Rapport de projet
Simulation de particules sur architectures parallèles

Yacine El Jarrari, Nicolas Heng

Juin 2014

1 Introduction

L'objectif de ce projet est de concevoir une application simulant l'interaction entre particules dans un environnement en trois dimensions. Ce projet met l'accent sur l'utilisation de programmes parallèles. Nous sommes partis d'un programme de base séquentiel, que nous avons remanié au fur et à mesure, afin de tirer un maximum profit des performances offertes par les processeurs multicoeurs grâce à OpenMP, ainsi que par les accélérateurs via OpenCL. Nous verrons ainsi à quel point la parallélisation d'un programme, en utilisant au mieux l'architecture des machines, peut améliorer ses performances par un facteur non négligeable.

2 Fonctionnalités - Calcul séquentiel

Dans cete partie nous verrons la structure du programme de base ainsi que ses performances en mode séquentiel

2.1 Fonctionnalités du programme

Tout d'abord, quelques rappels sur la structure du programme et des algorithmes de calcul sur les particules.

Mouvement des particules La position et la vitesse de chaque atome sont stockées dans deux tableaux de float distincts. Dans ces tableaux, sont d'abord stockées à la suite les composantes suivant x de tous les atomes, ensuite les composantes suivant y, puis selon z en fin de tableau.

A chaque itération de calcul, la fonction *seq_move*() met à jour le tableau des positions en ajoutant simplement à la position de chaque atome, sa vitesse. Le calcul s'effectue ainsi en temps linéaire.

Rebond sur les murs Deux valeurs *min_ext* et *max_ext* délimitent un espace contenant les atomes. Pour garantir le fait que les atomes restent dans cet espace, la fonction *seq_bounce*() fait rebondir les atomes lorsqu'ils atteignent les limites de l'espace. Pour réaliser le rebond, en testant pour chaque atome si l'un des bords de l'espace a été dépassé, et dans ce cas, on se contente de remplacer la composante de la vitesse selon le côté dépassé, par son opposé. Ce qui s'effectue également en temps linéaire.

Potentiel de Lennard-Jones L'interaction entre les atomes est calculée grâce au potentiel de Lennard-Jones. Celui-ci dépend de la distance qui sépare un couple d'atomes considéré. Pour éviter des calculs insignifiants, on se donne une distance maximale (ou rayon de coupure). Si la distance entre deux atomes dépasse ce rayon, le calcul de la force est négligée.

L'algorithme de base est de complexité en temps quadratique.

2.2 Performances

Nous allons vous présenter dans cette partie les performances obtenues lors d'une exécution séquentielle du programme. La configuration matérielle utilisée lors des tests est la suivante :

— CPU : Intel Core i5 2500K - 4 cœurs

— GPU : Nvidia GeForce 560Ti - 384 cœurs

Le graphique qui suivant montre le temps d'exécution d'une itération en fonction du nombre d'atomes simulés (Moyenne sur 100 itérations)

