Class of scalable parallel and vectorizable pseudorandom number generators based on non-cryptographic RSA exponentiation ciphers

Paul D. Beale* and Jetanat Datephanyawat

University of Colorado Boulder

(Dated: November 29, 2018)

Parallel supercomputer-based Monte Carlo and stochastic simulatons require pseudorandom number generators that can produce distinct pseudorandom streams across many independent processes. We propose a scalable class of parallel and vectorizable pseudorandom number generators based on a non-cryptographic version of the RSA public-key exponentiation cipher. Our method generates uniformly distributed IEEE double precision floating point pseudorandom sequences by encrypting pseudorandom sequences of 64-bit integer messages by modular exponentiation. The advantages of the method are: the method is parallelizable by parameterization with each pseudorandom number generator instance derived from an independent composite modulus, the method is fully scalable on massively parallel computing clusters because of the millions of 32-bit prime numbers, the seeding and initialization of the independent streams is simple, the periods of the independent instances are all different and greater than 8.5×10^{37}, and the method passes a battery of intrastream and interstream correlation tests. The calculations in each instance can be vectorized. and can produce more than 10^8 pseudorandom numbers per second on each multicore CPU.

PACS numbers: 02.70.-c, 05.10.-a, 05.10.Gg, 05.10.Ln, 05.40.-a, 07.05.Tp, 95.75.Wx

I. INTRODUCTION

Parallel supercomputer-based Monte Carlo and stochastic simulatons require pseudorandom number generators that can produce distinct pseudorandom streams across many independent processes. We have developed a new class of scalable parallel and vectorizable pseudorandom number generators for use in massively parallel supercomputer applications. The method we propose is based on a non-cryptographic version of the Rivest, Shamir and Adleman (RSA) public key exponentiation cipher.[1–6] The method creates pseudorandom streams by encrypting sequences of 64-bit integer plaintext *messages* m_k into *ciphertexts* c_k using the transformation

$$c_k = m_k^e \bmod n. \tag{1}$$

Each generator instance is based on an independent composite modulus $n = p_1 p_2$, where p_1 and p_2 are 32-bit safe primes, and the exponent e is a small odd number. Here and throughout, $x = y \bmod z$ means x is the remainder of y upon division by z, with $0 \le x < z$. This is *not* a cryptographically secure pseudorandom generator, which would need to operate on integers that are thousands of bits long. The algorithm is fully scalable by parametrization on parallel supercomputers since each node can be assigned independent pairs of primes. The algorithm is vectorizable, and can generate more than 10^8 pseudorandom numbers per second on each multicore supercomputer node.

The pseudorandom number generator algorithm described here cycles through a sequence of integer messages m_k with $k = 0, 1, 2, \ldots$ uniformly selected from $\mathbb{Z}_n = [0 \ldots n-1]$. The encryption step in equation (1) then gives a sequence of pseudorandom ciphertexts c_k that is uniformly distributed on $\mathbb{Z}_n$. Uniformly distributed double precision floating point pseudorandom numbers r_k on the real interval $[0, 1)$ are formed with a floating point division: $r_k = c_k/n$. Since $(abc) \bmod n = (a (bc \bmod n)) \bmod n$, repeated squaring and multiplying can be used to evaluate the exponentiation (1) with less than $2 \log_2 e$ modular multiplications on $\mathbb{Z}_n$.

Most pseudorandom number generators generate the next pseudorandom integer from either the previous pseudorandom integer in the sequence, or by operating on two or more pseudorandom integers from earlier in the sequence. In our method, the pseudorandom sequence arises from the encryption of a sequence of integer messages. In this way, it is similar to cryptographically secure pseudorandom number generators,[2,3] and pseudorandom number generators based on block ciphers.[7–10] The quality of the pseudorandom sequence produced by our method is based on modular exponentiation being a good one-way cryptographic function.[1–3,11]

Large-scale parallel programs that use pseudorandom numbers should utilize an algorithm that is scalably parallel. Otherwise, different processes risk sampling overlapping pseudorandom subsequences which would give results that are not statistically independent. Coddington[12] recommends parallel pseudorandom number generators should have the following characteristics (paraphrased here):

- The method should produce uncorrelated pseudorandom numbers in any number processes, and pass a battery of stringent tests of randomness within each process and between processes.

- The algorithm should have a provably long period that is far longer than a single process can exhaust in any conceivable run.

- The method should be able to create one instance, or a scalably large number of independent instances.

- To assist in debugging, the user should be able to seed the algorithm to give independent reproducible results in any number of processes

- The code should be portable across wide range of computer architectures.

- The generator should have fast initialization and execution speeds that use limited memory, and each process should run independently once initialized.

Two qualitatively different schemes have been used to create scalable systems of pseudorandom number generators: stream splitting and parameterization.[13] Parallelization by stream splitting is based on a single pseudorandom number generator with an extremely long period, with parallelization accomplished by subdividing the full period into non-overlapping subsequences. By contrast, parallelization by parameterization produces independent pseudorandom sequences by assigning different fixed parameters to each process. The most widely used classes of parallel pseudorandom number generators are based on the lagged Fibonacci method.[14,15,17–20] The algorithm is usually of the form $s_k = (s_{k-q} \odot s_{k-r}) \bmod 2^m$, where $\odot$ is one of the operations bitwise exclusive or, addition, subtraction, or multiplication, where $q < r$ are integer parameters chosen based on primitive polynomials modulo 2 that implement a Galois field of order 2^r.[20,21] The bitwise exclusive-or algorithms with $m = 1$ have periods of $P = 2^r - 1$, additive, subtractive, and word-wise exclusive-or algorithms have periods of $P = (2^r - 1)2^{m-1}$, and multiplicative algorithms have periods of $P = (2^r - 1)2^{m-2}$. The parameter r is typically chosen in the range of several hundred to several thousand. The state of the generator is defined by a table of r m-bit integers, which represent the most recent pseudorandom integers in the sequence. For $m > 1$, parallel implementation of these algorithms can be accomplished by either stream splitting or parameterization.[14,15,17,18]

Our method is parallelized by parameterization by assigning a unique modulus $n = p_1 p_2$ to each process. The number of independent streams is limited only by the number of prime pairs in the range defined by the implementation. Each independent generator can vectorized by simultaneously calculating a vector of pseudorandom numbers, which greatly speeds the calculation on vector processors or multicore CPUs.

