REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
28-07-2015	Final Report	20-Dec-2012 - 19-Mar-2015

4. TITLE AND SUBTITLE

Final Report: Random Number Generation for High Performance Computing

5a. CONTRACT NUMBER

5b. GRANT NUMBER

W911NF-13-C-0025

5c. PROGRAM ELEMENT NUMBER

665502

6. AUTHORS

Rajendra V. Boppana, Ph.D., P.I., UTSA, and Robert M Keller, Silicon Informatics, Inc.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES

Silicon Informatics, Inc.
6500 Parnell Ave.

Edina, MN 55435 -1515

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

62556-MA-ST2.1

12. DISTRIBUTION AVAILIBILITY STATEMENT

Approved for Public Release; Distribution Unlimited

13. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT

The primary objectives of the Phase II of the project are: (a) implement the context-aware parallel random number generator (CPRNG), developed in Phase I of this project, (b) implement the interstream correlation (ISC) test so that the quality of the random numbers (RNs) used by applications are evaluated and quality metrics are reported on demand. Both objectives have been accomplished.

Beyond these objectives, additional design and implementation contributions have been accomplished. A flexible

15. SUBJECT TERMS

CPRNG, Pseudorandom Number Generation for High Performance Computing, Interstream Correlation (ISC) Test, ISC Framework

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Alain Bopda
UU	UU	UU	UU		19b. TELEPHONE NUMBER 334-229-8396

Report Title

Final Report: Random Number Generation for High Performance Computing

ABSTRACT

The primary objectives of the Phase II of the project are: (a) implement the context-aware parallel random number generator (CPRNG), developed in Phase I of this project, (b) implement the interstream correlation (ISC) test so that the quality of the random numbers (RNs) used by applications are evaluated and quality metrics are reported on demand. Both objectives have been accomplished.

Beyond these objectives, additional design and implementation contributions have been accomplished. A flexible CPRNG-ISC Test (CIT) framework was developed and implemented so that a third party tester such as Dieharder or TestU01 can be run along with ISC test to corroborate or compare ISC test results with those from the well-known single-stream test batteries. The CPRNG Library facilitates implementation and use of other random number generators within the test framework easily.

To demonstrate the flexibility of the CIT framework, we implemented the MLFG generator from SPRNG package together with a number of other generators, some of which have become available since the beginning of this Phase II project.

Three versions of CPRNG were implemented: CPU-based context-free generator, a CPU-based context-aware generator, and GPU-based context-free generator.

Enter List of papers submitted or published that acknowledge ARO support from the start of the project to the date of this printing. List the papers, including journal references, in the following categories:

(a) Papers published in peer-reviewed journals (N/A for none)

Received	Paper

TOTAL:

Number of Papers published in peer-reviewed journals:

(b) Papers published in non-peer-reviewed journals (N/A for none)

Received	Paper

TOTAL:

Number of Papers published in non peer-reviewed journals:

(c) Presentations

Presentation to the HPC User Forum, Boston MA, in September 2013 by Rajendra V Boppana, Ph.D. An electronic copy of the presentation was provided to the COTR and is available online at www.hpcuserforum.com. At the HPC User Forum website, click on "previous meeting archive," then "September 2013 Boston," and then "RajBoppana.pdf."

Number of Presentations: 1.00

Non Peer-Reviewed Conference Proceeding publications (other than abstracts):

Received Paper

TOTAL:

Number of Non Peer-Reviewed Conference Proceeding publications (other than abstracts):

Peer-Reviewed Conference Proceeding publications (other than abstracts):

Received Paper

TOTAL:

Number of Peer-Reviewed Conference Proceeding publications (other than abstracts):

(d) Manuscripts

Received Paper

TOTAL:

Number of Manuscripts:

Books

Received	Book

TOTAL:

Received	Book Chapter

TOTAL:

Patents Submitted

Arising from Phase I of this project, there were two U.S. patent applications based on inventions that were developed by University of Texas at San Antonio (UTSA) during Phase I and subsequently reported to the Contracting Officer on DD Form 882. These patent applications include: "Generation of Distinct Pseudorandom Number Streams Based on Program Context," and "A Statistical Test Method to Quantify Inter-Streams Based on Program Context." A DD Form 882 describing these inventions was submitted to the Contracting Officer by UTSA on Feb 16, 2012.

Patents Awarded

On October 21, 2014, US Patent Number 8,868,630 B1, based on the latter patent application above, issued. Regarding the former, a US Patent Office Notice of Allowance has issued. The issued patent and patent application are included as appendices in the pdf file accompanying this final report.

Awards

Graduate Students

NAME	PERCENT_SUPPORTED	Discipline
Robin Schulze	0.43	
FTE Equivalent:	**0.43**	
Total Number:	**1**	

Names of Post Doctorates

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Names of Faculty Supported

NAME	PERCENT_SUPPORTED	National Academy Member
Rajendra V. Boppana	0.30	
Ram C. Tripathi	0.08	
Ravinderpal Singh Sandhu	0.04	
Ashok Srinivasan	0.08	
FTE Equivalent:	**0.50**	
Total Number:	**4**	

Names of Under Graduate students supported

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Student Metrics

This section only applies to graduating undergraduates supported by this agreement in this reporting period

The number of undergraduates funded by this agreement who graduated during this period: 0.00

The number of undergraduates funded by this agreement who graduated during this period with a degree in science, mathematics, engineering, or technology fields:..... 0.00

