoires alignées les unes aux autres. Cela permet également d'accélérer l'exécution du programme, car lorsque le processeur récupère des informations en mémoire, il les récupère par blocs de données.
Comme le programme applique les mêmes opérations sur l'ensemble des tableaux à chaque répétition des algorithmes, si les données sont alignées lorsque le processeur va les chercher en mémoire par blocs, il n'a pas besoin de faire des aller-retours en mémoire à chaque répétition. Le temps de récupération des données dans la mémoire est supérieur à celui dans les mémoires internes au processeur. Le gain de temps sur l'exécution générale du programme s'explique alors par l'alignement des données dans la mémoire qui permet un gain de temps sur cette phase.

2.4 Modifications des calculs flottants

Afin de diminuer la complexité des calculs flottants sans perdre de performance, plusieurs équations ont été transformées ou développées, telles que :

— $x^{\frac{3}{2}} = \left(x^{\frac{1}{2}}\right)^3 = (\sqrt{x})^3 = \sqrt{x} * \sqrt{x} * \sqrt{x} = \sqrt{x} * (\sqrt{x})^2 = \sqrt{x} * x$
La racine carrée est une opération moins complexe que l'opération puissance. Elle requiert donc moins de cycles CPU pour un même calcul. De plus, elle possède sa propre micro-architecture [5] ce qui permet de réduire le temps de calcul.

— $(a-b)^2 = a^2 - 2ab + b^2$
Le développement de certains termes peut conduire à l'apparition de "FMA" (Fused Multyply-Add) [6] qui est une équation très courante et qui possède également sa propre micro-architecture. Cette forme permet donc de réduire le nombre de cycles CPU et le temps de calcul.

— $\frac{a}{b} = a * \frac{1}{b}$
La division est une opération assez gourmande en cycles CPU et en registres. Comme la fonction *move_particles*() comporte une suite d'opération divisant toujours par le même b, décomposer ces fractions et stocker la valeur de $\frac{1}{b}$ permet de réduire ce nombre de cycles.

3 Mesures de performances

Afin d'étudier les optimisations réalisées, plusieurs mesures de performances ont été effectuées. Chacune de ces mesures compare les différentes versions du programme, implémenté dans le fichier nbody0.c (version non optimisée, non modifiée) jusqu'au fichier nbody8.c (version la plus optimisée).

3.1 Étude du taux de Gflop/s

Dans un premier temps, les taux de Gflop/s [7] [8] de la fonction *move_particles*() ont été étudiés. Ces chiffres ont pu être calculés grâce à la librairie OpenMP qui contient entre autres un compteur de cycles, ce qui nous permet de déterminer le nombre de cycles CPU de la fonction. La figure 1 illustre ces taux pour chacune des versions du programme.

5. Ensemble de petits circuits internes au processeur, optimisées pour une équation donnée.
6. Equation de la forme : $a * b + c$.
7. Le FLOPS (floating-point operations per second) est le nombre d'opérations en virgule flottante par seconde.
8. Plus un taux de Gflop/s est élevé plus cela traduit une augmentation du nombre d'instructions traitées en 1 seconde, donc plus moins les opérations ralentissent l'exécution.

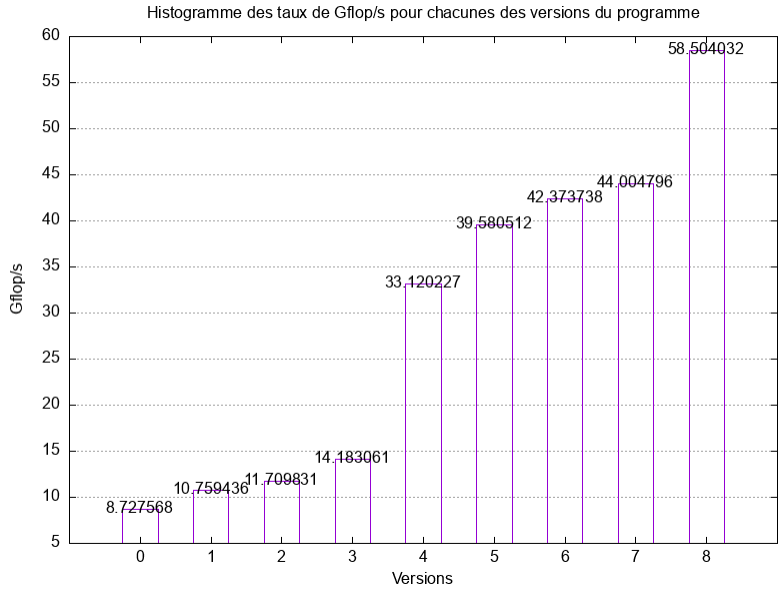

Figure 1

On remarque que l'on a 9 versions avec des taux croissants, et que le meilleur taux est de 58.504032 Gflop/s (version 8). Cette dernière version permet de multiplier la performance par 7 par rapport à la version 0 (non optimisée). Cette figure montre donc que les modifications expliquées précédemment ont bien permis d'accélérer le programme, en réduisant les opérations complexes et en travaillant avec des données mieux organisées.

