H H

August 1994

HLRZ-94-24

A Fast Random Number Generator for the Intel Paragon Supercomputer

F. Gutbrod

Deutsches Elektronen-Synchrotron DESY
Notkestr. 85, D-22603 Hamburg, Germany

SCAN-9412102

CERN LIBRARIES, GENEVA

SW9450

Abstract

A pseudo-random number generator is presented which makes optimal use of the architecture of the i860-microprocessor and which is expected to have a very long period. It is therefore a good candidate for use on the parallel supercomputer Paragon XP. In the assembler version, it needs 6.4 cycles for a real*4 random number. There is a FORTRAN routine which yields identical numbers up to rare and minor rounding discrepancies, and it needs 28 cycles. The FORTRAN performance on other microprocessors is somewhat better. Arguments for the quality of the generator and some numerical tests are given.

1 Introduction

Stochastic behaviour in deterministic systems arises from large ensembles and from absence of bound states in the dynamics of the interaction. It is natural to exploit this for the generation of almost random numbers on modern microprocessors, which usually have large and fast local memories (caches) and which can simulate an interaction in one or two cycles. The pseudo-random number generator presented here is based on this idea, and it is adapted to certain properties of the i860-microprocessor in the following way: It avoids integer multiplication, it uses right shift of 64-bit words, it ensures use of superscalarity and of pipelining of the few necessary floating point operations, and it uses quadrupel word data transport between floating point unit and memory. It acts on the content of a buffer of 512 32-bit words, and the period could therefore be of the order of $2^{16.384}$. A fast assembler code has been written, and an almost equivalent FORTRAN version runs well on all processors capable of the standard bit manipulation functions. The generator substantially weakens certain triple correlations present in the Kirkpatrick-Stoll-Greenwood algorithm [1].

The generator can be viewed as a physical system. One has to imagine N_{buff} objects ('*black balls*') which have an intrinsic property, the *spin*, and which have positions characterized by discrete angles. The spin is given by a n_{bit} -bit word. There is an extra *red* ball with its spin spin_r , out of which

$$(1) \quad L_b = \log_2 N_{\text{buff}}$$

nonleading bits are also interpreted as an angle. This angle determines an ensuing collision with the ball positioned at that angle, and the spin of the colliding ball, spin_c ,

is taken as output random number. The spin of the colliding ball is replaced by a bit-function of the two spins, $F_2(\text{spin}_c, \text{spin}_r)$, and the old value of spin_c will replace spin_r . The 'interaction' F_2 is given¹ by left circular shift by s bits and exclusive OR , which in FORTRAN notation is

$$(2) \quad F2(r1, r2) = \text{IEOR}(\text{ISHFTC}(r1, s, 32), r2).$$

The above procedure has many features in common with the scheme introduced by Bays and Durham [2].

If the spins were continuous variables, and N_{buff} sufficiently large, there would be no doubt about the chaotic properties of the system, since any small change in a spin would eventually lead to a different angle and thus to totally different collisions. The main problem is to know how large n_{bit} and N_{buff} have to be in order to get a good approximation to the continuum case.

First of all, one expects irregularities when the same black ball is hit in sequence, since at the second collision its spin is a function of the previous one, which may lead to correlations. To reduce the probability for this, N_{buff} has to be as large as possible. The price for increasing N_{buff} depends on the cache size and on the vector length. For vectors longer than N_{buff} , the overhead for a large buffer size N_{buff} is small, if the buffer fits into the cache. Present technology limits N_{buff} to about 1K or 2K 32-bit words.

Secondly, in the procedure given above, there exist traps, i.e. the sequence gets caught in very short subcycles ('bound states'). The two simplest traps are easy to understand and the probabilities to be trapped are known under certain assumptions. They can be tested numerically in simple cases. For a word length of n_{bit} bits, the most trivial trap (to be explained in the next section) is encountered with probability $2^{-2n_{\text{bit}}}$. Trapping is thus extremely rare for 32-bit words. Nevertheless, this and other traps should be cured. This will be done by increasing n_{bit} effectively to $n_{\text{bit}} = 128$ and by making the red angle depend also on an external variable, namely the loop count. To update the physical picture, one can imagine the angles of the black balls to rotate with constant speed. It is hoped that other, more complicated traps are cured likewise, and/or that they are less probable than the two identified ones.