The number of undergraduates funded by your agreement who graduated during this period and will continue to pursue a graduate or Ph.D. degree in science, mathematics, engineering, or technology fields:..... 0.00

Number of graduating undergraduates who achieved a 3.5 GPA to 4.0 (4.0 max scale):..... 0.00

Number of graduating undergraduates funded by a DoD funded Center of Excellence grant for Education, Research and Engineering:..... 0.00

The number of undergraduates funded by your agreement who graduated during this period and intend to work for the Department of Defense 0.00

The number of undergraduates funded by your agreement who graduated during this period and will receive scholarships or fellowships for further studies in science, mathematics, engineering or technology fields:..... 0.00

Names of Personnel receiving masters degrees

NAME
Total Number:

Names of personnel receiving PHDs

NAME
Total Number:

Names of other research staff

NAME	PERCENT_SUPPORTED

FTE Equivalent:
Total Number:

Sub Contractors (DD882)

1 a. Florida State University

1 b. Sponsored Research Administration
874 Traditions Way, Third Floor
Tallahassee FL 323064166

Sub Contractor Numbers (c): SI-2013-001
Patent Clause Number (d-1):
Patent Date (d-2):
Work Description (e): Provide subject matter expertise (testing of pseudorandom number generators) to the Prin
Sub Contract Award Date (f-1): 12/20/12 12:00AM
Sub Contract Est Completion Date(f-2): 12/29/14 12:00AM

---

1 a. Florida State University

1 b. 97 South Woodward Avenue, Third
Tallahassee FL 323060001

Sub Contractor Numbers (c): SI-2013-001
Patent Clause Number (d-1):
Patent Date (d-2):
Work Description (e): Provide subject matter expertise (testing of pseudorandom number generators) to the Prin
Sub Contract Award Date (f-1): 12/20/12 12:00AM
Sub Contract Est Completion Date(f-2): 12/29/14 12:00AM

---

1 a. University of Texas at San Antonio

1 b. One UTSA Circle
San Antonio TX 782491644

Sub Contractor Numbers (c): SI-2012-001
Patent Clause Number (d-1):
Patent Date (d-2):
Work Description (e): Serve as Principal Investigator for the overall Phase II effort. Provided direction to the Sil
Sub Contract Award Date (f-1): 12/20/12 12:00AM
Sub Contract Est Completion Date(f-2): 3/19/15 12:00AM

---

1 a. University of Texas at San Antonio

1 b. 6900 North Looop 1604 West
San Antonio TX 782491130

Sub Contractor Numbers (c): SI-2012-001
Patent Clause Number (d-1):
Patent Date (d-2):
Work Description (e): Serve as Principal Investigator for the overall Phase II effort. Provided direction to the Sil
Sub Contract Award Date (f-1): 12/20/12 12:00AM
Sub Contract Est Completion Date(f-2): 3/19/15 12:00AM

Inventions (DD882)

Scientific Progress

1. Accomplishments

A flexible CPRNG-ISC Test (CIT) framework was developed and implemented so that a third party tester such as Dieharder or TestU01 can be run along with the ISC test to corroborate or compare ISC test results with those from the well-known single-stream test batteries.

The CPRNG Library is implemented in a flexible manner to facilitate implementation and use of other random number generators within the test framework easily (see Figure 1 in the attached document).

To demonstrate the flexibility of the CIT framework, we implemented the MLFG generator from SPRNG package [7], drand48—available on standard Unix/Linux systems, and a parallel RNG based on cryptographic operations from the family of generators proposed by D.E. Shaw Group [12], and a pathological linear congruence generator (pLCG). In addition, we implemented within CPRNG Library to provide access to Intel's new digital random number generator (DRNG) and Nvidia's GPU-based generator MTGP32 [6], when the host system has the necessary hardware—newer processor chips or GPUs, respectively—to support these generators.

Three versions of CPRNG were implemented: CPU-based context-free generator, which is used to report results in this report, CPU-based context-aware generator, and GPU-based context-free generator. A context-aware generator automatically, without any changes to the application code, uses distinct RN streams when the application requests for RNs from a stream from different program contexts.

2. Performance Analysis

CPRNG, the new parallel random generator developed in Phase I of this project, was implemented in the SPRNG package in Phase I. In Phase II, it was implemented as a standalone library package with a simple application programming interface. The results given in Figure 2 of the attached document indicate that the time to initialize a RN stream is decreased slightly, and the time to obtain a RN is reduced by 20-30%. CPRNG generates RNs with very low overhead.

The ISC Test was used to determine the interstream correlations for MLFG and CPRNG in Phase I. In Phase II, several other random number generators implemented within the CPRNG Library have been evaluated for interstream correlations using the CPRNG-ISC Test framework. The results, given in Figure 3 of the attached document, show that CPRNG generates a large number of parallel RN streams with low interstream correlations.

The CIT framework is used to compare the quality metrics—DR and KS statistics [8],[9]—by ISC test with the Dieharder [11] and TestU01 [10] test batteries that are commonly used in literature. In general, the two test batteries corroborate each other's test results for a given stream of RNs. The Ising model simulation [5], which simulates the spread of energy in a 2-D lattice, and for which the exact theoretical results are available, is the application we used to corroborate or refute the results by various test methods.

For the pathological linear congruence generator (pLCG), which is designed to have high interstream correlations, the ISC Test and the two test batteries indicate significant correlations among RNs. This is confirmed by the Ising model simulations. On the other hand, drand48, a sequential generator commonly available on Linux and Unix systems, is reported to have high correlations by Dieharder and TestU01. However, ISC test does not indicate any correlations; the Ising model simulations confirm the ISC test results.