3.2 Étude du code assembleur

Une autre analyse du programme a été effectuée sur le code assembleur généré lors de la compilation de la fonction *move_particles*(). La figure 2 présente donc le taux d'occurence de certaines instructions [9] en code assembleur. Les instructions choisies [10] (*mul*, *add*, *fma*, *broadcast*, *mov*, *sqrt*) se regroupent en trois catégories :

- des instructions "scalaires" : traite les données une par une ;
- des instructions "quad-words" : traite les données 2 par 2 (ou 4 par 4, selon la taille des données) ;
- des instructions "packed" : traite les données par blocs ;

Les deux dernières catégories traduisent donc une vectorisation des instructions et des données.

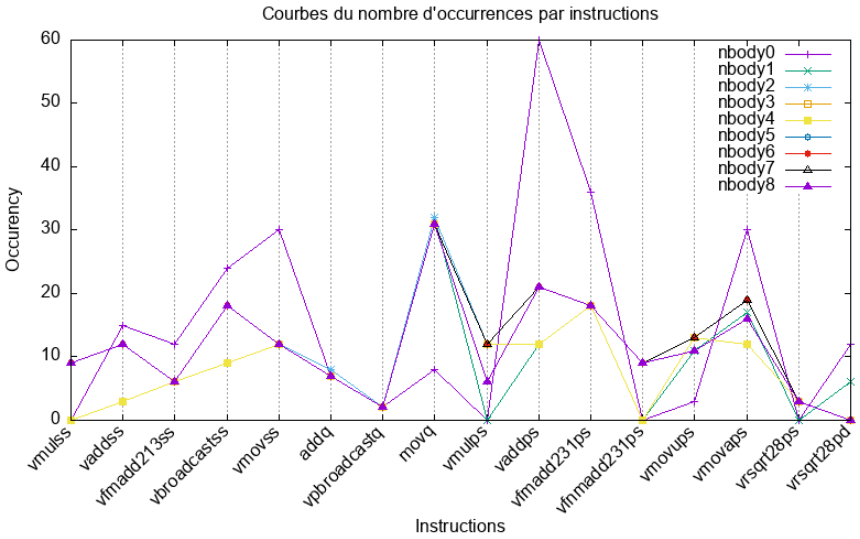

Figure 2

On remarque une trame générale pour les courbes pour l'ensemble des versions du programme, ce qui peut être expliqué par le fait que les instructions du programme en elles-mêmes n'ont pas vraiment été modifiées.

9. Une instruction est une opération la plus simple possible (ex : addition, multiplication, récupération de données,...) pouvant être traduite directement en instruction machine.

10. Les instructions présentées ont été choisies arbitrairement après une rapide étude du code assembleur généré.

Pour plus de lisibilité, la figure 3 ne représente ces données que pour la version 0 (initiale) et pour la version 8 (finale).

