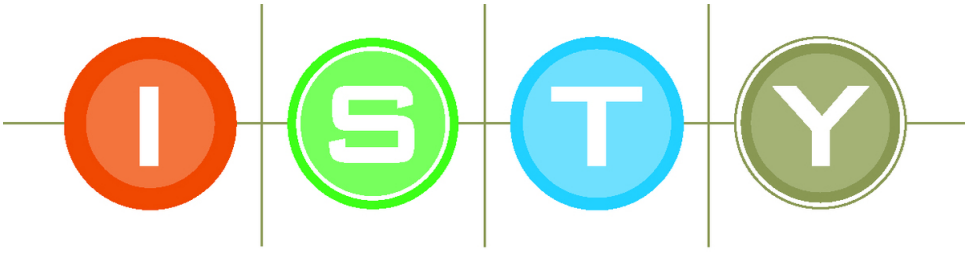
Analyse de performance et optimisation de code

Pierre AYOUB

5 février 2019



INSTITUT DES SCIENCES ET
TECHNIQUES DES YVELINES

Résumé

La simulation numérique est un procédé informatique visant à modéliser un phénomène par ordinateur, s'agissant le plus souvent d'un phénomène physique. Cette modélisation prend forme par des systèmes d'équations décrivant l'état du système physique représenté à chaque instant. De nombreux domaines scientifiques convergent vers la simulation informatique, tel que certaines branches de la physique, de l'analyse et de l'optimisation mathématique, ou encore le calcul haute performance en informatique. Enfin, la simulation trouve naturellement de nombreuses applications concernant des sujets variés, tel que la simulation du climat et des évènements météorologiques, la simulation d'essais nucléaires, de l'effet d'un médicament sur un corps, ou encore des astres et de l'univers. Ce rapport s'articulera donc autour de la simulation de fumée, phénomène impliquant les lois de la mécanique des fluides. Notre travail portera sur l'aspect du calcul haute performance de cette simulation.

Table des matières

1 Introduction

Le projet que nous vous présentons aujourd'hui consiste à analyser puis, grâce à nos mesures, optimiser un code de simulation numérique. Ce dernier nous offre une interface graphique permettant d'ajouter de la fumée dans un espace confiné et, ainsi, d'en observer le comportement. Nous pouvons influencer la quantité de fumée et sa vélocité dans l'espace. De plus, l'application nous donne le contrôle sur la résolution de la simulation, cela revient à dire sur sa précision, qui détermine principalement la performance du programme.

Le déroulement du projet s'est effectué en plusieurs étapes distinctes :

Analyse du code Cette phase consiste à analyser le programme d'un point de vue mathématique et informatique. De cette première approche, il s'agira de comprendre les opérations du programme sur les équations qui régissent le système physique. De l'autre approche, il convient d'étudier l'architecture logicielle de l'application, ainsi que les choix mis en œuvres afin d'implémenter le ou les algorithmes nécessaires.

Protocole expérimental Une fois l'analyse effectuée, nous pouvons en déduire le moyen le plus adapté afin de mesurer les performances de notre implémentation. Nous allons donc mettre en avant les critères théoriques à atteindre dans nos mesures, puis nous exposerons la manière dont nous avons mis ceci en pratique.

Optimisations et mesures Grâce au protocole mis en place, nous pouvons quantifier la performance du programme. De ce fait, nous serons en mesure d'expérimenter différentes techniques d'optimisation sur le programme et d'en calculer l'accélération.

2 Analyse du code

L'analyse du code est la première étape à effectuer afin d'optimiser un code. Il s'agit d'identifier les différentes sections et fonctions qui pourraient être critiques, ainsi que de comprendre mathématiquement quelles fonctions remplissent quels rôles.

L'interface du programme et l'interaction avec l'utilisateur est géré par les fichiers « demo.html », « FluidSolver.java » et « WebStart.java ». À priori, ces fichiers nous importe peu dans notre processus d'analyse et d'optimisation. Le cœur de la simulation se déroule dans le fichier « fluid.c », dans lequel se situe les fonctions de calcul des phénomènes physique.

