Prospects for Monte Carlo Methods in Million Processor-core Era and Beyond

July 9, 2014

Kenichi Miura, Ph.D.

Professor Emeritus
National Institute of Informatics
Tokyo, Japan

Outline

- Monte Carlo Methods as scalable algorithms
- Desirable characteristics of Pseudo-random Number Generator(PRNG)
- Parallelization of PRNGs
- Application Areas

Trends in HPC Architecture

- Higher degree of parallelism (>Million cores)
- Diminishing memory size per core
- Imbalance between CPU power and Bandwidth of Interconnecting network
- More expensive to move data than F.P. computation
- Fault Resilience becoming very critical issue

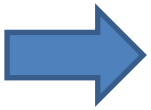
Advantages of Monte Carlo Methods

- Many traditional algorithms are to be re-examined because of bad scalability

- Monte Carlo Methods are **“embarrassingly parallel” and highly scalable**

- Monte Carlo Methods require **small memory size per core** because of geometry without discretization (no grid!)

- Monte Carlo Methods are fault-resilient (to some extent)



Wider use of Monte Carlo Method should be considered in various application areas, such as solving the Elliptic Partial Differential Equations

John von Neumann wrote,

JOHN VON NEUMANN
1903–1957

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin." (1951)

- Middle-square method
- Mapping function: $f(x) = 4x(1-x)$

Various Techniques Used in Connection With Random Digits, Collected Works Vol. 5 pp.768-770 (1961)

Important Features of PRNG Algorithms

- Long Period

- Good statistical quality (both serial and parallel)

- High speed generation

- Massively parallelizable in short time
  (ability to jump-ahead / parallel seeds)

Random Number Generation Algorithms

(1) Linear Congruential Generator(LCG): $X_n = (a X_{n-1} + c) \bmod M$

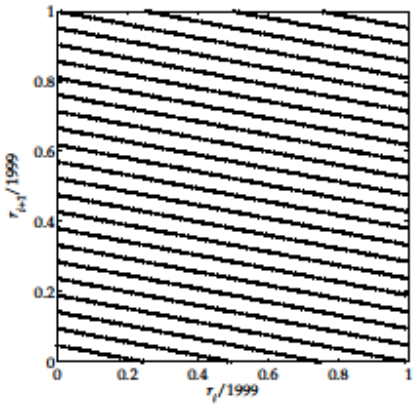
Multiplicative： c=0. → Period = 2^{j-2}

Mixed： c .ne. 0. → Period = 2^j

where $M=2^j$（usually Machine Word Size)

- The period is short (especially if j=32)
- Undesirable characteristics for certain applications

(i.e. hyperplanes in higher dimensions)



64-bit LCGs Recommended by Forrest Brown(LANL)

LA-UR-05-4983 Fundamentals of Monte Carlo Particle Transport

L'Ecuyer's 63-bit LCGs

Los Alamos

- **L'Ecuyer suggested 63-bit LCGs with good lattice structures.**
  Math. Comp., 68, 249–260 (1999)

  – Good multipliers were chosen based on the **spectral test**.

  – **Multiplicative LCGs**
    - LCG(3512401965023503517, 0, 2^{63})
    - LCG(2444805353187672469, 0, 2^{63})
    - LCG(1987591058829310733, 0, 2^{63})

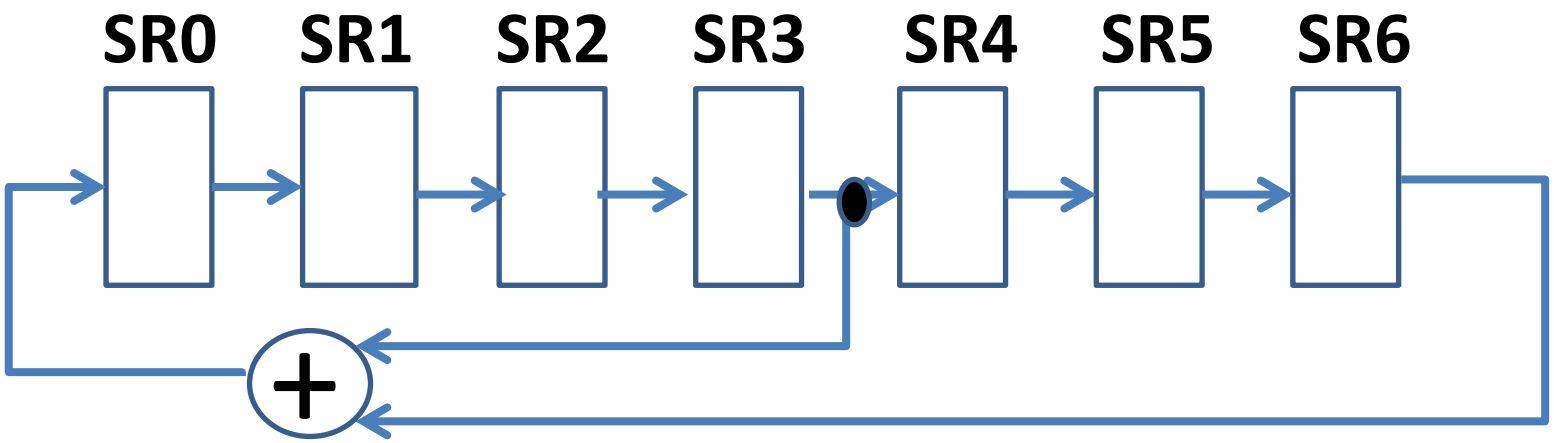
  – **Mixed LCGs**
    - LCG(9219741426499971445, 1, 2^{63})
    - LCG(2806196910506780709, 1, 2^{63})
    - LCG(3249286849523012805, 1, 2^{63})

