A package of Linux scripts for the parallelization of Monte Carlo simulations

Andreu Badal, Josep Sempau

Institut de Tècniques Energètiques, Universitat Politècnica de Catalunya, Diagonal 647, 08028 Barcelona, Spain

Received 21 March 2006; received in revised form 22 May 2006; accepted 28 May 2006

Available online 26 July 2006

Abstract

Despite the fact that fast computers are nowadays available at low cost, there are many situations where obtaining a reasonably low statistical uncertainty in a Monte Carlo (MC) simulation involves a prohibitively large amount of time. This limitation can be overcome by having recourse to parallel computing. Most tools designed to facilitate this approach require modification of the source code and the installation of additional software, which may be inconvenient for some users. We present a set of tools, named clonEasy, that implement a parallelization scheme of a MC simulation that is free from these drawbacks. In clonEasy, which is designed to run under Linux, a set of “clone” CPUs is governed by a “master” computer by taking advantage of the capabilities of the Secure Shell (ssh) protocol. Any Linux computer on the Internet that can be ssh-accessed by the user can be used as a clone. A key ingredient for the parallel calculation to be reliable is the availability of an independent string of random numbers for each CPU. Many generators—such as RANLUX, RANECU or the Mersenne Twister—can readily produce these strings by initializing them appropriately and, hence, they are suitable to be used with clonEasy. This work was primarily motivated by the need to find a straightforward way to parallelize PENELOPE, a code for MC simulation of radiation transport that (in its current 2005 version) employs the generator RANECU, which uses a combination of two multiplicative linear congruential generators (MLCGs). Thus, this paper is focused on this class of generators and, in particular, we briefly present an extension of RANECU that increases its period up to \( \sim 5 \times 10^{27} \) and we introduce seedsMLCG, a tool that provides the information necessary to initialize disjoint sequences of an MLCG to feed different CPUs. This program, in combination with clonEasy, allows to run PENELOPE in parallel easily, without requiring specific libraries or significant alterations of the sequential code.

Program summary 1

Title of program: clonEasy
Catalogue identifier: ADYD_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/ADYD_v1_0
Program obtainable from: CPC Program Library, Queen’s University of Belfast, Northern Ireland
Computer for which the program is designed and others in which it is operable: Any computer with a Unix style shell (bash), support for the Secure Shell protocol and a FORTRAN compiler
Operating systems under which the program has been tested: Linux (RedHat 8.0, SuSe 8.1, Debian Woody 3.1)
Compilers: GNU FORTRAN g77 (Linux); g95 (Linux); Intel Fortran Compiler 7.1 (Linux)
Programming language used: Linux shell (bash) script, FORTRAN 77
No. of bits in a word: 32
No. of lines in distributed program, including test data, etc.: 1916
No. of bytes in distributed program, including test data, etc.: 18 202
Distribution format: tar.gz

* This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.

E-mail address: andreu.badal@upc.edu (A. Badal).

0010-4655/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2006.05.009
Nature of the physical problem: There are many situations where a Monte Carlo simulation involves a huge amount of CPU time. The parallelization of such calculations is a simple way of obtaining a relatively low statistical uncertainty using a reasonable amount of time.

Method of solution: The presented collection of Linux scripts and auxiliary FORTRAN programs implement Secure Shell-based communication between a “master” computer and a set of “clones”. The aim of this communication is to execute a code that performs a Monte Carlo simulation on all the clones simultaneously. The code is unique, but each clone is fed with a different set of random seeds. Hence, clonEasy effectively permits the parallelization of the calculation.

Restrictions on the complexity of the program: clonEasy can only be used with programs that produce statistically independent results using the same code, but with a different sequence of random numbers. Users must choose the initialization values for the random number generator on each computer and combine the output from the different executions. A FORTRAN program to combine the final results is also provided.

Typical running time: The execution time of each script largely depends on the number of computers that are used, the actions that are to be performed and, to a lesser extent, on the network connexion bandwidth.

Unusual features of the program: Any computer on the Internet with a Secure Shell client/server program installed can be used as a node of a virtual computer cluster for parallel calculations with the sequential source code. The simplicity of the parallelization scheme makes the use of this package a straightforward task, which does not require installing any additional libraries.

