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

Paul D. Beale
Department of Physics
University of Colorado Boulder
paul.beale@colorado.edu

This document describes the public functions available in randomRSA, a scalable parallel and vectorizable pseudorandom number generator based on non-cryptographic RSA exponentiation ciphers. 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. Uniformly distributed pseudorandom numbers r_k on $(0, 1]$ are created using a real divide $r_k = c/n$. The algorithm is fully scalable by parametrization on parallel supercomputers, since each node can be assigned independent pairs of primes. In addition, the vectors of independent messages, skips, and ciphertexts can be updated simultaneously. Vectorized code can produce more than 10^8 pseudorandom numbers per second on each multicore supercomputer node.

The pseudocode for updating the message, skip, and ciphertext vectors for the internal calculation is

$$\boldsymbol{s} := a\boldsymbol{s} \bmod q, \tag{2a}$$
$$\boldsymbol{m} := (\boldsymbol{m} + \boldsymbol{s}) \bmod n, \tag{2b}$$
$$\boldsymbol{c} := \boldsymbol{m}^e \bmod n, \tag{2c}$$
$$\boldsymbol{r} := \boldsymbol{c}/n. \tag{2d}$$

where $\boldsymbol{m}$ is a vector of 64-bit messages, $\boldsymbol{s}$ is a vector of 64-bit pseudorandom skips, $\boldsymbol{c}$ is a vector of 64-bit ciphertexts, and $\boldsymbol{r}$ is a vector of 64-bit double precision floating point pseudorandom numbers that are uniformly distributed on $[0, 1)$. The entire calculation can be accomplished using fast native unsigned 64-bit arithmetic by taking advantage of 32-bit restricted primitive roots $a < \sqrt{q}$ in the skip calculation (2a), and the Chinese Remainder Theorem in the exponentiation (2c).

The two key public functions available to the user are:

int vrandomRSA(double * randomRSA, long nrandomRSA); returns a vector of double precision floating point pseudorandom numbers uniformly distributed on $[0, 1)$. This is our recommended way to use the generator, as it returns large numbers of pseudorandom numbers to the user most efficiently. The user, of course, should use each pseudorandom number once, and only once, to assure all random tests in the user code are uncorrelated. The user defined vector **double * randomRSA** is a double precision array created by the user with **long nrandomRSA** elements.

and

double randomRSA();. returns one double precision floating point pseudorandom number per call. The pseudorandom sequence is uniformly distributed on $[0, 1)$. This function is available so the user can easily substitute this RSA-based pseudorandom number generator in place of another generator that returns one pseudorandom number per call.

Both functions will initialize themselves into unique states on the first call in each MPI process using the current time in microseconds since 00:00:00 1/1/1970, and values mpirank and mpisize determined by MPI calls. If the user is not using multiple MPI processes, or MPI has not been initialized before the first call to either generator, the values are set to mpirank=0 and mpisize=1. The user can initialize the generator in the same manner using

int randomRSA_init();.

If the user would like to initialize the generator into a reproducible state using a user chosen seed for debugging purposes, the user can call **int randomRSA_init_seed(uint64_t seed);** before calls to **randomRSA**. or **vrandomRSA**.

int randomRSA_init_seed(uint64_t seed); will initialize each MPI process with the user value **seed**, and the MPI-determined mpirank and mpisize values.

If the user would like to initialize the generator into a reproducible state in each MPI process using a user chosen seed for debugging purposes, the user can use
int randomRSA_init_seed_MPI(uint64_t seed, int mpirank, int mpisize); which will initialize each MPI process using **seed** and user chosen values for **mpirank** and **mpisize**.

If any of the initialization routines are called after the generator has already been initialized, the state of the generator is reinitialized.

The user can access the RSA encryption parameters using the routines

int randomRSA_get_exponent(uint32_t *exponent); which returns the RSA exponent e,

int randomRSA_get_primes(uint32_t *prime1, uint32_t *prime2, uint64_t *composite); which returns the 32-bit RSA primes p_1 and p_2, and 64-bit RSA composite $n = p_1 p_2$,

int randomRSA_get_skipprime(uint64_t *skipprime); which returns the 64-bit prime $q = 2^{63} - 25$ used in the linear congruential skip generator (q is the largest prime less than 2^{63}),

int randomRSA_get_primitiveroot(uint32_t *primitiveroot); which returns the 32-bit primitive root mod q used in the linear congruential skip generator.

The user can change the values of the exponent e using
int randomRSA_set_exponent(uint32_t exponent);, where **exponent** must be an odd integer in the range 3 to 257 (recommended values are $e = 3, 5, 9, 17$),

The user can change the values of primes p_1 and p_2 using
int randomRSA_set_primes(uint32_t prime1, uint32_t prime2);, where **prime1** and **prime2** set the values of p_1 and p_2. The primes p_1 and p_2 must be *safe primes*, i.e p_1, $(p_1 - 1)/2$, p_2, and $(p_2 - 1)/2$ must all be prime. The generator also requires $2^{32} > p_1 > p_2 > 2^{31}$.

The skip prime q, and the primitive root a cannot be changed by the user since they are selected from a very limited set of well-tested values.

The user can get the full internal state of the generator using

int randomRSA_get_state(uint32_t *prime1, uint32_t *prime2, uint32_t *exponent, uint32_t *primitive-root, uint32_t *index, uint64_t *hash, uint64_t *messages, uint64_t *skips);. The user must also declare 64-bit arrays **uint64_t *messages** and **uint64_t *skips** with length **int randomRSA_vectorsize();** before calling **int randomRSA_get_state**.

The user can reset the generator using the saved state by calling

int randomRSA_set_state(uint32_t prime1, uint32_t prime2, uint32_t exponent, uint32_t primitive-root, uint32_t index, uint64_t hash, uint64_t *messages, uint64_t *skips);. The user must call **int randomRSA_get_state** before calling **int randomRSA_set_state**, and the user may not make any changes to the stored state before resetting the generator. This is controlled using a hash function to set and test **hash**. The unsigned 64-bit arrays **messages** and **skips** must be created with length **RSAVECTORSIZE**.

The user can get the total number of pseudorandom numbers returned to the user by calling
long randomRSA_randsreturned();.

The code can be compiled into an object file using the intel MPI C compiler and the OpenMP library using
mpiicc -O2 -qopenmp -c randomRSA.c

The code is vectorized using OpenMP to share the vector calculation across multiple cores on each independent MPI process. The user needs to inform the operating system of the number of cores commited to the calculation before running user code that calls the generator by using the interactive terminal mode, bash shell command
export OMP_NUM_THREADS=XX

When using the code in slurm batch mode, one can set the number of cores assigned to each process using
#SBATCH –nodes=YY
#SBATCH –cpus-per-task=XX
On the University of Colorado Boulder Summit supercomputer, the code can generate between 100 million and 200 million pseudorandom numbers per second per node when 24 cores were assigned to each node. If a job is limited to a single core, the code generates 10 million to 15 million per second.