II. RSA PUBLIC KEY ENCRYPTION

Asymmetric or public-key cryptography was first publicly proposed Ralph Merkle.[22] Whitfield Diffie and Martin Hellman[23] were the first to publish a practical algorithm for key exchange based on modular exponentiation in a prime field, and Ron Rivest, Adi Shamir and Leonard Adleman[1] published their public key cryptosystem paper in 1978. (It is important to note that all of these methods were discovered earlier by British GCHQ mathematicians in highly classified work: James Ellis proposed the idea of asymmetric ciphers, or what he called *non-secret encryption*, Malcolm Williamson developed a key exchange method identical to Diffie–Hellman, and Clifford Cocks developed a version of the RSA algorithm. These discoveries were not revealed publicly until their work was declassified in 1997.[24])

Our generator mimics the manner that RSA is used in practice to establish a secure communications channel between Alice and Bob.[2] Alice wants to send an encrypted message to Bob even though they have never met to securely exchange a secret symmetric encryption/decryption key. Bob first creates a public key consisting of a composite number $n = p_1 p_2$ where p_1 and p_2 are two large secret prime numbers. Bob publicly shares the product n and a small exponent e that is coprime to $p_1 - 1$ and $p_2 - 1$. Alice uses Bob's public key to encrypt her message m into ciphertext $c = m^e \bmod n$, and sends the ciphertext to Bob over an open channel. Bob can decrypt the ciphertext using $m = c^d \bmod n$, where Bob's decryption exponent d can be determined from e and the two secret primes. RSA is usually used to securely share a secret symmetric key K. Alice encrypts a random key K using Bob's public key, and sends the ciphertext to Bob who decrypts the key. Both Alice and Bob can then use their shared secret key K in a fast symmetric encryption algorithm.

Alice and Bob assume that an eavesdropper Eve will be able to intercept the ciphertext c. The security of RSA is based on both modular exponentiation mod n, and multiplying p_1 and p_2 to determine n are both good one-way functions, i.e. multiplying and exponentiating are easy, while factoring and solving the discrete logarithm problem are hard.[2–5] *Easy* and *hard* are distinguished by whether or not a function can be calculated in a time proportional to a power of $\log_2 n$ (*polynomial time*). Cryptographic security currently requires that n should be thousands of bits long. No known classical algorithm can factor a large composite or calculate a discrete logarithm in a polynomial time. It is these problems that a successful quantum computer with thousands of qubits could potentially crack.[25]

III.   NUMBER THEORY

For every prime number p, the set of integers $\mathbb{Z}_p = [0 \ldots p-1]$ forms a finite field, i.e. $\mathbb{Z}_p$ is closed under addition and subtraction modulo p, and the set of nonzero elements $\mathbb{Z}_p^* = [1 \ldots p-1]$ forms a group that is closed under multiplication and division modulo p. Division is defined since for every integer $a \in \mathbb{Z}_p^*$ there exists a unique multiplicative inverse $a^{-1} \in \mathbb{Z}_p^*$ such that $aa^{-1} \bmod p = 1$.

The message m can be decrypted from the ciphertext c using a decryption exponent d:[1,3–6]

$$c = m^d \bmod n. \tag{2}$$

The decryption exponent d exists and is unique if e and $(p_1 - 1)(p_2 - 1)$ are co-prime. Decryption is based on Fermat's little theorem:[4–6] for any prime p and for all $m \in \mathbb{Z}_p^*$, $m^{p-1} \bmod p = 1$. For the case of composite moduli of the form $n = p_1 p_2$, the generalization of Fermat's little theorem is for all m co-prime to n, $m^{\phi(n)} \bmod n = 1$, where $\phi(n) = (p_1-1)(p_2-1)$ is Euler's totient function, the number of elements in $\mathbb{Z}_n^*$ that are coprime to n. The decryption exponent is given by

$$d = e^{-1} \bmod \phi(n) = e^{-1} \bmod (p_1 - 1)(p_2 - 1), \tag{3}$$

and can be calculated using the extended Euclidean algorithm by anyone who knows e, p_1 and p_2.[3–6,26] The validity of equation (2) is demonstrated as follows:

$$c^d \bmod n = m^{de} \bmod n = m^{1+u\phi(n)} \bmod n = (m(m^{\phi(n)})^u) \bmod n = m \bmod n = m. \tag{4}$$

The decryption in equation (2) works for all $c \in \mathbb{Z}_n$. Encryption and decryption are one-to-one mappings of $\mathbb{Z}_n$ onto the same set, so any message sequence $\{m_k\}$ that uniformly samples $\mathbb{Z}_n$ will produce a ciphertext sequence $\{c_k\}$ that uniformly samples $\mathbb{Z}_n$.