In the next section the generator is described in detail, and the correlations, the traps and their remedies are explained. Since beyond that no mathematical theory can be given, the generator is studied numerically by some of the standard tests [3], also for small values of n_{bit} . Such tests will be presented in section 3. After the conclusions, some justification for the choice of the generator will be given in the Appendix.

2 The Generator

2.1 The Simple Code

Since the physical picture of the generator has already been given, the FORTRAN code of the simple version may be understandable immediately. For a fixed word length of 32 bits and for a buffer length $\text{nb} = 128$, the code is:

¹The index 2 signals that we consider a function of two variables

```

*****
      subroutine ranshi(n, a)                                !1 put n random
***** !2 numbers into array a
      parameter(nb = 128, nbit = 32, is = 17, amin = 1.e-10) !3
      parameter(nc = nb + 2, mask = nb -1, scalin = 2.**(1-nbit)) !4
      common /ranbuf/ mbuff(nc)                             !5 buffer
      real*4 a(*)                                             !6 output randoms

      iangle = mbuff(nb + 1)                                !7 load red angle
      ic      = mbuff(nb + 2)                                !8 load red spin

      do i = 1, n                                           !9

          ir = mbuff( iangle )                               !10 hit ball iangle
          a(i)= float(ishft(ir,-1)) * scalin + amin !11 avoid zero output
          iro = ishftc( ir, is, nbit)                       !12 circ. left shift by
                                                           !   is bits, see below
          mbuff( iangle ) = ieor( iro, ic)                   !13 replace black spin
          ic = ir                                              !14 replace red spin
          iangle = iand(mask , ir + i ) + 1                  !15 boost spin and
                                                           !16 mask for new angle
      enddo                                                  !17 end of loop

      mbuff(nb + 1) = iangle                                !18 store red angle
      mbuff(nb + 2) = ic                                     !19 store red spin

      return                                                 !20
      end                                                    !21

***** code to avoid the potentially slow circular shift ****
***** replace line 12 by the following three lines ****
      irl = ishft(ir, is)                                   !12a avoid slow circular
      irr = ishft(ir,-is)                                   !12b shift by right and
      iro = ior(irl, irr)                                   !12c left shift and IOR
***** code for using a function with three arguments *****
***** insert the lines after lines 8, 12 and 13 respectively
      im = -1                                               !8a start with ad hoc
                                                           !   third number
      iro = ieor(iro, im)                                   !12a XOR once more
      im = ic                                                !13a replace third number
***** end of alternative code *****

```

Fig. 1: The simple generator for 32-bit words

2.2 Explanations

1. Before calling the subroutine `ranshi`, the buffer-array `mbuff(nc)` with size $nc = N_{\text{buff}} + 2$ has to be initialized with 32-bit integers taken from another simple generator, and possible deficiencies of such a generator will be eliminated by a few warming-up calls to `ranshi`. The red angle, stored in `mbuff(nc)`, has to be bounded by $1 \leq \text{mbuff}(nc) \leq N_{\text{buff}}$. The initializing routine `ranvin` is listed in appendix B.

The initialization is not critical. Even a rather exotic one, namely setting only one bit of the $32 \times N_{\text{buff}}$ bits of the whole buffer to 1, and all others to 0, does not lead to detectable deficiencies after producing $O(10.000)$ random numbers. In the more sophisticated version presented later, one can even initialize everything by -1, the generator recovering quickly.

2. The creation of a new random word is controlled by the spin of the red ball, `ic`, (see lines 8 and 14) and by its angle, `iangle`, (see lines 7 and 15). The buffer content `mbuff(iangle)`, `ir`, will be converted to a floating point number, where the value 0.0 is avoided (see line 11). Then it is circularly left shifted (line 12), and XOR-ed with the red spin (line 13). The result is substituted back for `mbuff(iangle)` (line 13), and the red spin is replaced by the black spin (line 14).
3. The new red angle is found by masking the last L_b bits, after boosting it with the loop count (line 15). This boosting will eliminate at least the two most probable traps, which will be explained in the next subsection.

