

Architecture Parallèle Projet N-body

YANG Yizhi

Janvier 2024

Table des matières

1	Introduction	2
2	Environnement	2
3	Expérience	3
4	Comparaison	6
5	Conclusion	8

1 Introduction

Dans le problème de N corps, on s'intéresse à une modélisation simple de prédire l'état final d'un système contenant N particules qui se réagissent entre elles par l'effet gravitationnel, dont la masse est supposée équivalente.

Notre objectif est de mesurer et optimiser la performance de la fonction suivante :

```
1 void move_particles(particle_t *p, const f32 dt, u64 n)
2 {
3     const f32 softening = 1e-20;
4     for (u64 i = 0; i < n; i++)
5     {
6         f32 fx = 0.0;
7         f32 fy = 0.0;
8         f32 fz = 0.0;
9         for (u64 j = 0; j < n; j++)
10        {
11            const f32 dx = p[j].x - p[i].x;
12            const f32 dy = p[j].y - p[i].y;
13            const f32 dz = p[j].z - p[i].z;
14            const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening;
15            const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0);
16            fx += dx / d_3_over_2;
17            fy += dy / d_3_over_2;
18            fz += dz / d_3_over_2;
19        }
20        p[i].vx += dt * fx;
21        p[i].vy += dt * fy;
22        p[i].vz += dt * fz;
23    }
24    for (u64 i = 0; i < n; i++)
25    {
26        p[i].x += dt * p[i].vx;
27        p[i].y += dt * p[i].vy;
28        p[i].z += dt * p[i].vz;
29    }
30 }
```

Listing 1 – Final Positions and Velocities: Original Version

On sauvegarde les six paramètres (la position x , y et z , la vitesse v_x , v_y et v_z) pour chaque particule, donc la complexité de mémoire de de l'ordre $O(N)$.

Chaque boucle contient N itérations, et elle s'effectue N fois (car il y a N particules), donc la complexité du temps est de l'ordre $O(N^2)$.

2 Environnement

On utilise le serveur *hsw03*, un système Archi Linux, ses informations sont extraites ci-dessous :

On-line CPU(s) list:	0-47
Vendor ID:	GenuineIntel
Model name:	Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz

CPU family:	6
Model:	63
Thread(s) per core:	2
Core(s) per socket:	12
Socket(s):	2
CPU max MHz:	3300.0000
CPU min MHz:	1200.0000
L1d cache:	768 KiB (24 instances)
L1i cache:	768 KiB (24 instances)
L2 cache:	6 MiB (24 instances)
L3 cache:	60 MiB (2 instances)
NUMA node(s):	2
NUMA node0 CPU(s):	0-11,24-35
NUMA node1 CPU(s):	12-23,36-47

Par défaut, on choisit $N = 16384 = 2^{14}$, cela occupe un espace de mémoire de taille 384 KiB, donc notre programme reste dans la cache L1.

3 Expérience

Premièrement, on compile le prgramme avec *GCC* option *-O0*, on sauvegarde l'état initial (la position et la vélocité des particules) dans un fichier et l'état final dans un autre en tant que référence pour la suite.

À chaque exécution, on utilise la même condition initiale et on compare les résultats obtenus afin d'analyser la stabilité numérique des calculs. La fonction Δ est définie comme une moyenne empirique de toutes les particules et de toutes les répétitions après l'échauffement (les trois premières exécutions) :

$$\Delta x = \frac{1}{N} \sum_{i=1}^N (x_i^{\text{ref}} - x_i), \quad \Delta y = \frac{1}{N} \sum_{i=1}^N (y_i^{\text{ref}} - y_i), \quad \Delta z = \frac{1}{N} \sum_{i=1}^N (z_i^{\text{ref}} - z_i)$$

$$\Delta = \frac{1}{3} (\Delta x + \Delta y + \Delta z)$$

On choisit les différentes compilations avec *GCC* et *CLANG*, options *-O0*, *-O2*, *-O3* et *-Ofast*.

