

# M1 - Techniques d'optimisation (TOP 2021-2022)

## TD1

Rappels sur gcc, make, gdb, valgrind,  
AddressSanitizer, gprof, maqao et  
perf



hugo.taboada@cea.fr  
julien.jaeger@cea.fr



Click here or scan to download TD'files from pcloud  
link.

## I Utilisation des Makefile

**Q.1:** Écrivez un Makefile pour le code VECTOR en écrivant toutes les règles explicitement (sans utiliser de variable personnalisée ni de variable propre à Make).

**Q.2:** Améliorez le Makefile en employant cette fois-ci des variables personnalisées (*CC*, *CFLAGS*, *LDFLAGS*, *EXEC*) et variables internes à Make (`%`, `$$`, `$$@`, `$$<`).

## II Compiler un programme avec gcc

**Q.3:** Compilez sans optimisation (sans flag `-O` ou avec `-O0`) le programme situé dans répertoire *VECTOR*. Exécutez le programme. Que constatez-vous sur l'affichage des valeurs du vecteur *V3* ?

**Q.4:** Rajoutez le flag `-Wall` et corrigez tous les avertissements émis par le compilateur (warnings).

**Q.5:** Exécutez le programme corrigé et relevez le temps affiché.

**Q.6:** Compilez plusieurs versions du code avec avec les flags `-O1`, `-O2`, `-O3`, `-Ofast` et exécutez et relevez à nouveau les temps d'exécutions. Que constatez-vous ? Comparez l'assembleur.

### III Les passes de compilation de gcc

**Q.7:** Étudiez le programme dans le répertoire *SAXPY*, puis compilez-le avec l'option *-fdump-tree-all*. Utilisez la commande *ls*. Qu'observez-vous ?

**Q.8:** A quoi cela correspond-t-il ? Combien en observez-vous ?

**Q.9:** Effacez les fichiers qui sont apparus, et compilez à nouveau le fichier, en ajoutant le flag *-O1*. Qu'observez-vous ? Y a-t-il des fichiers manquants par rapport à la précédente compilation ?

### IV Prise en main de gdb ou cgdb

Vous pouvez télécharger et installer cgdb à l'adresse suivante <https://cgdb.github.io/>. Il s'agit d'un frontend de gdb permettant la coloration syntaxique entre autre.

Étudiez le programme dans le répertoire *BUGS*, puis compilez le avec le flag *-g*. Exécutez le programme compilé avec gdb :

```
gdb nom_de_l_executable
```

gdb dispose d'une aide interactive. Commencez par parcourir le menu d'aide en saisissant d'abord *help* pour avoir la liste des commandes. Puis, *help* suivi d'une commande pour obtenir des informations sur celle-ci.

**Q.10:** Afin de repérer la source du premier bug, tapez *run* sous gdb. Quittez gdb (*quit*), corrigez l'erreur et recompilez le programme.

**Q.11:** Procédez de la même manière pour corriger le bug suivant (i.e. *gdb executable*, puis *run*). Utilisez la commande *backtrace* (ou *bt*) pour afficher la pile des appels de fonctions et obtenir plus d'informations sur la source de l'erreur.

**Q.12:** Identifiez la prochaine erreur après avoir recompile le programme. Quel est le problème et peut on le corriger ?

**Q.13:** Nous allons maintenant résoudre le dernier bug avec d'autres fonctionnalités élémentaires de gdb. Démarrez gdb sans utiliser la commande *run* pour le moment. Fixez un point d'arrêt sur la fonction *launch\_fibonacci* (*breakpoint launch\_fibonacci* ou *b launch\_fibonacci*). Utilisez ensuite la commande *run*. Le programme va s'arrêter lors de la première entrée dans la fonction *launch\_fibonacci*. Essayez maintenant d'accéder à la valeur *fibonacci\_values* → *max* avec la commande *print fibonacci\_values* → *max* que constatez-vous ? Saisissez maintenant *up* pour se placer avant l'appel de la fonction puis *list* pour afficher les lignes de code autour du point d'arrêt. Entrez maintenant la commande *print fibonacci\_values*. Corrigez le problème et recompilez le programme.