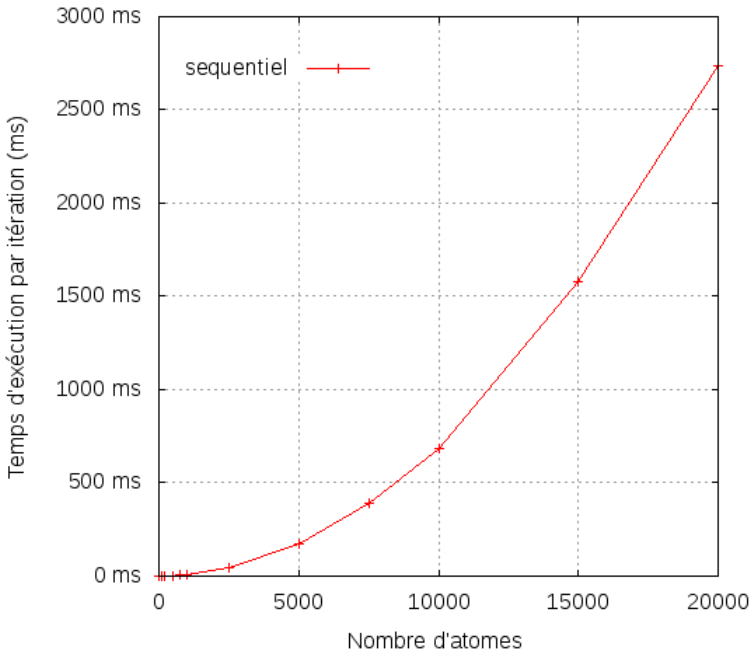

FIGURE 1 – Temps d'exécution d'une itération en fonction du nombre d'atomes

On peut remarquer que le programme actuel est très rapidement à la peine lorsque le nombre d'atomes augmente. A tel point qu'il n'est pas possible par exemple d'effectuer de simulation sur le fichier de configuration *biochoc*1 (contenant plus de 40000 atomes) en un temps raisonnable. La forme de la courbe laisse fortement entrevoir une complexité quadratique (liée au calcul de la force).

Dans le même temps, nous ne comptons sur les quelques optimisations que le compilateur ou que le matériel peuvent trouver. A ceux-ci près, le programme est entièrement séquentiel, et nous sommes très en deçà des capacités théoriques du matériel. De plus, la quasi totalité du temps d'exécution est dû au calcul de la force, ce qui n'est pas surprenant puisqu'il s'effectue en $O(n^2)$ alors que les autres processus ne sont qu'en $O(n)$. Nous voyons ainsi que pour obtenir de meilleurs résultats, il est nécéssaire de revoir l'algorithme de calcul de la force. C'est l'objet des parties qui vont suivre.

```
[PERF]    #atoms        time/i (µs)     Matoms/i/s
[PERF]      1984             27102            0.1
[PERF] Detailed performance report for device [Fake CPU Device]
[PERF]   All kernels performed in 0.000000 µs with inf Matoms/i/s
[INFO] Finalizing...
Calculation time after 100 iterations : 2710.2ms
Details per iteration : time_move 2.7us - time_force 27099.1us - time_bounce 0.0us
```

FIGURE 2 – Détails du temps de calcul sur 100 itérations de choc1.conf

3 Parallélisation avec OpenMP

Dans cette partie nous verrons comment l'utilisation d'OpenMP pour tirer profit de l'architecture multicœur des processeurs permet d'améliorer sensiblement les performances du programme

3.1 Approche naïve

Une première étape dans la parallélisation du programme consiste à y insérer des *#pragma omp parallel for* afin de paralléliser le parcours des atomes. On obtient ainsi une accélération qui tend vers des valeurs situées entre 3 et 4, ce qui est conforme à ce qu'on s'attend à voir, avec une machine tournant sur un processeur à 4 cœurs, en prenant en considération le fait que seul les processus de calculs sont parallélisées et non pas l'intégralité du programme (d'où des valeurs inférieures à 4 seloin la loi d'Amdahl).