Le flag *-S* nous permet de générer les codes d'assembleur, qui nous aident à effectuer les optimisations du code source.

```
gcc -S -o nbody.s nbody.c
```

Puis on optimise les codes comme dessous :

```
1 //enrolling by 4 and parallelizing the outer loop
2 void move_particles(particle_t *p, const f32 dt, u64 n)
3 {
4     const f32 softening = 1e-20;
```

```

5  #define UNROLL4 4
6  #pragma omp parallel for
7  for (u64 i = 0; i < n; i++)
8  {
9      f32 fx[UNROLL4] = {0.0};
10     f32 fy[UNROLL4] = {0.0};
11     f32 fz[UNROLL4] = {0.0};
12     #pragma omp parallel for
13     for (u64 j = 0; j < (n - (n % UNROLL4)); j += UNROLL4)
14     {
15         #pragma unroll
16         for (int k = 0; k < UNROLL4; k++)
17         {
18             const f32 dx = p[j + k].x - p[i].x;
19             const f32 dy = p[j + k].y - p[i].y;
20             const f32 dz = p[j + k].z - p[i].z;
21             const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening;
22             const f32 d_3_over_2 = d_2 * sqrt(d_2);
23             fx[k] += dx / d_3_over_2;
24             fy[k] += dy / d_3_over_2;
25             fz[k] += dz / d_3_over_2;
26         }
27     }
28     #pragma omp simd reduction(+:fx,fy,fz)
29     for (u64 j = (n - (n % UNROLL4)); j < n; j++)
30     {
31         const f32 dx = p[j].x - p[i].x;
32         const f32 dy = p[j].y - p[i].y;
33         const f32 dz = p[j].z - p[i].z;
34         const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening;
35         const f32 d_3_over_2 = d_2 * sqrt(d_2);
36         fx[0] += dx / d_3_over_2;
37         fy[0] += dy / d_3_over_2;
38         fz[0] += dz / d_3_over_2;
39     }
40     #pragma omp critical
41     {
42         p[i].vx += dt * (fx[0] + fx[1] + fx[2] + fx[3]);
43         p[i].vy += dt * (fy[0] + fy[1] + fy[2] + fy[3]);
44         p[i].vz += dt * (fz[0] + fz[1] + fz[2] + fz[3]);
45     }
46 }
47 #pragma omp parallel for
48 for (u64 i = 0; i < n; i++)
49 {
50     p[i].x += dt * p[i].vx;
51     p[i].y += dt * p[i].vy;
52     p[i].z += dt * p[i].vz;
53 }
54 }

```

Listing 2 – Final Positions and Velocities: Optimized Version

L'utilisation de `#pragma unroll` permet au compilateur de générer du code qui déroule la boucle interne. Cela signifie que plusieurs itérations de la boucle peuvent être exécutées en parallèle, ce qui peut améliorer les performances en exploitant le parallélisme d'instructions du processeur.

La directive `#pragma omp simd` indique au compilateur que la boucle peut être vectorisée. Cela

permet d'exploiter les instructions SIMD (Single Instruction, Multiple Data) du processeur pour effectuer des opérations sur plusieurs éléments de données en parallèle. La clause *reduction*(+ : fx, fy, fz) spécifie que les variables f_x , f_y , et f_z doivent être réduites en utilisant une opération d'addition.

Les directives *#pragma omp parallel for* indiquent que la boucle externe peut être exécutée en parallèle. Cela permet d'exploiter les ressources de plusieurs cœurs de processeur pour accélérer le traitement.

En utilisant des tableaux temporaires pour stocker les composantes des forces (fx, fy, fz) pour chaque particule, le code exploite la localité spatiale, ce qui peut améliorer les performances en réduisant les accès mémoire dispersés.

La directive *#pragma omp critical* est utilisée pour garantir qu'une seule thread à la fois exécute la section critique du code. Cela est nécessaire pour éviter les conditions de concurrence lors de la mise à jour des vitesses v_x , v_y , et v_z des particules.

La fonction $\text{pow}(d.2, 3.0/2.0)$ a été remplacée par $d.2 * \text{sqrt}(d.2)$, car la racine carrée est généralement plus rapide à calculer que la puissance pour des exposants constants et, la conversion de l'exposant en un nombre à virgule flottante (3.0/2.0) peut introduire une surcharge de calcul.

