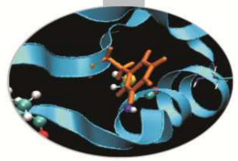
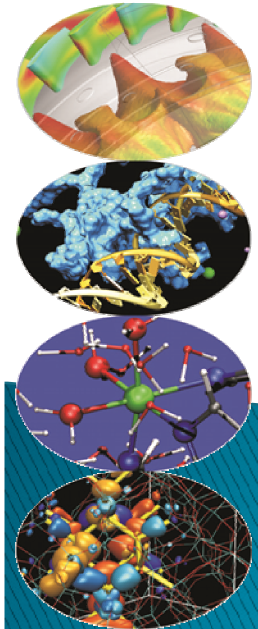
Random Number on a HPC system: An overview





HPC Numerical Libraries
26-28 April 2016
CINECA – Casalecchio di Reno (BO)

Massimiliano Guarrasi- m.guarrasi@cineca.it
Super Computing Applications and Innovation Department

Part 1: Randomness and Random Number

Randomness and Random Number

- Random numbers are useful for a variety of purposes
- A random number is one that is drawn from a set of possible values, each of which is equally probable, i.e., a uniform distribution.
- When discussing a sequence of random numbers, each number drawn must be statistically independent of the others.
  - However, surprising as it may seem, it is difficult to get a computer to do something by chance.
  - A computer follows its instructions blindly and is therefore completely predictable. (A computer that doesn't follow its instructions in this manner is broken.)

Usefulness Of Random Numbers

- Simulation
  - In many scientific and engineering fields, computer simulations of real phenomena are essential to understanding. When the real phenomena are affected by unpredictable processes, such as radio noise or day-to-day weather, these processes must be simulated using random numbers.
- Statistical Sampling
  - Statistical practice is based on statistical theory which, itself is founded on the concept of randomness. Many elements of statistical practice depend on the emulation of randomness through random numbers.
- Analysis
  - Many experiments in physics rely on a statistical analysis of their output. For example, an experiment might collect X-rays from an astronomical source and then analyze the result for periodic signals.

Usefulness Of Random Numbers

- Computer Programming
  - Most computer programming languages include functions or library routines that purport to be random number generators. They are often designed to provide a random byte or word, or a floating point number uniformly distributed between 0 and 1.
- Cryptography
  - A ubiquitous use of unpredictable random numbers is in cryptography which underlies most of the attempts to provide security in modern communications (e.g., confidentiality, authentication, electronic commerce, etc.).
- Decision Making
  - There are reports that many executives make their decisions by flipping a coin or by throwing darts, etcIt is important to make a completely "unbiased" decision. Randomness is also an essential part of optimal strategies in the theory of games.

Random Number Generators

- There are many different methods for generating random data. These methods may vary as to how unpredictable or statistically random they are, and how quickly they can generate random numbers.
- Before the advent of computational random number generators, generating large amount of sufficiently random numbers required a lot of work. Results would sometimes be collected and distributed as random number tables.

Random Number Generation Methods

- *Random numbers should not be generated with a method chosen at random* – Donald Knuth

- There are two main approaches to generating random numbers using a computer:
  - Pseudo-Random Number Generators (PRNGs)
  - True Random Number Generators (TRNGs).

Pseudo Random Number Generators

- Pseudo-random numbers are not random in the way you might expect, at least not if you're used to dice rolls or lottery tickets.
- Essentially, PRNGs are algorithms that use mathematical formula or simply precalculated tables to produce sequences of numbers that appear random.
- A good deal of research has gone into pseudo-random number theory, and modern algorithms for generating pseudo-random numbers are so good that the numbers look exactly like they were really random.
- Effectively, the numbers appear random, but they are really predetermined.

Pseudo Random Number Generators

- PRNGs are
  - Efficient (can produce many numbers in a short time)
  - Deterministic (a given sequence of numbers can be reproduced)
  - Periodic (the sequence will eventually repeat itself)
- These characteristics make PRNGs suitable for applications where many numbers are required and where it is useful that the same sequence can be replayed easily.
- Popular examples of such applications are simulation and modeling applications.
- PRNGs are not suitable for applications where it is important that the numbers are really unpredictable, such as data encryption and gambling.

True Random Number Generators

- In comparison with PRNGs, TRNGs extract randomness from physical phenomena and introduce it into a computer
- A really good physical phenomenon to use is a radioactive source. The points in time at which a radioactive source decays are completely unpredictable, and they can quite easily be detected and fed into a computer .
- Another suitable physical phenomenon is atmospheric noise, which is quite easy to pick up with a normal radio.
- A common technique is hashing a frame of a video stream from an unpredictable source.
- Most notable perhaps was Lavarand which used images of a number of lava lamps.
- Lithium Technologies uses a camera pointed at the sky on a windy and cloudy day.