2.3 Traps and Correlations

Without boosting the angle, the first trap is encountered, if the spin of the ball with angle 1 has value 0 and if $\text{spin}_r = 0$. Then all future output numbers will be zero. The probability for this trap to occur is $2^{-2n_{\text{bit}}}$, if the distribution of integer numbers is random up to the occurrence of the trap. For practical applications, with $n_{\text{bit}} = 32$, this probability is small enough for the trap to be irrelevant. For testing purposes with smaller values of n_{bit} (and for nervous users), one should be able to avoid the trap, which is easy. If the red angle is boosted by force, i.e. by an external variable, one will escape the trap. This has been done in the above code by adding the loop count to the red spin before masking it to obtain the angle (see line 15).

The next trap occurs, when the red ball has spin $s_{-1} = 111\dots$, the 1st ball also has spin s_{-1} and the N_{buff} -th ball has spin 0. The angle will then jump between direction 1 and N_{buff} in a three-state cycle. This trap is met, under the same assumptions as above, with probability 3×2^{-3n} . It is likewise cured by boosting the angle externally.

In addition to the traps, there exist deficiencies in the probability for obtaining two specified numbers consecutively, i.e. the serial test [3] is not perfect for integer numbers. Specific errors in the correlation occur for certain numbers r_n due to an anomalous probability to hit the same black ball twice and obtaining output numbers r_{n+1} which agree with r_n in the last L_b bits. Because of eq. 2, the condition for this to occur is ² that

²I will forget, for simplifying the discussion, the boost of the angle.

the bits of r_n with positions $n_{\text{bit}} - s$ to $n_{\text{bit}} + L_b - s$ are all 0. The set of bits with these positions will be called X_s for reference. For numbers r_n with $X_s = 0$, the probability for r_{n+1} to follow r_n is $(2N_{\text{buff}} - 1)/N_{\text{buff}}^2$ instead of the correct probability $1/N_{\text{buff}}$. This error by almost a factor 2 is balanced by a corresponding change in the probability for those $N_{\text{buff}} - 1$ numbers r_{n+1} which differ from r_n in the last bits. Considering also the case $X_s \neq 0$, the serial test there has an error of order $O(1/N_{\text{buff}})$, since the sequence cannot proceed via selection of the same ball.

Choosing N_{buff} large will not reduce the error in the serial test for $X_s = 0$, but will reduce the probability for these numbers to occur, and it will reduce the error for the case $X_s \neq 0$. Anyhow, these errors will not be detectable in floating point applications, where only 23 leading bits will be used. In this case one averages over an interval of 9 low order bits, which is equal to or larger than the largest value of L_b chosen here.

As for any other generator based on a function of two recently created numbers, there must be triple correlations, as explained in the Appendix. Such correlations, as well as the error in the serial test can be eliminated by extending the algorithm slightly. One may store one more used spin and perform one more IEOR. Then triple correlations disappear in favour of quadrupel correlations, which are certainly less harmful. The necessary change of the code is indicated in fig.1 by the lines given at the bottom of the text.

Another way to avoid the traps is to exclude the collision with the same ball. In this way also the serial correlation will be corrected. In view of the improvements explained in the last paragraph and those given in the next subsection, this is probably not necessary.

2.4 The Code for 128 Bit Rotations

The fact that the simple traps are related to the rotationally invariant spins 0 and s_{-1} , suggests to increase the word length to make such states even less probable. In the final code I therefore combine four consecutive spins³ and shift them cyclically, thus arriving at an effective word length of 128 bits. In order to obtain a fast assembler code, a further blocking to eight output random numbers has been introduced. The penalty for this is that now the length of the array into which the random numbers are stored, should be a multiple of 8. Otherwise, a segmentation violation may occur. If the vector length is not a multiple of 8, too many random numbers will be calculated. Of course, the code can be modified to eliminate these limitations.

The red angle is chosen to point alternately into the lower and upper half of the buffer. Since subsequent accesses to the same buffer position are no longer possible, this eliminates the error in the sequential test. For speed reasons, the angle is derived no longer from the least significant bits but from those left-shifted by two positions. In the FORTRAN code, this change shows up by the shift operation in the determination of `iangle`. In a pure FORTRAN application, this shift can be omitted.