C.E.Haynes: LCG(6364136223846793005 ,0,2^64) (Knuth, Vol.2, p.107-108)

(2) Binary M-sequence with Primitive Trinomial:

$X_n = (X_{n-m} \oplus X_{n-k}) \bmod 2 \rightarrow$ Period $= 2^k-1$ $(m<k)$

(Shift Register Sequence)



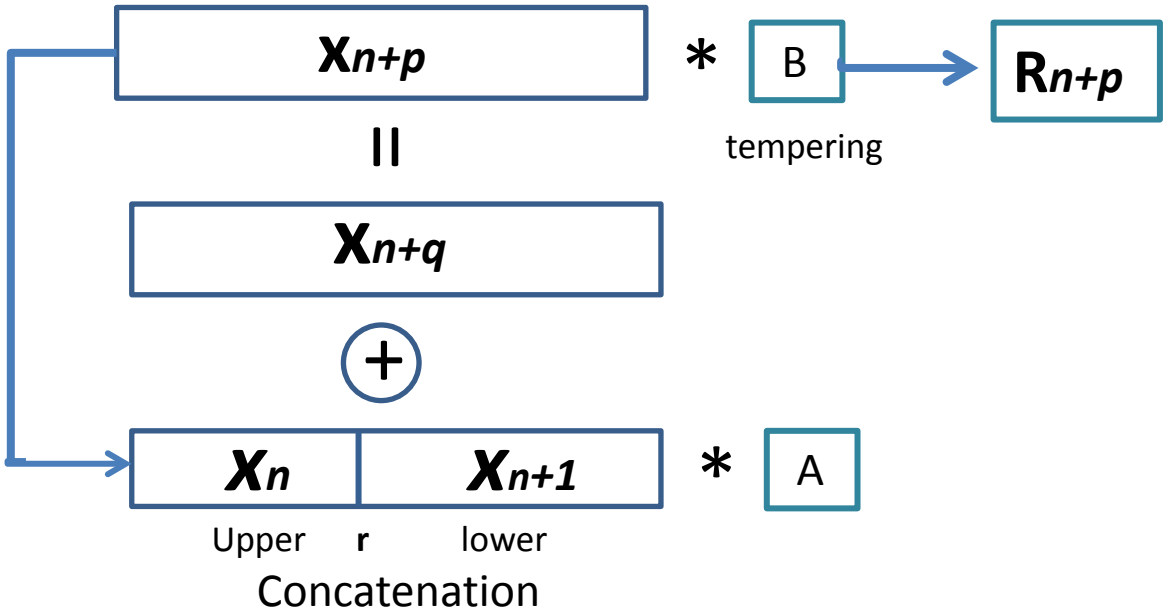
Mersenne Twister

- Evolved from a matrix formulation of M-sequence

  $X_{n+p} := x_{n+q} + (x_n \mid x_{n+1})\, A.$

- Very very long period（2^19937 －1）
- Logical operations only
- May have issues of generating the seeds for MPP (My opinion)