Program summary 2

Title of program: seedsMLCG
Catalogue identifier: ADYE_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/ADYE_v1_0
Program obtainable from: CPC Program Library, Queen’s University of Belfast, Northern Ireland
Computer for which the program is designed and others in which it is operable: Any computer with a FORTRAN compiler
Operating systems under which the program has been tested: Linux (RedHat 8.0, SuSe 8.1, Debian Woody 3.1), MS Windows (2000, XP)
Compilers: GNU FORTRAN g77 (Linux and Windows); g95 (Linux); Intel Fortran Compiler 7.1 (Linux); Compaq Visual Fortran 6.1 (Windows)
Programming language used: FORTRAN 77
No. of bits in a word: 32
Memory required to execute with typical data: 500 kilobytes
No. of lines in distributed program, including test data, etc.: 492
No. of bytes in distributed program, including test data, etc.: 5582
Distribution format: tar.gz

Nature of the physical problem: Statistically independent results from different runs of a Monte Carlo code can be obtained using uncorrelated sequences of random numbers on each execution. Multiplicative linear congruential generators (MLCG), or other generators that are based on them such as RANECU, can be adapted to produce these sequences.

Method of solution: For a given MLCG, the presented program calculates initialization values that produce disjoint, consecutive sequences of pseudo-random numbers. The calculated values initiate the generator in distant positions of the random number cycle and can be used, for instance, on a parallel simulation. The values are found using the formula \( S_1 = (a^J S_0) \mod m \), which gives the random value that will be generated after \( J \) iterations of the MLCG.

Restrictions on the complexity of the program: The 32-bit length restriction for the integer variables in standard FORTRAN 77 limits the produced seeds to be separated a distance smaller than \( 2^{31} \), when the distance \( J \) is expressed as an integer value. The program allows the user to input the distance as a power of 10 for the purpose of efficiently splitting the sequence of generators with a very long period.

Typical running time: The execution time depends on the parameters of the used MLCG and the distance between the generated seeds. The generation of \( 10^8 \) seeds separated \( 10^{12} \) units in the sequential cycle, for one of the MLCGs found in the RANECU generator, takes 3 s on a 2.4 GHz Intel Pentium 4 using the g77 compiler.

© 2006 Elsevier B.V. All rights reserved.

PACS: 02.70.Uu

Keywords: Monte Carlo; Parallelization; Pseudo-random number generator; Multiplicative linear congruential generator; RANECU; PENEOPE

1. Introduction

Monte Carlo (MC) methods are used in a vast range of applications [6]. The reason for their success lies, in part, in the conceptual simplicity and in the relative easiness with which they can be coded on a computer. One important application of these methods is the description of radiation transport. In this case, particle tracks are generated according to the cross sections of the different atomic processes and, after simulating a sufficiently large number of independent random tracks, the mean value and variance of the quantities of interest (for instance, the energy deposited in a certain detector) are calculated using statistical procedures. In fact, the present paper has been motivated by our work on the simulation of electromagnetic showers using the PENEOPE code [20,21] and, although the tools to be introduced are not restricted to a particular code, the main goal is to allow PENEOPE users to run parallel simulations using a simple and reliable procedure.

A common feature of MC calculations is that the statistical variance of the result is, when everything else remains the same, inversely proportional to the simulation time. In spite of the fact that fast computers are available nowadays at low cost, there...
are still many situations where the amount of time needed to obtain a reasonably low statistical uncertainty with a MC simulation is prohibitively large. This limitation can be overcome by having recourse to parallel computing. Although the parallel execution cannot reduce the total CPU time needed by your program to finish the calculation, the use of more than one CPU at the same time may effectively reduce the real time spent in the simulation, that is, the time you have to wait to get your work done. Various strategies and tools have been developed to facilitate the implementation of this solution, e.g., MPI, openMP, PVM and openMOSIX. All these have been successfully used with MC, but they all have the drawback of requiring the modification of the sequential code and the installation of additional software, which may be inconvenient for some users.

In general, the parallelization of an algorithm is not a straightforward task and it may even be unfeasible. Fortunately, it can be achieved relatively easily in the case of an MC code for which each random history is sampled independently from the others and, therefore, the calculation can be divided into several batches that can be executed on different CPUs. The results yielded by independent runs of the same code can be combined a posteriori to obtain a single final result with a reduced statistical uncertainty. Some authors call this approach process replication (or domain replication) to distinguish it from a possible alternative, domain decomposition, which involves segmenting the space of possible states of the system, e.g., by splitting the geometry or the energy range into pieces and assigning each piece to a different CPU. The “natural” simplicity of the process replication scheme has led to the denomination “embarrassingly parallel” used by some authors to refer to MC algorithms (see e.g. [22]).