The Chinese remainder theorem (CRT)[2–6] is used to speed up the modular exponentiations modulo n in RSA cryptographic applications. Every integer $m \in \mathbb{Z}_n$ with $n = p_1 p_2$ can be uniquely represented in terms of two numbers $m_1 \in \mathbb{Z}_{p_1}$ and $m_2 \in \mathbb{Z}_{p_2}$ given by

$$m_1 = m \bmod p_1, \tag{5a}$$
$$m_2 = m \bmod p_2, \tag{5b}$$

and the value of m can be recovered from m_1 and m_2 using Garner's formula:

$$m = (((m_1 - m_2)(p_2^{-1} \bmod p_1)) \bmod p_1)p_2 + m_2. \tag{6}$$

The exponentiation $c = m^e \bmod n$ can then be accomplished by calculating $c_1 = c \bmod p_1$ and $c_2 = c \bmod p_2$ by exponentiating m_1 and m_2:

$$c_1 = m_1^e \bmod p_1, \tag{7a}$$
$$c_2 = m_2^e \bmod p_2, \tag{7b}$$
$$c = (((c_1 - c_2)(p_2^{-1} \bmod p_1)) \bmod p_1)p_2 + c_2. \tag{7c}$$

Since we choose p_1 and p_2 to be 32-bit primes, the exponentiations can be accomplished using native 64-bit arithmetic. In RSA cryptographic applications, this CRT-based speedup can only be used in the decryption step, since only Bob knows p_1 and p_2.

IV.   PSEUDORANDOM SKIPS

The sequence of messages to be encrypted can be expressed in terms of an integer skip sequence $\{s_k\}$ chosen from $\mathbb{Z}_n$:

$$m_k = (m_{k-1} + s_k) \bmod n, \tag{8}$$

Note that if the skip sequence uniformly and *randomly* (not pseudorandomly) samples $\mathbb{Z}_n$, then the sequence of messages m_k forms a uniform random sequence on $\mathbb{Z}_n$. Each message is, in effect, a one-time pad encryption of the previous message.[3] We will approximate this by choosing the skips s_k pseudorandomly.

In encryption it is essential to avoid *cribs*, i.e. messages that result in easily decoded ciphertexts. For example, the messages $m = 0, 1, n-1$ are cribs for all allowed exponents e since $m^e \bmod n = 0, 1, n-1$, respectively. RSA-based cryptographic applications often use encryption exponents as small as $e = 3, 5$ for efficiency.[2] Messages with $m^e < n$ and $(n-m)^e < n$ result in trivially decodable ciphertexts, so exponents $e < \log_2 n$ result in additional cribs. In cryptographic applications, messages are randomly padded[2,3] to avoid cribs. For our purposes, it is not necessary to eliminate cribs, since they would appear in any long random sequence of messages, but rather to prevent correlated sequences of cribs. Our goal is to select a simple skip pattern that ensures a uniform sampling of the set of all messages, avoids correlated cribs, is computationally fast, has a long period, and allows the use of small encryption exponents.

The simplest skip sequence that uniformly samples $\mathbb{Z}_n$ is the unit skip, i.e. $s_k = 1$ for all k. Encryption of this sequence gives a block cipher operating in counter mode which is used in some cryptographically secure pseudorandom number generators.[2,3] The message sequence is $m_k = (m_0 + k) \bmod n$, with period $P = n$. In spite of the cribs near $m = 0$ and n, pseudorandom sequences derived from a unit skip empirically pass the U01 battery of statistical correlations tests[10] for exponents $e \geq 9$. A constant skip $s_k = b$ with $1 < b < n-1$, can eliminate sequential cribs. but the pattern produced by constant skips is not substantially better than the unit skip pattern since

$$(kb)^e \bmod n = ((b^e \bmod n)(k^e \bmod n)) \bmod n, \tag{9}$$

is just a constant multiplier permutation of the unit skip sequence.

We choose a skip sequence produced by a prime number linear congruential pseudorandom number generator:[20,29–31]

$$s_k = a s_{k-1} \bmod q = s_0 a^k \bmod q, \tag{10}$$

with prime modulus q. The multiplier a is chosen to be a primitive root $\bmod q$[4–6] that delivers a full period, well-tested pseudorandom sequence. The period of the skip generator is $q-1$ since every skip in Z_q^* will appear once and only once before the skip sequence repeats. If n is co-prime to q and $q-1$, message/skip sequence does not begin to repeat until after period $P = (q-1)n$ steps. Since a is a primitive root $\bmod q$, after $q-1$ steps s_k will have cycled through every value in $\mathbb{Z}_q^*$, so $s_{k+q-1} = s_k$ and $m_{k+q-1} = \left(m_k + \sum_{s=1}^{q-1} s \right) \bmod n = (m_k + q(q-1)/2) \bmod n$. If $\gcd(q(q-1)/2, n) = 1$, then $b = q(q-1)/2 \bmod n \neq 0$. Therefore the skip and message values shifted forward by $(q-1)u$ steps are given by $s_{k+(q-1)u} = s_k$ and $m_{k+(q-1)u} = (m_k + ub) \bmod n$. Therefore, every subsequence of messages of length $q-1$ is different from every other subsequence. Since $s_{k+(q-1)n} = s_k$ and $m_{k+(q-1)n} = m_k$, the period of the generator is $P = (q-1)n$. Using Fermat's little theorem, the state of the generator after $k = u(q-1)+v$ steps, with $u = \lfloor k/(q-1) \rfloor$ and $v = k \bmod (q-1)$, is given by

$$s_k = s_0 a^k \bmod q = s_0 a^v \bmod q = a^{v_0+v} \bmod q, \tag{11a}$$

$$m_k = \left(m_0 + \sum_{j=1}^{k} s_0 a^j \bmod q \right) \bmod n,$$

$$= \left(m_0 + ub + \sum_{j=1}^{v} a^{v_0+j} \bmod q \right) \bmod n, \tag{11b}$$

$$c_k = m_k^e \bmod n, \tag{11c}$$

where $s_0 = a^{v_0} \bmod q$. Even though the message and ciphertext sequence can be expressed in closed form, calculating the values of m_k and c_k for large k requires $O(q)$ steps unless k is close to a multiple of $q-1$. Likewise, determining the value of $k = (q-1)u+v$ that gives a particular state (m, s) requires $O(q)$ steps. The method has the following properties:

- The algorithm is based on elementary number theory and cryptography, and satisfies all of Coddington's criteria.[12]

- Using a pseudorandom skip extends the period of the generator to $P = (q-1)n$, and provides a uniform sampling of ciphertexts over the full period of the generator. Each message $m \in \mathbb{Z}_n$, and hence each ciphertext $c \in \mathbb{Z}_n$, will appear exactly $q-1$ times in the full-period sequence.

- The method is parallelizable by parameterization, with each independent process derived from a distinct composite modulus $n = p_1 p_2$. The method is fully scalable on massively parallel supercomputers due to the millions of 32-bit primes.

- Pseudorandom sequences that result from different moduli are independent, and have different periods.

- The seeding and initialization of the independent streams is simple.

- The state of each generator is defined by a few fixed integer parameters $\{n = p_1 p_2, e, q, a\}$, and a few integer state values $\{m, s, c\}$ that change during each call to the generator.

- By using 32-bit primitive roots mod q for the skip generator, 32-bit primes p_1 and p_2, the Chinese Remainder Theorem, and small exponents e, the implementation below requires only a few native 64-bit integer operations per pseudorandom number.

- The calculation in each independent process can be vectorized by operating simultaneously on vectors of messages and skips that belong to widely separated subsequences.

- If needed, one can use equation (11) to jump far forward in the sequence, as long as the jump distance is close to a multiple of $q - 1$.

- The method passes a battery of strong randomness tests, within each stream and between streams.

V.   IMPLEMENTATION IN 64 BITS

We choose the pseudorandom skip modulus to be $q = 2^{63} - 25$, the largest prime less than 2^{63}. Equation (10) can be implemented using 64-bit arithmetic if the primitive root is chosen from the restricted set of values $a < \sqrt{q}$.[27–29] This can be shown by expressing q in the form $q = a q_1 + q_2$, where $q_1 = \lfloor q/a \rfloor$ and $q_2 = q \bmod a$:

$$
\begin{aligned}
as \bmod q &= (as - \lfloor s/q_1 \rfloor q) \bmod q \\
&= (as - \lfloor s/q_1 \rfloor (a q_1 + q_2)) \bmod q \\
&= (a(s - \lfloor s/q_1 \rfloor q_1) - q_2 \lfloor s/q_1 \rfloor)) \bmod q \\
&= (a(s \bmod q_1) - q_2 \lfloor s/q_1 \rfloor)) \bmod q.
\end{aligned} \tag{12}
$$

If $2^{31} < a < \sqrt{q}$, then a, q_1, and q_2 are all 32-bits, and intermediate results $s_1 = a(s \bmod q_1)$ and $s_2 = q_2 \lfloor s/q_1 \rfloor$ in the final line of equation (12) are both less than q. L'Ecuyer, Blouin, and Couture,[30] and Sezgin and Sezgin[31] give a handful of restricted primitive roots $a < \sqrt{q}$ that have good spectral test properties and pass all U01 Crush and BigCrush tests.[10] By using restricted primitive roots, 32-bit primes, and the Chinese Remainder Theorem, the entire RSA calculation can be implemented using fast native unsigned 64-bit integer arithmetic. The algorithm uses three pre-calculated values $q_1 = \lfloor q/a \rfloor$, $q_2 = q \bmod a$, and $p_2^{-1} \bmod p_1$. The pseudocode for generating the next double precision floating point pseudorandom number r on the interval $[0, 1)$ is given by:

$$
\begin{aligned}
s_1 &:= a\,(s \bmod q_1), & \text{(13a)} \\
s_2 &:= q_2 \lfloor s/q_1 \rfloor, & \text{(13b)} \\
s &:= (s_1 - s_2) \bmod q, & \text{(13c)} \\
m_1 &:= (m_1 + s) \bmod p_1, & \text{(13d)} \\
m_2 &:= (m_2 + s) \bmod p_2, & \text{(13e)} \\
c_1 &:= m_1^e \bmod p_1, & \text{(13f)} \\
c_2 &:= m_2^e \bmod p_2, & \text{(13g)} \\
c &:= \left(\left((c_1 - c_2)(p_2^{-1} \bmod p_1) \right) \bmod p_1 \right) p_2 + c_2, & \text{(13h)} \\
r &:= c/n. & \text{(13i)}
\end{aligned}
$$

Care needs to be taken in steps (13c) and (13h) to avoid negative intermediate results and, since $n > 2^{53}$, step (13i) needs to include a test to avoid returning the upper limit $r = 1.0$ in the IEEE double precision floating point format.

There are 98,182,656 primes in the range $[2^{31}, 2^{32}]$,[35] but cryptography theory suggests choosing p_1 and p_2 from the set of *safe primes*, i.e. primes p for which $(p-1)/2$ is also prime.[3,32] Since the 64-bit ciphertexts are determined by $c_1 \in \mathbb{Z}_{p_1}$ and $c_2 \in \mathbb{Z}_{p_2}$ using Garner's formula, we performed many statistical tests using 32-bit prime moduli $n = p$.[33,34] Most primes work fine, but occasionally primes in which $p - 1$ contains only small factors fail some of the U01 tests. For those two reasons, we recommend using only safe primes to construct $n = p_1 p_2$. This does not seriously limit the scalability of our generator since there are 3,060,794 safe primes in the range $[2^{31}, 2^{32}]$,[36] so there

are 4.68×10^{12} safe prime pairs. Using safe primes has the added advantage that every small odd exponent e is coprime to $\phi(n)$.