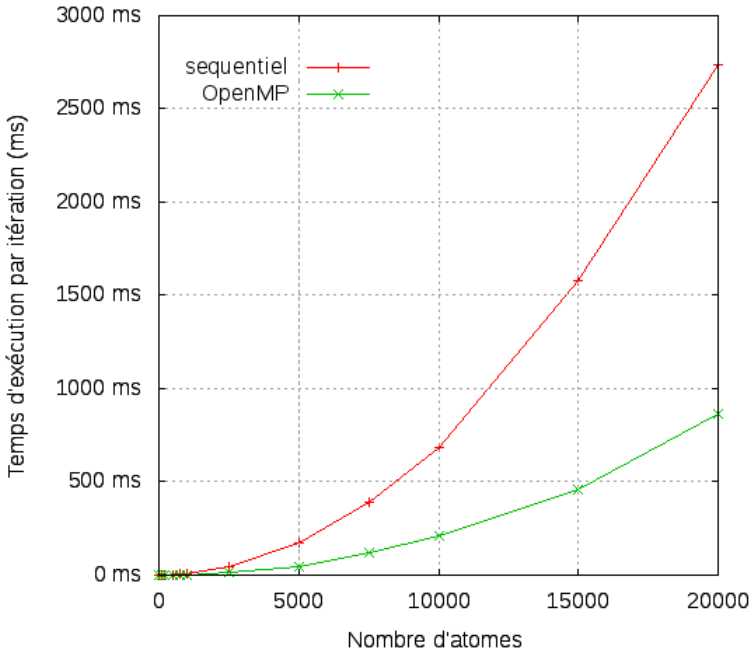

FIGURE 3 – Comparatif séquentiel/parallèle

Toutefois, l'accélération n'est toujours pas suffisante pour pouvoir exécuter une simulation comportant un très grand nombre d'atomes comme *biochoc*1. Ceci étant dit, nous pouvons encore améliorer l'algorithme en triant les atomes en fonction de leurs coordonnées, de façon à ne pas parcourir à chaque fois l'intégralité des atomes, ce qui permet de limiter le nombre de calculs (le calcul de la distance étant particulièrement coûteux).

3.2 Tri des atomes

Nous avons vu que le calcul de la force est la partie qui demande le plus de ressources. Ainsi, si nous pouvons éviter des calculs inutiles dans cette partie, les performances du programme n'en seront que meilleures. Etant donné que nous faisons le choix de négliger la force entre deux atomes lorsque ceux-ci sont éloignés l'un de l'autre d'une distance au moins égale au rayon de coupure. Ainsi, en triant les atomes, nous pouvons faire en sorte de ne parcourir qu'un sous-ensemble restreint d'atomes situés à proximité de l'atome considéré.

Tri par Z Dans un premier temps, les atomes sont triés par leurs coordonnées selon Z. Ainsi, lorsque l'on effectue le parcours du tableau d'atomes, nous pouvons nous arrêter dès lors que l'on voit que l'écart sur l'axe Z dépasse le rayon de coupure. Le tri est effectué selon l'algorithme du tri par tas, dont la complexité est en $O(nlog(n))$

Il y a cependant un défaut à ce tri. En effet, il n'est d'aucune aide si tous les atomes sont situés sur un même plan orthogonal à l'axe Z.

Tri par boîtes Désormais, l'espace est découpé en petits cubes de taille égale au rayon de coupure. Chaque atome se retrouve alors dans un des cubes de l'espace. Pour chacun d'entre eux, nous pouvons alors ne considérer que les atomes situés dans les 26 cubes situés autour de celui auquel il appartient. Le processus de tri est toutefois un peu plus élaboré que pour le tri par Z. Pour ce faire, nous avons défini en tant que variables globales, deux tableaux intermédiaires dans lesquels vont être stockés les positions et les vitesses des atomes suivant leur nouvel indice. Nous avons également défini un tableau qui associe à chaque numéro de boîte le nombre d'atomes présents à l'intérieur ainsi qu'un tableau donnant le nouvel indice d'un atome en fonction de son numéro de boîte.

Dans un premier temps, le but est de calculer le nombre d'atomes présents dans chaque boîtes. Ceci se fait aisément en parcourant la table des atomes et en calculant pour chaque atome l'indice de boîte à laquelle ils appartiennent en fonction de leur position dans l'espace.

Numéro de boite	0	1	2	3	...	n
Nombre d'atomes	1	0	15	6	...	10

Numéro de boite	0	1	2	3	...	n
Cumulé	0	1	1	16	...	$nb\_atomes[n-1] + cumule[n-1]$

TABLE 1 – Tableau supérieur : Nombre d'atomes par boite
Tableau inférieur : Tableau des cumulés associé

Dans un second temps, une fois le nombre d'atomes par boîte calculé, on remplit un tableau des cumulés (voir la table ci-dessus). Cela nous permet de relier un numéro de boîte d'un atome à son nouvel indice. Nous pouvons désormais copier vers les tableaux intermédiaires, les positions et vitesses des atomes, aux nouveaux indices que nous donne le tableau des cumulés (Dans l'exemple ci-dessus, le premier atome de la boîte 3 rencontrée devra être placé à l'indice 16).