In order to sample the probability distributions that characterize the system to be simulated it is necessary to have a source of randomness, typically a string of numbers uniformly distributed between 0 and 1. True random numbers can be generated using physical devices such as radiation detectors or semiconductors, but they are too slow compared with computer speeds and their unpredictable nature makes it difficult to debug the programs or check the results. For these reasons most MC codes use pseudo-random number generators (PRNG), which provide a cyclic sequence of numbers produced by deterministic algorithms [8,10]. These algorithms must be carefully chosen so that the correlations between the generated numbers are not apparent. Thus, they are required to pass several statistical tests intended to check the uniformity and independence of the produced sequence [1,22].

Along these lines, two related tools are presented in this work. Firstly, we introduce a set of Linux scripts and FORTRAN programs, named clonEasy, that implement the process replication approach on a set of “clone” CPUs governed by a “master” computer. This software package readily permits parallel computations to be carried out without requiring additional software or significant modifications of the original, sequential, code. Secondly, a tool named seedsMLCG, which provides the information necessary to initialize disjoint sequences of any PRNG based on a multiplicative linear congruential algorithm, is presented. As a particular case, this auxiliary tool can be used to obtain disjoint sequences of RANECU, the PRNG used in PENELOPE.

The rest of the paper is organized as follows. In Section 2 we briefly describe the basic properties of some of the most popular PRNGs used in MC simulations and discuss their use in parallel calculations in order to show how they can be employed in combination with clonEasy. We focus our attention on RANECU (including an extension to enlarge its period) and on the method employed to obtain disjoint sequences with it. The tools clonEasy and seedsMLCG are introduced in Section 3. Finally, some conclusions are drawn in Section 4.

2. Pseudo-random number generators and parallel simulations

The best PRNGs are those that perform as well as the true random number generators in the randomness tests and have a solid mathematical basis that justifies their essential properties and cycle length. A theoretical proof of their quality is an important point since biased results due to subtle correlations have been reported in the past (see e.g. [2]).

Among the most well studied and extensively used algorithms for PRNGs there are those based on recursions with modular arithmetic, which take a few input integer numbers, called seeds, and produce a cyclic sequence of integers that can be transformed into real values uniformly distributed between 0 and 1. A family of algorithms that belong to this class is the so-called multiplicative linear congruential generators (MLCGs). Although more recent generators are known to perform better in some aspects and have therefore superseded MLCGs for most applications, it can be argued that the latter are conceptually simpler and their weaknesses well understood. In fact, PRNGs based on combinations of MLCGs (such as RANECU, see below) have been extensively used in the past and are still in use by some computer codes. PENELOPE, for instance, has been amply benchmarked against experimental results and other MC codes since its first release in 1996, and no bias that can be attributed to its PRNG has been detected to date.

An MLCG is initialized with a single seed, an integer value \( S_0 \). It produces each term of the sequence \((S_i, i = 0, 1, \ldots)\) by multiplying the previous value by an integer \(a\) and calculating the modulo \(m\), i.e., computing the remainder of the integer division by \(m\). A real value \(u_i\) in the interval \([0, 1)\) is obtained by dividing \(S_i\) by \(m\). The resulting sequence can thus be expressed by

\[
S_{i+1} = (aS_i) \mod m, \quad u_i = \frac{S_i}{m}. \quad (1)
\]

The possible remainders are all the integers from 0 to \(m - 1\). A null remainder, however, should be avoided to prevent the generator from collapsing into a sequence of zeroes. The largest possible period is therefore \(m - 1\), or \(\sim 2 \times 10^9\) if \(m\) is restricted to be representable by 32-bit-long signed integers, which is clearly insufficient for present-day applications. This drawback can be overcome by combining several MLCGs, as explained later.

A known weakness common to all MLCGs is, as pointed out by Marsaglia [13], that if groups of \(n\) successive random values
are used as the Cartesian coordinates of points in an n-dimensional space, they do not uniformly fill up the volume. Instead, they lie on a relatively small number of parallel hyperplanes producing a lattice structure. The maximal distance between adjacent hyperplanes is a convenient measure of the quality of the generator and its determination is the goal of the so-called spectral test [8]. When the distance between hyperplanes is small the illusion that points are uniformly distributed in the hypercube is reinforced. This criterion is thus frequently employed to find the best multiplier and modulus for an MLCG.