References

[1] R.V. Boppana, "Context-aware parallel pseudorandom number generators for large parallel computations," In Proceedings Of DoD High Performance Computing Modernization Program Users Group Conference, pp. 634-641, 2011.
[2] R.V. Boppana, "Generation of distinct pseudorandom number streams based on program context," UT Patent Approved (application number 13/426,028), June 2015.
[3] R.V. Boppana and R.C. Tripathi, "Verification of pseudorandom number streams," US Patent 8.868,630 B1, October 2014.
[4] R.V. Boppana, "Final Report," STTR Project: Random number generation for high performance computing, ARO Contract # W911NF-11-C-0026, May 2011.
[5] P. Coddington, Tests of random number generators using Ising model simulations. Int. J. of Mod. Phys. 7, 3, 295-303, 1996.
[6] NVIDIA, Inc. CUDA Programming Guide, version 1.0, 1997.
[7] M. Mascagni and A. Srinivasan, Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. ACM Transactions on Mathematical Software (TOMS), 26(3), pp. 436-461, 2000.
[8] A. Donner and B. Rosner. On inferences concerning a common correlation coefficient. Applied Statistics, 29(1):69-76, 1980.
[9] P. V. Rao. Statistical Research Methods in the Life Sciences. Brooks/Cole, 1998.
[10] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. ACM Trans. Math. Softw. 33, 4, August 2007.
[11] R. G. Brown, D. Eddelbuettel and D. Bauer. Dieharder Test Package, v 3.29.0, Duke University. Online: http://www.phy.

duke.edu/~rgb/General/dieharder.php. Retrieved on Nov. 2010.
[12] J.K. Salmon et al., “Parallel Random Numbers: As Easy as 1, 2, 3,” In Procs. of Supercomputing, 2011.

3. Business and Dissemination Plan

The major components supporting long term sustainability of CPRNG include:

- Preservation of all the software files and documentation,
- Development and growth of a body of users, and
- Continuation of CPRNG commercial licensing efforts, with a primary objective of licensing to a strategic partner such as a processor manufacturer (for inclusion in its tools library), or system manufacturer or vendor, or a major software provider.

3.1. Rationale

In order to accomplish the above, Silicon Informatics plans to assign its interest in CPRNG software, the “GetCPRNG.com” domain and associated website creative files to the University of Texas (UT) System. To the extent practicable, Silicon Informatics will continue to support the development of a user community and commercial licensing efforts.

The innovations developed during the course of this project are extensive and a bit ahead of their time than was anticipated at the outset of Phase II of this project. Despite our extensive outreach to the government, academic and private sectors, users’ needs appear to be met with conventional tools, including those that have been introduced since the inception of this project. One possible explanation is that much of the high performance computing software used for production runs has yet to be adapted and optimized to run on GPGPU-enhanced machines. [This information was presented by IDC in conjunction with at the SC14 Conference in New Orleans, LA, Nov 2014, a copy of which was provided to the COTR]. Despite these software issues, semiconductor manufacturers continue to innovate in areas of parallelization, processor-coprocessor integration and memory. One example is Intel Corporation’s Knights Landing processor, a “many integrated core” architecture that competes with GPGPU products. Knights Landing, with first shipments expected in 2015, will work as a host processor, capable of running an OS and applications on its own, while at the same time functioning as a coprocessor. Another example is Nvidia’s Titan X GPGPU, announced at Nvidia’s 2015 GPU Technology Conference, which will deliver up to 7 teraflops of single precision performance. We cite these processor trends, as they exemplify the future of High Performance Computing (HPC) where the circumstances on the demand side will eventually ripen for CPRNG’s innovations.

In the meantime, while applications software is adapted and optimized (or “rewritten” according to IDC), the greatest challenge is to gain CPRNG user experience. Toward that end, the University of Texas at San Antonio (UTSA) is well positioned. Unhindered by jointly held intellectual property rights, UTSA will have the ability to make our CPRNG software available not only to Government users, but also to users throughout the UT System. Under sole ownership, the process of licensing CPRNG commercially will be streamlined.

3.2. Further Work

Based on our evaluation of CPRNG and several other generators, CPRNG appears to be a high quality random number generator suitable for parallel applications that require a large number independent random number (RN) streams. The ISC test is a unique test to evaluate correlations among streams without being limited by the number of streams or number of random numbers. It also has the capability to evaluate the random numbers used by an application on the fly and provide a quality metric on the correlations among the random numbers used. The CIT framework provides a flexible framework to (a) evaluate new random number generators easily and compare them to the existing ones and (b) compare and calibrate the new random number generator test packages against the current test packages. CIT framework is very powerful for the design and testing of new random number generators and test suites.

The research and software produced by this project can be extended in making CIT framework more accessible to researchers that use a wide variety of computing platforms including multicore, GPU and many integrated core (MIC) architectures. Another direction for further work is to implement CPRNG and some of the other generators for MIC architectures such as Intel Xeon Phi. Currently, the ISC test can analyze RNs based a pre-specified grouping of streams and and interleaving method. However, the ISC test can be made even more powerful by recoding it to analyze random numbers consumed by application in multiple ways simultaneously.