**Q.14:** La suite est incorrecte. Nous devrions obtenir les nombres suivants :

$\mathcal{F}_0 = 0$ ,  $\mathcal{F}_1 = 1$ ,  $\mathcal{F}_2 = 1$ ,  $\mathcal{F}_3 = 2$ ,  $\mathcal{F}_4 = 3$ ,  $\mathcal{F}_5 = 5$ ,  $\mathcal{F}_6 = 8, \dots$

Pour repérer d'où vient l'erreur, nous allons afficher pas par pas les valeurs de la suite en surveillant les modifications de la variables `fibo_values`  $\rightarrow$  `result`.  
Démarrez gdb et saisissez les commandes suivantes :

```
b main
run
watch fibo_values->result
```

Entrez ensuite *continue* (ou *c*) pour avancer pas à pas à chaque modification de `fibo_values`  $\rightarrow$  `result`. Trouvez où l'erreur se situe.

## V Prise en main de valgrind et AddressSanitizer

Valgrind est un outil de programmation libre pour déboguer, effectuer du profilage de code et mettre en évidence des fuites mémoires.(Wikipedia)

AddressSanitizer (or ASan) is an open source programming tool that detects memory corruption bugs such as buffer overflows or accesses to a dangling pointer (use-after-free). AddressSanitizer is based on compiler instrumentation and directly mapped shadow memory. AddressSanitizer is currently implemented in Clang (starting from version 3.1), GCC (starting from version 4.8[2]), Xcode (starting from version 7.) and MSVC (widely available starting from version 16.9). (Wikipedia)

**Q.15:** Utilisez valgrind sur le code SAXPY. Qu'observez-vous ?

**Q.16:** Utilisez address sanitizer sur le code SAXPY. Qu'observez-vous ?

**Q.17:** Corrigez le problème.

## VI Prise en main de gprof et perf

Gprof est un logiciel GNU Binary Utilities qui permet d'effectuer du profilage de code. lors de la compilation et de l'édition de liens d'un code source avec gcc, il suffit d'ajouter l'option `-pg` pour que, lors de son exécution, le programme génère un fichier `gmon.out` qui contiendra les informations de profilage. Il suffit ensuite d'utiliser gprof pour lire ce fichier, en spécifiant les options. (Wikipedia)

Perf (sometimes called `perf_events` or `perf tools`, originally Performance Counters for Linux, PCL) is a performance analyzing tool in Linux, available from Linux kernel version 2.6.31 in 2009. Userspace controlling utility, named `perf`, is accessed from the command line and provides a number of subcommands; it is capable of statistical profiling of the entire system (both kernel and userland code). (Wikipedia)

Le code MolDyn fourni est une maquette d'une simulation de dynamique moléculaire dans un gaz (interaction entre les molécules du gaz).

```
Makefile pour compiler :  
  make NPART=MINI  
=> 1372 particules (i.e. molécules)  
  make NPART=MEDIUM  
=> 4000 particules  
  make NPART=MAXI  
=> 13500 particules  
fichier binaire md.
```

**Q.18:** Utilisez `gprof` sur le code MolDyn. Quelle est la fonction la plus couteuse en calcul ?

**Q.19:** Utilisez `perf` sur le code MolDyn. Quelle est la fonction la plus couteuse en calcul ?

## VII Prise en main de maqao

Télécharger maqao sur le site <http://www.maqao.org/>.

**Q.20:** Utilisez `maqao lprof` sur le code MolDyn. Quelle est la boucle la plus cher en calcul ?

**Q.21:** Utilisez `maqao cqa` sur la boucle la plus cher en calcul. Est-ce que le code est bien vectorisé ?

**Q.22:** Utilisez les conseils de `maqao cqa` pour améliorer les performances du code MolDyn

**Q.23:** Quels problèmes peut-on rencontrer lors de l'utilisation de maqao prématurément ?

## VIII Quid d'un debugger pour les programmes parallèles ?

Il existe des debuggers spécifiques aux besoins des programmes parallèles. Nous pouvons citer TotalView ou Arm DDT par exemple. Bien que ces logiciels soient très puissants, ces logiciels sont payant. Néanmoins, il est possible d'utiliser `gdb` pour débbugger des programmes parallèles à petite échelle. Pour les programmes multi-threadés, il suffit d'utiliser la commande `info threads` dans `gdb`. Nous pouvons choisir de visualiser un thread en particulier avec la commande `thread #thread_number`. Pour les programmes MPI, nous pouvons utiliser l'astuce suivante : `mpirun -np 2 xterm -e gdb -args ./monprogramme arg1 arg2`. Cela ouvrira un terminal par processus MPI.

**Q.24:** Repartez du code MolDyn d'origine et parallélisez la fonction la plus couteuse du programme MolDyn en openMP ou pthread.

**Q.25:** Utilisez les méthodes ci-dessus pour déboguer votre code parallèle sur le code MolDyn.

**Q.26:** Faites un programme MPI et essayez la méthode de debug avec xterm+gdb.