Another widely known PRNG based on a recursion with modular arithmetic is RCARRY, which implements an extension of the lagged Fibonacci algorithm called subtract-and-borrow [14]. Its initialization requires 24 seeds ($S_i$, $i = 0, \ldots, 23$) and a carry bit $c_{23}$ (equal to either 0 or 1) and generates the elements of the sequence as

$$S_i = (S_{i-10} - S_{i-24} - c_{i-1}) \mod 2^{24}, \quad i > 23,$$  \hspace{1cm} (2)

where $c_{i-1}$ ($i > 24$) is 0 if ($S_{i-11} - S_{i-25} - c_{i-2} > 0$ and it is 1 otherwise. This generator has a long period ($\sim 5 \times 10^{171}$) but it fails several tests [23] and, therefore, its use may compromise the reliability of the simulation results. By studying RCARRY from the viewpoint of the dynamics of chaotic systems Lüscher [12] showed that the detected correlations are short-ranged and that they can be eliminated by simply discarding ($p - 24$) elements of every $p$ consecutive elements of the sequence, where $p$ is a fixed user-defined parameter that determines the quality (i.e., randomness) of the resulting generator. It can be argued that the corresponding computer code, named RANLUX [5], has therefore been proven to produce random sequences of the highest quality, but the price to be paid is a low efficiency, that is, the quantity of pseudo-random numbers produced per unit time is relatively small compared with other algorithms.

Another heavily used generator is based on the algorithm called Mersenne Twister (MT) proposed by Matsumoto and Nishimura [16]. It implements a version of the linear feedback shift register [8], which involves binary operations with the seed bits. The MT is becoming increasingly popular due to the fact that it is considerably faster than RANLUX and RANECU and that it has passed the most relevant randomness tests available. However, although it has been demonstrated that the generated values are equidistributed in 623 dimensions and that the cycle period is of the order of $10^{6001}$, the theoretical basis for its random properties is not as established as it is for RANLUX and its possible weaknesses are not well understood.

The string of random numbers employed by each processor participating in a parallel MC simulation should be uncorrelated with those used by the other processors in order to guarantee the statistical independence of the different partial results. The ability to produce these sequences is thus an important feature of a PRNG. While some algorithms can generate long independent strings in a simple way (RANLUX produces one of these strings for each integer value given to its initialization routine, see [5]) more elaborated techniques are required in other cases (for the MT they can be produced using a slightly modified version of the generator in each node, as proposed in [17]).

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Parameters of the MLCGs that are used in RANECU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modulus ($m$)</td>
<td>Multiplier ($a$)</td>
</tr>
<tr>
<td>1st generator</td>
<td>2 147 483 563</td>
</tr>
<tr>
<td>2nd generator</td>
<td>2 147 483 399</td>
</tr>
</tbody>
</table>

RANECU, developed by L’Ecuyer in 1988 [9], combines the sequences $S^{(1)}_i$ and $S^{(2)}_i$ ($i = 0, 1, \ldots$) from a pair of MLCGs with moduli $m^{(1)}$ and $m^{(2)}$ and multipliers $a^{(1)}$ and $a^{(2)}$, respectively, to produce a new sequence $S_i$ defined by

$$S_i = (S^{(1)}_i - S^{(2)}_i) \mod (m^{(1)} - 1).$$  \hspace{1cm} (3)

The parameters of these two MLCGs, chosen so as to yield optimal results in the spectral test, are shown in Table 1. The period of the combination in Eq. (3) is the least common multiple of the periods of $S^{(1)}_i$ and $S^{(2)}_i$ and its lattice structure is considerably better than that of its individual components (see below). This algorithm was coded in FORTRAN 77 by James [4] for computers that use registers with a minimum of 32 bits. The FORTRAN code included in PENELOPE (version 2005), which differs from the one proposed by James mainly in that it returns a single value at each call instead of an array of values, is displayed in Fig. 1. The reliability of RANECU stems from the well known mathematical basis of congruential generators and the fact that it has successfully passed a number of statistical tests [1,3,8,9]. Moreover, it has been satisfactorily used for many years in a number of MC codes without showing any apparent artefact. A version of RANECU (called RAN2) that incorporates an additional shuffling algorithm is supplied with the book Numerical Recipes [19].

RANECU’s two MLCGs fulfil the following conditions:

1. $m$ is a large prime number and $a$ is a primitive root modulo $m$.
2. $a^2 < m$.
3. $(m^{(1)} - 1)/2$ and $(m^{(2)} - 1)/2$ are relatively prime.