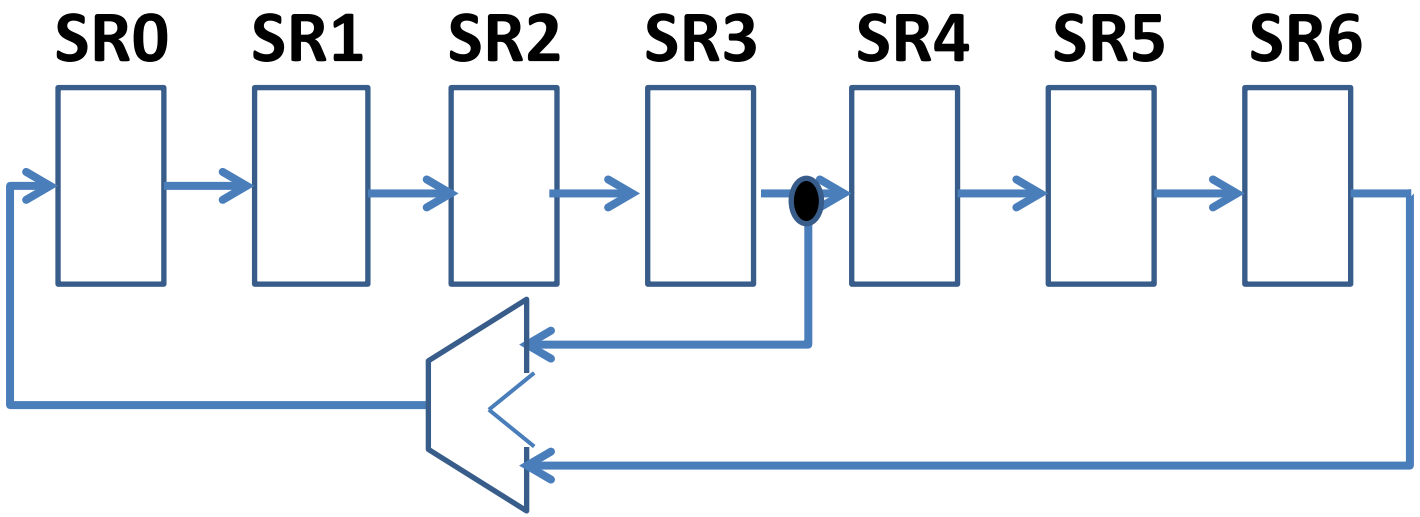
P=624
q=397
r=31



(3) Generalized Fibonacci Method with Primitive Trinomial:

$X_n = (X_{n-m}\ \text{op.}\ X_{n-k}) \bmod M \rightarrow \text{Period} = (2^k-1)2^{j-1} \sim 2^{k+j-1}$

where op. is {+, -, *}.



* Ranlux is a modification to Generalized Fibonacci Method

(4) Generalized Recursive Method With Large Prime Modulus

(Multiple Recursive Generator or MRG)

The Period is $p^k-1 \sim 2^{j*k}$,where $p \sim 2^j$ is a Prime Number (e.g., $2^{31} - 1$)

Characteristics (and Advantages) of MRG with 8^{th}-order Full Primitive Polynomials

From here on, consider $p=2^{31}-1$ and $k = 8$.

$$f(x) = (x^8 - a_1x^7 - a_2x^6 - a_3x^5 - a_4x^4 - a_5x^3 - a_6x^2 - a_7x - a_8) \ mod\ (p)$$

(1) Better chance of finding a generator which possesses good Lattice Structure with non-constrained full coefficients

$$d_t \geq 1/\mathrm{sqrt}(1 + a_1^2 + a_2^2 + a_3^2 + a_4^2 + a_5^2 + a_6^2 + a_7^2 + a_8^2)$$

(*Ref*. P.L'Ecuyer , INFORMS Journal on Computing, Vol.9, No.1,pp.57-60, 1997)

(2) Reasonably long period: $(2^{31}-1)^8 - 1 \sim 4.5*10^{74}$

(3) Ease of applying to vector/parallel processing, due to small dimension of the states

(4) Many primitive polynomials to choose from:

$\phi (p^k - 1)/k/(p^k - 1) = 2.2\%$

(5) Simple and straight-forward implementation is possible

A Sample Polynomial with Good Lattice Structure (LatMRG)

Ref: Prof. Pierre L'Ecuyer, Univ. Montreal

$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3} + a_4 x_{n-4} + a_5 x_{n-5} + a_6 x_{n-6} + a_7 x_{n-7} + a_8 x_{n-8} \quad mod(\ p\ )$$

where

a_1 = 1089656042 a_5 = 189748160

a_2 = 1906537547 a_6 = 1984088114

a_3 = 1764115693 a_7 = 626062218

a_4 = 1304127872 a_8 = 1927846343

$p = 2^{31} - 1 = 2147483647$

Figure of Merit: M32=.62729

Out of 345597 tested polynomials with a_8 as primitive roots, 31843 are primitive polynomials（9.2%）.

Other Reported Full Polynomials with Good Lattice Structure

3rd Order: $x_n = a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3} \mod(p),$ where $p = 2^{31} - 1 = 2147483647$

Grube-Dieter	Dieter	L'Ecuyer
$a_1 = 518175991$	$a_1 = 388425559$	$a_1 = 2021422057$
$a_2 = 510332243$	$a_2 = 227651891$	$a_2 = 1826992351$
$a_3 = 71324449$	$a_3 = 5412951$	$a_3 = 1977753457$

4th Order: $x_n = a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3} + a_4 x_{n-4} \mod(p),$ where $p = 2^{31} - 1 = 2147483647$