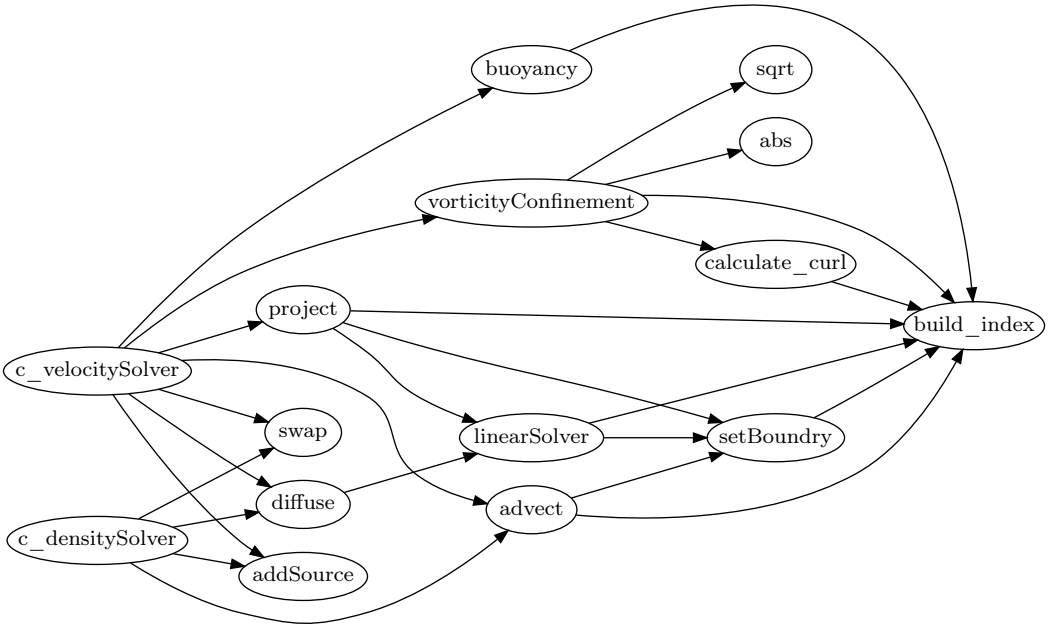


FIGURE 1 – Graphe d'appel du fichier *fluid.c*

Avant d'observer et d'analyser attentivement le code du fichier « fluid.c », nous avons utilisé **Cflow** afin d'obtenir un graphe d'appel pour avoir une vue d'ensemble et observer les liens entre les fonctions. Le résultat est présenté dans la figure 1. Nous décrivons ici les fonctions principales. On remarque très rapidement les deux points d'entrés du programme : « c_densitySolver() » et « c_velocitySolver() ». Ces deux fonctions sont utilisées conjointement pour assurer les deux fonctionnalités de notre simulation : la première permet de calculer la position d'une particule de fumée ajouté par la souris, la deuxième permet de calculer leur déplacement en fonction des différentes vélocités dans l'espace. Nous observons ensuite deux fonctions qui sont ici pour assurer des opérations physique : « project() » et « diffuse() ». Le nom laisse sous entendre que la première permet de faire une projection (des coordonnées d'un système à un autre ?) et la seconde permet de diffuser (une particule dans l'espace ?), mais nos connaissances en mécanique des fluides nous arrêtent ici pour les suppositions. Cependant, il est intéressant d'observer que ces deux fonctions font appel à une nouvelle fonction : « linearSolver() ». Cette fonction, comme le nom le laisse entendre, est le cœur de calcul de l'application : elle permet de résoudre

un système d'équation linéaire. Il ne fait aucune doute que c'est dans cette fonction que de nombreuses optimisations seront possibles ! Enfin, on observe quelques autres fonctions de physique (« vorticityConfinement() », « advect() », « buoyancy() ») ou des fonctions de mathématiques (« sqrt() », « abs() ») qui gravitent autour des fonctions précédemment décrites. Enfin, une fonction semble être appelée par quasiment toutes les autres de manière récursive : la fonction « build_index() » . Cette fonction, très courte, permet de passer d'un couple de coordonnées (i, j) identifiant une donnée à un offset correspondant à son emplacement mémoire par rapport à la base de la matrice. Cette fonction, présente à de nombreux endroits dans le code, devrait aussi avoir droit à une attention particulière.

