Quand la performance compte : retour d'expérience sur la micro-optimisation

Guillaume Crognier

LocalSolver, 24 avenue Hoche, 75008 Paris, France
gcrognier@localsolver.com

Mots-clés : *micro-optimisation, solveur, cpp*

1 Introduction

LocalSolver est un solveur d'optimisation globale, mêlant des techniques de recherche opérationnelle exactes et des techniques heuristiques, comme la recherche locale [1]. De type *model-and-run*, il permet de modéliser divers problèmes d'optimisation (combinatoires, continus, mixtes, ...) et de les résoudre sur des instances de grande taille.

La performance des méthodes de résolution pour résoudre un problème de recherche opérationnelle est un enjeu majeur. Cela s'applique particulièrement au cas d'un solveur, qui a pour objectif d'être le plus rapide possible pour résoudre le problème auquel il est confronté. Lorsque des méthodes sont implémentées dans un but de performance, les critères qui rentrent en jeu sont en général les complexités théoriques en temps et en mémoire des algorithmes implémentés ainsi que les structures de données utilisées.

Bien que ces critères soient de très bon indicateurs pour qualifier la performance d'une implémentation, il peut parfois manquer un dernier niveau d'optimisation "bas niveau" pour que l'exécution du code soit encore plus rapide. Ce dernier niveau d'optimisation peut être crucial surtout dans des approches très itératives comme la recherche locale ou la recherche par voisinage. En effet, pour exprimer le maximum de leur potentiel ces approches nécessitent une implémentation très efficace en plus d'une bonne complexité algorithmique.

L'ensemble des modifications locales du code dans le but de générer un programme un peu plus efficace s'appelle la micro-optimisation. Les aspects de micro-optimisation sont en général ignorés pendant les développements car ils rendent le code plus difficile à maintenir pour des gains en performance souvent mineurs, voire nuls.

Le but de cette présentation est de faire un retour d'expérience sur un développement C++ au sein de LocalSolver pour lequel la micro-optimisation représente un enjeu majeur. Différentes techniques de micro-optimisation seront présentées en lien avec des sujets souvent peu abordés dans les conférences de recherche opérationnelle comme la prédiction de branchement, les indirections et l'*inlining*. Les résultats obtenus sur des exemples concrets dans le cadre du développement du solveur seront également présentés.

2 Possibilités de micro-optimisation

2.1 Prédiction de branchement

```
1 void f() {
2     // First part of function f
3     // [...]
4     if (condition) g();
5     else h();
6 }
```

Les méthodes de prédiction de branchement sont des techniques utilisées par les processeurs dans le but d'anticiper certains calculs sans être certain que ces calculs seront nécessaires. Un exemple simple est présenté ci-dessus.

Lorsque le code va être exécuté, le processeur va commencer à lancer les instructions correspondant à g (ou à h) avant même que la condition soit testée. Si finalement le test conduit à la branche prédite, le processeur a économisé du temps. Sinon, il doit s'assurer que son état est valide pour effectuer les calculs dans l'autre branche, quitte à annuler certaines opérations qu'il avait anticipées.

En pratique, il est possible de créer des cas d'école très simples pour lesquels on peut observer un facteur 10 en temps du seul fait de ce mécanisme. Lors de la présentation, un cas d'usage précis au sein du code de LocalSolver sera également présenté.

2.2 Indirections et inlining

En C++, la chaîne de construction d'un programme passe en général par une phase de compilation puis par une phase d'édition de liens (ou *link*). Si on le lui demande, le compilateur peut faire beaucoup d'optimisations sur un code donné [2]. Or celui-ci ne travaille que sur des unités de compilation, qui sont un ensemble de fichiers nécessaires à la compilation d'un seul fichier objet. L'implémentation de certaines méthodes (qui peuvent se trouver dans d'autres unités de compilation) est donc potentiellement inconnue au moment de la compilation, empêchant de fait d'effectuer certaines optimisations.

Dans le cas simple ci-dessous, le compilateur ne peut pas voir lors de la compilation de *other.cpp* l'implémentation de la méthode *f*, car celle-ci se trouve dans une autre unité de compilation (*utils.cpp*). S'il avait pu le voir, le compilateur aurait pu faire de l'*inlining*, c'est à dire qu'il aurait remplacé le contenu de *g* directement par le contenu de *f*.

```
// utils.hpp
void f();
```

```
// utils.cpp
#include "utils.hpp"
void f() { ... } // do something
```

```
// other.cpp
#include "utils.hpp"
void g() { f(); }
```

En pratique, cela signifie entre autres choses qu'à l'exécution la méthode *g* va appeler la méthode *f* (on parle d'indirection) alors que *g* aurait pu exécuter directement le contenu de *f*. L'indirection a un surcoût temporel qui peut ne pas être négligeable.

Lors de la présentation, des cas d'école ainsi qu'un cas d'usage au sein de LocalSolver seront détaillés.

Références

[1] F. Gardi, T. Benoist, J. Darlay, B. Estellon, et R. Megel. *Mathematical Programming Solver Based on Local Search*, Wiley, 2014

[2] GCC, the GNU Compiler Collection
https ://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Optimize-Options.html