Pour la version de base, on applique la commande dessous pour définir l'affinité CPU : c'est sur le coeur numéro 3 que la tâche s'exécute.

```
taskset -c 3 ./nbody3D
```

Pour le code optimisé, il y a trois options d'exécution :

```
taskset -c 3 ./nbody_opt
OMP_PROC_BIND=close OMP_PLACES=cores numactl --membind=0 ./nbody_opt
./nbody_opt
```

OMP_PROC_BIND = close signifie que les threads OpenMP doivent être liés à un seul processeur, cela évite les migrations de threads entre différents processeurs, ce qui peut entraîner une surcharge due à la perte de données dans le cache du processeur. On peut améliorer la localité des données et réduire le temps d'accès à la mémoire. L'option *--membind = 0* spécifie que la mémoire doit être allouée sur le nœud NUMA 0.

Pour chaque version, on garde bien la précision numérique, soit un écart de position en moyenne de l'ordre 10^{-2} .

```
-----
Average delta:  1.490e-02
-----
```

4 Comparison

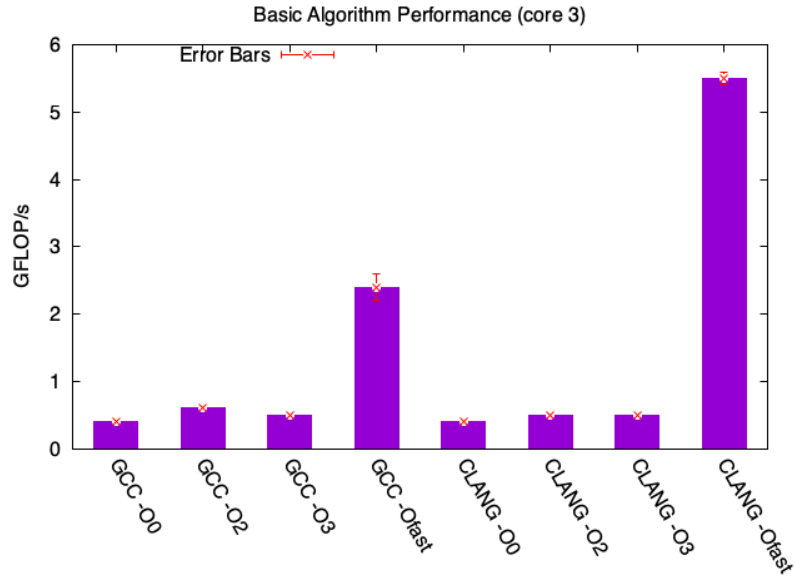


FIGURE 1 – Basic Source Code Version

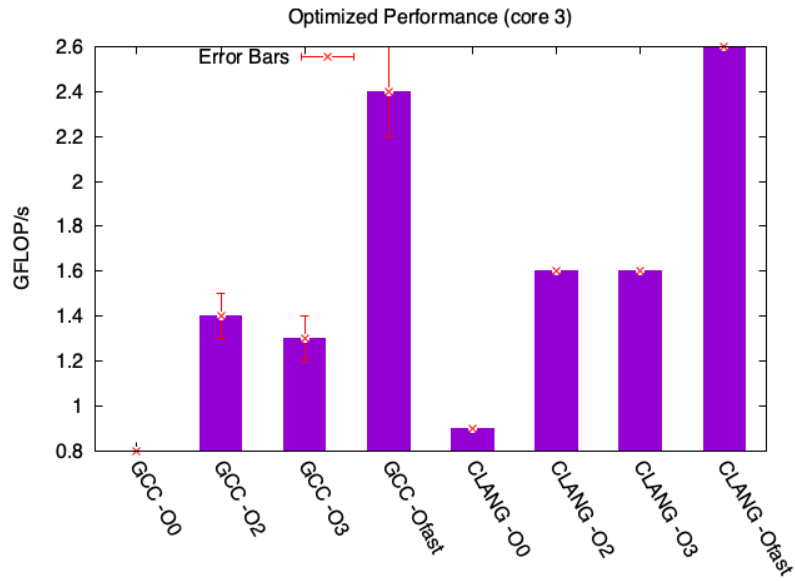


FIGURE 2 – Optimized Code pinning to Core 3

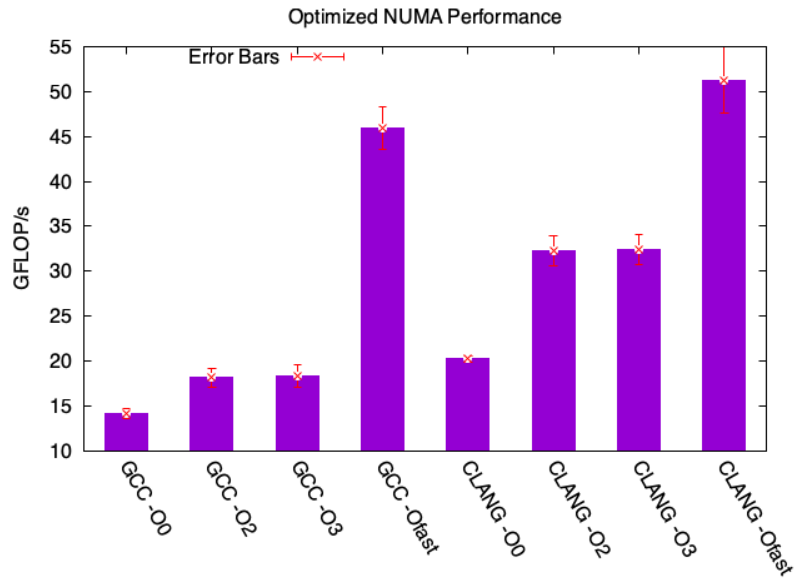