TRNGs vs PRNGs

Characteristic	Pseudo-Random Number Generators	True Random Number Generators
Efficiency	Excellent	Poor
Determinism	Determinstic	Nondeterministic
Periodicity	Periodic	Aperiodic

TRNGs vs PRNGs

Application	Most Suitable Generator
Lotteries and Draws	TRNG
Games and Gambling	TRNG
Random Sampling (e.g., drug screening)	TRNG
Simulation and Modelling	PRNG
Security (e.g., generation of data encryption keys)	TRNG

Part 2:
Most common Random Generators

Uniform Random Generators

- It is quite common finding random number generators that creates uniformly distributed between 0 and 1.
- It is always possible to transform a random number (or a series) that follow a particular distribution in another one that follow a complete different distribution.
  - We will see in the following the most common methods to make this.
- In the following slides we will see some of the most common (and used, of course) algorithms to generate a series of pseudo-random numbers uniformly distributed:
  - Linear Congruental generator (LCG)
  - Lagged Fibonacci Generator (LFG)
  - Blum Blum Shoub (BBS)

Linear Congruential Generator

- Linear congruential generators (LCGs) represent one of the oldest and best-known pseudorandom number generator algorithms.
- It generates an uniform distributed sequelce of random number between 0 and M
- LCGs are defined by the recurrence relation:

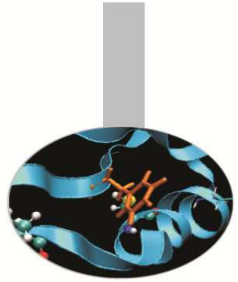
$$x_{i+1} = \left(Ax_i + C\right)\mathrm{mod}\left(M\right)$$

- Where x_n is the sequence of random values and A, C and M are generator-specific integer constants.
- X_n is an external seed
- A is a multiplier
- C is a shift factor
- M is the modulus

Linear Congruential Generator

- The period of a general LCG is at most M, and in most cases less than that. The LCG will have a full period if:
  - C and M are relatively prime
  - A-1 is divisible by all prime factors of M.
  - A-1 is a multiple of 4 if M is a multiple of 4
  - M > max (A, B, V0)
  - A > 0, B > 0
- Neither this, nor any other LCG should be used for applications where high-quality randomness is critical.
- They should also not be used for cryptographic applications
- LCGs may be the only option in an embedded system, the amount of memory available is often very severely limited

Linear Congruential Generator



Fortran 90

```fortran
module lcgs
  implicit none

  integer, parameter :: i64 = selected_int_kind(18)
  integer, parameter :: a1 = 1103515245, a2 = 214013
  integer, parameter :: c1 = 12345, c2 = 2531011
  integer(i64), parameter :: m = 2147483648_i64

  contains

  function bsdrand(seed)
    integer :: bsdrand
    integer, optional, intent(in) :: seed
    integer(i64) :: x = 0

    if(present(seed)) x = seed
    x = mod(a1 * x + c1, m)
    bsdrand = x
  end function
end module


program lcgtest
  use lcgs
  implicit none
  integer :: i

  write(*, "(a)") "      BSD "
  do i = 1, 10
    write(*, "(i12)") bsdrand()
  end do
end program
```

C

```c
#include <stdio.h>

/* always assuming int is at least 32 bits */
int rand();
int rseed = 0;

inline void srand(int x)
{
        rseed = x;
}

#ifndef MS_RAND
#define RAND_MAX ((1U << 31) - 1)

inline int rand()
{
        return rseed = (rseed * 1103515245 + 12345) & RAND_MAX;
}

#else /* MS rand */

#define RAND_MAX_32 ((1U << 31) - 1)
#define RAND_MAX ((1U << 15) - 1)

inline int rand()
{
        return (rseed = (rseed * 214013 + 2531011) & RAND_MAX_32) >> 16;
}

#endif/* MS_RAND */

int main()
{
        int i;
        printf("rand max is %d\n", RAND_MAX);

        for (i = 0; i < 100; i++)
                printf("%d\n", rand());

        return 0;
}
```

Output:

```
       BSD
      12345
 1406932606
  654583775
 1449466924
  229283573
 1109335178
 1051550459
 1293799192
  794471793
  551188310
```

Lagged Fibonacci Generator

- This class of random number generator is aimed at being an improvement on the 'standard' linear congruential generator. These are based on a generalisation of the Fibonacci sequence.
- Fibonacci sequence is defined as:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n > 1 \end{cases}$$

- The Lagged Fibonacci Algorithm is:

$$S_n = \left(S_{n-j} * S_{n-k}\right) \bmod (m)$$

$$0 < j < k$$

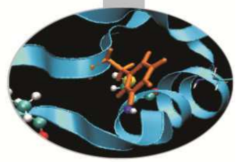
- In which case, the new term is some combination of any two previous terms.
- m is usually a power of 2 (m = 2M).
- The operator denotes a general binary operation. This may be either addition, subtraction, multiplication, or the bitwise arithmetic exclusive-or operator (XOR).
- The theory of this type of generator is rather complex, and it may not be sufficient simply to choose random values for j and k.
- These generators also tend to be very sensitive to initialization.