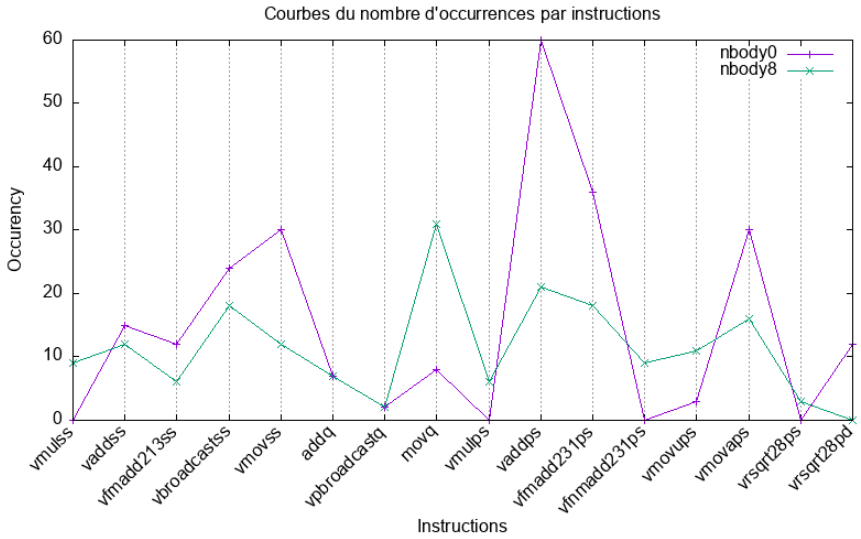

Figure 3

On remarque que pour 8 instructions sur les 16 présentées, elles sont plus présentes dans la version 0 que dans la version 8. On peut alors déduire que l'ensemble des transformations du code présentées précédemment ont pu réduire le nombre d'instructions scalaires pour la version 8, ce qui a contribué à l'augmentation de la performance.
Par ailleurs, ces différences peuvent également être interprétées par le fait que les transformations ont conduit à changer les appels instructions assembleur avec par exemple : remplacer des divisions par des multiplications, ou faire apparaître des FMA.

3.3 Compléments : MAQAO et Perf

Afin d'étudier plus précisément le programme et d'étudier les optimisations faites, des rapports d'analyses [11] par les outils MAQAO [12] et Perf (Linux) ont été générés. Ces analyses ont été faites uniquement sur la version 0 (initiale) et la version 8 (finale).

Global Metrics

Total Time (s)		586.56
Profiled Time (s)		586.56
Time in analyzed loops (%)		0.05
Time in analyzed innermost loops (%)		0.05
Time in user code (%)		99.9
Compilation Options		OK
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		100
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	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.00
	Nb Loops to get 80%	1

Figure 9 - Analyse générales de la version 0

Global Metrics

Total Time (s)		105.05
Profiled Time (s)		105.05
Time in analyzed loops (%)		99.9
Time in analyzed innermost loops (%)		99.2
Time in user code (%)		99.9
Compilation Options		OK
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		100
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	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.34
	Nb Loops to get 80%	1

Figure 10 - Analyse générale de la version 8

(toutes deux extraites du rapport d'analyse MAQAO)

D'après les analyses générales de MAQAO (figures 9 et 10), on remarque que les deux versions ont pu être correctement analysées et sont globalement bien implémentées. Il n'y a pas de point majeur diminuant les performances des versions.

11. Les rapports MAQAO et Perf complets se trouvent dans le sous-dossier "annexes" de l'archive du TP.
12. Outil d'analyse et d'optimisation de qualité d'assemblage modulaire développé par l'Université de Versailles St. Quentin et le laboratoire Exascale Computing Research (ECR).

Loop id	Source Location	Source Function	Coverage run_0 (%)	Nb Threads run_0	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized
3	nbody0.i - nbody0.c: 49-75 [...]	main	0.04	1	23.08	11.06	1.2	1.64	15.14
6	nbody0.i - nbody0.c: 23-38	init	0.01	1	20	10	1.65	1.82	16

Figure 11 - Analyses des boucles de la version 0 (extrait du rapport d'analyse MAQAO)

Loop id	Source Location	Source Function	Coverage run_0 (%)	Nb Threads run_0	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized
3	nbody8.i - nbody8.c:61-81	main	99.19	1	100	100	1	1	1

Figure 12 - Analyses des boucles de la version 8 (extrait du rapport d'analyse MAQAO)

Grâce aux figures 11 et 12 qui analysent plus précisément les boucles et parties du codes qui prennent le plus de temps, on remarque que la version 0 est principalement ralentie par un manque de vectorisation dans deux fonctions. Comme la version 8 a 100% de vectorisation dans son unique boucle la plus lente, cela signifie que cette version 8 est effectivement plus optimisée et mieux vectorisée que la version initiale.

Les rapports fournis par l'outil Perf (figures 13 et 14) permettent d'appuyer ces conclusions : la version 8 a notamment un temps d'exécution plus court que la version 0 et elle possède environ 3 fois moins de cycles CPU.

```
Performance counter stats for 'numactl --cpunodebind=1 --membind=1 ./nbody0.i':

        7,263.85 msec task-clock                #    0.986 CPUs utilized
              22      context-switches          #    0.003 K/sec
               1      cpu-migrations            #    0.000 K/sec
           2,411      page-faults               #    0.332 K/sec
  10,842,089,950      cycles                    #    1.493 GHz                      (49.99%)
   6,767,486,483      instructions              #    0.62  insn per cycle           (75.02%)
      93,452,153      branches                  #   12.865 M/sec                    (75.01%)
         718,075      branch-misses             #    0.77% of all branches          (75.00%)

     7.363959394 seconds time elapsed

     7.229776000 seconds user
     0.035988000 seconds sys
```