L'Ecuyer

$a_1 = 2001982722$

$a_2 = 1412284257$

$a_3 = 1155380217$

$a_4 = 1668339922$

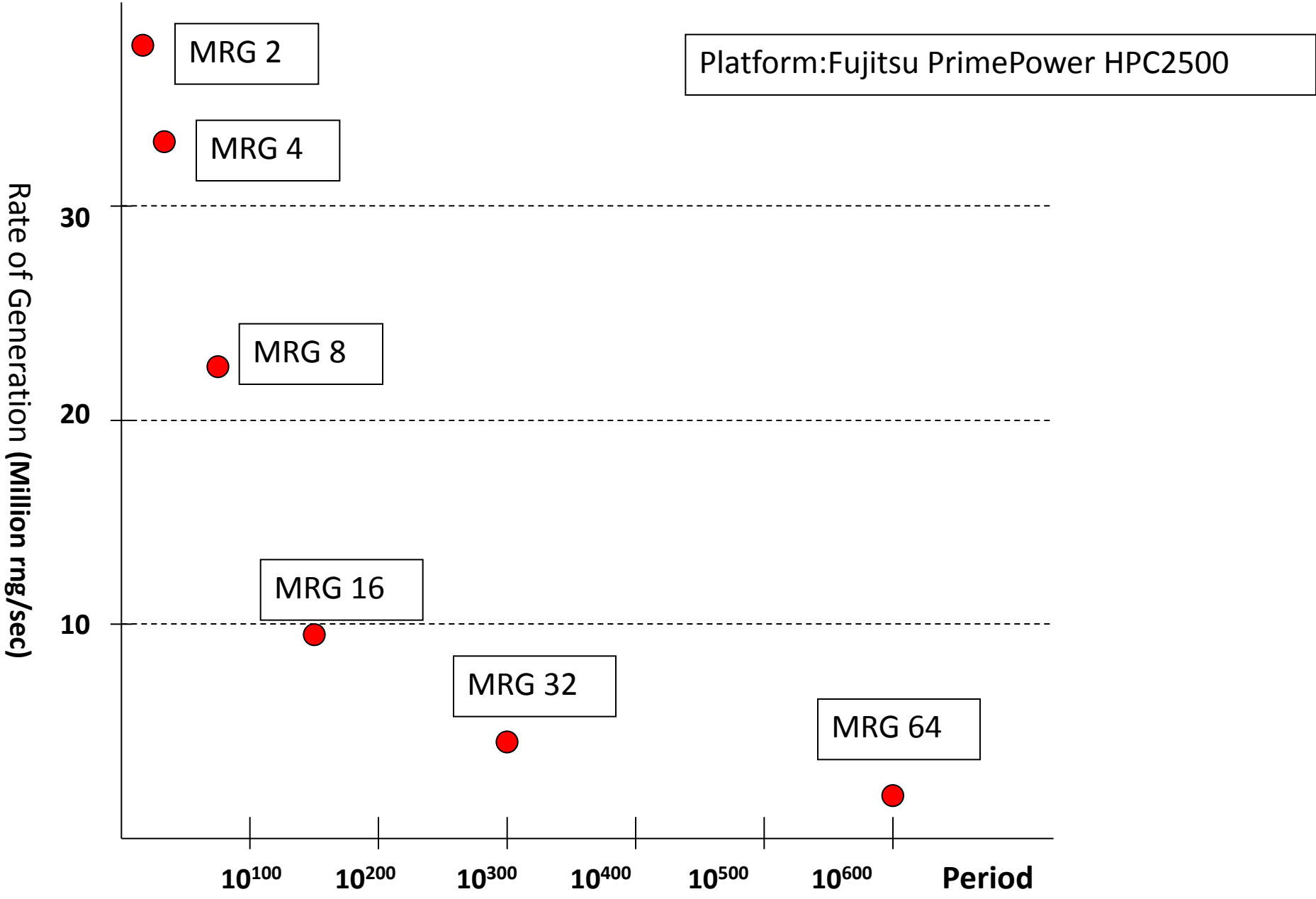
Ref. http://crypto.mat.sbg.ac.at/results/karl/server/node7.html

Performance Measurement of MRG8 on Machines with Different Architectures

As of 8/10/06
Rev. 06/14/11

System	Architecture	Clock (GHz)	Peak Performance (Gflops)	Rate of Generation (10^6 dp rng/sec)
Fujitsu VPP5000	Vector (Proprietary)	.3	9.6	201.8
NEC SX-6	Vector (Proprietary)	.5	8.0	224.3
Fujitsu PrimePower HPC2500	Scalar (Sparc64V)	1.3	5.2	23.2
IBM p-series 690	Scalar (Power4)	1.3	5.2	11.4
SGI Altix3700	Scalar (Itanium 2)	1.3	5.2	39.7
AMD Opteron	Scalar	2.0	4.0	34.0
Intel Woodcrest Xeon	Scalar (1 core)	2.66	10.6	108.1
Fujitsu Primergy RX200S2	Scalar (Xeon EM64T)	3.6	7.2	79.9
Fujitsu PrimeQuest480	Scalar (Itanium 2)	1.5	6.0	80.3
Fujitsu PrimeQuest RXI300	Scalar (Itanium2)	1.6	6.4	83.5
RIKEN/Fujitsu "K"	Scalar (SPARC64VIIIfx) (1 core)	2.0	16.0	34.5 (2 Integer Ops./clock)

Order of Recurrence vs Rate of Generation



Empirical Test of PRNG (TestU01)

- TestU01 is a set of utilities for testing RNG
  - P. L'Ecuyer and R. Simard, University of Montreal
- Three pre-defined batteries of tests
  - SmallCrush
    - 15 statistical tests, uses 2^7 random numbers
  - Crush
    - 186 statistical tests, uses 2^{35} random numbers
  - BigCrush
    - 234 statistical tests, uses 2^{38} random numbers

openpetascale LIBRARIES

BigCrush Test of PRNGs

- **SPRNG**
  - **Pass:** LFG, MLFG, CMRG
  - **Fail:** LCG, LCG64, PMLCG
- **TRNG**
  - **Pass:** LCG64_shift, MRG3, MRG4, MRG5, MRG3s, MRG5s, YARN2, YARN3, YARN4, YARN5, YARN3s, YARN5s
  - **Fail:** LCG64, MRG2, MT19937, MT19937_64z
- **Random123**
  - **Pass:** Threefry, Philox, AES NI, ARS
- **MRG8**
  - **Pass:** MRG8