Appendixes:
A. Phase I Final Technical Report
B. Paper, Context-Aware Parallel Pseudorandom Number Generators for Large Parallel Computations, 2011 DoD High Performance Computing Modernization Program (HCPMP) Users Group Conference, Rajendra V. Boppana, June 2011
C. US Patent 8,868,630 B1, entitled Verification of Pseudorandom Number Streams, Inventors Rajendra V. Boppana and Ram C. Tripathi
D. US Patent Application 13/426,028, entitled Generation of Distinct Pseudorandom Number Streams based on Program Context, inventor Rajendra V. Boppana

Technology Transfer

Silicon Informatics together with subcontractor University of Texas at San Antonio have engaged in outreach to several DoD Defense Supercomputing Resource Centers and service laboratories and have organized demonstrations to researchers at ARL and NRL. Our outreach extended to NASA and DoE laboratories as well as to private corporations. Most of these organizations have participated in the HPC User Forum (www.hpcuserforum) which is organized by IDC and convened twice annually at various locations throughout the USA. During the course of this project, we have attended four of the HPC User Forum meetings and were invited to present at the meeting held in Boston MA in 2013. We have also attended two IEEE/ACM Supercomputing conferences, SC13 and SC14, where we met with representatives from Government and private industry. In addition, we participated in the DoD SBIR/STTR Beyond Phase II conference in San Antonio in December 2014 where we participated in several one-on-one meetings with DoD lab, agency and industry representatives.

Technical Progress Report and Final Report

STTR Phase II Project: Random Number Generation for High Performance Computing

Period of Performance: December 20, 2012 — March 19, 2015

1. Accomplishments

The primary objectives of the Phase II of the project are: (a) implement the context-aware parallel random number generator (CPRNG), developed in Phase I of this project [1],[2],[4], with simple application programming interface (API) and scalability to accommodate applications running on a large number of processor cores or general purpose graphics processing unit (GPU) cores; (b) implement the interstream correlation (ISC) test so that the quality of the random numbers (RNs) used by applications are evaluated and quality metrics are reported on demand [3]. Both objectives have been accomplished. The following additional design and implementation contributions have been accomplished in this project.

A flexible CPRNG-ISC Test (CIT) framework was developed and implemented so that a third-party tester such as Dieharder or TestU01 can be run along with ISC test to corroborate or compare ISC test results with those from the well-known single-stream test batteries.

The CPRNG Library is implemented in a flexible manner to facilitate implementation and use of other random number generators within the test framework easily (see Figure 1).

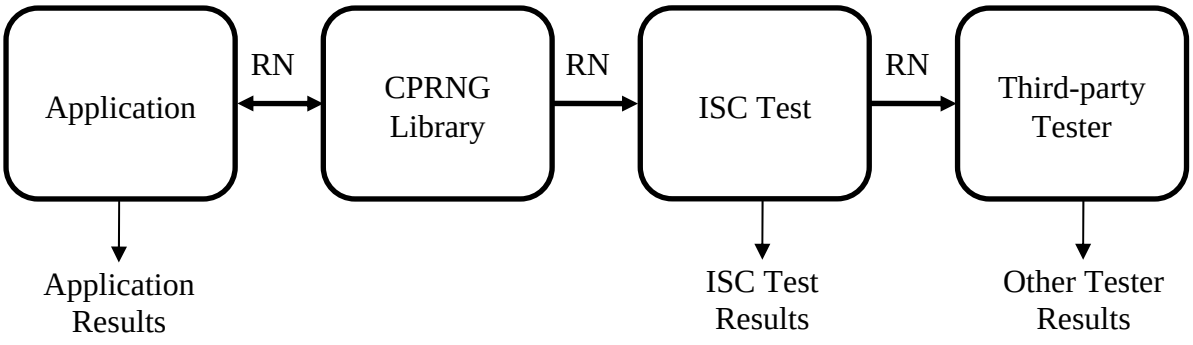


Figure 1. CPRNG-ISC Test framework. ISC Test can be run concurrently with application and quality of the random numbers (RNs) consumed by the application can be provided periodically or upon the completion of the application. In addition, a third-party tester such as the single-stream offline test packages could be used to provide an alternate method to assess the quality of the RNs.

CPRNG Library is designed to accommodate a wide variety of random number generators with a simple interface and compare their suitability for a given application.

Also, new random number generator test packages can be evaluated by comparing their performance against ISC Test or other known test packages.

To demonstrate the flexibility of the CIT framework, we implemented the MLFG generator from SPRNG package [7], drand48—available on standard Unix/Linux systems, and a parallel RNG based on cryptographic operations from the family of generators proposed by D.E. Shaw Group [12], and a pathological linear congruence generator (pLCG). In addition, we implemented within CPRNG Library to provide access to Intel's new digital random number generator (DRNG) and Nvidia's GPU-based generator MTGP32 [6], when the host system has the necessary hardware—newer processor chips or GPUs, respectively—to support these generators.

Three versions of CPRNG were implemented: CPU-based context-free generator, which is used to report results in this report, CPU-based context-aware generator, and GPU-based context-free generator. A context-aware generator automatically, without any changes to the application code, uses distinct RN streams when the application requests for RNs from a stream from different program contexts.

2. Performance Analysis

CPRNG, the new parallel random generator developed in Phase I of this project, was implemented in the SPRNG package in Phase I. In Phase II, it was implemented as a standalone library package with a simple application programming interface. The results given in Figure 2 indicate that the time to initialize a RN stream is decreased slightly, and the time to obtain a RN is reduced by 20-30%. CPRNG generates RNs with very low overhead.

