About an implementation "detail": a few ideas to speed up the execution and the C++ programming (in Operations Research)

Daniel Cosmin Porumbel

Sur un « détail »d'implémentation : quelques idées pour accélerer l'exécution et la programmation C++ pour la RO

Daniel Porumbel[1]

[1] Conservatoire National des Arts et Métiers, CEDRIC, 75002, Paris, France
{daniel.porumbel}@cnam.fr

[...] il se fait donc que le commodore sur la dunette inhale la plupart du temps un air de seconde main, si je puis dire, lequel air il reçoit, c'est l'évidence même, exhalé par les hommes du gaillard d'avant. Il croit, lui, qu'il le respire en premier; mais non pas.
Chapitre I "Mirages", Moby-Dick, trad. Armel Guerne

Introduction

La vitesse d'exécution de tout algorithme pratique dépend d'un facteur constant de complexité δ étroitement lié à la qualité de l'implémentation. Je présente des communications à la Roadef depuis dix ans et je viens seulement de réaliser que tous mes exposés se terminent invariablement avec une phrase comme « *Et pour finir, je vais vous épargner les détails d'implémentation parce que je préfère me focaliser sur l'essentiel et sur les concepts de haut niveau* ». Malgré ses mérites, cette phrase est trop simpliste. Elle est de nature à éloigner le lecteur d'attention moyenne de certaines vérités. Comme une porte dérobée dans un logiciel, une conséquence inavouée est la suivante : le cout en temps de calcul est très peu impacté par (le facteur constant de complexité associé à) la qualité de l'implémentation et on peut l'ignorer. Ce facteur constant δ est aussi invisible lorsqu'on calcule des complexités théoriques comme $O(n^2)$, $O(n^2 \log n)$, *etc*. Mais imaginer un vendeur qui dit que le coût à payer est de 1000€ multiplié par un facteur constant δ=2 ou δ=6 qui n'a aucune importance dans l'absolu. N'en déplaise à certains, je ne me laisserais pas convaincre si facilement.

S'il y a 15 ans on pouvait relativiser le facteur constant de complexité δ persuadés que la fréquence d'horloge des CPUs ne cessera jamais d'augmenter exponentiellement, cette idée a été un mirage en partie alimenté par une interprétation superficielle de la loi de Moore. Bien que le nombre de transistors dans un CPU peut en effet doubler tous les 18 mois, la fréquence d'horloge tend à stagner depuis 2004 et certaines limites physiques ont été atteintes. Même si j'ai aussi écrit des articles sans implémentation uniquement avec des preuves/calculs de complexités (e.g., dans la théorie des fonctions sous-modulaires), je ne veux relativiser ni le facteur constant de complexité ni le temps de programmation nécessaire pour implémenter n'importe quelle méthode pratique. Si une méthode est géniale en théorie mais trop difficile à implémenter, elle n'aura pas beaucoup de succès (penser à la méthode de l'ellipsoïde).

Bien que la théorie propose beaucoup de leviers pour déterminer le temps de calcul, la pratique est trop complexe et échappe facilement aux simplifications et axiomatisations de la théorie. Le nombre de facteurs qui rentrent en jeux est incalculable. J'ai l'habitude d'indiquer dans mes articles des facteurs comme : le langage de programmation (C++), la fréquence d'horloge, la RAM et le système d'exploitation. On comprend aisément que j'ai l'habitude de passer sous silence beaucoup d'autres aspects, comme la hiérarchie des mémoires cache, les optimisations des compilateurs, l'implémentation des structures de données, la vitesse du bus système qui gère l'accès du CPU à la mémoire, la qualité du matériel, etc. – cette liste pourrait s'allonger à perte de vue.

Il me paraît impossible de prendre en compte tous ces aspects et toutes leurs interactions pour déterminer le facteur constant δ. Il y a un seul fait qui peut simplifier la vie d'un programmeur en RO: on n'a pas forcement besoin d'artifices avancés de programmation en RO; les fonctionnalités d'un langage assez minimaliste comme le C sont suffisantes, au moins pour écrire des prototypes. La programmation en RO n'a pas forcement besoin de concepts ésotériques de programmation, comme sémantique dénotationnelle, dispatch multiple, manipulations de continuations, méthodes fabriques, polymorphisme, auto-boxing, relations complexes de typage ou d'héritage, des classes amies,etc.

Cependant, je peux rappeler que la programmation orientée objet a été inventée par des chercheurs en RO qui avaient besoin de beaucoup d'organisation dans leur code. Cela à conduit au langage Simula qui a largement influencé Stroustrup pour concevoir le C++. On ne peut pas dire que la RO ne peut jamais avoir d'influence positive sur l'informatique, ou qu'il serait inutile de s'intéresser aux idées/pratiques qui permettraient d'alléger le fardeau des programmeurs en RO.

Quelques Pratiques de Programmation

Mon but est de parler de plusieurs pratiques de programmation que j'ai utilisées en C++. Voici un premier principe: la performance est une priorité en optimisation et il faut écrire un code aussi transparent que possible, c.à.d., il faut savoir précisément ce qui se passe quand un bloc de code est exécuté. Je ne vais *pas* essayer de donner des conseils de bonne conduite en programmation. Le concepteur du C++ a résolu cette question sans équivoque: «C++ is deliberately designed to support a variety of styles rather than a would-be "one true way"». En plus, j'assume le fait que j'ai un style C++ assez atypique. En fait, je suis une preuve vivante du fait que Linus Torvalds a (presque) eu raison en disant « the only way to do good, efficient, and system-level and portable C++ ends up to limit yourself to all the things that are basically available in C. »

Grâce aux pratiques ci-dessus (http://cedric.cnam.fr/~porumbed/CODE_GUIDELINES) j'ai pu implémenter des algorithmes d'optimisation à une vitesse relativement satisfaisante. Par exemple, mon deuxième papier proposé à cette édition de la ROADEF est basé sur un code d'environ 14000 lignes et la programmation a pris environ un tiers du temps total du travail (2-3 mois). J'ai généralement programmé moi-même la plupart des algorithmes associés aux papiers que j'ai publiés, mais le temps dédié à la programmation n'a jamais été trop important. En programmant moi-même les algorithmes, j'ai aussi bénéficié d'un avantage vieux comme le monde : la meilleure façon de découvrir tout chose c'est toujours par expérience directe et non pas par l'intermédiaire des explications données par quelqu'un d'autre – dans l'esprit de la devise de cette communication.