FIGURE 3 – Optimized Code with Memory Binding

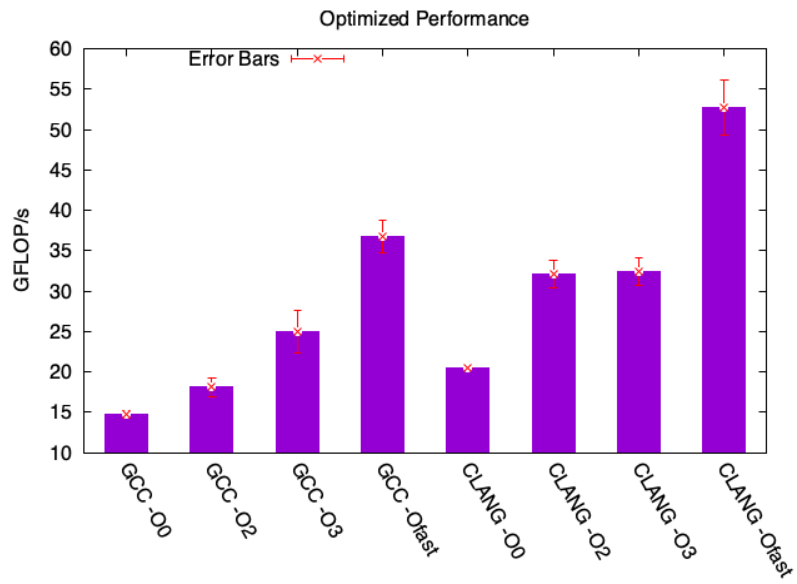


FIGURE 4 – Optimized Code without Core pinning

La performance du départ est :

Average performance:	0.4 +- 0.0 GFLOP/s
----------------------	--------------------

On constate que la performance du compilateur *CLANG* est toujours plus élevée que celle du *GCC* (sous les mêmes conditions).

Dans le cas du parallélisme, il est plus performant de ne pas pinner la tâche sur un coeur de CPU, car en cas de parallélisme, les charges de travail peuvent être réparties dynamiquement entre les threads en fonction de la disponibilité des ressources.

La meilleure performance est obtenue avec *CLANG – Ofast* sans pinné au coeur de CPU :

Average performance:	52.7 +- 3.4 GFLOP/s
----------------------	---------------------

Ce qui augmente de 130 fois.

5 Conclusion

Il existe d'autres algorithmes plus efficaces, comme l'algorithme de Barnes-Hut et la méthode multipolaire rapide. Ces deux méthodes concernent à regrouper les particules "lointaines" en un effet d'équivalence depuis leur barycentre vers la particule à laquelle on s'intéresse. Cela diminuera le nombre d'itérations donc la complexité totale.