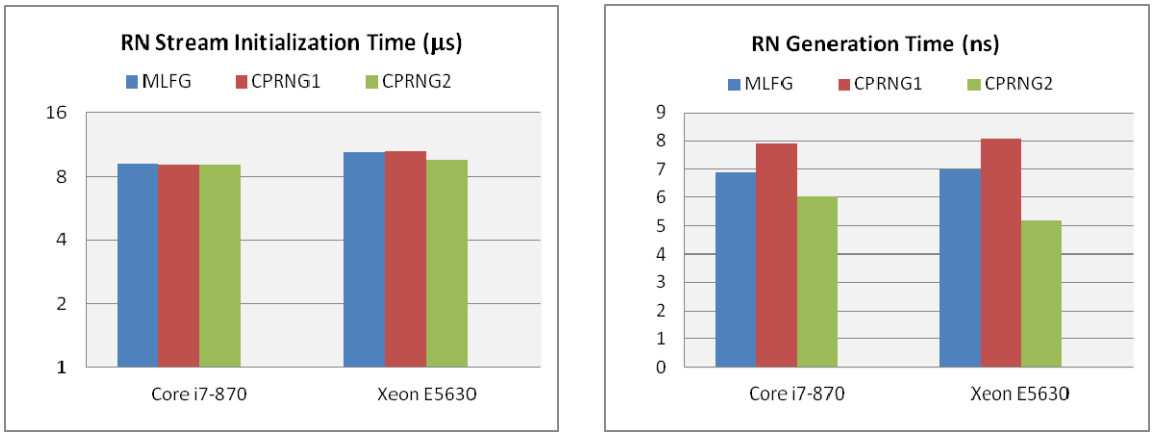


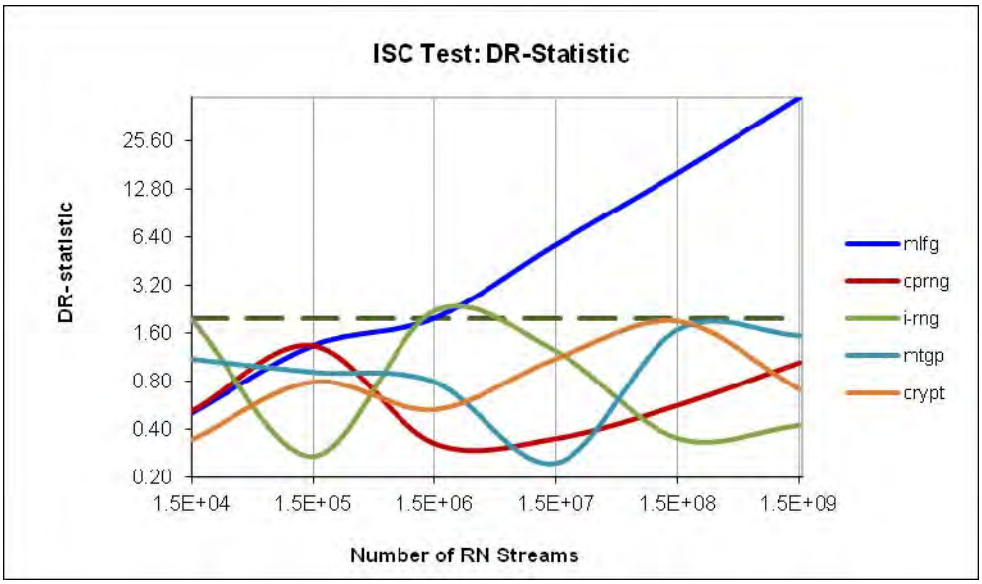
Figure 2. Comparison of RN stream initialization and RN generation times for two implementations of CPRNG. CPRNG 2 is the Phase II implementation in the CPRNG Library. CPRNG1 is the Phase I implementation of CPRNG in SPRNG package. For comparison purposes, the times for MLFG, a parallel RNG in SPRNG package are shown.

The ISC Test was used to determine the interstream correlations for MLFG and CPRNG in Phase I. In Phase II, several other random number generators implemented within the CPRNG

Library have been evaluated for interstream correlations using the CPRNG-ISC Test framework. The results, given in Figure 3, show that CPRNG generates a large number of parallel RN streams with low interstream correlations.

The CIT framework is used to compare the quality metrics—DR and KS statistics [8],[9]—by ISC test with the Dieharder [11] and TestU01 [10] test batteries that are commonly used in literature. In general, the two test batteries corroborate each other's test results for a given stream of RNs. The Ising model simulation [5], which simulates the spread of energy in a 2-D lattice, and for which the exact theoretical results are available, is the application we used to corroborate or refute the results by various test methods.

For the pathological linear congruence generator (pLCG), which is designed to have high interstream correlations, the ISC Test and the two test batteries indicate significant correlations among RNs. This is confirmed by the Ising model simulations. On the other hand, drand48, a sequential generator commonly available on Linux and Unix systems, is reported to have high correlations by Dieharder and TestU01. However, ISC test does not indicate any correlations; the Ising model simulations confirm the ISC test results.