Choosing $n \approx q$ has the advantage that D-dimensional message sequence $\{m_k, m_{k+1}.., m_{k+D}\}$ samples $\mathbb{Z}_n^D$ nearly uniformly, so each message is a pseudorandom one-time pad-like encryption of the previous message. This choice further reduces the total number of moduli available, but there are still over ten million independent values of n that differ from q by less than one part in a million. The full period of each generator is then $P = (q-1)n \approx 2^{126} \approx 8.5 \times 10^{37}$.

One can use fast primality tests to select the primes p_1 and p_2. The Rabin-Miller test,[2,3,6,37,38] which is the same as Algorithm P in Knuth,[20] provides a simple probabilistic test for primality. Every odd prime $p = 1 + 2^u t$ with t odd satisfies one of the following conditions for every base $g \in \mathbb{Z}_p^*$: either $g^t \bmod p = 1$, or $g^{2^j t} \bmod p = p - 1$ for some some j in the range $0 \le j < u$. A composite modulus n satisfying these criteria is called a strong pseudoprime to base g. For any odd composite, the number of bases for which n is a strong pseudoprime is less than $n/4$. If the test is applied repeatedly with M randomly chosen bases in $\mathbb{Z}_n^*$, the probability that a composite will pass every test is less than 4^{-M}.[6,20,37,38] Better yet, the Rabin-Miller test can deterministically identify all primes below 2^{64}. There are no composite numbers below 2^{64} that are strong pseudoprimes to all of the twelve smallest prime bases ($g = 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37$).[39–43] Therefore, any number less than 2^{64} that passes the Rabin-Miller test for all twelve of these bases is prime. Likewise, any number less than 2^{32} that passes the Rabin-Miller test for all of the five smallest prime bases ($g = 2, 3, 5, 7, 11$) is prime. For efficiency, one first checks to see if any small primes divide the modulus before applying the Rabin-Miller test.

VI. PARALLELIZATION AND VECTORIZATION

In a multiprocessor supercomputer environment, independent parallel pseudorandom streams can be created by assigning distinct parameters p_1, p_2, and a to each process. If the number of choices for p_1, p_2, and a are N_{p_1}, N_{p_2}, and N_a, respectively, each process can be assigned a unique parameter $0 \le \beta < N_{p_1} N_{p_2} N_a$ based on a time stamp t and process identifier α. For example, if we label the M_p processes with identifiers $\alpha = 0, .. M_p - 1$, we can choose $\beta = (t + \alpha\sigma)^\epsilon \bmod (N_{p_1} N_{p_2} N_a)$ where σ is an integer close to $\lfloor (N_{p_1} N_{p_2} N_a)/M_p \rfloor$ that is coprime to $N_{p_1} N_{p_2} N_a$, and exponent ϵ that is coprime to $\phi(N_{p_1} N_{p_2} N_a)$. One can then choose p_1 in process α to be the $\beta \bmod N_{p_1}$-st prime, etc. Message Passing Interface (MPI) calls can be used to initialize the M_p parallel processes with independent parameters.

Algorithm (13) can be vectorized in each process. Vectors of messages $\boldsymbol{m}$ and skips $\boldsymbol{s}$, with M_v 64-bit elements each, can be can be updated simultaneously with fixed parameters $n = p_1 p_2$, e, q, and a to return a vector of pseudorandom reals $\boldsymbol{r}$. The vector pseudocode is

$$\boldsymbol{s} := a\boldsymbol{s} \bmod q, \tag{14a}$$

$$\boldsymbol{m} := (\boldsymbol{m} + \boldsymbol{s}) \bmod n, \tag{14b}$$

$$\boldsymbol{r} := (\boldsymbol{m}^e \bmod n)/n. \tag{14c}$$

The elements of the vectors can be updated simultaneously in parallel on a vector processor, or the calculation can be shared among multiple cores available to each process using Open Multiprocessing (OpenMPI). As described above, we can use restricted primitive roots and the CRT to update the vectors using only 64-bit arithmetic. Our state vector $(\boldsymbol{m}, \boldsymbol{s})$ consists of only $2M_v$ 64-bit words. By contrast, vectorizing lagged Fibonacci generators with recursion relation $s_k = (s_{k-q} \odot s_{k-r}) \bmod 2^m$ requires much larger state vectors of rM_v m-bit words.

To ensure that the M_v sub-streams labelled $\gamma = 0, 1, .., M_v - 1$ sample greatly separated subsequences, we use the jump ahead property of the skip generator $s_k = s_0 a^k \bmod q$ to widely distribute the skips across the period of the skip generator. This can be accomplished by setting initial values of the elements of the skip vector to be $s_0^{(\gamma)} = s_0^{(0)} a^{\gamma \lfloor (q-1)/M_v \rfloor} \bmod q$. The skip sequences in successive sub-streams will not begin to overlap until the vector has been updated about $\lfloor (q-1)/M_v \rfloor$ times, i.e. after a total of about $q - 1$ pseudorandom numbers have been generated. Since $q \approx 2^{63}$, this would take hundreds of years for one node to accomplish. Even after the skip sequences in the sub-streams begin to overlap, it is very improbable that any two message subsequences become synchronized since that probability is of the order of M_v/n. (Note that equation (11) allows one to subdivide the period $P = (q-1)n$ into n subsequences, each with length $q - 1$. However, in that case the skips in each subsequence are the same, and the messages in different subsequences differ by multiples of $b = q(q-1)/2 \bmod n$. We recommend avoiding the use such strongly correlated message patterns.)