Parallelization（1）
～ Random Initialization ～

Same as “Birthday Paradox” (e.g., Feller)

- Period of Random Numbers : N
  Usage of Random Numbers in each process/thread : n
  Total number of chunks which should not overlap: N/n = m
  Total Number of Cores : p

- Probability of no collision = 1・(1-1/m)・(1-2/m)・・・・(1-(p-1)/m)
  ~= 1 -1/m-2/m- ・・・・-(p-1)/m = 1 - (p-1)p/(2 m)

Probability that at least one collision occurs ~= （p-1)p/(2m)

Example : N=10^74, n=10^15, m=10^59, p=10^6 → **~ 5・10^(- 48)**

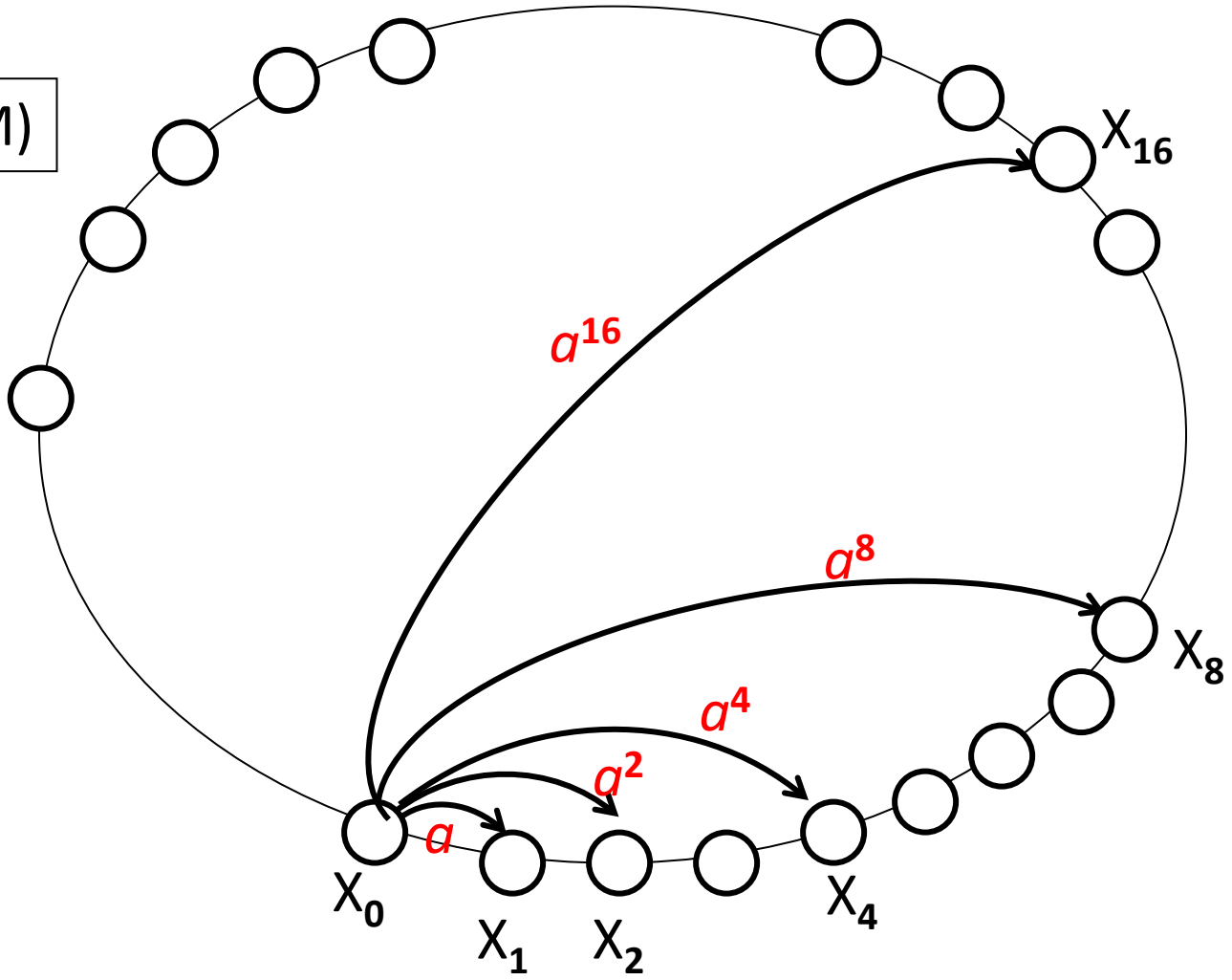
But, Donald Knuth’s advice is:

“Random number generators should not be chosen at random”

Parallelization（2）

~ Jump-Ahead (First Order Recurrence) ~

$X_{n+1} = a\, X_n \mod(M)$



Parallelization（2）
~ Jump-Ahead (the 8th Order Recurrence) ~

$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3} + a_4 x_{n-4} + a_5 x_{n-5} + a_6 x_{n-6} + a_7 x_{n-7} + a_8 x_{n-8} \quad mod(p)$$