The code is now

³The starting address may be anywhere in the first N_{buff} elements of the buffer, which has been extended by a few words.

```

*****
      subroutine ranshi4(n, a)                                ! put n randoms 1
*****
*****                               ! in array a.
*****
***** subroutine only works, if n is a multiple of 8 *****
***** otherwise segmentation violation may occur *****

      parameter( nb = 512, nbit = 32, is = 15)                !2
      parameter( nc = nb + 12, mask = 2*nb - 1, mupper = nb/2 + 4) !3
      parameter( isl = nbit - is, isr = - is)                  !4
      parameter( lbuff = 2 * nb -1, scalin = 2.**(1 - nbit) ) !5
      parameter( amin = 1.e-10)                                !6
      real a(*)                                                  !7
      common/ranbuf4/ mbuff(lbuff)                             !8

      iangle = mbuff(nb + 9)                                     ! load red angle          9
      ic1     = mbuff(nb + 5)                                   ! load red spin          10
      ic2     = mbuff(nb + 6)                                   ! load red spin          11
      ic3     = mbuff(nb + 7)                                   ! load red spin          12
      ic4     = mbuff(nb + 8)                                   ! load red spin          13
      nboost = n - 16                                           ! comply with assembler 14

      do i = 1, n, 8                                           !                               15

          j = i + 4                                             ! comply with assembler 16

          ir1 = mbuff(iangle )                                  !                               17
          ir2 = mbuff(iangle + 1)                               !                               18
          ir3 = mbuff(iangle + 2)                               !                               19
          ir4 = mbuff(iangle + 3)                               !                               20

          ip1 = ishft(ir1, -1)                                  ! get rid of sign        21
          ip2 = ishft(ir2, -1)                                  ! bit                     22
          ip3 = ishft(ir3, -1)                                  !                           23
          ip4 = ishft(ir4, -1)                                  !                           24

          a(j ) = float(ip1) * scalin + amin ! add amin to          25
          a(j + 1) = float(ip2) * scalin + amin ! avoid zero            26
          a(j + 2) = float(ip3) * scalin + amin ! output                27
          a(j + 3) = float(ip4) * scalin + amin !                      28

          io1 = ior(ishft(ir1, isl), ishft(ir2, isr)) ! circular shift 29
          io2 = ior(ishft(ir2, isl), ishft(ir3, isr)) ! of a 128-bit 30
          io3 = ior(ishft(ir3, isl), ishft(ir4, isr)) ! word            31
          io4 = ior(ishft(ir4, isl), ishft(ir1, isr)) !                  32

          mbuff(iangle ) = ieor( io1, ic1) ! replace black spin 33

```

```

mbuff(iangle + 1) = ieor( io2, ic2)      ! replace black spin  34
mbuff(iangle + 2) = ieor( io3, ic3)      ! replace black spin  35
mbuff(iangle + 3) = ieor( io4, ic4)      ! replace black spin  36

iangle = ishft(iand(mask ,ir1),-2) + mupper ! mask for angle 37
                                           ! upper half
***** repeat one cycle in order to agree with assembler *****

ic4 = mbuff(iangle      )                ! load new red spins  38
ic3 = mbuff(iangle + 1)                ! in different order  39
ic2 = mbuff(iangle + 2)                !                        40
ic1 = mbuff(iangle + 3)                !                        41

j    = i + 8                            ! next quadrupel      42
if(j .gt. n) j = 1                      ! at end of loop,      43
                                           ! do first 4 words     44

ip1 = ishft(ic1, -1)                    ! get rid of sign bit  45
ip2 = ishft(ic2, -1)                    !                        46
ip3 = ishft(ic3, -1)                    !                        47
ip4 = ishft(ic4, -1)                    !                        48

a(j    ) = float(ip1) * scalin + amin    ! add amin to avoid  49
a(j + 1) = float(ip2) * scalin + amin    ! zero output        50
a(j + 2) = float(ip3) * scalin + amin    !                    51
a(j + 3) = float(ip4) * scalin + amin    !                    52

io1 = ior(ishft(ic1,isl),ishft(ic2,isr)) ! circular shift     53
io2 = ior(ishft(ic2,isl),ishft(ic3,isr)) !                    54
io3 = ior(ishft(ic3,isl),ishft(ic4,isr)) !                    55
io4 = ior(ishft(ic4,isl),ishft(ic1,isr)) !                    56

mbuff(iangle      ) = ieor( io1, ir1)    !replace black spin  57
mbuff(iangle + 1) = ieor( io2, ir2)      !                    58
mbuff(iangle + 2) = ieor( io3, ir3)      !                    59
mbuff(iangle + 3) = ieor( io4, ir4)      !                    60

iangle = ishft(iand(mask ,ic1 + nboost),-2) + 1 !boost angle 61
                                           ! lower half
nboost = nboost -8                        ! reset boost        62
enddo ! i                                  !                    63

mbuff(nb + 9) = iangle                    ! store red angle    64
mbuff(nb + 5) = ic1                      ! store red spin     65
mbuff(nb + 6) = ic2                      ! store red spin     66
mbuff(nb + 7) = ic3                      ! store red spin     67