We developed and tested our code on the University of Colorado Boulder Summit supercomputer, which uses 2.50GHz Intel Xeon E5-2680 v3 processors and 24 cores per node.[51] We tested the speed of the code by averaging sequences of 10^9 pseudorandom numbers, and use OpenMP to share the calculation across multiple cores on each node. By assigning all 24 cores to each process, the code generates more than 10^8 pseudorandom numbers per second per process for exponents as large as $e = 257$. We recommend exponents $e \le 17$, whose 24 core speed ranges from

1.25×10^8 to 1.72×10^8 pseudorandom numbers per second per process. Eight core speeds range from 5.6×10^7 to 9.3×10^7 pseudorandom numbers per second per process By comparison, the highly optimized (but not necessarily scalable parallel or vectorized) pseudorandom number generators in the Intel MKL library deliver from 2.7 to 6.0×10^8 pseudorandom numbers per second per process.

VII. TESTS

We applied the well-established pseudorandom number test suites DIEHARD,[44] NIST,[45] and TestU01,[10] to ensure the generator passes a wide variety of tests, and calculated χ^2 and the associated p-value for the following fourteen additional chi-squared tests.

- One-dimensional frequency test:[20] We distributed sequences of pseudorandom numbers into a one-dimensional histogram with 2^{20} bins, and compared the histogram to a uniform Poisson distribution.

- Serial test in D=2, 3, 4, 5, and 6 dimensions:[20] We distributed sequences of D successive pseudorandom numbers $\{r_1, \ldots, r_D\}$ into a D-dimensional histogram with either 2^{20} or 10^6 bins, and compared the histogram to a uniform Poisson distribution. This tests for D-dimensional sequential correlations in the sequence.

- Poker test:[20] We used groups of five pseudorandom numbers and counted the number of pairs, three-of-a-kind etc. formed from five cards with sixteen denominations, and compared the resulting histogram to a Poisson distribution derived from the exact probabilities. This tests for a variety of five-point correlations in the sequence.

- Collision tests:[20] We used the pseudorandom stream to distribute 2^{14} balls into 2^{20} urns, and compare the distribution of the number of collisions with the exact distribution. We used this to test for correlations in the twenty most significant bits of each pseudorandom number, and the most significant bit of twenty sequential pseudorandom numbers.

- Gaps test:[20] We compared the histogram of the length of runs of 0's ($r \leq 0.5$) and 1's ($r > 0.5$) to the exact Poisson distribution. This tests for correlations in the leading bits.

- Max-of-t test:[20] We compared the distribution of the maximum value among $\{r_1, r_2, \ldots, r_t\}$ for $t = 32$ with the exact probability distribution.

- Permutations test:[20] We compared the permutation ordering number of t successive pseudorandom numbers $\{r_1, r_2, \ldots, r_t\}$ for $t = 10$, with the uniform distribution of $t!$ possibilities.

- Fourier test:[14] We used a fast Fourier transform[46] to calculate the Fourier coefficients of sequences of $M = 2^{20}$ pseudorandom numbers,

$$\hat{x}_k = \frac{1}{\sqrt{M}} \sum_{j=0}^{M-1} x_j e^{2\pi i j k / M}, \tag{15}$$

where $x_j = (r_{2j} - 0.5) + i(r_{2j+1} - 0.5)$. We compared the distribution of the real and imaginary parts of $\hat{x}_k$ with the exact normal distribution with zero mean and variance $1/12$. This test exposes long-range pair correlations in the pseudorandom sequence.

- Two-dimensional Ising model energy distribution test:[47,48] We performed Wolff algorithm[49] Monte Carlo simulations at the critical point of the two-dimensional Ising model on a 128×128 square lattice, and compared the energy histogram to a Poisson distribution derived from the exact probabilities[47,48] Since the Wolff algorithm is based on stochastically growing fractal critical clusters that can span the system, this tests for long-range correlations in the pseudorandom sequence, and has proven to be effective at identifying weak generators.[47,48,50] See figure 1.