Blum Blum Shub

- Blum Blum Shub (BBS) is a pseudorandom number generator proposed in 1986 by Lenore Blum, Manuel Blum and Michael Shub.

$$x_i = \left(x_0^{2^i \bmod (p-1)(q-1)} \right) \bmod (M)$$

- Where M=pq is the product of two large primes p and q.
- The generator is not appropriate for use in simulations, only for cryptography, because it is not very fast.



Part3:
How to Change Probability Density Function

Method of inversion

- The **method of Inversion** (or the **inverse cdf method** as it is sometimes called) can be used to obtain transformations for many distributions.
- Consider the problem of generating a random variable C from a cumulative dinstribution function F, and suppose that F is continuous and strictly increasing and $F^{-1}(u)$ is well-defined for $0 \le u \le 1$. If U is a random variable from U(0,1), then it can be shown that $X = F^{-1}(U)$ is a random variable from the distribution F.
- Useful only if the function can be easily inverted.

Method of inversion in some simple steps

- Start writing the PDF: f(x) where <u>A < x < B</u>
- Generate the Cumulative Distribution Function (CDF):

$$u = F(x) = \int_A^x f(x)dx$$

- Create the inverse function: $F^{-1}(u) = x$

- Generate a series of uniformly distributed random number:

$$u_i \in U(0,1)$$

- $x_i = F^{-1}(u_i)$ is a series of random number that follow the PDF f(x)

Method of inversion: Example 1

- From U(0,1) to f≡U(A,B)
- Clearly f is:

$$f = \begin{cases} \dfrac{1}{B-A} & 0 \le x \le 1 \\ 0 & otherwhise \end{cases}$$

- The corresponding CDF is: $$u = \frac{1}{B-A}(X-A)$$
- The inverse function $F^{-1}(u)$ is: $x = A + (B-A)u$
- If $u_i \in U(0,1)$

$$x_i = A + (B-A)u_i \in U(A,B)$$

Method of inversion: Example 2

- From U(0,1) to $f\left(0,+\infty\right) \rightarrow f(x) = \lambda e^{-\lambda x}$
- The corresponding CDF is: $u = 1 - e^{-\lambda x}$
- The inverse function $F^{-1}(u)$ is: $x = -\frac{1}{\lambda}\log\left(1-u\right)$
- If $u_i \in U\left(0,1\right)$

$$x_i = -\frac{1}{\lambda}\log\left(1-u_i\right) \in f\left(0,+\infty\right)$$

Method of inversion: Limitations

- To use this method the PDF need to be:
  - Continuous
  - With CDF
  - strictly increasing
  - $F^{-1}(u)$ is well-defined
- If not, the method is not valid.
- Some other method can be used! E.g.:
  - Box-Muller (only for Normal distribution)
  - Rejection Methods (always valid, but computational expensive)

Box-Muller

- If U_1, U_2 are two independently distributed U(0,1) random variables, then it can be proved that:

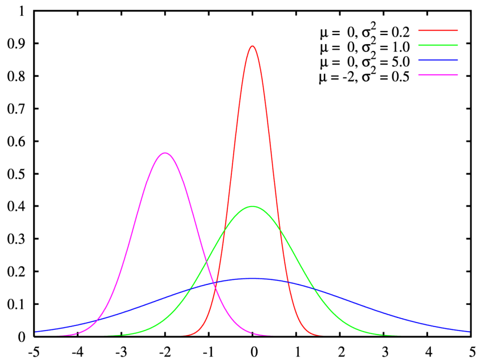
$$\begin{cases} X_1 = \sqrt{-2\log(U_1)}\sin(2\pi U_2) \\ X_2 = \sqrt{-2\log(U_1)}\cos(2\pi U_2) \end{cases}$$

- Are independent Standard Normal random Variables, i.e.:
  - Follow the PDF: $N(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \qquad x \in \Re$

- In order to change the centre and the amplitude of the Normal distribution, i.e.: $N(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-x_0)^2}{2\sigma^2}} \qquad x \in \Re$