3 Protocole expérimental

La mise en place d'un protocole expérimental de mesure est une étape nécessaire et cruciale dans tout optimisation de code. D'une part, le but de ce protocole est de mettre en lumière les points chauds du programme, c'est-à-dire les parties du code qui ralentissent considérablement l'exécution de la simulation. Ces points chauds seront les cibles de nos optimisations. D'autre part, après chaque tentative d'optimisation, le protocole doit nous permettre de mesurer l'impact de cette dernière, qu'il soit positif ou négatif, et enfin de le quantifier.

3.1 Théorie

Le protocole se doit d'être le plus représentatif possible d'une utilisation typique du programme. Pour cela, on ne doit pas uniquement mesurer la simulation exécutant une fonctionnalité particulière, mais l'ensemble des fonctionnalités. Il est bon de noter que ceci est vrai dans le cas d'un unique protocole, comme ici : en effet, on pourrait imaginer plusieurs protocoles différents pour optimiser chaque fonctionnalité indépendamment des autres. Ici, le programme nous offre deux fonctionnalités : ajouter de la fumée dans un espace confiné et ajouter une vélocité (c'est-à-dire un vecteur vitesse) modifiant ainsi la trajectoire de la fumée. Notre protocole doit donc faire intervenir ces deux possibilités dans notre expérience.

Lors de nos expériences, il ne faut pas oublier que le hasard ou l'aléa des mesures peuvent biaiser un résultat. Afin d'éviter cela, il faut donc utiliser une valeur moyenne ou une valeur médiane.

3.2 Pratique

Pour effectuer nos mesures, il nous faut utiliser un outil de d'analyse de performance. Plusieurs possibilités s'offrent à nous, parmi lesquels nous pouvons notamment citer : Performance Counters for Linux (perf), OProfile, Gprof, MAQAO, Callgrind/Kcachegrind, et bien d'autre. Parmi ces choix-là, nous avons choisis d'utiliser « perf ». Nous pouvons lui citer plusieurs avantages qui nous ont intéressés : intégration en ligne de commande, outil bas-niveau, développement stable, accès aux compteurs hardware. Nous pouvons donc en déduire que c'est un outil tout à fait adapter à l'analyse de code bas-niveau (C, assembleur).

Le programme de simulation utilise une interface graphique pour rentrer en interaction avec l'utilisateur. De ce fait, nous ne pouvons pas lui donner un jeu de données défini à l'avance qui serais constant entre les différentes mesures. Nous devons alors définir des actions à réaliser de la manière la plus similaire possible entre les exécutions. De manière « arbitraire », nous définissions un run de la manière suivante : une fois l'applet lancé, nous traçons une ligne de fumée en diagonale, du point en bas à droite au point en haut à gauche [clic gauche enfoncé], puis nous refaisons la même opération en appliquant une vélocité sur ce trajet [clic droit enfoncé], dans le même sens.

Cependant, même en appliquant les mêmes actions entre chaque expérience, les temps d'exécution seront irrémédiablement différents. Dès lors, pour mesurer la performance d'une fonction, nous n'allons pas regarder le nombre de cycles passés dans cette fonction mais son pourcentage de cycle par rapport à l'ensemble du programme. En effet, étant donné que le pourcentage est un rapport sur le temps de l'ensemble du programme, ce dernier peux varier tout en gardant un pourcentage similaire pour une même fonction, car le nombre de cycles passé dans cette fonction augmentera proportionnellement.

4 Optimisations et mesures

4.1 Mise en place de la compilation

4.2 Déroulage de boucle

4.3 Vectorisation

4.4 Inlining

5 Conclusion

Acronymes

CPU Central Processing Unit, processeur central de l'ordinateur
RAM Random Access Memory
HT Hyper-Threading