Define a Transfer Matrix A:

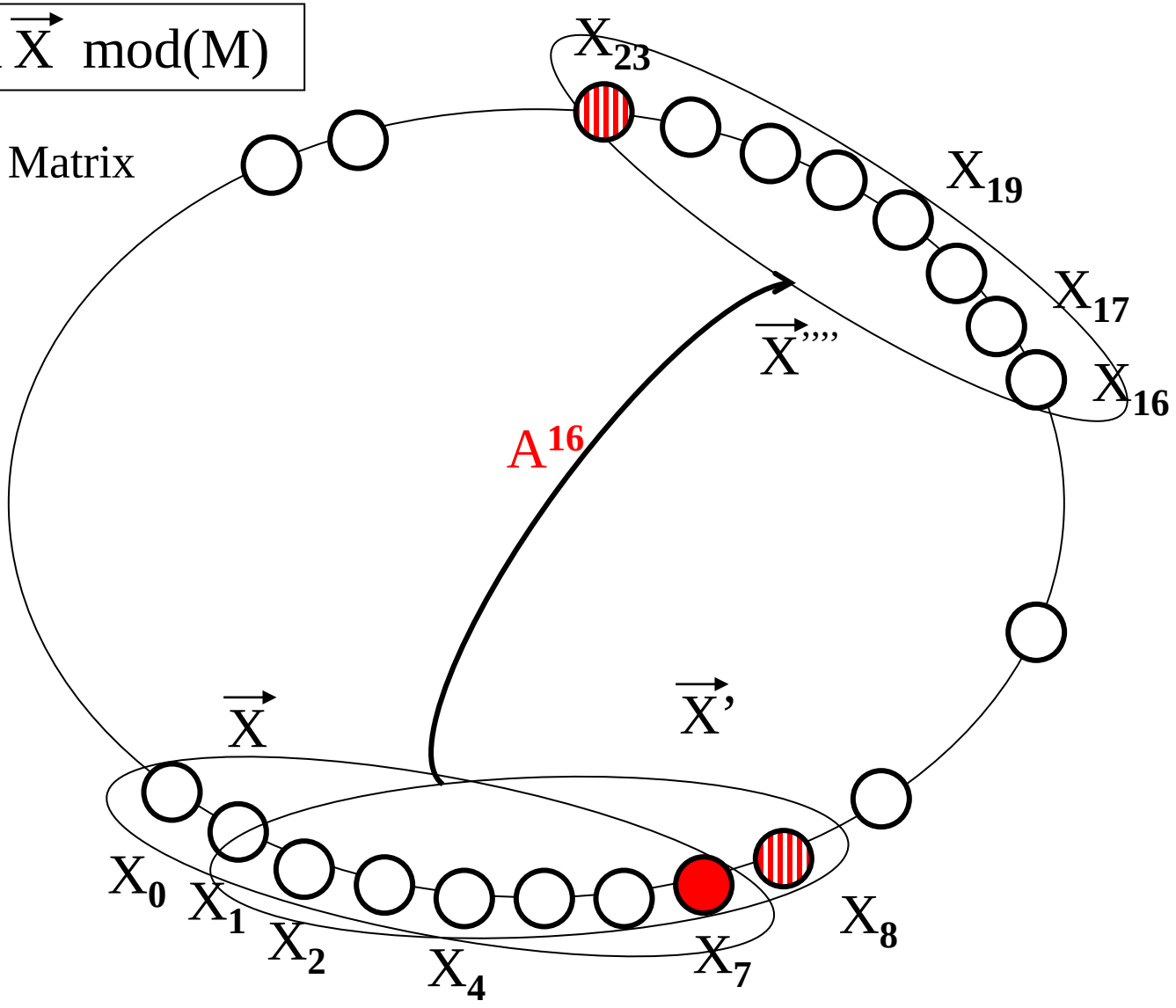
$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad x = \begin{pmatrix} x_{n-1} \\ x_{n-2} \\ x_{n-3} \\ x_{n-4} \\ x_{n-5} \\ x_{n-6} \\ x_{n-7} \\ x_{n-8} \end{pmatrix}, \quad x' = \begin{pmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ x_{n-3} \\ x_{n-4} \\ x_{n-5} \\ x_{n-6} \\ x_{n-7} \end{pmatrix}$$

Then $x' = A\,x \bmod (p)$.

Jumping Ahead the Sequence of MRG

$$\vec{X}' = A\,\vec{X} \bmod(M)$$

A: Transfer Matrix



Jump-Ahead for Arbitrary Distance

In order to compute $x_n = A^n x_0 \; mod(p)$ for an arbitrary n:

(1) Compute and store $A_j = A^{2^j} \; mod(p)$ (j=0,1,2,3,4,……).