```

```

mbuff(nb + 8) = ic4                                ! store red spin    68

return                                              !                        69
end                                                !                        70
*****
*****

```

Fig. 2: Improved code, equivalent to an assembler version

The assembler version for the i860-microprocessor differs in the conversion `integer` → `floating point number` such that rounding may be different in some cases.

3 Numerical Tests

Out of the classical tests (see [3]), I have concentrated on the run test, the gap test and the maximum-of-t-test, apart from testing the expectation values $\langle r_n^2 \rangle$ and $\langle r_n r_{n+d} \rangle$ with several values of d . Tests been performed with up to 4.8×10^{10} random numbers, and for $n_{bit} = 32$ no failures have been observed for both codes.

3.1 The Case of Short Words

The simple code given in fig. 1 can easily be used in the case of small n_{bit} ($n_{bit} = 4, 5, 6, \dots$), and there it is possible to detect the deficiencies of the generator. Traps can be found either by direct monitoring or by watching for a rapid deviation from expectation of some measured average .

Especially, without boosting the red angle, one gets quickly caught by one of the two traps mentioned above, and the measured probabilities for this agree with the quoted numbers within 15 %. More precisely, they are smaller than the predictions by this amount. This can be understood by the wrong probabilities in the serial test for certain numbers, which themselves have been clearly observed. No other traps have been found, which means that they occur with still smaller probabilities, if they exist.

If the red angle is boosted by the loop index, no traps have been observed. However, for $n_{bit} = 4$ and $N_{buff} = 8$, the 16 different output numbers deviate from equi-distribution by about 10 %. This again can be ascribed to the wrong correlation of certain numbers. For $n_{bit} = 6$, the deviations reduce to about 1 %, and they disappear if we introduce the 'three-body' interaction, as specified at the end of fig. 1, or if the immediate recurrence of the same angle is excluded by an obvious modification of the same program. The deviations are no longer detectable for $n_{bit} = 8$.

For $n_{bit} = 8$, there are errors in the run test of the order of 1%, which is understandable as the run test is ill-defined if output numbers may be equal with significant probability. The errors are no longer observable for $n_{bit} = 16$. The maximum-of-t test with $t = 32$ and with 10^7 random numbers still fails at $n_{bit} = 12$, but looks o.k. at $n_{bit} = 16$. This seems to be due to the finite spacing of the random numbers and not due to correlations, since the remedies used in the last paragraph do not improve the situation.

3.2 The Case of Long Words

3.2.1 The Run Test

The run test has been performed 6 times with $8 \cdot 10^8$ random numbers each, taken from **ranshi4**, and the relative deviations of the number of runs from expectation are 10^{-4} for runs of lengths 1 and 2. The $\chi^2/d.f.$ for runs up to length 9 (which all have more than 100 hits) were 0.86, 2.0, 1.7, 0.64, 1.44 and 0.31. This looks a bit noisy, and it is due to one 3.5σ deviation for runs of length 5, which look normal in the other runs. I believe in a large fluctuation.

3.2.2 The Gap Test

Gaps were observed in 32 intervals between 0.0 and 1.0, with lengths up to 100. This amounts to a decrease in probability from length 1 to length 100 by a factor 50. From 10^8 random numbers, the distribution in the 3200 categories has been studied. The result was a $\chi^2/d.f. = 0.98$ and a normal error distribution was observed within 2σ around the maximum.