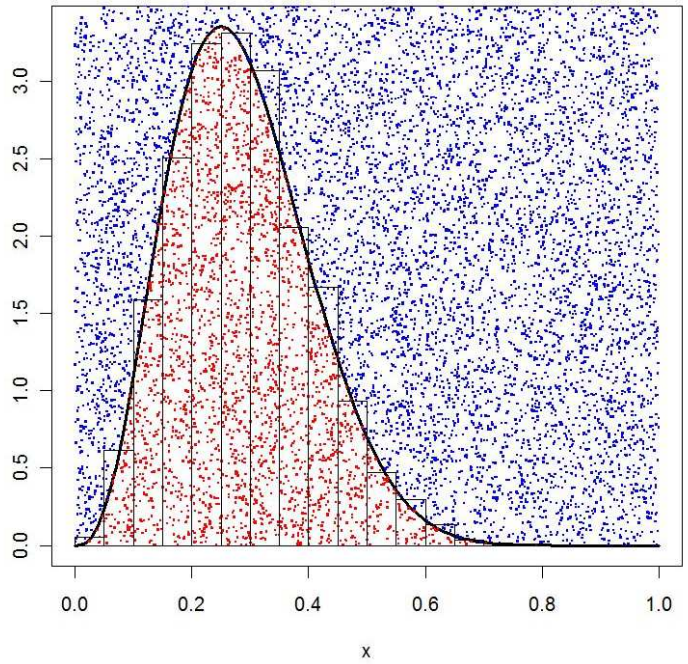
- You can apply a corollary of the inversion method:

- $$\begin{cases} Y_1 = x_0 + \sigma X_1 & Y_1 \in N(x_0, \sigma) \\ Y_2 = x_0 + \sigma X_2 & Y_2 \in N(x_0, \sigma) \end{cases}$$



Rejection Methods

- Suppose it is required to generate a random variable froma a distribution with density f(x).
- Let g(y) be another variable defined in the support of f that $f(x) \le c \cdot g(x)$, where $c > 1$is a known constant, hold for all x in the support.
- REJECTION ALGHOIRITHM:
  - Repeat
    - Generate y from g(y)
    - Generate u from U(0,1)
  - Until
    - $U \le f(y)/[c \cdot g(y)]$
  - Return X = y
- Always feasible
- Computational intensive



Part4:
Default Random Generators

FORTRAN Random Generators - 1

- FORTRAN
  - RAND (GNU extension):
    - RAND(FLAG) returns a pseudo-random number from a uniform distribution between 0 and 1. If FLAG is 0, the next number in the current sequence is returned; if FLAG is 1, the generator is restarted by CALL SRAND(0); if FLAG has any other value, it is used as a new seed with SRAND.
    - Syntax:

      ```
      RESULT = RAND(I)
      ```

    - Example:

      ```
      program test_rand
         integer,parameter :: seed = 86456
         call srand(seed)
         print *, rand(), rand(), rand(), rand()
         print *, rand(seed), rand(), rand(), rand()
      end program test_rand
      ```

FORTRAN Random Generators - 2

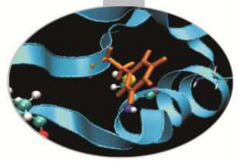
- FORTRAN
  - IRAND (GNU extension):
    - IRAND(FLAG) returns a pseudo-random number from a uniform distribution between 0 and a system-dependent limit (which is in most cases 2147483647). If FLAG is 0, the next number in the current sequence is returned; if FLAG is 1, the generator is restarted by CALL SRAND(0); if FLAG has any other value, it is used as a new seed with SRAND.
    - Syntax:
      - RESULT = IRAND(I)
    - Example:

```
program test_irand
        integer,parameter :: seed = 86456
        call srand(seed)
        print *, irand(), irand(), irand(), irand()
        print *, irand(seed), irand(), irand(), irand()
end program test_irand
```

  - SRAND (GNU extension):
    - reinitializes the pseudo-random number generator called by RAND and IRAND. The new seed used by the generator is specified by the required argument SEED.
    - Syntax:

      CALL SRAND(SEED)
    - See RAND and IRAND for Examples
    - NOTE: The Fortran 2003 standard specifies the intrinsic RANDOM_SEED to initialize the pseudo-random numbers generator and RANDOM_NUMBER to generate pseudo-random numbers. Please note that in GNU Fortran, these two sets of intrinsics (RAND, IRAND and SRAND on the one hand, RANDOM_NUMBER and RANDOM_SEED on the other hand) access two independent pseudo-random number generators.