Condition 1 is equivalent to requiring that the MLCG attains its maximal period $m - 1$, i.e. that all the integer values between (and including) 1 and $m - 1$ are produced once before repeating the initial seed. The second condition permits an MLCG to be coded in an efficient and portable way with integers of bit length $b$ such that $m < 2^{2b-1}$ by having recourse to the so-called approximate factoring (AF) method [11]. This method is used to compute the product of the two integers $S_i$ and $a$ modulo $m$ without overflow hazard (i.e. without exceeding the length of a signed long integer) by taking advantage of the identity

$$(aS_i) \mod m = [a(S_i \mod q) - \lfloor S_i/q \rfloor r] \mod m,$$  \hspace{1cm} (4)
where \( q = [m/a] \) and \( r = m \mod a \) are the quotient and remainder, respectively, of the integer division of \( m \) by \( a \). Since RANECU uses moduli slightly smaller than \( 2^{31} \), its MLCGs can be coded using integer arithmetic in 32 or more bits. Finally, the third condition ensures that the combination of the two MLCGs produces a generator which also attains its maximal possible period, \( (m(1) - 1)(m(2) - 1)/2 \), which is the least common multiple of the individual periods \( m(1) - 1 \) and \( m(2) - 1 \). For the moduli of Table 1 this yields a total period of \( 2^{305} \times 10^{18} \).

It is a well-known fact that some generators have long-range correlations and, hence, it is not advisable to use a large fraction of the sequence in a single simulation. This, compounded with the increasing computing power available per monetary unit and the widespread use of computer clusters, is bound to render the increasing computing power available per monetary unit of the sequence in a single simulation. This, compounded with correlations and, hence, it is not advisable to use a large fraction of the sequence in a single simulation. Particularly, the MLCG with multiplier \( a(3) = 45742 \) and modulus \( m(3) = 2147482739 \), also studied by L’Ecuyer [9], can be used to produce an extension of RANECU with a sequence defined by

\[
S_t = (S_{t-1} - S_{t-2} + S_{t-3}) \mod (m(1) - 1). \tag{5}
\]

This “extended” RANECU uses three initial seeds and has a period of \( \sim 5 \times 10^{27} \).

The structural properties of combined MLCGs are much better than those of single MLCGs. To study the lattice structure of the generators, a 2D plot of points with Cartesian coordinates consisting of two consecutive random values in the interval \((0,1)\) can be employed. It can be calculated that for the two MLCGs in Table 1 and the third one proposed above as an extension, the number of parallel hyperplanes (straight lines in this case) that contain all these points is \( 65535 \) at most [13]. In contradiction, RANECU requires of the order of \( 10^9 \) lines. As shown in Fig. 2, the lattice structure of the MLCGs is apparent, whereas the points from the extended RANECU (and similarly for the original version) seem to be truly randomly distributed due to the much larger number of hyperplanes present.

2.2. Parallel execution with an MLCG

Statistically independent sequences of pseudo-random numbers for parallel executions can be obtained from an MLCG [11, 18] by applying the methodology described by L’Ecuyer [9]. As he states, an important property of MLCGs is that any term in the cycle can be obtained without calculating the intermediate values. Indeed, given an initial seed \( S_0 \), the term \( S_t \) can be found directly using (cf. Eq. (1))

\[
S_t = (a^t S_0) \mod m = [(a^t \mod m) S_0] \mod m. \tag{6}
\]

In order to compute \( a^t \mod m \) the right-to-left binary method for exponentiation described by Knuth [8, p. 462] adapted to

FUNCTION RAND(DUMMY)
C This is an adapted version of subroutine RANECU written by F. James
C (Comput. Phys. Commun. 60 (1990) 329-344), which has been modified to
give a single random number at each call.
C
C The ‘seeds’ ISEED1 and ISEED2 must be initialized in the main program
c and transferred through the named common block /RSEED/.
C
C Some compilers incorporate an intrinsic random number generator with
C the same name (but with different argument lists). To avoid conflict,
C it is advisable to declare RAND as an external function in all sub-
C programs that call it.
C
IMPLICIT DOUBLE PRECISION (A-H,O-Z), INTEGER*4 (I-N)
PARAMETER (USCALE=1.0D0/2.147483563D9)
COMMON/RSEED/ISEED1,ISEED2
C
I1=ISEED1/53668
ISEED1=40014*(ISEED1-I1*53668)-I1*12211
IF(ISEED1.LT.0) ISEED1=ISEED1+2147483563
C
I2=ISEED2/52774
ISEED2=40692*(ISEED2-I2*52774)-I2*3791
IF(ISEED2.LT.0) ISEED2=ISEED2+2147483399
C
I2=ISEED1-ISEED2
IF(I2.LT.1) I2=I2+2147483562
RAND=I2*USCALE
END
RETURN
END
The value of $a$ is that a high power of modular arithmetic is used. The basic idea behind this method is that a high power of $a$ can be obtained by successively squaring $a$. Thus, for instance, since 6 is written as 110 in binary, $a^6$ can be expressed (by reading the binary from right to left) as $(a^2)^2 \times a^2$, which involves evaluating only three products. Notice that the AF method described above (see Eq. (4)) may not be applicable here to compute $a^6$ (or any other power) because some intermediate factors, $a^2$ and $(a^2)^2$ in our example, may not fulfill the condition that their squares are less than $m$, as required by condition 2 in the previous section. To overcome this difficulty and to calculate the product without overflow, we have recourse to the algorithm proposed by L’Ecuyer and this difficulty and to calculate the product without overflow, required by condition 2 in the previous section. To overcome the aforesaid condition 2, at which point the AF method is applied, we double the other until the halved factor complies with the integer division of 2 and open circles to the third MLCG included in the extended generator (see text).