3.2.3 The Maximum-of-t Test

The maximum-of-t-test has been performed with lengths $t = 16$ and $t = 32$, in order to look for short term fluctuations ('collective motions'). The total number of randoms for $t = 32$ was as in the run-test, and the χ^2 distribution was similar, apart from the absence of the one large deviation.

3.2.4 Average Values

The average $\langle r_n^2 \rangle$ has been measured repeatedly on 80 nodes on the Paragon XP/S10, and a χ^2 has been derived from the scatter of the average across the nodes. For three runs with a total of 1.6×10^{10} random numbers each, I found the values $\chi^2/d.f. = 1.03$, 1.16 and 1.07. With the same statistics, the averages $\langle r_n r_{n+d} \rangle$ with $d = 1, 4$ and 16 were measured, and they agreed with the value $1/4$ within errors, i.e. with an relative accuracy of better than 10^{-5} .

3.3 Performance on Microprocessors

The performance of the code of course depends on the vector length and on the memory location of the output vector and of the buffer. The numbers given in the abstract for the processor i860 refer to the optimal case, with a vector length of 256. If 10^6 random numbers are calculated in sequence, the generator **ranshi4** (in FORTRAN) performs as follows on other processors:

IBM RS6000/580 (62.5 MHz) :	0.21	$\mu sec/number$
HP 9000/735 (99 MHz) :	0.16	$\mu sec/number$
SG R4400(IP19) (150 MHz) :	0.17	$\mu sec/number$
PC 486DX (33 MHz) :	25	$\mu sec/number$

When comparing these numbers with those on the i860-processor (FORTRAN: $0.55 \mu\text{sec}$, assembler: $0.13 \mu\text{sec}$), one has to consider the relatively low frequency (50 MHz) of this processor. In view of this, the assembler version is quite fast.

4 Conclusions

For small word length, the generator has clear deficiencies which show up in deviations from equi-distribution and in tests sensitive to the density of output random numbers. The former can be corrected by introducing 'three-body' interactions, and all observed deficiencies quickly decrease with increasing word length. Specifically, for a word length of 16 bits, no defects were observed with moderate statistics, and for 32-bit words, several high statistics tests were passed successfully. Still, one expects certain errors in the probability for successive occurrence of numbers correlated in the last bits, which however are not observable in floating point applications, and which theoretically are cured in the extended version `ranshi4` of fig. 2. Also the well known triple correlations [4] have to be mentioned (see Appendix A), which show up if the accuracy is better than 4×10^{-6} , and which have a random delay.

The sizes of the 'physical ensemble' we have used, $N_{\text{buff}} = 128$ and 512, are somewhat ad hoc. A large value of N_{buff} will lead to a long random delay for the triple correlation, and it will reduce the error in the serial test. Since the chance to detect both deficiencies is minute for long word lengths, the above value may be an overkill.

Naively, the period of the generator is of the order of $2^{n_{\text{bit}} N_{\text{buff}}}$, but unfortunately no mathematical theory can be given for the actual period. One can remark that, due to the boost of the angle with the loop count, the sequence of random numbers depends on how many numbers are calculated within one subroutine call. By making this number depend on the physical problem under study, one introduces an external noise into the system such that the actual period is determined by that of the external system. Perhaps this can be studied for the case of smaller word length.

As indicated by the steps undertaken to improve the statistical properties, the generator is flexible, and elimination of the known limitations and of others should be possible without great loss in performance. The strongest tools seem to be the introduction of 'many-body' interactions and an increase of the effective word length. Also the generation of `real*8` random numbers, for which a similarly fast assembler version already exists, poses no serious problems.

Acknowledgement

The code of the generator has been developed and tested on the Supercomputers Intel iPSC/860 and on the Paragon XP/S with 140 nodes, both at the HLRZ, KFA Jülich. The author is grateful to the Paragon Operating Team and, for useful assistance, to M. Weber (ZAM at KFA Jülich). Test were made also on the SP1 of IfH-DESY, Zeuthen, and useful support by the computer department of the IfH is gratefully acknowledged.