FORTRAN Random Generators - 3



- FORTRAN
  - RANDOM_NUMBER (F95 standard)
    - Returns a single pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range 0 \leq x < 1. The runtime-library implements George Marsaglia's KISS (Keep It Simple Stupid) random number generator (RNG). This RNG combines:
      - The congruential generator x(n) = 69069 \cdot x(n-1) + 1327217885 with a period of 2^32,
      - A 3-shift shift-register generator with a period of 2^32 - 1,
      - Two 16-bit multiply-with-carry generators with a period of 597273182964842497 > 2^59.
    - The overall period exceeds 2^123.
    - Please note, this RNG is thread safe if used within OpenMP directives, i.e., its state will be consistent while called from multiple threads. However, the KISS generator does not create random numbers in parallel from multiple sources, but in sequence from a single source. If an OpenMP-enabled application heavily relies on random numbers, one should consider employing a dedicated parallel random number generator instead.
    - Syntax:

      ```
      RANDOM_NUMBER(HARVEST)
      ```
    - Arguments:
      - *HARVEST* : Shall be a scalar or an array of type REAL.
    - Example:

      ```
      program test_random_number
          REAL :: r(5,5)
          CALL init_random_seed()          ! see example of RANDOM_SEED
          CALL RANDOM_NUMBER(r)
      end program
      ```

FORTRAN Random Generators - 4

- RANDOM_SEED (F95 standard)
  - Restarts or queries the state of the pseudorandom number generator used by RANDOM_NUMBER. If RANDOM_SEED is called without arguments, it is initialized to a default state. The example below shows how to initialize the random seed with a varying seed in order to ensure a different random number sequence for each invocation of the program. Note that setting any of the seed values to zero should be avoided as it can result in poor quality random numbers being generated.
  - Syntax:
    - CALL RANDOM_SEED([SIZE, PUT, GET])
  - Arguments:
    - *SIZE* (Optional): Shall be a scalar and of type default INTEGER, with INTENT(OUT). It specifies the minimum size of the arrays used with the *PUT* and *GET* arguments.
    - *PUT* (Optional): Shall be an array of type default INTEGER and rank one. It is INTENT(IN) and the size of the array must be larger than or equal to the number returned by the *SIZE* argument.
    - *GET* (Optional): Shall be an array of type default INTEGER and rank one. It is INTENT(OUT) and the size of the array must be larger than or equal to the number returned by the *SIZE* argument.
  - Example:
    - See: https://gcc.gnu.org/onlinedocs/gfortran/RANDOM_005fSEED.html#RANDOM_005fSEED

C Random Generators: ISO C Random

- This section describes the random number functions that are part of the ISO C standard.
- To use these facilities, you should include the header file stdlib.h in your program.
  - Macro: int RAND_MAX :
    - The value of this macro is an integer constant representing the largest value the rand function can return.
  - Function: **int rand (void)** :
    - The rand function returns the next pseudo-random number in the series. The value ranges from 0 to RAND_MAX.
  - Function: **void srand (unsigned int seed)** :
    - This function establishes seed as the seed for a new series of pseudo-random numbers.
    - If you call rand before a seed has been established with srand, it uses the value 1 as a default seed.
    - To produce a different pseudo-random series each time your program is run, do srand (time (0)).
  - Function: **int rand_r (unsigned int *seed)** :
    - This function returns a random number in the range 0 to RAND_MAX just as rand does.
    - However, all its state is stored in the seed argument.
    - This means the RNG's state can only have as many bits as the type unsigned int has. This is far too few to provide a good RNG.

Random Number on ESSL

- Random number generation subroutines generate uniformly distributed random numbers or normally distributed random numbers using one of the following algorithms:
  - SIMD-oriented Mersenne Twister algorithm
  - Multiplicative congruential methods
  - Polar methods
  - Tausworthe exclusive-or algorithm

Subrouitine	Descriptive Name and Location
INITRNG	INITRNG (Initialize Random Number Generators)

Short-Precision Subroutine	Long-Precision Subroutine	Descriptive Name and Location
SURNG	DURNG	SURNG and DURNG (Generate a Vector of Uniformly Distributed Pseudo-Random Numbers)
SNRNG	DNRNG	SNRNG and DNRNG (Generate a Vector of Normally Distributed Pseudo-Random numbers)
SURAND	DURAND	SURAND and DURAND (Generate a Vector of Uniformly Distributed Random Numbers)
SNRAND	DNRAND	SNRAND and DNRAND (Generate a Vector of Normally Distributed Random Numbers)
SURXOR*	DURXOR*	SURXOR and DURXOR (Generate a Vector of Long Period Uniformly Distributed Random Numbers)

*This subroutine is provided for migration from earlier releases of ESSL and is not intended for use in new programs