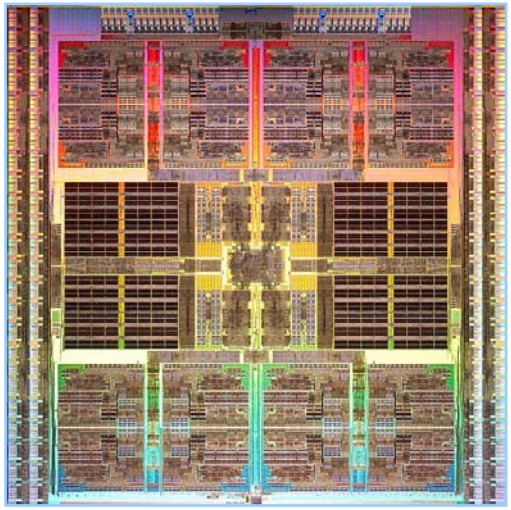
(2) Represent "n" in the binary form,

e.g., $(b_{m-1},\ldots\ldots,b_2,b_1,b_0)$.

(3) Multiply $A_j \; mod(p)$'s together only when $b_j=1$ (j=0,….,m-1), to obtain A^n.

Note: The same strategy works with the polynomial formulation with fewer arithmetic operations (Knuth).

Fujitsu SPARC64 IXfx specifications



Item	Specification
No. of cores	16
Level 2 cache	12 MB
Operating frequency	1.848 GHz
Process technology	40-nm CMOS
Die size	21.9 mm × 22.1 mm
No. of transistors	Approx. 1.87 billion
Peak performance	236 gigaflop/s
Memory bandwidth	85 GB/s (theoretical peak value)
Power consumption	110 W (process condition: TYP)

Implementation of MRG8 on Fujitsu FX10 (16 cores/node, clock:1.848 GHz)

- Parallelization for 16 threads with OpenMP
- Initialization Time = 0.208 + .696*M (□□sec)
  where M is the number of 1's in the binary representation of the Jump distance N
- Generation Time of a PRN = 0.031 (□□sec)

  => 32 Million PRN/sec/thread
- Work in Progress (MPI version across nodes)

Random Number Testing on Quantum Diffusion Monte Carlo Application

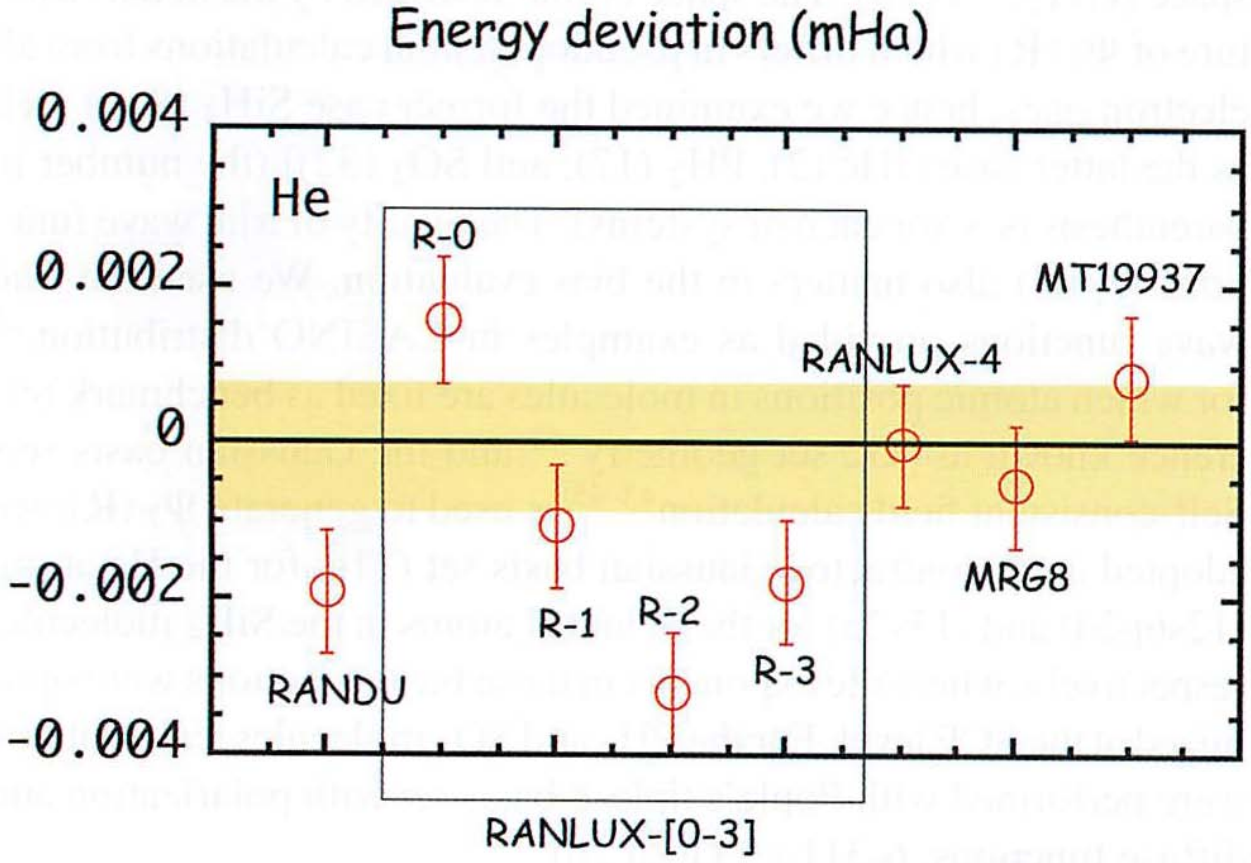


Figure 1. The ground state energies of He atom evaluated by several RNGs in VMC. Data is shown as the deviation from that by RANLUX-4.