References

- [1] S. Kirkpatrick, E.P. Stoll, *Journal of Computational Physics* **40** (1981), 517
- [2] C. Bays and S.D. Durham, *A.C.M. Trans. Math. Software* **2**, (1976), 59
- [3] D. Knuth, *Seminumerical Algorithms*, 2nd ed., vol. 2 (Reading, Mass.: Addison-Wesley)
- [4] M. Lüscher, *Computer Physics Communications* **79** (1994), 100
- [5] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, Cambridge 1988
- [6] G. Marsaglia and A. Zaman, *Ann. Appl. Prob.* **1** (1992) 462

Appendix

A Some Generalities

In order to justify the present choice of the generator, I will start with some remarks on those random number generators which might be useful on supercomputers. There one requires a long period, small correlations, and a performance of few cycles per random number. Let us first discuss the standard linear congruential generator and its extensions, as summarized in ref. [5].

These generators belong to the general class in which a newly generated number, to be used immediately or later, is a function of the preceeding one:

$$(3) \quad r_n = F_1(r_{n-1})$$

For 32-bit words, the period of this basic sequence clearly is insufficient. It has been proposed by Bays and Durham [2] to reorder such sequences by

1. storing the r_n 's in a buffer of length N_{buff} (e.g. $N_{\text{buff}} = 128$)
2. selecting the output random number from the buffer with the help of a second random sequence which may be identical to the first one.

In this way, the period of the output is practically infinite, but for small N_{buff} , there exist macroscopic fluctuation with the period of the basic sequence. This is because when taking the average over n N_{buff} consecutive numbers with $n = O(10)$ say, all elements put into the buffer have been used once with a probability close to one, such that the average differs little from that of the basic series. Thus one needs longer basic periods. A 64-bit version would be sufficient, but 64-bit integer multiplication is not easily available. As a compromise, one could use floating point multiplication and truncation, but this seems to be time consuming and also difficult to pipeline. Thus, the linear congruential generator and its reordered versions do not seem to be good candidates for our purpose.

The next, more complicated class of generators uses a function of two 'recent' random numbers⁴ to generate a new one:

$$(4) \quad r_n = F_2(r_{n-l_1}, r_{n-l_2})$$

⁴If the lags l_1 and l_2 are fixed, one may interpret the two arguments of F_2 as different parts of a longer word, and thus the distinction of the two classes is not clearcut.

where the lags l_1 and l_2 can be specially adjusted to the function F_2 . For this function, convenient choices are subtraction [6] or bit-wise exclusive OR [1]. The periods obtainable in this way are now sufficiently large. There exist, however, triple correlations between the output random numbers, which may lead to significant errors in some applications. Suppose that we have two parts of the sequence with numbers r_n and $r_{n'}$ and

$$(5) \quad |r_{n-l_i} - r_{n'-l_i}| < \epsilon, \quad i = 1, 2$$

for some n and n' , then the sequence will have a correlation between the pair of random numbers r_n and $r_{n'}$. The way how the correlation depends on ϵ , may be vastly different for different F_2 . If F_2 is given by subtraction [6], we have

$$(6) \quad |r_n - r_{n'}| < 4\epsilon$$

i.e. the correlation is weakened⁵. If we only use bit-wise exclusive OR (IEOR) as it is done in the Kirkpatrick-Stoll-Greenwood algorithm [1], there is no softening of the correlation: identical bits will give identical bits at the same position. Since one of the lags is very large in the standard implementation⁶ ($l_1 = 532$), this may be tolerable. I prefer the generator to be discussed next for its statistical quality, its simplicity and flexibility towards further improvement.

If we use for F_2 left circular shift by s bits and exclusive OR, see eq. 2, the correlation problem for floating point numbers is not severe, since the correlation occurs between leading and nonleading bits: If the numbers r_{n-l_i} and $r_{n'-l_i}$, $i = 1, 2$ differ in the s -th nonleading bit, the resulting pair of the sequence will differ in the sign bit, i.e. the 'distance' between the output numbers is maximal. If we take $s = 17$, a value of $\epsilon = 4 \cdot 10^{-6}$ will already lead to complete decorrelation, and this can be tolerable for most applications. If we furthermore apply the trick of ref. [2], there are delays d and d' before the correlated numbers appear as output, which are random delays of order N_{buff} . Especially for such Monte Carlo simulations, where one proceeds in a regular way through a lattice, these irregular delays may be helpful. There is little overhead created by this technique, if the selection of the output random number from the buffer is controlled by the random sequence itself (i.e. if no additional congruential generator is employed). To complete the specification, I set $l_1 = 1$ and $l_2 = 2$. The algorithm is compact and very fast, when written in assembler language, but also the FORTRAN versions show good speed on most processors.