By iterating the former expression, a division with null remainder is eventually found, yielding the GCD as the last non-null remainder. The extended algorithm uses the quotients and remainders of the intermediate iterations to find integers $\tilde{m}$ and $\tilde{a}$ such that

$$a\tilde{a} + m\tilde{m} = \text{GCD}(m, a)$$

by recursively expressing each residue as a linear combination of the original $a$ and $m$ values. When $m$ and $a$ are coprime, as is the case with the MLCGs considered here, we find

$$a\tilde{a} + m\tilde{m} = 1,$$

known as Bézout’s identity—despite some authors attribute it to Bachet de Méziriac. Now, since $m \text{ MOD } m = 0$, it follows that $\tilde{a} \text{ MOD } m$ is the sought inverse. We have found that the extended Euclid’s algorithm takes less computer time than evaluating $a^{(m-2)}$ and, hence, the former is the method of choice in our routines.

With all these ingredients, each CPU in a parallel calculation can be fed with a seed that initiates a string of pseudo-random numbers that does not overlap other CPU sequences. Indeed, if $J$ is an integer larger than the number of calls to the PRNG performed by any of the CPUs during the simulation, then the $k$th CPU ($k = 0, 1, \ldots, K - 1$) is provided with the sequence splitting and has the advantage of using the original generator without alterations. A drawback of this approach are the long-range correlations of the MLCG, which may become particularly relevant if $J$ is related to $m$, for instance if both are powers of 2.

A different procedure to obtain disjoint sequences is the jumping or leapfrog technique. This method consists of jumping a fixed distance $K$ along the generator cycle before the next seed is obtained. The resulting subsequence for the $k$th CPU is thus $\{S_{jK+i} \mid j = 0, 1, \ldots, J - 1\}$. Since, from Eq. (6), $S_{(Kj)} = (a^K)^{i/S_0} \text{ MOD } m$, an MLCG can be modified to leapfrog the sequence by merely replacing the multiplier $a$ by $a^K$ or, equivalently, by $a^K \text{ MOD } m$. Nonetheless, the generator with the new multiplier may be cumbersome to code in a portable way because it may not comply with condition 2 mentioned above. Furthermore, it is not clear whether the resulting sequence of pseudo-random numbers is as good as the original since all the tests (e.g. the spectral test) have been performed using the latter. In other words, the correlations between numbers $K$ positions apart in the original sequence are, generally speaking, less well known. Taking these considerations into account, we have opted for the sequence splitting method.

3. Description of the programs

3.1. clonEasy, a simple parallelization package

The clonEasy package is a collection of Linux scripts and auxiliary FORTRAN programs that implement Secure Shell-based communication between a “master” computer and a set
of “clones”. They perform simple operations, such as upload and download files or compile and execute programs, with the aim of running a code that performs a MC simulation on all the clones at the same time. The scripts that are provided with clonEasy are briefly explained below:

- **clon-setup**: sets up the environment for clonEasy to work.
- **clon-upload**: uploads files from the master to the clones.
- **clon-download**: downloads files from the clones to the master.
- **clon-remove**: removes files and directories from all clones.
- **clon-make**: executes a compilation script on each clone.
- **clon-run**: uploads the PRNG seeds and executes a program on all clones.

Most of these scripts take arguments; when not enough of them are provided, a message is issued to the screen to inform the user. The scripts are written using the syntax of the bash shell and, consequently, they may run under various Unix flavours, although we have only tested them on Linux distributions. The README file distributed with the package gives a detailed description of their use.

Secure Shell (ssh) is used to execute commands on a remote machine and Secure Copy (scp) is used to move files across the net. Thus, any computer on the Internet with a Secure Shell and Secure Copy (scp) is used to move files across the net. After all the nodes with the same initial conditions, the user should carefully choose these seeds so as to produce independent sequences in each processor. The sample file `clon.tab` distributed with the package is displayed in Fig. 3.

