TP : Flux stochastiques – modèles du hasard - Nombres quasi-aléatoires - Parallélisme

Génération de flux parallèles de nombres pseudo-aléatoires

Confrontation avec une « vraie » bibliothèque – CLHEP

Mail : bibliothèque CLHEP, code de calcul intégral

But principal du TP : se confronter à une bibliothèque de taille professionnelle pour la simulation (Physique des Hautes Energies : CLHEP) et réaliser un calcul stochastique distribué. ATTENTION : bien l**ire le sujet et les TIPS** avant d'attaquer les questions.

1) Installer la bibliothèque (observer la hiérarchie des répertoires, où sont les sources, les include, les codes de test, regarder les nouveaux répertoires qui sont créés, la création des bibliothèques de leurs chemins d'accès…). Tester cette bibliothèque avec le code C++ donné en fin de TP et le code de test fourni (`test.cc`)

2) Archiver quelques statuts pour le générateur Mersenne Twister (MT) avec de très petites séquences d'une longueur donnée (10 nombres par exemple). Tester la restauration à un statut donné. Réfléchir et proposer une façon de lancer **en parallèle** 10 réplications d'une simulation de Monte Carlo sur un serveur multi-cœur (Etud = 48 cœurs logiques) avec des flux stochastiques indépendants. Utiliser les méthodes saveSatus et restoreStatus qui prennent un nom de fichier en paramètre.

3) Faire un petit code de test de la méthode de « Sequence splitting » (découpage des séquences tous le N nombres avec N très petit) avec le générateur Mersenne Twister.

4) Utiliser la technique du séquence splitting pour calculer en parallèle 10 réplications tirant chacune 1 milliards de points pour calculer PI (2 milliards de nombres par réplication).

5) Utiliser MT pour générer des mots et des portions de phrases avec des caractères tirés au hasard. Par exemple écrire un programme qui essayer de générer au hasard l'oligopeptide : « gattaca » (avec pour chaque tirage de lettre seulement 4 possibilité ('A', 'C', 'G', 'T') pour les 4 bases nucléiques, compter le nombre d'essais pour une réussite. Cela vous donnera tout d'abord avoir une idée du nombre de tirages nécessaires sur une seule réplication, puis faire 40 réplications indépendantes pour donner une probabilité moyenne (ne pas réinitialiser le générateur si on est en séquentiel, puis utiliser la technique du « sequence splitting » en restaurant des statuts de MT espacés de façon adaptée). Quand vous aurez fait plusieurs simulations, vous pourrez calculer une moyenne de probabilité et un écart type (dans l'idéal on donnerait un intervalle de confiance). Comparer avec la probabilité exacte. Essayer de générer au hasard la phrase : « Le hasard n'écrit pas de programme. » en se basant sur les anciens codes ASCII. Puis estimer la probabilité d'obtenir par hasard cette chaine et enfin estimer la probabilité d'avoir par hasard la chaîne fonctionnelle d'un humain avec de 3 milliards de bases (taille de votre ADN).

TIPS : Partir de la version CLHEP modifiée par Romain Reuillon (Random.tgz - CLHEP)

```
tar zxvf random.tgz                 On dézippe avec l'option Z, puis eXtract Verbose

./configure --prefix=$PWD           Ici PWD est le répertoire d'installation
make                                Un make –j NbCoeurs exploite le parallélisme
make install
```

TIPS (in English for CLHEP)

Install the CLHEP library (given in .tgz). Check the hierarchy of folders, where are the sources, include… **before and after installation** – list files & folders with dates. **Look at ALL the tips below** before installing. The Unix command tar below (Tape ARchive) will unzip the .tgz with a verbose formatted manner).

```
$ tar zxvf CLHEP-Random.tzg
```

In your installation process you will first make a sequential installation. First, find the configure script in the *Random* directory that has been created, launch it before making the compilation, check and note the total compilation time.

```
./configure --prefix=$PWD // where PWD is the installation directory
time make                 // look at how much time is needed for sequential compilation
```

Then, remove the created directories (`rm –fr ./Random`). Now you will compile in parallel to exploit the full potential of the SMP machine you are using for the labs. The etud server proposes many logical cores. Check and note how much time you save with a parallel compilation (though you are all sharing the same machine). The user time will approximatively be the same, the sys time also, but the real time (wall clock) – **your real waiting time** – will be reduced thanks to this parallelism.

```
./configure --prefix=$PWD // where PWD is the installation directory
time make –j32            // specifies a parallel compilation with 32 logical cores
make install              // this rule of the Makefile finishes the installation
```

Test the code proposed at the end of this file (in the Lab2 directory) and also the main test proposed in the package (Full testing – the code is present in the test Directory of the installed library).

