Architecture Parallèle Projet N-body

YANG Yizhi

Janvier 2024

Table des matières

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 eux par effet gravitationnel, dont la masse est supposée équivalente.

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

```
void move_particles(particle_t *p, const f32 dt, u64 n)
{
    const f32 softening = 1e-20;
    for (u64 i = 0; i < n; i++)
    {
        f32 fx = 0.0;
        f32 fy = 0.0;
        f32 fz = 0.0;
        for (u64 j = 0; j < n; j++)
        {
         const f32 dx = p[j].x - p[i].x;
         const f32 dy = p[j].y - p[i].y;
         const f32 dz = p[j].z - p[i].z;
         const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening;
         const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0);
         fx += dx / d_3_over_2;
         fy += dy / d_3_over_2;
         fz += dz / d_3_over_2;
        }
        p[i].vx += dt * fx;
        p[i].vy += dt * fy;
        p[i].vz += dt * fz;
    }
    for (u64 i = 0; i < n; i++)
    {
        p[i].x += dt * p[i].vx;
        p[i].y += dt * p[i].vy;
        p[i].z += dt * p[i].vz;
    }
}
```

Listing 1 – Final Positions and Velocities: Original Version

On sauvegarde 6 paramètres (position x, y et z, 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 (il y a N particules), donc la complexité du temps est de l'ordre $O(N^2)$.

2 Environnement

3 Expérience

Premièrement, on compile le prgramme avec GCC option $-O0$, 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 tous les particules et de toutes les répétitions après l'échauffement :

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

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

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

Le flag $-S$ nous permet de générer les condes 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 :

```
//enrolling by 4 and parallelizing the outer loop
void move_particles(particle_t *p, const f32 dt, u64 n)
{
    const f32 softening = 1e-20;
    #define UNROLL4 4
    #pragma omp parallel for
    for (u64 i = 0; i < n; i++)
    {
        f32 fx[UNROLL4] = {0.0};
        f32 fy[UNROLL4] = {0.0};
        f32 fz[UNROLL4] = {0.0};
        #pragma omp parallel for
        for (u64 j = 0; j < (n - (n % UNROLL4)); j += UNROLL4)
        {
            #pragma unroll
            for (int k = 0; k < UNROLL4; k++)
            {
                const f32 dx = p[j + k].x - p[i].x;
                const f32 dy = p[j + k].y - p[i].y;
                const f32 dz = p[j + k].z - p[i].z;
                const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening;
                const f32 d_3_over_2 = d_2 * sqrt(d_2);
                fx[k] += dx / d_3_over_2;
                fy[k] += dy / d_3_over_2;
                fz[k] += dz / d_3_over_2;
            }
```

```
        }
        #pragma omp simd reduction(+:fx,fy,fz)
         for (u64 j = (n - (n % UNROLL4)); j < n; j++)
        {
            const f32 dx = p[j].x - p[i].x;
            const f32 dy = p[j].y - p[i].y;
            const f32 dz = p[j].z - p[i].z;
            const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening;
            const f32 d_3_over_2 = d_2 * sqrt(d_2);
            fx[0] += dx / d_3_over_2;
            fy[0] += dy / d_3_over_2;
            fz[0] += dz / d_3_over_2;
        }
        #pragma omp critical
        {
            p[i].vx += dt * (fx[0] + fx[1] + fx[2] + fx[3]);
            p[i].vy += dt * (fy[0] + fy[1] + fy[2] + fy[3]);
            p[i].vz += dt * (fz[0] + fz[1] + fz[2] + fz[3]);
        }
    }
    #pragma omp parallel for
    for (u64 i = 0; i < n; i++)
    {
        p[i].x += dt * p[i].vx;
        p[i].y += dt * p[i].vy;
        p[i].z += dt * p[i].vz;
    }
}
```

Listing 2 – Final Positions and Velocities: Optimized Version

L'utilisation de *pragmaunroll* 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 *pragmaompsimd* 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 $clausereduction(+ : fx, fy, fz)$ spécifie que les variables f_x, f_y, et f_z doivent être réduites en utilisant une opération d'addition.

Les directives $pragmaompparallelfor$ 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 *pragmaompcritical* 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 $pow(d\_2, 3.0/2.0)$ a été remplacée par $d\_2 * sqrt(d\_2)$, car a 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 $\frac{3.0}{2.0}$ peut introduire une surcharge de calcul.