Lorsqu'un atome est placé à son nouvel indice, le tableau des cumulés est incrémenté à la case correspondant à son numéro de boîte. De cette façon lorsqu'on rencontre un autre atome de la même boîte, on pourra lire directement dans cette case l'indice auquel cet autre atome devra être placé.

Une fois tous les atomes déplacés vers leurs nouvelles positions, le contenu des tableaux intermédiaires est recopié vers le tableau global.

Finalement, le calcul des atomes proches se fait aisément, en calculant le numéro des boîtes voisines et en reliant ces numéros aux indices des atomes par le tableau des cumulés

Résultats On peut voir sur ces résultats que le tri par boîtes améliore grandement les performances dans le cas général, notamment lorsque les atomes sont répartis uniformément dans l'espace. Cependant, avec une configuration identique à $3\_atoms\_dim\_400$ où il n'y a que peu d'atomes mais un grand nombre de boîtes, le tri par boîtes se retrouve finalement en-deçà de celui du tri par Z (et même de la version non triée).

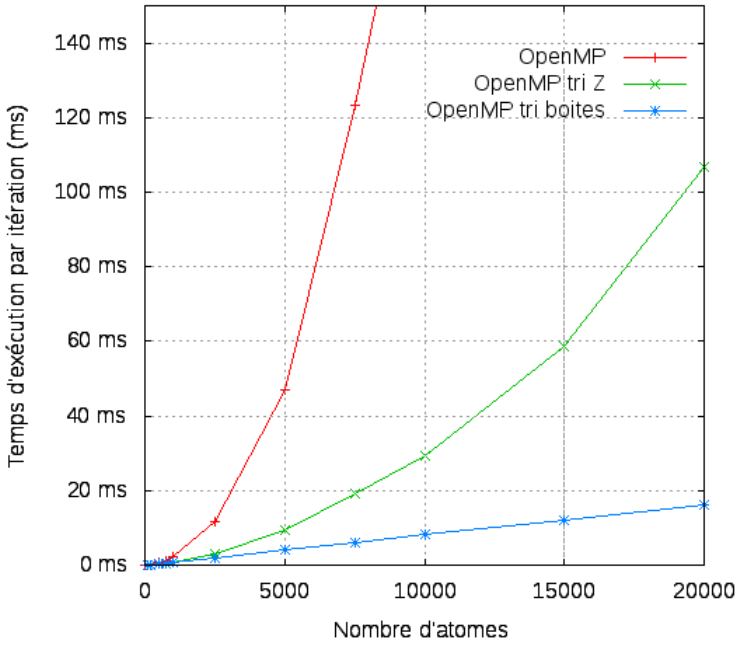

FIGURE 4 – Comparatif OpenMP

A l'inverse, l'algorithme est bien meilleur lorsque les atomes sont tous groupés comme c'est le cas dans la configuration *dense* (Le calcul est effectué 5 fois plus vite par rapport à la simulation avec l'option $-n$ 1000, où il y a le même nombre d'atomes mais répartis uniformément dans l'espace). On remarquera également les faibles résultats obtenus sur *choc*2 avec la version tri par Z qui confirment effectivement l'inefficacité de ce tri lorsque beaucoup d'atomes forment un plan orthogonal à l'axe Z

Fichier	#atomes	#boites	Temps/i (µs) tri par Z	Temps/i (µs) tri par boîtes
3_atoms_dim_400	3	330k	26	3023
dense	1000	1.3k	836	159
-n 1000	1000	1.2k	783	743
choc2	5200	176k	82389	6443
biochoc1	40344	1.3M	130399	33235

TABLE 2 – Comparaison des tris

4 Parallélisation avec OpenCL

Dans cette partie nous verrons comment l'utilisation d'OpenCL permet d'améliorer les performances du programme par rapport au code séquentiel.