The various processes that run in parallel do not need to communicate with each other. When all of them have finished, their outputs are downloaded to the master where they can be combined to obtain the global result with a reduced statistical uncertainty. The FORTRAN program `combine`, supplied with clonEasy, can be used to perform this task. Its input file can be adapted to read a wide variety of output file formats. `combine` looks for some user specified keywords inside the output files and reads the value of the quantity of interest $q_k$ ($k = 0, 1, \ldots, K - 1$) produced by each clone. It can also process output files containing a data set, e.g. an energy spectrum. After $N_k$ histories have been simulated by the $k$th clone producing the value $q_k$ with variance $\sigma^2(q_k)$, the global average value $\bar{q}$ and its standard deviation $\sigma(\bar{q})$ are obtained using the relations

$$\bar{q} = \frac{1}{N} \sum_{k=1}^{K-1} N_k q_k$$  \hspace{1cm} (13)

and

$$\sigma(\bar{q}) = \frac{1}{N} \sqrt{\sum_{k=1}^{K-1} N_k^2 \sigma^2(q_k)},$$  \hspace{1cm} (14)

where

$$N = \sum_{k=1}^{K-1} N_k$$  \hspace{1cm} (15)

is the total number of histories simulated. Notice that Eq. (14) is derived from (13) by taking advantage of the independence of

---

2 For a description of the features and use of an open distribution of Secure Shell tools, the reader may consult [http://www.openssh.com](http://www.openssh.com).
the various $q_k$'s. The overall relative uncertainty $\Delta$ is obtained as the percentage
\[ \Delta = 100 \frac{\sigma(\bar{q})}{\bar{q}}. \tag{16} \]

The intrinsic and absolute simulation efficiencies are defined by
\[ \varepsilon_N = \frac{1}{N/\Delta^2}, \tag{17} \]
and
\[ \varepsilon = \frac{N}{I} \varepsilon_N, \tag{18} \]
respectively, where $N/I$ stands for the total simulation speed, in histories per unit clock time (i.e. real time), which would be achieved if all the clones were running in single-process mode.

We have
\[ N/I = \sum_{k=1}^{K-1} \frac{N_k}{I_k}, \tag{19} \]
where $I_k$ is the CPU time (that is, user time) employed by the $k$th clone to simulate $N_k$ histories. Note that $\varepsilon_N$ depends only on the performance of the simulation algorithm per unit simulated history, regardless of any timing considerations—hence the term “intrinsic”.

The absolute simulation efficiency increases approximately linearly with total CPU power, or what is equivalent in the case that all CPUs are identical, with the number of processors. This assertion is quite obvious from the considerations made above, since no time is spent intercommunicating clones or sending information between these and the master, except when required by the user. In consequence, $N/I$ is proportional to the computer power available and, since $\varepsilon_N$ is independent of this quantity, Eq. (18) reflects the claimed linearity.

3.2. seedsMLCG, sequence splitting of an MLCG

The program seedsMLCG provides, using Eq. (6), the initial seeds required to feed a set of different CPUs in order to implement the sequence splitting technique. On execution, the user is prompted to introduce the modulus and multiplier of an MLCG and the separation (that is, the number $J$ of intermediate random values) to be jumped between consecutive seeds. Since long integers with sign are usually restricted to 32 bits, input separation is limited to $2^{31}$ – 1. For this reason, the program also allows the user to input the separation as the exponent of a power of 10 to efficiently split the sequence of generators with a very long period. Negative distances can be entered to jump the sequence backwards with the aid of Eq. (7). This capability may be useful for debugging purposes; for instance, to reproduce a certain past history.

The FORTRAN 77 source code of seedsMLCG is accompanied by three sample input files containing the parameters that define the two MLCGs employed by RANECU and the MLCG used in the extended version proposed in previous sections. These files are read by the program by redirecting the keyboard input with the ‘<’ sign from the system command line. In Table 2 a set of ten seeds found with the three sample input files is presented. Each seed starts a disjoint subsequence with $10^{15}$ numbers. On average, the generation of each seed takes a few µs on a modern computer.

4. Conclusion

MC codes can be parallelized in a straightforward way without significantly changing the source code or using sophisticated software libraries. A simple communication system based on the Secure Shell protocol that distributes a simulation job among multiple CPUs and collects all the output files has been implemented in the script package clonEasy. An auxiliary program that combines the various output files to yield the global result with reduced statistical uncertainty is also included.