Ref. K.Hongo, R.Maezono and K.Miura, J. Comp. Chem. **31** ,pp.2186-2194 2010

Revisiting Monte Carlo Methods

The Monte Carlo Methods (MCMs) were systematically studied in the early days of computing in various application areas.

- Elliptic Partial Differential equations
  - $\triangle u = 0$ (Laplace Equation
  - $\triangle u = f$ (Poisson's Equation)
  - $\triangle u - \kappa^2 u = 0$ (Linearized Poisson-Boltzmann Eq.)
    with u=g on the boundary (Dirichlet B.C.)

- System of Linear Equations
- Eigenvalue Problems

Example: Laplace's Equation

Potential calculation in electromagnetics etc

$$\Delta\Psi = 0 \text{ in } \Gamma$$
$$\Psi = \text{g on } \gamma$$

1. Finite Difference Method/finite Element Method
2. Boundary Element Method
3. Monte Carlo Methods
   - Walk on Spheres Method
   - Walk on Rectangles Method
   - Walk on Boundary Method

Comparison of Three Methods

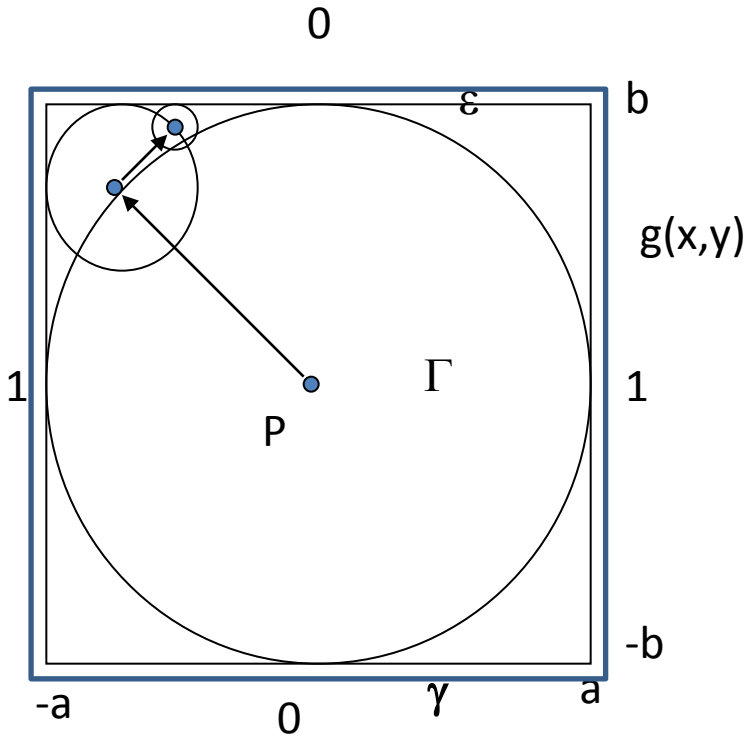
- FDM/FEM → All the values on the grid need to be calculated
- BEM → The boundary equation has to be re-calculated everytime the boundary values are changed
- MCM →
  - There is no Grid points (Direct calculation)
  - even if the boundary values are dynamically modified, calculation can be done immediately

Example: Laplace's equation (cont.)

No need to perform random walks on grid points

→Walk on Spheres (WOS) Method*

$$\Delta \Psi = 0 \text{ in } \Gamma$$
$$\Psi = g \text{ on } \gamma$$



ε: Error Bound

* **Some Continuous Monte Carlo Methods for the Dirichlet Problem**

Mervin E. Muller

The Annals of Mathematical Statistics. Vol. 27. No. 3 (Sep., 1956), 569-589.

Example: (x=0, y=0, ε =.0001)

No. of Walks	Solution	Final Radius	No. of RNs
10,000	.5098	1.3661 10^{-5}	121,291
40,000	.50265	1.06 012 10^{-5}	484,677
1,000,000	.499734	1.71094 10^{-6}	12,083,215
Exact	.50000		

$\varepsilon = 10^{-4}$: takes ~12 Walks on the average
(Theoretically proportional to log(ε)

Conclusion

- With the recent trends in HPC architecture in the Million-core era, some of the traditional numerical algorithms need to be reconsidered due to their poor scalability.
- Monte Carlo Methods can be used to efficiently solve wider class of problems
- Random number support for massively parallel processing is essential.

Acknowledgement

The author would like to thank

- Prof. P.L'Ecuyer : Sample Primitive Polynomials
- Dr. C.Chen at Fujitsu Computer System :
  Measurement on VPP5000, Primergy, PrimeQuest
- Dr. M.Kurokawa, RIKEN:
  Measurement on SX-6, Woodcrest
- Mr. Hayakawa, AMD Japan:
  Measurement on AMD Opteron
- Mr. Hotta, Fujitsu Limited (Makuhari):
  Measurement on Sparc64VIIIfx
- Mr. Yamanaka and Mr.Takeshige, Fujitsu Limited:
  Parallelization of MRG8 and measurement on FX10


Thank you !