4 Comparaison

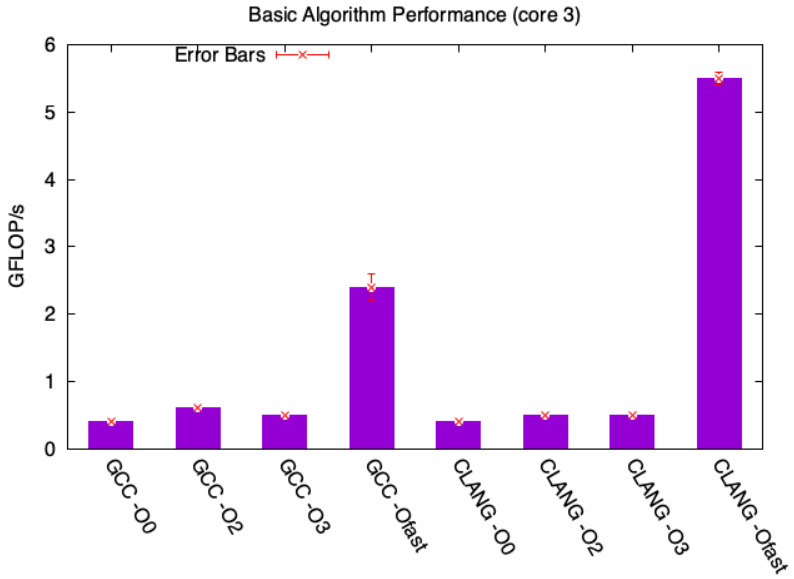

FIGURE 1 – Basic Source Code Version

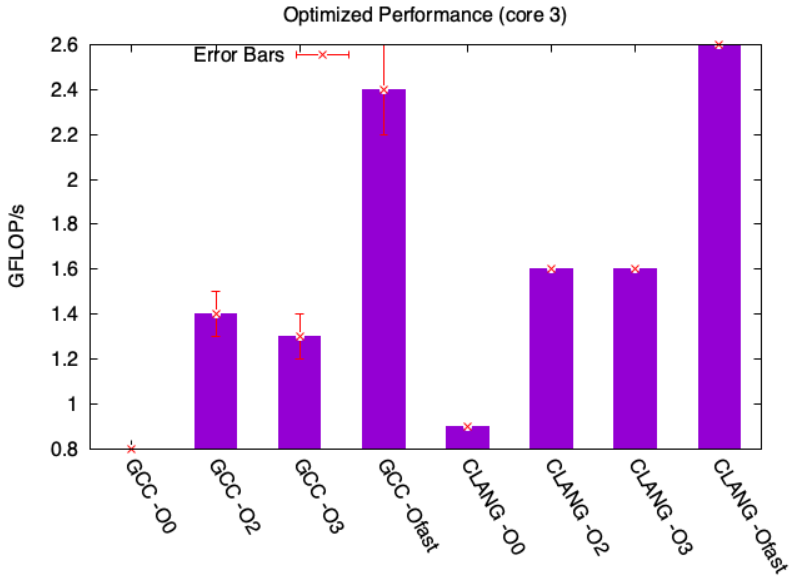

FIGURE 2 – Optimized Code pinning to Core 3

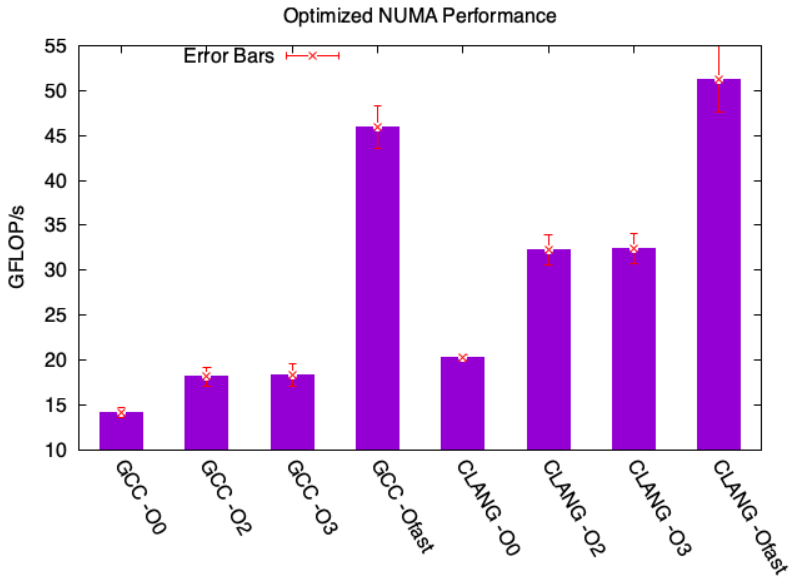

FIGURE 3 – Optimized Code with Memory Binding

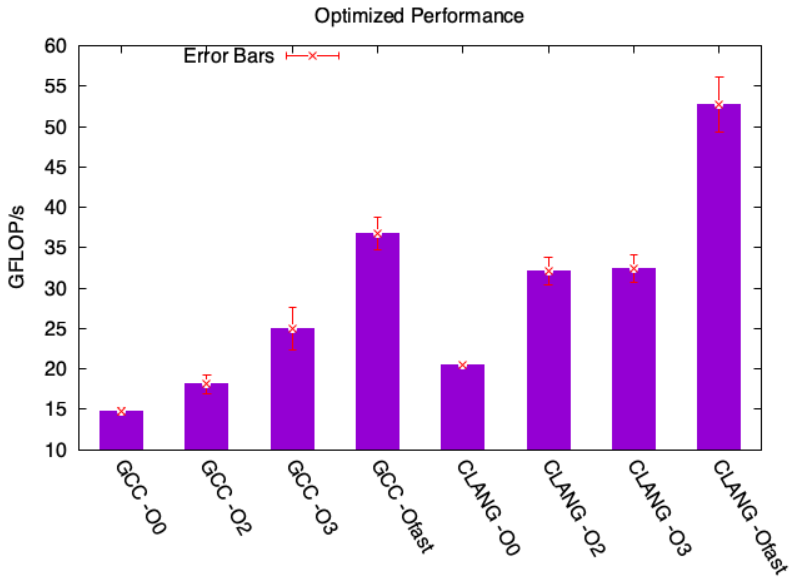

FIGURE 4 – Optimized Code without Core pinning

5 Conclusion