The statistical independence of the output from the different executions is guaranteed when each computer uses an independent sequence of pseudo-random numbers. Such sequences can be produced with the initialization values supplied by the program seedsMLCG for MC codes that rely on MLCGs or a combination of them. The tools presented here have been successfully applied to the MC code PENEOPE, which utilizes the PRNG RANECU. A brief explanation of the mathematical properties of this generator has also been provided with the aim of proposing an extension that has a much longer period, a feature that may become necessary in future applications.

clonEasy and seedsMLCG can be freely downloaded from http://www.upc.es/inte/downloads/clonEasy.htm and http://www.upc.es/inte/downloads/seedsMLCG.htm, respectively.

Acknowledgements

We would like to thank Prof. Francesc Salvat (Universitat de Barcelona), co-author of PENEOPE, for giving us valuable suggestions. The authors gratefully acknowledge financial support from the Fondo de Investigación Sanitaria (Spain), Project No. 03/0980.
Appendix A

A.1. Test run output for ‘clonEasy’

(Note: the following example assumes a Linux system with the bash shell. The file clon.tab shown in Fig. 3 has been used.)

```bash
user@master: ~/clonEasy$ source clon-setup
user@master: ~/clonEasy$ cd test

user@master: ~/test$ clon-upload
Usage: [thisScript] [FileToUpload] [ClonTableFile] [ClonDir]
%all can be used for [FileToUpload]

user@master: ~/test$ clon-upload program.f clon.tab JobDir
program.f 100% |******************************| 83 00:00
program.f 100% |******************************| 83 00:00
program.f 100% |******************************| 83 00:00
user@master: ~/test$ clon-upload make_ifc clon.tab JobDir
make_ifc 100% |******************************| 233 00:00
make_ifc 100% |******************************| 233 00:00
make_ifc 100% |******************************| 233 00:00
user@master: ~/test$ clon-upload make_g95 clon.tab JobDir
make_g95 100% |******************************| 233 00:00
make_g95 100% |******************************| 233 00:00
make_g95 100% |******************************| 233 00:00

user@master: ~/test$ clon-make
Usage: [thisScript] [ClonTableFile] [ClonDir]

user@master: ~/test$ clon-make clon.tab JobDir

user@master: ~/test$ clon-run
Usage: [thisScript] [ClonTableFile] [ClonDir] [ExecutableFile]
(Optionally: [RedirectionName])

user@master: ~/test$ clon-run clon.tab JobDir program.x
rngseed.in 100% |******************************| 21 00:00
clon-run_aux 100% |******************************| 438 00:00
rngseed.in 100% |******************************| 21 00:00
clon-run_aux 100% |******************************| 438 00:00
rngseed.in 100% |******************************| 21 00:00
clon-run_aux 100% |******************************| 438 00:00

user@master: ~/test$ clon-download
Usage: [thisScript] [ClonTableFile] [ClonDir] [FileToDownload]

user@master: ~/test$ clon-download clon.tab JobDir program.out
program.out 100% |******************************| 21 00:00
program.out 100% |******************************| 21 00:00
program.out 100% |******************************| 21 00:00

user@master: ~/test$ clon-remove
Usage: [thisScript] [ClonTableFile] [ClonDir] [FileToRemove]
if [FileToRemove] is %all then [ClonDir] is deleted completely
USE WITH CARE!!

user@master: ~/test$ clon-remove clon.tab JobDir %all

user@master: ~/test$ combine.x < combine.in > program.out
```
A.2. Test run output for ‘seedsMLCG’

***************************************************
** CALCULATING FUTURE OR PAST ELEMENTS OF THE **
** SEQUENCE OF A MULTIPLICATIVE LINEAR **
** CONGRUENTIAL GENERATOR (MLCG) **
**
**
***************************************************

- Initial integer seed: \( S(i) = 1 \)
- Number of new seeds to be calculated = 10
- Is the interval to the next seed a power of 10? [0=no/1=yes] Yes
- Distance between seeds (negative value for a backward jump): \( j = 10^{15} \), \( k = 15 \)
- Multiplier of the MLCG: \( a = 40014 \)
- Modulus of the MLCG: \( m = 2147483563 \)

- The input generator is the first MLCG from RANECU.

- RESULTS: 11 seeds, separated \( 10^{15} \) units:

\[
1 \\
918882992 \\
2069007070 \\
944675654 \\
149156960 \\
360537627 \\
1446789139 \\
888673974 \\
258943 \\
1434784182 \\
698429770
\]

References