If the triple correlation is regarded as too dangerous, one can improve the algorithm at little extra cost. There is no immediate limitation to build algorithms constructing r_n out of m preceeding numbers, with F_m also built by IEO and (optionally) a shift operation. Since the basic operation can be performed in two cycles, the impact on speed is minor for $m = 3$ or $m = 4$. For the present algorithm, we restrict the case to $m = 2$ and $s = 17$, except for testing purpose, where also $m = 3$ is studied.

B Initialization of the Buffer

The buffer has to be filled with the statement

⁵It has been shown in ref. [4] that discarding a definite number of intermediate randoms will lead to complete decorrelation within floating point accuracy

⁶The generator has been implemented by G. Groten, ZAM-KFA-Jülich on the Paragon. The conversion of this FORTRAN code to assembler looks somewhat tedious.

```
call ranvin(iseed, iwarm)
```

where the routine `ranvin` is described below (together with the version `ranvin4`), and where `iseed` is any integer, and for `iwarm` something like 10 is sufficient. A subsequent generation of `n` random numbers is obtained by

```
real*4 a(1024)
call ranshi(n, a)
```

The corresponding sequence for `ranshi4` is

```
real*4 a(1024)
call ranvin4(iseed, iwarm)
call ranshi4(n, a)
```

The routine `ranvin4` is obtained by the indicated modifications in lines 1, 4, 6, 17 and 19 in the following code:.

```
*****
      subroutine ranvin (iseed, iwarm)          ! 1
c      subroutine ranvin4(iseed, iwarm)         ! 1a /128-bit code
***** initializes buffer for ranshi generator *****
      real*8 rtval, modu, modi, aa, ac, xint, scale ! 2
      parameter(modu = 1771875.0d0, aa = 2416.d0, ac = 37444.d0) !3
      parameter(nb = 128, nc = nb + 2, nbit = 32)          !4
c      parameter(nb = 512, nc = nb + 9, nbit = 32, lbuff = 2*nb-1) !4a
      parameter( scale = 2.d0 ** nbit )                  ! 5
      common /ranbuf/ mbuff(nc)                          ! 6
c      common /ranbuf4/ mbuff(lbuff)                    ! 6a /128-bit code
***** initializes buffer for ranshi generator *****

      real*4 a(nb)                                         ! 7
      modi = 1.d0 / modu                                  ! 8
      rtval = float(4 * iseed+1)                          ! 9 take an odd seed
      do j = 1, nc                                         ! 10 lin. congr. generator
        xint = rtval * aa + ac                            ! 11
        ntval = xint * modi                               ! 12 truncate
        rtval = xint - ntval * modu                       ! 13 modulus
        rnanf = rtval * modi                              ! 14 scale to = 0. ./ 1.0
        mbuff(j) = scale * (rnanf-0.5)                   ! 15 expand to all bits
      enddo ! j                                             ! 16 buffer is filled
      mbuff(nb+1)= iand(mbuff(nb+1), nb - 1) + 1         ! 17 restrict
                                                         ! red angle
c      mbuff(nb+9)= iand(mbuff(nb+9), nb - 1) + 1         ! 17a 128 bit
      do j = 1, iwarm                                     ! 18 warm up a
        call ranshi (nb, a)                               ! 19 few times
c        call ranshi4(nb, a)                             ! 19a 128 bit
      enddo ! j                                           ! 20 warming up
                                                         ! finished
      return
```

`end`

Fig. 3: Code to initialize the buffer by a linear congruential random generator

In the lines 10 to 16, a standard linear congruential generator is used to fill the buffer, and the scaling in line 15 uses also negative integers to set the sign in 50% of the cases. In line 17, the integer is truncated to the range allowed by the size of the buffer, and the code closes with the warming-up steps.