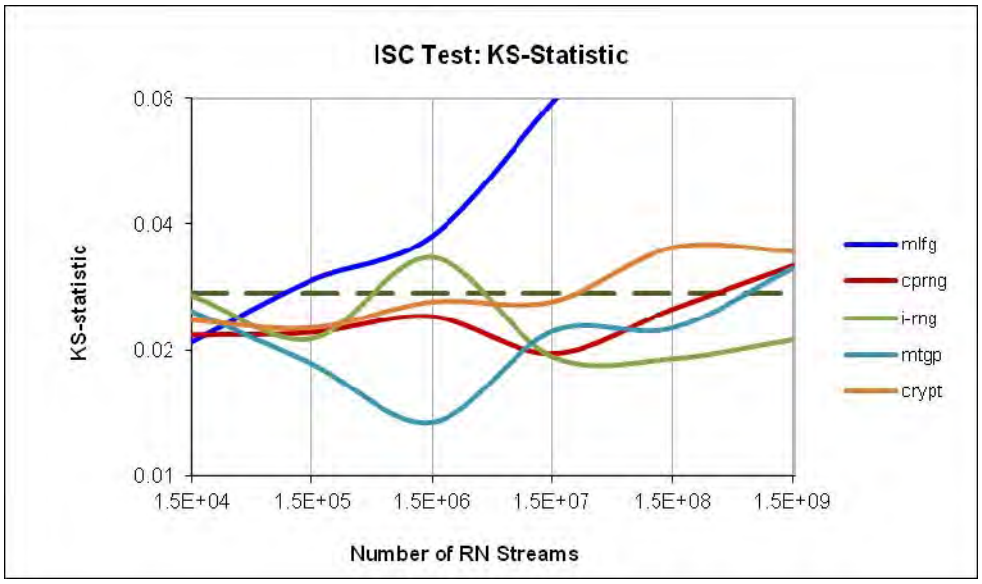


Figure 3. ISC Test of interstream correlations for various random number generators implemented or accessed through CPRNG Library. The generators implemented within the CPRNG Library include mlfg—a generator from SPRNG package—and crypt—a cryptographic operations-based generator from a family of generators designed by D.E. Shaw Group. The generators accessed through CPRNG Library include mtgp—a GPU-based RNG by Nvidia—and i-rng—a new RNG by Intel which has hardware-support for seed generation with high entropy and cryptographic operations for RN generation.

Two statistical tests, Donner-Rossner and Kolmogorov-Smirnov tests, are used to accept or reject the hypothesis that the parallel streams extracted from an RNG are not correlated. The significance levels are 0.05 for the DR test and 0.01 for the KS test. The dashed lines indicate the critical value below which the test statistics (DR or KS) should remain to validate the hypothesis.

References

[1] R.V. Boppana, "Context-aware parallel pseudorandom number generators for large parallel computations," In Proceedings Of DoD High Performance Computing Modernization Program Users Group Conference, pp. 634-641, 2011.
[2] R.V. Boppana, "Generation of distinct pseudorandom number streams based on program context," UT Patent Approved (application number 13/426,028), June 2015.
[3] R.V. Boppana and R.C. Tripathi, "Verification of pseudorandom number streams," US Patent 8.868,630 B1, October 2014.
[4] R.V. Boppana, "Final Report," STTR Project: Random number generation for high performance computing, ARO Contract # W911NF-11-C-0026, May 2011.
[5] P. Coddington, Tests of random number generators using Ising model simulations. Int. J. of Mod. Phys. 7, 3, 295-303, 1996.
[6] NVIDIA, Inc. CUDA Programming Guide, version 1.0, 1997.
[7] M. Mascagni and A. Srinivasan, Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. ACM Transactions on Mathematical Software (TOMS), 26(3), pp. 436-461, 2000.
[8] A. Donner and B. Rosner. On inferences concerning a common correlation coefficient. Applied Statistics, 29(1):69-76, 1980.
[9] P. V. Rao. Statistical Research Methods in the Life Sciences. Brooks/Cole, 1998.
[10] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. ACM Trans. Math. Softw. 33, 4, August 2007.
[11] R. G. Brown, D. Eddelbuettel and D. Bauer. Dieharder Test Package, v 3.29.0, Duke University. Online: http://www.phy.duke.edu/~rgb/General/dieharder.php. Retrieved on Nov. 2010.
[12] J.K. Salmon et al., "Parallel Random Numbers: As Easy as 1, 2, 3," In Procs. of Supercomputing, 2011.

3. Business and Dissemination Plan

The major components supporting long term sustainability of CPRNG include:

- Preservation of all the software files and documentation,
- Development and growth of a body of users, and
- Continuation of CPRNG commercial licensing efforts, with a primary objective of licensing to a strategic partner such as a processor manufacturer (for inclusion in its tools library), or system manufacturer or vendor, or a major software provider.

3.1. Rationale

In order to accomplish the above, Silicon Informatics plans to assign its interest in CPRNG software, the "GetCPRNG.com" domain and associated website creative files to the University of Texas (UT) System. To the extent practicable, Silicon Informatics will continue to support the development of a user community and commercial licensing efforts.

The innovations developed during the course of this project are extensive and a bit ahead of their time than was anticipated at the outset of Phase II of this project. Despite our extensive outreach

to the government, academic and private sectors, users' needs appear to be met with conventional tools, including those that have been introduced since the inception of this project. One possible explanation is that much of the high performance computing software used for production runs has yet to be adapted and optimized to run on GPGPU-enhanced machines.[1] Despite these software issues, semiconductor manufacturers continue to innovate in areas of parallelization, processor-coprocessor integration and memory. One example is Intel Corporation's Knights Landing processor, a "many integrated core" architecture that competes with GPGPU products. Knights Landing, with first shipments expected in 2015, will work as a host processor, capable of running an OS and applications on its own, while at the same time functioning as a coprocessor. Another example is Nvidia's Titan X GPGPU, announced at Nvidia's 2015 GPU Technology Conference, which will deliver up to 7 teraflops of single precision performance. We cite these processor trends, as they exemplify the future of High Performance Computing (HPC) where the circumstances on the demand side will eventually ripen for CPRNG's innovations.

