HPC - Lab # 2 and Home Work
Random number statuses & Parallel Stochastic streams

Generation of parallel random streams
Practice with a scientific library used for High Performance Computing in High Energy Physics

In www.isima.fr/~hill/HPC/HPC-Lab2-Docs you will find a *.tgz of the CLHEP library (C++), a C code for a Monte Carlo simulation - computing of PI and an efficient code for 1D quasi-random generation*

Computer Science skills required: *Unix / C / C++ & and a little of "sysadmin"*

A) Basics: Generation of pseudorandom numbers with a high quality generator

Find and test the current implementation in C of Mersenne Twister (MT) (on **Makoto Matsumoto Home Page – Mersenne Twister – explanation & C code – (improved initialization)** March 2004 version. Compare your results with the expected output archived by the authors of MT - reproducibility is the essence of the scientific method. This will ensure that the generator has been well ported.

FYI - no need to test this : A new version by Saito has been proposed in 2006. It is named SFMT and it uses the vector facilities of modern processors (maximum period 2^{216091}-1) and it is two times faster on modern architecture which uses SSE (Intel SIMD instructions set). Saito is also proposing a GP-GPU version of MT (since 2009) and a TinyMT.

Test the speed of MT (2002), with and without optimization (-O2 sufficient) for drawing 1 billion numbers (1 000 000 000 numbers). Use simply the Unix time command ("man time" for beginners to read the manual pages for this Unix command). Note the computing time of each (optimized and non-optimized) and the achieved speedup and remember to always check for reproducibility. What if I am much faster but without the correct results! Beware for instance that –O3 could change the output. In this case the speedup when compiling with both O3 and O2 are significant. **Launching un-optimized parallel code is a considerable waste of time and energy.**

B) Use of a scientific library in C++ for stochastic simulation (CLHEP)

1) 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
```

2) 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
```

3) 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.

```
(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.

4) Modify the basic example code first proposed (testRand.cpp) to test a parallelization technique of random streams known as « sequence splitting » or fixed spacing. Use the saveSatus method (of the RandomEngine) class to archive 10 statuses of the Mersenne Twister, each with a very small number of drawings (10 drawings for instance, name the files like this: status1 to status10). Check that it works fine with the restoreStatus method and a basic display of the drawn numbers.

5) When it works do the same but with '1 000 000' numbers and name the statuses files Status01 to Status10. This will enable the execution of 10 parallel independent simulations consuming less than 1 million pseudo-random numbers. The etud server has 32 logical cores. Thus at least 3 students car run the test in pseudo-parallelism on etud without competition for resources…

<u>C) Parallel Monte Carlo simulation and Quasi Monte Carlo (Home Work for fast students and those interested)</u>

1) Check Xmtc.c code given by email to compute PI (NOTE : If Xlib is installed you can compile and display the random drawings under X-windows (find the path for Xlib and link with –lX11 ; otherwise remove the graphic part and just look at the PI computation part).

2) Rename the file and remove the graphical part to have a code that will be run in parallel

    a. with the Linux basic rand() as is it ;
    b. with MersenneTwister as implemented thanks to CLHEP. This means incorporating the MonteCarlo computation inside the CLHEP example given.

3) Thanks to CLHEP and the work done in the previous questions, consider doing independent replicates, each with 1 000 000 drawings to provide a mean and a standard deviation. If you are a simulationists you would prefer computing confidence intervals.

   Use the different statuses previously archived to run in parallel 10 replications. Provide a script or code that will give the mean and the standard deviation.

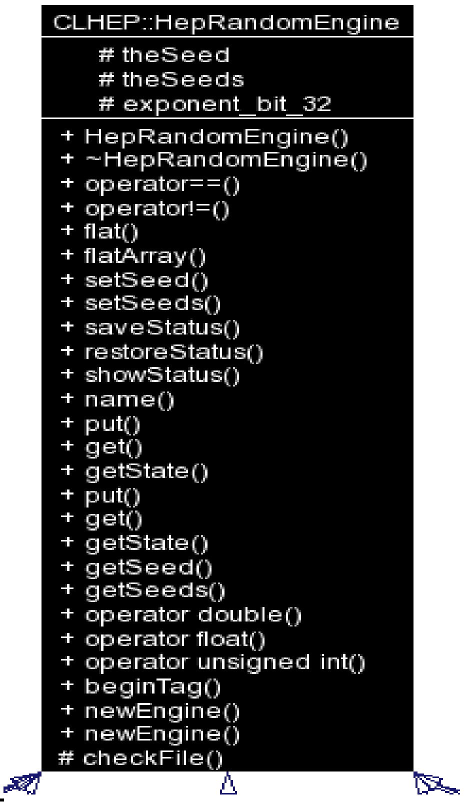
   FYI: many parallelization techniques exist to propose parallel random streams, the sequence splitting (or blocking) is a basic one (it can show long range correlation for large simulations). Go back to Matsumoto Home page to see what is proposed by the "Dynamic Creator" approach.

4) Compile and test the last code given – a quasi-random number generator in 1 dimension.

5) How would you adapt it to 2 Dimensions and would you need replicates if you replaced pseudo-random number drawings by quasi-random drawings?

<u>The CLHEP::HepRandomEngine proposes the following subclasses: it is possible to use the following random number generators</u>

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



Superclass Methods

Example of code using CLHEP and methods for saving and restoring MT statuses:

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

#include "CLHEP/Random/MTwistEngine.h"

using namespace std;

int main ()
{
   double       sum;
   unsigned int nbr;

// Creation of a random stream using MT
   CLHEP::MTwistEngine * rs = new CLHEP::MTwistEngine();

   rs->saveStatus(“MT-Status0”);              // Saves the current status of the MT Engine

   for(int i = 1; i < 1000000; i++)
   {
     sum += rs->flat();                        // Draws random numbers between 0 and 1 and sum them
   }
   cout << sum / 1000000. << endl;

   rs->retsoreStatus(“MT-Status0”);            // Comes back to the MT Status before the loop

   delete rs;

   return 0;
}
```