Figure 13 - Rapport d'analyse de Perf pour nbody0

```
Performance counter stats for 'numactl --cpunodebind=1 --membind=1 ./nbody8.i':

        1,985.75 msec task-clock                #    0.958 CPUs utilized
              18      context-switches          #    0.009 K/sec
               1      cpu-migrations            #    0.001 K/sec
           2,402      page-faults               #    0.001 M/sec
   2,963,545,101      cycles                    #    1.492 GHz                      (50.07%)
   4,079,403,192      instructions              #    1.38  insn per cycle           (75.00%)
     176,290,254      branches                  #   88.778 M/sec                    (74.90%)
         768,884      branch-misses             #    0.44% of all branches          (75.02%)

     2.073882620 seconds time elapsed

     1.963647000 seconds user
     0.024093000 seconds sys
```

Figure 14 - Rapport d'analyse de Perf pour nbody8

4 Conclusion

Pour conclure, afin d'optimiser le programme nbody3D, huit autres versions de celui-ci ont été produites, de plus en plus performantes. Ces optimisations ont pu être possibles par le biais de modifications de la structure du code et de son contenu, ainsi que par la manière de le compiler. Cela a conduit à l'apparition de meilleurs taux de vectorisation et donc de meilleurs temps d'exécution.
Au final, la meilleure performance atteinte est de 58.504032 Gflop/s.

5 Références

— Liste de certains flags de compilation avec gcc (GNU) : https ://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html ;
— Liste de certains flags de compilation avec icc (Intel) : https ://www.spec.org/accel/flags/icc2015-openmp.html#Section_
— Liste des instructions assembleur : https ://www.felixcloutier.com/x86/index.html ;

6 Annexe

Comparaison des taux de Gflop/s entre les deux compilateurs gcc (GNU) et icc (Intel)

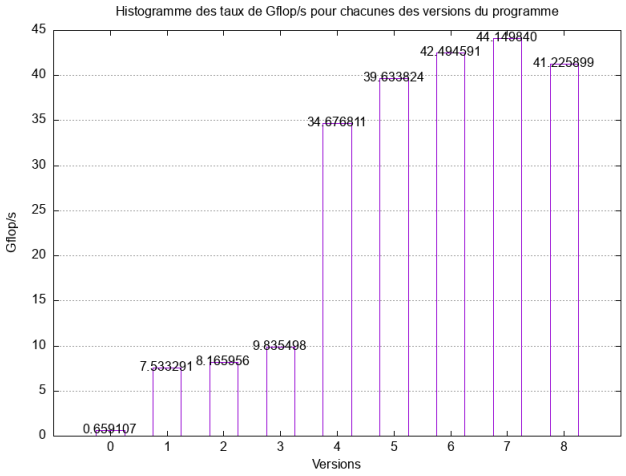

Figure 15 - Compilateur gcc

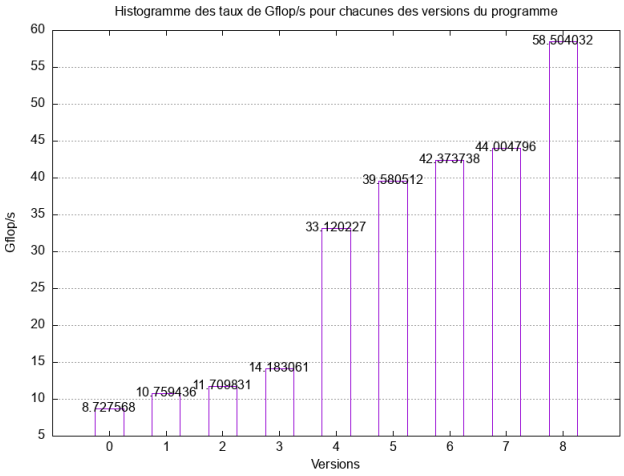

Figure 16 - Compilateur icc

En comparant les deux figures ci-dessus, on remarque que pour les quatre premières versions, le compilateur icc est plus performant que gcc. De la 4^e à la 7^e version, c'est gcc qui est plus adapté. Toutefois, on remarque une grande différence (d'environ 20 Gflop/s) pour la dernière version, avec icc comme compilateur plus performant.

Cependant, le compilateur icc permet d'avoir un facteur 7 entre les deux taux extrêmes (version 0 et version 8), tandis que gcc permet d'avoir un meilleur facteur avec *version* 7 $\approx$ 73 x *version* 0. On peut donc conclure que les deux compilateurs sont quasiment égaux en termes de performances, mais sont plus ou moins adaptés à différents types d'implémentations. Pour les analyses ci-dessus, le choix du compilateur a reposé sur les valeurs maximales de taux et non sur le plus grand ratio entre les versions extrêmes.