We tested the generator with thousands of different safe prime pairs for exponents as small as $e = 3$. Every instance passed all of our correlations tests, some with as many as 10^{13} pseudorandom numbers per test. In no case did a chi-squared test produce a p-value less than 10^{-6} or greater than $1 - 10^{-6}$. We also counted the number of tests that produced p-values less than 10^{-3} or greater than $1 - 10^{-3}$, and confirmed the number was consistent with the expected rate of one per one-thousand for each. We also applied the 1D frequency test, D-dimensional serial tests, the poker test, collisions tests, and gaps tests to the least significant bits, which also passed every test.
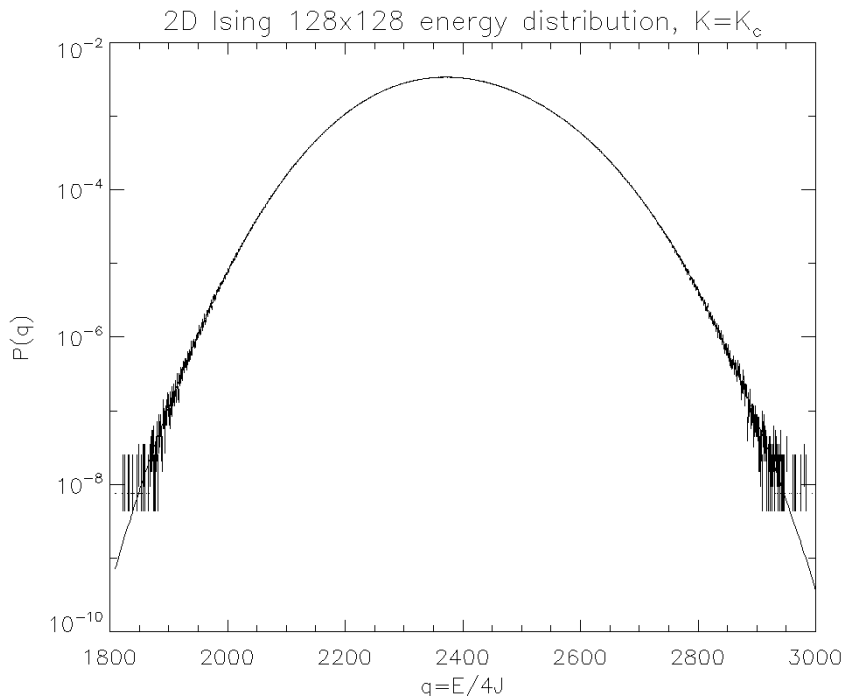
FIG. 1: The exact energy distribution[47,48] for a 128×128 square-lattice two-dimensional Ising model at the critical temperature (solid line) on a log scale, and the distribution calculated from 1.34×10^8 configurations chosen from 4.36×10^8 Monte Carlo steps per spin. The simulation was performed using from a Wolff[49] algorithm (error bars). The Wolff algorithm effectively eliminates critical slowing down, so the selected configurations are nearly uncorrelated with correlation time $\tau = 0.44$. The absissa is the energy above the ordered ground state in units of four times the coupling constant J. The simulation was about 8×10^8 Monte Carlo steps per spin. The simulation included 32 independent parallel processes, using approximately 10^{13} pseudorandom numbers generated withexponent $e = 9$. The result was $\chi^2 = 990$ with 1026 degrees of freedom, for a p-value of $p = 0.79$.

We confirmed that the algorithm displayed lack of correlation between streams. Each of M_p distinct streams labeled $\alpha = 0, 1, ..M_p - 1$ was assigned a different prime pairs p_1 and p_2. Our interstream correlations tests drew the pseudorandom numbers from the M_p streams in the order $r_1^{(0)}, r_1^{(1)}, r_1^{(2)}, \ldots, r_1^{(N_p-1)}, r_2^{(0)}, r_2^{(1)}, \ldots$, with various values of M_p. To ensure that seeding coincidences do not cause correlations, we performed the interstream correlations tests using the same primitive root a in each stream, and initialized every sequence with the same values $m_0 = 0$ and $s_0 = 1$. The interstream correlations passed all of the U01 SmallCrush, Crush and BigCrush tests, even for $e = 3$.

To examine the resilience of the generator, we tested various intentionally weakened versions the generator. As noted before, the generator passes the U01 tests with a unit or constant skip for $e \geq 9$. We tested the generator with $e = 1$, i.e. $c_k = m_k$ to test our use of $n \approx q$. Since the messages nearly uniform sample of Z_n^D, the messages themselves pass gentle randomness tests such as U01 SmallCrush. To test the sensitivity of the generator to the quality of the skip sequence, we tested the skip generator with $a = 3, 6, 7, 10, 11$, the smallest, and arguably worst, primitive roots mod q. The generator passes the U01 Crush tests with $e = 3$ even with these bad primitive roots.

VIII.   CONCLUSION

We propose a new class of parallel pseudorandom number generators based on a non-cryptographic RSA exponentiation cipher operating on 64-bit messages. The method is fully scalable based on parametrization since each process can be assigned a unique composite modulus $n = p_1 p_2$, where p_1 and p_2 are 32-bit safe primes, and the period of each instance greater than 8×10^{37}. By vectorizing the calculation, the method can produce over one-hundred million pseudorandom numbers per second on each a node of a multi-core supercomputer. We tested thousands of different pseudorandom streams, and all passed a battery of statistical tests. The C source code is available at https://github.com/PDBeale/randomRSA.git.

IX. ACKNOWLEDGEMENTS

---

[*] Electronic address: paul.beale@colorado.edu

[1] R. Rivest, A. Shamir, and L. Adelman, Commun. ACM **21**, 120 (1978).

[2] N. Ferguson, B. Schneier, T. Kohno, *Cryptography Engineering: Design Principles and Applications*, (Wiley, Indianapolis, 2010).

[3] B. Schneier, *Applied Cryptography* (Wiley, New York, 1994).

[4] T. Koshy, *Elementary Number Theory and Applications* (Academic, San Diego, 2002).

[5] J.H. Silverman, *A Friendly Introduction to Number Theory* (Pearson, New York, 2006).

[6] N. Koblitz, *A Course in Number Theory and Cryptography* (Springer-Verlag, New York, 1987).

[7] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, (Cambridge, New York, 1992), 2nd ed.

[8] V. Rijmen, A. Bosselrs, and P. Barreto. Optimised ANSI C code for the Rijndael cipher (now AES), 2000. Public domain software.

[9] J.K. Salmon, M.A. Moraes, R.O. Dror, and D.E. Shaw, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11), New York, NY: ACM, 2011. doi:10.1145/2063384.2063405 .

[10] P. L'Ecuyer and R. Simard, ACM Trans. Math. Software **33**, 22 (2007); see `http://www.iro.umontreal.ca/~simardr/testu01/tu01.html`.