In order to compile the code you need to look where the '*include'* and '*lib'* libraries have been installed. If the installation went correctly, you will find two versions of the CLHEP library. The date/time for both files will correspond to your compilation. One will be the static version with a '.a' extension (like libm.a for the static version of the standard math library). You can see the object files inside a static library with the regular *ar* Unix command. Ex: `ar –t libm.a` lists all the object files of the standard C math library.

```
libCLHEP-Random-2.1.0.0.a
```

The other one will be a dynamic version '.so' – for shared objects – this corresponds to DLLs (Dynamic Link Libraries) under Windows). In this version, only one version of each compiled object file will be loaded in memory for all executables. Objects will be shared and re-entered at runtime by executing processes.

```
libCLHEP-Random-2.1.0.0.so
```

To compile you test file (given below), you can first start with a separate compilation ('*-c' option*) to first avoid the linking stage. The *'-I'* option gives the path where we find the include directory – in this case we suppose that we have the testRand.cpp file place just above the *include* directory. You can use the absolute path (it will be longer to strike).

```
g++ -c testRand.cpp –I./include
```

If this phase is ok, you can go further to add the linking stage after compilation. This will be done if we remove the *'-c'* option and also precise the path of the *lib* directory with the *'-L'* option. Then we ask for an executable result with the output *'-o'* option

```
g++ testRand.cpp –I./include –L./lib –o myExe
```

This will not be enough, since we have not mentioned that we use the CLHEP library at this linking stage. (like if we had forgotten to precise *'-lm'* for a C program that uses a function of the math library). Depending on the server installation context (changing every year with updates) you have to find the right compilation line that will work. Here are some lines below; the first is in a dynamic link context. Then we go more and more static (if needed you can add –static to force the usage of a static library).

```
(1)g++ -o myExe testRand.cpp -I./include -L./lib -lCLHEP-Random-2.1.0.0
(2)g++ -o myExe testRand.cpp -I./include -L./lib -lCLHEP-Random-2.1.0.0 -static
(3)g++ -o myExe testRand.cpp -I./include -L./lib ./lib/libCLHEP-Random-2.1.0.a
(4)g++ -o myExe testRand.cpp -I./include ./lib/libCLHEP-Random-2.1.0.a
```

When everything works fine, you can go in the `test` directory and compile the `testRandom.cc` file which makes a much wider usage of all objects & methods proposed in the library. On modern C++ compilers, you may have to include `stdlib.h` for the `exit` function. This is achieved by achieved by a `#include <cstdlib>` (note that a 'c' is added to the standard name and that the '.h' extension is not mentioned in modern C++ when we include old C libraries. If you are familiar with Unix, you can use the `ldd` Unix command to see the dependencies of an executable – you may see that the LD_LIBRARY_PATH global variable has to be updated to specify the correct PATH and to enable a dynamic linking with the shared object library.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD/CLHEP/lib
```

When compiled the executable will give you an idea of the utility of this library for Monte Carlo simulations which simulates the major probability.

Super classe

Pour tirer un nombre utiliser la méthode : `flat();`
Pour sauver un statut : `saveStatus(nomDeFichier);`
Pour restaurer le générateur à un statut : `restoreStatus(nomDeFichier);`

Sous classes

La bibliothèque CLHEP propose de nombreux générateurs, selon les connaissances actuelles, 2 générateurs seulement sont corrects : Ranlux64Engine (très lent avec un 'luxury' level de 64, et Mersenne Twister de Matsumoto – non cryptosecure). Les autres générateurs sont conservés par souci de reproductibilité numériques des anciens travaux. Voici la liste des générateurs de cette bibliothèque.

```
JamesRandom             RanecuEngine
TripleRand              RanshiEngine
DRand48Engine           Ranlux64Engine
DualRand                RanluxEngine
Hurd160Engine           Hurd288Engine
MTwistEngine            RandEngine
NonRandomEngine
```

Exemple de code CLHEP :

Objectif du code ci-dessous générer des nombres dans un fichier binaire (pour test avec le logiciel DIE HARD de G. Marsaglia)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <unistd.h>

#include "CLHEP/Random/MTwistEngine.h"

int main ()
{
   CLHEP::MTwistEngine * s = new CLHEP::MTwistEngine();

   int          fs;
   double       f;
   unsigned int nbr;

   fs = open("./qrngb",O_CREAT|O_TRUNC|O_WRONLY,S_IRUSR|S_IWUSR);

   for(int i = 1; i < 3000000; i++)
   {
     f = s->flat();
     nbr = (unsigned int) (f * UINT_MAX);

     // printf("%f\n", f); ou si iostream       cout << f << endl;

     write(fs,&nbr,sizeof(unsigned int));
   }

   close(fs);

   delete s;

   return 0;
}
```