MYCHP203 — TOP : TD 1

Gabriel Dos Santos
gabriel.dossantos@cea.fr
gabriel.dos-santos@uvsq.fr

Hugo Taboada
hugo.taboada@cea.fr

,

Cliquer ici pour accéder
aux fichiers du TD

26 mars 2024

TD1 : Rappels sur les compilateurs, moteurs de production, débogueurs, code sanitizers et profilers pour le HPC.

I Utilisation des systèmes de build `GNU Make` et `CMake`

Q.1 : Écrivez un `Makefile` pour le code `vector`. Écrivez toutes les régles 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` , etc.) et variables internes à Make (`%` , `$^` , `$@` , `$<` , ...).

Q.3 : Remplacer le `Makefile` par un `CMakeLists.txt` .

II Compilation

Q.4 : Compilez sans optimisation (sans flag `-O` ou avec `-O0`) le programme situé dans le répertoire `vector` . Exécutez le programme. Que constatez-vous sur l'affichage des valeurs du vecteur `V3` ?

Q.5 : Rajoutez les flags `-Wall -Wextra -Werror` et corrigez tous les avertissements émis par le compilateur afin que le programme compile de nouveau.

Q.6 : Exécutez le programme corrigé et relevez le temps affiché.

Q.7 : 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 Passes de compilation avec GCC

Q.8 : Étudiez le programme dans le répertoire `saxpy` , puis compilez-le avec l'option `-fdump-tree-all` . Utilisez la commande `ls` . Qu'observez-vous ?

Q.9 : À quoi cela correspond-il ? Combien en observez-vous ?

Q.10 : 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

Étudiez le programme dans le répertoire `bugs`, puis compilez le avec le flag `-g3`. Exécutez le programme compilé avec le GNU Debugger (GDB) :

```
gdb <BIN_NAME>
```

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.11 : Afin de repérer la source du premier bug, tapez `run` (ou `r`) sous GDB. Quittez (`quit`), corrigez l'erreur et recompilez le programme.

Q.12 : Procédez de la même manière pour corriger le bug suivant. 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.13 : Identifiez la prochaine erreur après avoir recompilé le programme. Quel est le problème et peut on le corriger ?

Q.14 : Nous allons maintenant résoudre le dernier bug avec d'autres fonctionnalités élémentaires de GDB. Démarrez le débogueur sans utiliser la commande `run` pour le moment. Fixez un point d'arrêt sur la fonction `launch_fibonacci()` :

```
breakpoint launch_fibonacci
# or
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 `fibo_values->max` :

```
print fibo_values->max
```

Que constatez-vous ? Utilisez `up` pour vous 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 fibo_values`. Corrigez le problème et recompilez le programme.

Q.15 : La suite est incorrecte. Nous devrions obtenir les nombres suivants :

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8, \ldots$$

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->result`.

Démarrez GDB et saisissez les commandes suivantes :

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

Entrez ensuite `continue` (ou `c`) pour avancer pas à pas à chaque modification de `fibo_values->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. (voir Page Wikipédia)

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). (see Wikipedia page)

Q.16 : Utilisez Valgrind sur le code `saxpy`. Qu'observez-vous ?

Q.17 : Utilisez AddressSanitizer sur le code `saxpy`. Qu'observez-vous ? Corrigez le(s) problème(s).

Q.19 : Que sont que `dhat`, `memcheck`, `cachegrind`, `callgrind`, `massif` ?

Q.20 : Que sont que `kcachegrind`, `massif-visualizer` ?

VI Prise en main de `Linux perf` et `gprof`

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.

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 userspace code).

Le code `mol-dyn` fourni est une maquette d'une simulation de dynamique moléculaire dans un gaz (intéraction entre les molécules du gaz).

Makefile pour compiler :

```
make NPART=MINI
```

$\Longrightarrow$ 1372 particules (i.e. molécules)

```
make NPART=MEDIUM
```

$\Longrightarrow$ 4000 particules

```
make NPART=MAXI
```

$\Longrightarrow$ 13500 particules

Q.21 : Utilisez Gprof sur le code `mol-dyn`. Quelle est la fonction la plus couteuse en calcul ?

Q.22 : Utilisez Perf sur le code. Quelle est la fonction la plus couteuse en calcul ?

Q.23 : Qu'est ce qu'un *hotspot* ?

VII Prise en main de `MAQAO`

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

Q.24 : Utilisez le module `lprof` de MAQAO sur le code `mol-dyn`. Quelle est la boucle la plus chère en calcul ?

Q.25 : Utilisez le module `cqa` de MAQAO sur la boucle la plus chère en calcul. Est-ce que le code est bien vectorisé ?

Q.26 : Utilisez les conseils de CQA pour améliorer les performances du code `mol-dyn`.

Q.27 : Quels problèmes peut-on rencontrer lors de l'utilisation prématurée de MAQAO ?

VIII Prise en main de Tau et de Scalasca

Q.28 : Installez Tau avec `spack`, et refaites les manipulations sur `mol-dyn`.

Q.29 : Faites de même avec Scalasca.

IX 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 payants. Néanmoins, il est possible d'utiliser GDB pour déboguer 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 <NPROC> <TERM_EMULATOR> -e gdb -args <MY_BIN> [ARGS ... ]
```

Cela ouvrira un terminal par processus MPI.

Q.30 : Repartez du code `mol-dyn` d'origine et parallélisez la fonction la plus couteuse du programme en OpenMP ou Pthread.

Q.31 : Utilisez les méthodes ci-dessus pour déboguer votre code parallèle sur le code `mol-dyn`.

Q.32 : Écrivez un programme MPI et essayez la méthode de débogue avec l'émulateur de terminal + GDB.