In the meantime, while applications software is adapted and optimized (or "rewritten" according to IDC), the greatest challenge is to gain CPRNG user experience. Toward that end, the University of Texas at San Antonio (UTSA) is well positioned. Unhindered by jointly held intellectual property rights, UTSA will have the ability to make our CPRNG software available not only to Government users, but also to users throughout the UT System. Under sole ownership, the process of licensing CPRNG commercially will be streamlined.

3.2. Further Work

Based on our evaluation of CPRNG and several other generators, CPRNG appears to be a high quality random number generator suitable for parallel applications that require a large number independent random number (RN) streams. The ISC test is a unique test to evaluate correlations among streams without being limited by the number of streams or number of random numbers. It also has the capability to evaluate the random numbers used by an application on the fly and provide a quality metric on the correlations among the random numbers used. The CIT framework provides a flexible framework to (a) evaluate new random number generators easily and compare them to the existing ones and (b) compare and calibrate the new random number generator test packages against the current test packages. CIT framework is very powerful for the design and testing of new random number generators and test suites.

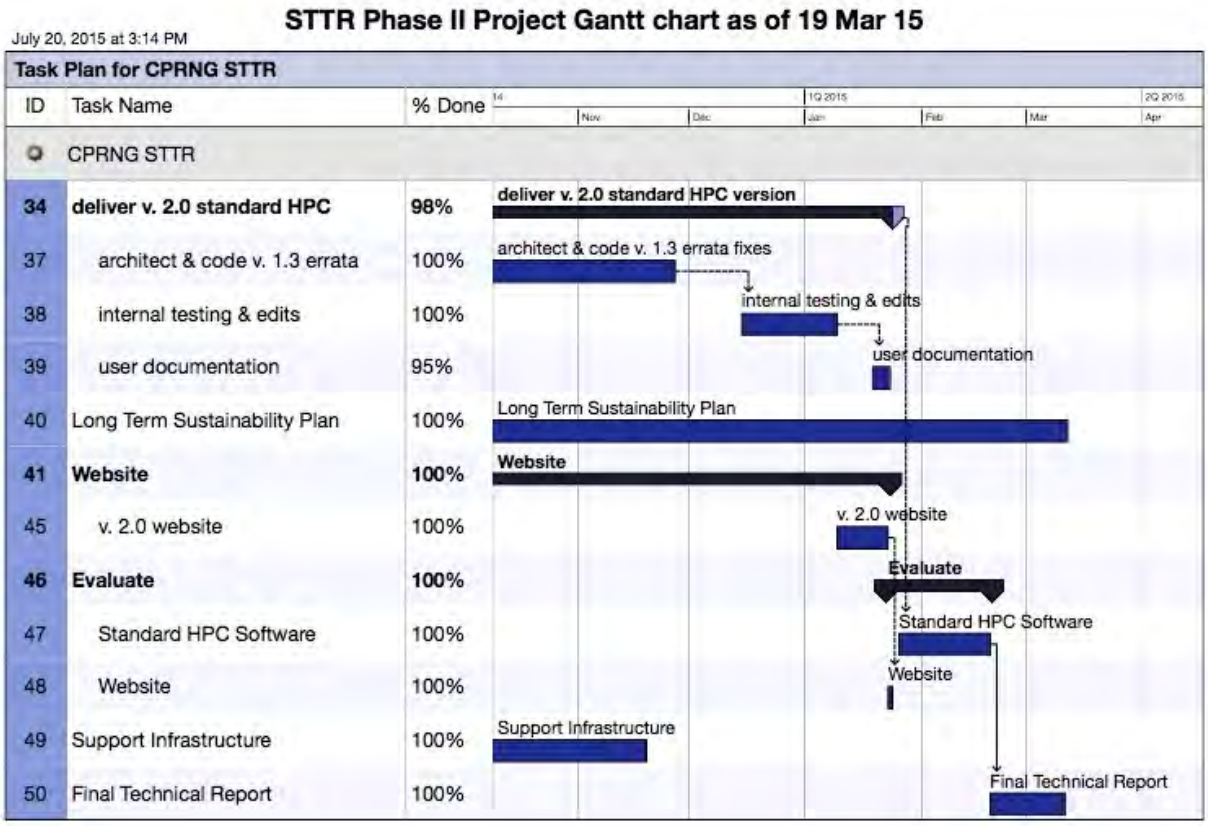
The research and software produced by this project can be extended in making CIT framework more accessible to researchers that use a wide variety of computing platforms including multicore, GPU and many integrated core (MIC) architectures. Another direction for further

[1] Presentation by IDC, "IDC at SC14" slide 93 of 95, Nov 18, 2014: "software is the #1 roadblock; better management software is needed, parallel software is lacking for most users, (and) many applications will need a major redesign."

work is implement CPRNG and some of the other generators for MIC architectures such as Intel Xeon Phi. Currently, the ISC test can analyze RNs based a pre-specified grouping of streams and and interleaving method. However, the ISC test can be made even more powerful by recoding it to analyze random numbers consumed by application in multiple ways simultaneously.

Programmatic issues: None.

4. Schedule Update



5. Milestone Update

Milestones completed to date: The fourth and final of the software is released in July 2015 to ARL researchers. This version supersedes the prior releases. Initial version of the project website is active and hosted by Rackspace at the URL getcprng.com.