Random Number on ESSL: Subroutine 1

- INITRNG
  - FORTRAN:
    - CALL INITRNG (iopt, irepeat, iseed, liseed, istate, listate)
  - C or C++:
    - initrng (iopt, irepeat, iseed, liseed, istate, listate);
  - Arguments:
    - **Iopt:**
      - Admitted value:1 or 2.
      - Indicates the random number generator desired for use, where:
        - If iopt = 1, a single-precision, SIMD-oriented Mersenne Twister pseudo-random number generator with a period of $2^{19937}-1$ is used.
        - If iopt = 2, a long-precision, SIMD-oriented Mersenne Twister pseudo-random number generator with a period of $2^{19937}-1$ is used.
    - **Irepeat:**
      - Admitted value: 0 or 1
      - Indicates whether repeatable or non-repeatable pseudo-random number sequences will be generated, where:
        - If irepeat = 0, the pseudo-random number generator uses values from iseed to generate repeatable pseudo-random number sequences.
        - If irepeat = 1, the pseudo-random number generator uses hardware-generated values to generate non-repeatable pseudo-random number sequences.
    - **Iseed:**
      - If irepeat = 0, iseed is an array containing the initial seed values to use in initializing the pseudo-random number generator to generate repeatable pseudo-random number sequences.
      - If irepeat = 1, iseed is ignored.
      - Specified as: a one-dimensional integer array of (at least) length max(1,liseed).
    - **liseed:**
      - Is the number of elements in array ISEED, where:
        - If irepeat = 0, liseed is determined as follows:
          - 32-bit integer environment: liseed ≥ 624
          - 64-bit integer environment: liseed ≥ 312
        - If irepeat = 1, liseed is ignored.
    - **istate:**
      - Is an array containing information about the current state of the pseudo-random number generator.
    - **listate:**
      - If listate ≠ -1, listate is the number of elements in the array istate
  - For further information, please see: http://www-01.ibm.com/support/knowledgecenter/SSFHY8_5.4.0/com.ibm.cluster.essl.v5r4.essl100.doc/am5gr_initrng.htm%23am5gr_initrng?lang=it

Random Number on ESSL: Subroutine 2

- SURNG (short precision) and DURNG (double precision)
  - These subroutines generate a repeatable or non-repeatable vector x of uniform pseudo-random numbers uniformly distributed over the interval [a, b].
  - For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG.
  - FORTRAN:
    - CALL SURNG (n, a, b, x, istate, listate) or CALL DURNG (n, a, b, x, istate, listate)
  - C or C++:
    - surng | durng (n, a, b, x, istate, listate); or surng | durng (n, a, b, x, istate, listate);
  - Arguments:
    - n:
      - Is the number of pseudo-random numbers to be generated. Specified as: an integer; n ≥ 0.
    - a:
      - Is the mean value of the distribution.
    - b:
      - Is the standard deviation value of the distribution.
    - x:
      - Is a vector of length n, containing the uniformly distributed pseudo-random numbers. Returned as: a one-dimensional array of (at least) length n
    - istate:
      - Is an array containing information about the current state of the pseudo-random number generator.
    - listate:
      - is the number of elements in the array istate and depends on both the environment the subroutine is running in and the value of iopt specified on the previous call to INITRNG
  - For further information, please see: http://www-01.ibm.com/support/knowledgecenter/SSFHY8_5.4.0/com.ibm.cluster.essl.v5r4.essl100.doc/am5gr_sdurng.htm%23am5gr_sdurng?lang=it

Random Number on ESSL: Subroutine 3

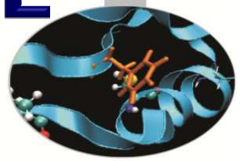
- SNRNG (short precision) and DNRNG (double precision)
  - These subroutines generate a repeatable or non-repeatable vector x of normally distributed pseudo-random numbers normally distributed with a mean of rmean and a standard deviation of sigma, using the BoxMuller2 method.
  - For the initial call to these subroutines, you must initialize the pseudo-random number generator with a preceding call to INITRNG
  - FORTRAN:
    - CALL SNRNG(n, rmean, sigma, x, istate, listate) or CALL DNRNG (n, rmean, sigma, x, istate, listate)
  - C or C++:
    - SNRNG(n, rmean, sigma, x, istate, listate); or SNRNG(n, rmean, sigma, x, istate, listate);
  - Arguments:
    - **n:**
      - Is the number of pseudo-random numbers to be generated. Specified as: an integer; n ≥ 0.
    - **rmean :**
      - Is the left boundary of the interval [a, b].
    - **sigma :**
      - Is the right boundary of the interval [a, b].
    - **x:**
      - s a vector of length n, containing the normally distributed pseudo-random numbers. Returned as: a one-dimensional array of (at least) length n
    - **istate:**
      - Is an array containing information about the current state of the pseudo-random number generator.
    - **listate:**
      - is the number of elements in the array istate and depends on both the environment the subroutine is running in and the value of iopt specified on the previous call to INITRNG
  - For further information, please see: http://www-01.ibm.com/support/knowledgecenter/SSFHY8_5.4.0/com.ibm.cluster.essl.v5r4.essl100.doc/am5gr_sdnrng.htm%23am5gr_sdnrng?lang=it