Résultats Il est à noter que les performances obtenues avec OpenCL sont meilleures qu'avec OpenMP en considérant un nombre peu élevé d'atomes. Mais cela est surtout dû à la puissance de calcul offerte par la carte graphique. L'algorithme utilisé pour la version OpenCL reste en $O(n^2)$ et n'optimise pas les calculs en fonction de la position des atomes, d'où les faibles performances comparées à la version OpenMP trié par boîte passé la barre des 10000 atomes. Cela dit, les performances restent bien meilleurs qu'avec la version non triée d'OpenMP.

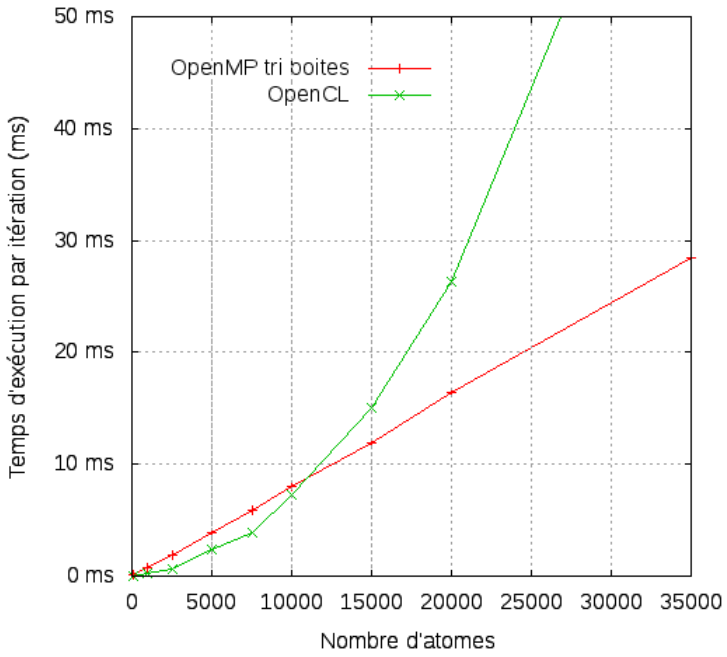

FIGURE 5 – Comparatif OpenMP boites/OpenCL

Nous avons également pensé à une amélioration possible du code en créant au sein de chaque workgroup, un tableau de taille $TILE\_SIZE$. Chaque thread d'un workgroup s'occupe alors de charger les coordonnées d'un atome pour le ranger dans ce tableau. Une fois ce tableau rempli, on continue le calcul de la force, puis on charge un nouveau bloc de coordonnées, etc... Jusqu'à ce que tous les atomes soient parcourus. On divise ainsi le nombre d'accès aux coordonnées par la taille d'un workgroup. Cependant, en pratique, cela n'augmente pas les performances.

5 Conclusion

Ce projet a permis de mettre en lumière que le simple fait d'utiliser du matériel plus puissant, notamment avec plus de cœurs, n'est pas une condition suffisante pour accélérer les calculs. Au contraire, c'est en adaptant le code qu'il est possible d'améliorer les temps de calcul. Par exemple avec l'utilisation d'OpenMP afin de tirer profit de chaque cœur du processeur.

Mais c'est surtout en adaptant le code aux problèmes qu'il doit résoudre qu'on atteint de meilleurs résultats, on le voit dans ce projet par l'influence des tris dans les courbes de performances : Un mauvais algorithme en $O(n^2)$, même s'il est effectué sur un GPU d'une centaine de de cœurs, face à un problème de grande échelle sera quand même bien moins performant qu'un algorithme de complexité réduite par une bonne analyse du problème, tournant sur un CPU.