[11] I.E. Shparlinski, Math. Comp. **70**, 801 (2000).

[12] P.D. Coddington, Northeast Parallel Architecture Center. Paper 13. `https://surface.syr.edu/npac/13/` (1997)

[13] H. Bauke and S. Mertens, Phys. Rev. E **75**, 066701 (2007).

[14] M. Mascagni and A. Srinivasan, ACM Trans. Math. Software **26**, 436 (2000).

[15] The Scalable Parallel Random Number Generators Library (SPRNG), `http://www.sprng.org`.

[16] M. Mascagni, Parallel Comput. **24**, 923 (1998).

[17] M. Mascagni, M. L. Robinson, D. V. Pryor and S. A. Cuccaro, Lec. Notes Statistics **106**, 263 (1995).

[18] M. Mascagni, S. A. Cuccaro, D. V. Pryor and M. L. Robinson, J. Comp. Phys. **119**, 211 (1995).

[19] M. Matsumoto and T. Nishimura, ACM Trans. Model. Comput. Sim. **8(1)**, 3 (1998).

[20] D. Knuth, *The Art of Computer Programming*, vol. 2 (Addison-Wesley, Reading, Massachusetts, 1999).

[21] N. Zierler and J. Brillhart, Info. and Control **13**, 541 (1968).

[22] R.C. Merkle, Communications of the ACM. **21 (4)**: 294 (1978).

[23] W. Diffie, and M.E. Hellman, IEEE Transactions on Information Theory. **22 (6)**, 644 (1976).

[24] S. Singh, *The Code Book*, (Doubleday, New York, 1999).

[25] P.W. Shor, SIAM J. Comp. **26**, 1484 (1977); `https://arxiv.org/abs/quant-ph/9508027`.

[26] Fpr the case $n = p_1 p_2$, a smaller value of d can be obtained using $d = e^{-1} \bmod \lambda(n)$, where $\lambda(n) = (p_1 - 1)(p_2 - 1)/\gcd(p_1 - 1, p_2 - 1)$, is Carmichael's totient function. This does not make a difference for RSA applications that use CRT since the exponents used in the decryption $d_1 = d \bmod (p_1 - 1)$ and $d_2 = d \bmod (p_2 - 1)$ that result are the same as using $\phi(n)$.

[27] B.A Wichmann, and I.D. Hill, I.D, Appl. Stat. **31**, 188 (1982).

[28] P. Bratley, B. L. Fox,, and L.E. Schrage,. *A Guide to Simulation*, 2nd ed.. (Springer-Verlag, New York, 1987).

[29] P. L'Ecuyer, Commun. ACM **31**, 741 (1988).

[30] P. L'Ecuyer. F.-O. Blouin, and R. Couture, ACM Trans. Model. Comput. Simul.. **3**, 87 (1993). They recommend primitive root $a = 2307085864$.

[31] F. Sezgin, T. M. Sezgin, Comp. Phys. Comm. **184**, 1889 (2013). Not all of the multipliers in Table 4 are primitive roots mod $q = 2^{63} - 25$. We have confirmed that multipliers
$a = \{3157107955, 3163786287, 3200261722, 3211103532, 3338736601, 3423977237, 3465965455, 3474009732, 3512424704\}$ are primitive roots mod q and pass all U01 Crush and BigCrush tests.

[32] This makes $(p - 1)/2$ a Sophie Germain prime.

[33] S. Pohlig and M. Hellman, IEEE Transactions on Information Theory IEEE Trans. Inform. Theory **(24)**, 106 (1978).

[34] P.D. Beale, `https://arxiv.org/abs/1411.2484`.

[35] *The On-Line Encyclopedia of Integer Sequences*, `http://oeis.org/A036378`.

[36] *The On-Line Encyclopedia of Integer Sequences*, `http://oeis.org/A211395`; `http://oeis.org/A211397`.

[37] G.L. Miller, J. Comp. Sys. Sci. **13**, 300 (1976).

[38] M.O. Rabin, J. Number Theory **12**, 128 (1980).

[39] C. Pomerance, J. L. Selfridge and S. S. Wagstaff, Jr., Math. Comp **35**, 1003 (1980).

[40] *The On-Line Encyclopedia of Integer Sequences* `http://oeis.org/A014233`.

[41] Zhenxiang Zhang, Math. Comp. **70**, 863 (2001).

[42] Y. Jiang and Y. Deng, `http://arxiv.org/abs/1207.0063v1`.

[43] See `http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html`.

[44] G. Marsaglia, DIEHARD: a battery of tests of randomness (1996); see `http://stat.fsu.edu/pub/diehard/`.

[45] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, NIST special publication 800-22, National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, USA, 2001; see `http://csrc.nist.gov/rng/`.

[46] J.W. Cooley, J.W. Tukey, Math. Comp. **19**, 297 (1965). .

[47] P.D. Beale, Phys. Rev. Lett. **76**, 78 (1996).

[48] R.K. Pathria and P.D. Beale, *Statistical Mechanics* 3rd ed., (Academic, Boston, 2011).

[49] U. Wolff, Phys. Rev. Lett. **62**, 361 (1989).

[50] A.M. Ferrenberg, D.P. Landau and Y.J. Wong, Phys. Rev. Lett. **69**, 3382 (1992).

[51] Jonathon Anderson, Patrick J. Burns, Daniel Milroy, Peter Ruprecht, Thomas Hauser, and Howard Jay Siegel. 2017. Deploying RMACC Summit: An HPC Resource for the Rocky Mountain Region. In Proceedings of PEARC17, New Orleans, LA, USA, July 09-13, 2017, 7 pages. DOI: 10.1145/3093338.3093379