Random Numbers on MKL

- Intel MKL VS routines are used to generate random numbers with different types of distribution.
- Continuous Distribution Generators:

Type of Distribution	Data Types	BRNG DataType	Description
vRngUniform	s, d	s, d	Uniform continuous distribution on the interval [*a*,*b*)
vRngGaussian	s, d	s, d	Normal (Gaussian) distribution
vRngGaussianMV	s, d	s, d	Multivariate normal (Gaussian) distribution
vRngExponential	s, d	s, d	Exponential distribution
vRngLaplace	s, d	s, d	Laplace distribution (double exponential distribution)
vRngWeibull	s, d	s, d	Weibull distribution
vRngCauchy	s, d	s, d	Cauchy distribution
vRngRayleigh	s, d	s, d	Rayleigh distribution
vRngLognormal	s, d	s, d	Lognormal distribution
vRngGumbel	s, d	s, d	Gumbel (extreme value) distribution
vRngGamma	s, d	s, d	Gamma distribution
vRngBeta	s, d	s, d	Beta distribution

Random Numbers on MKL



- Intel MKL VS routines are used to generate random numbers with different types of distribution.
- Discrete Distribution Generators:

Type of Distribution	Data Types	BRNG DataType	Description
vRngUniform	i	d	Uniform discrete distribution on the interval [*a*,*b*)
vRngUniformBits	i	i	Underlying BRNG integer recurrence
vRngUniformBits32	i	i	Uniformly distributed bits in 32-bit chunksUniformly distributed bits in 64-bit chunks
vRngUniformBits64	i	i	Uniformly distributed bits in 64-bit chunks
vRngBernoulli	i	s	Bernoulli distribution
vRngGeometric	i	s	Geometric distribution
vRngBinomial	i	d	Binomial distribution
vRngHypergeometric	i	d	Hypergeometric distribution
vRngPoisson	i	s (for VSL_RNG_METHOD_POISSON_POISNORM) s (for distribution parameter $\lambda \geq 27$) and d (for $\lambda < 27$) (for VSL_RNG_METHOD_POISSON_PTPE)	Poisson distribution
vRngPoissonV	i	s	Poisson distribution with varying mean
vRngNegBinomial	i	q	Negative binomial distribution, or Pascal distribution

The Scalable Parallel Random Number Generators Library (SPRNG)

- Where to Get SPRNG
  - The main web site for SPRNG is located at URLs: http://sprng.cs.fsu.edu or http://www.sprng.org
- Many versions available.
- Latest version 4.0 which is C++
- The 4.0 page gives info pages to 4.0 page info:
  - Quick Start
  - Quick Reference
  - User's Guide
  - Reference Manual
  - Examples

How to Build SPRNG

- How to Build SPRNG:
  - zcat sprng4.tar.gz | tar xovf -
  - cd sprng4
  - Run
  - ./configure
  - Run make
  - NB: Sometimes 'make' has errors on some parts which can be ignored. In these cases, 'make -k' can be used to continue compiling even if there are errors.
  - The MPI programs sometimes need special configuring.

Testing SPRNG

- How to check the build:
  - Go to directory check, and run ./checksprng
  - This program checks to see if SPRNG has been correctly installed.
  - The check folder contains a single program which generates known sequences and checks this against a data file

SuperComputing Applications and Innovation

How SPRNG is Structured

- How SPRNG is Structured:
  - Directories in SPRNG
  - SRC – Source code for SPRNG
  - EXAMPLES – Examples of SPRNG usage. All MPI examples are placed in subdirectory mpisprng. If MPI is installed on your machine, then all MPI examples will be automatically installed.
  - TESTS – Empirical and physical tests for SPRNG generators. All MPI tests are stored in subdirectory mpitests. If MPI is installed on your machine, then all MPI tests will be automatically installed.
  - Check – contains executables ./checksprng and ./timesprng.
  - Lib – contains SPRNG library libsprng after sucessful installation.
  - include – SPRNG header files.

Predefined Generators

- Types of generators:
  - 0: Modified Lagged-Fibonacci Generator (lfg)
  - 1: 48-Bit Linear Congruential Generator w/Prime Addend (lcg)
  - 2: 64-Bit Linear Congruential Generator w/Prime Addend (lcg64)
  - 3: Combined Multiple Recursive Generator (cmrg)
  - 4: Multiplicative Lagged-Fibonacci Generator (mlfg)
  - 5: Prime Modulus Linear Congruential Generator (pmlcg)
- The number represents the type of generator in the Class interface

Specific Generator Details

- lfg: Modified-Lagged Fibonacci Generator (the default generator)
  - $z_n = x_n \text{ XOR } y_n$
  - $x_n = x_{n-k} + x_{n-l} (\text{mod } M)$
  - $y_n = y_{n-k} + y_{n-l} (\text{mod } M)$
- lcg: 48-Bit Linear Congruential Generator w/Prime Addend
  - $x_n = a x_{n-1} + p (\text{mod } M)$
  - p is a prime addend
  - a is the multiplier
  - M for this generator is 2^{48}
- lcg64: 64-Bit Linear Congruential Generator w/Prime Addend
  - The 48-bit LCG, except that the arithmetic is modulo 2^{64}
  - The multipliers and prime addends for this generator are different from those for the 48-bit generator

Specific Generator Details

- cmrg: Combined Multiple Recursive Generator
  - $z_n = x_n + y_n*2^{32} (\mathrm{mod}\ 2^{64})$
  - x_n is the sequence generated by the 64 bit Linear Congruential Generator
  - y_n s the sequence generated by the following prime modulus Multiple Recursive Generator
- mlfg: Multiplicative Lagged-Fibonacci Generator
  - $x_n = x_{n-k} * x_{n-l} (\mathrm{mod}\ M)$
  - l and k are called the lags of the generator, with convention that $l > k$
  - M is chosen to be 2^{64}
- pmlcg: Prime Modulus Linear Congruential Generator
  - $x_n = a*x_{n-1} (\mathrm{mod}\ 2^{61}-1)$

Default Interface

- Default Interface:
  - Sprng(int streamnum, int nstreams, int seed,int param) (Constructor)
  - double sprng() – The next random number in [0,1) is returned
  - int isprng() – The next random number in $[0,2^{31})$ is returned

Simple Interface

- Simple Interface:
  - int * init_sprng(int seed, int param, int rng_type = 0)
  - double sprng() – The next random number in [0;1) is returned
  - int isprng() – The next random number in $[0;2^{31})$ is returned

Random Number Parameter

- Random Number Parameters:
  - Parameter is the number of predefined families defined
    - Modified Lagged Fibonacci Generator – 11
    - 48 Bit Linear Congruential Generator – 7
    - 64 Bit Linear Congruential Generator – 3
    - Combined Multiple Recursive Generator – 3
    - Multiplicative Lagged Fibonacci Generator – 11
    - Prime Modulus Linear Congruential Generator – 1

Usage Example - Default Interface

```
#define PARAM SPRNG_LFG
int gtype = 1;
seed = make_sprng_seed();
Sprng *gen1;
gen1 = SelectType(gtype);
gen1->init_sprng(0,ngens,seed,PARAM);
int random_int = gen1->isprng();
double random_float = gen1->get_rn_flt_simple();
gen1->free_sprng();
```

Usage Example - Simple Interface

```
#define PARAM SPRNG_LFG
int gtype = 1;
seed = make_sprng_seed();
gen = init_sprng(seed, PARAM, gtype);
int random_int = isprng();
double random_float = get_rn_flt_simple();
```

Other Parts - Examples Folder

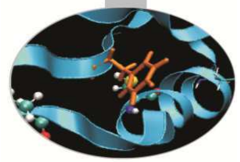
- Examples Folder Examples Folder
  - convert.cpp – Used to be an example of converting old code to new, but mostly empty
  - pi-simple.cpp – Compute pi using Monte Carlo integration
  - spawn.cpp – Small sample program to get you started
  - Fortran versions as well

Other Parts - Tests Folder

- Tests Folder:
  - Statistical Tests
    - chisquare.cpp – Chi-Square and Kolmogorov-Smirnov Probability Functions
    - collisions.cpp – Collision test
    - coupon.cpp – Coupon test
    - equidist.cpp – Equidistribution test
  - Other Tests
    - fft.cpp – FFT test
    - metropolis.cpp – Metropolis Algorithm
    - random_walk.cpp – Random Walk Algorithm

CINECA SCAI
SuperComputing Applications and Innovation

References



- M. Mascagni and H. Chi (2004)]
  Parallel Linear Congruential Generators with Sophie-Germain Moduli,
  Parallel Computing,30: 1217-1231.
- [M. Mascagni and A. Srinivasan (2004)]
  Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators,
  Parallel Computing,30: 899-916.
- [M. Mascagni and A. Srinivasan (2000)]
  Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation,
  ACM Transactions on Mathematical Software,26: 436-46

CINECA

Thank you for your attention