This report is supplemented by a 4-part Appendix consisting of the technical documents produced as part of the project and the final report from Phase I of this project.

Prepared by: Rajendra V. Boppana, Ph.D., P.I., University of Texas at San Antonio, and

Robert Keller, Project Director, Silicon Informatics.

July 22, 2015

Final Technical Report
CLIN: 0001AF
Period: 26 Oct 10– 25 Apr 11

Silicon Informatics, Inc.
Academic Partner: University of Texas at San Antonio
Contract #: W911NF-11-C-0026

Final Report

STTR Project: Random Number Generation for High Performance Computing

---

1. Summary of Work Completed

This project has two primary objectives: (a) design and implement prototypes of highly scalable, high quality parallel random number generators (PRNGs) for a variety of computing and programming models including multicore/multithreaded, message passing, and general purpose graphics processing unit (GPGPU) models; (b) design and implement test methods that evaluate the independence of a large number of parallel random number (RN) streams and provide easy to use quality metrics. Both objectives have been accomplished.

The rest of the final report is organized as follows. First, the main contributions are summarized. Descriptions of the work done for various tasks that were pursued to accomplish the project objectives are given next. Technical details and performance data are provided in two attachments: a supplementary report and a technical paper that will be presented at DoD HPC Users Group Conference, June 2011.

Main contributions

- A new statistical test, called ISC test, to evaluate interstream correlations of a large number of RN streams is designed and implemented. The ISC test is a significant contribution to the state-of-the art in PRNG testing. It can be used to evaluate billions of RN streams simultaneously and obtain an overall quality metric. This test has low computational overhead and can be adapted for online testing—in which the RNs consumed in an application are analyzed in parallel with the application and a quality metric is provided at the end of the application execution. To the best of our knowledge, this is the first such test. The ISC test identified potential correlations among the streams of a popular and widely used PRNG in the SPRNG package. The test results were further confirmed with a new DTMC simulation application we developed in this project.

- ISC test is a first-level test method with applications of Ising model simulations and other applications forming the next level test methods. A new application based on the simulations of a discrete-time Markov chain (DTMC) model is implemented. This application can be used to test both intrastream correlations and interstream correlations for a large number of RN streams.

- Online version of ISC test and additional physical modeling applications such as fracture analysis, multiscale modeling, and CTH will be added to the test package that will be implemented in Phase II.

- A new context-aware parallel random number generator (CPRNG) is designed and implemented. CPRNG is highly scalable and supports applications that require a large and unpredictable number (at the beginning of the execution) of distinct RN streams. The current version is based on the multiplicative lagged Fibonacci generator (MLFG) technique. Additional CPRNGs based on other RN generation techniques will be designed and implemented in Phase II.

- CPRNG implementation supports various computing/programming models: sequential, multicore/multithreaded, message passing (MPI), and GPGPU. We tested the functional correctness of all these implementations extensively. The prototype CPRNG implementation is free of memory leaks and race conditions; it can supply billions of RN streams easily.

- CPRNG prototype implementation is tuned extensively for efficient initialization of RN Streams and generation of RNs. With respect to timing costs, CPRNG compares well with the basic MLFG, which does not provide the same level of flexible and scalable generation of streams dynamically.

- Several code optimizations that reduce the overheads and improve the speed of CPRNG have been identified. With these optimizations incorporated (in Phase II implementation), CPRNG will perform faster with less overhead.

Description of work completed

To accomplish the project objectives, several tasks were identified and pursued during the project period. The work completed for each proposed task and the contributions are described below.

A. Comparison and assessment of current parallel random number generators (PRNGs) and their evaluation techniques.

As part of this task, we identified several PRNG software packages and sequential test packages. The SPRNG package from Florida State University, the Dieharder test package from Duke University, and the TestU01 package from Universite de Montreal obtained for this task are extensively used in the remainder of the project work.

Regarding the currently available PRNGs, we identified the multiplicative lagged Fibonacci generator (MLFG), a parameterized approach to generate independent parallel RN streams, as the most suitable candidate for the design of highly scalable and high quality context-aware parallel random number generators (CPRNGs). We used version 2 of SPRNG package, which include 6 PRNGs, as the platform on which we implemented CPRNGs. The Ising model simulations (both Metropolis and Wolff algorithms) implemented in SPRNG have been extensively used to test and compare CPRNGs with the MLFG and other generators in SPRNG.

B. Implementation of PRNGs on multicore and GPGPUs

SPRNG provides MPI (message passing interface)—based interface for parallel applications designed with MPI interface for interprocess communication. In addition, SPRNG is implemented in such a way that multithread programs can also use the package transparently. However, the burden is placed on the application developer/user to ensure that total number of streams used is known and the streams are allocated to different threads/processes suitably.

We developed a test code to evaluate the time taken to initialize a new RN stream and the time to generate a random number from a stream using Intel's timestamp counters. The timing tests are repeated several times and averaged to obtain representative timing data.

Mersenne twister (MT), in particular, Nvidia developed MTGP generator, is extensively used by parallel applications that use GPUs. However, MTGP is a single-stream generator; it needs to be segmented and segments must be allocated to different threads. Our investigation did not find a truly scalable PRNG with small state-space and highly independent RN streams needed for large-scale GPGPU computing.

C. Evaluation of PRNGs using known statistical and application-based tests

Single-stream test methods have been extensively studied in literature. Many single-stream tests were implemented in various test packages including the Dieharder and TestU01 packages, which we used extensively. Parallel random number streams are interleaved using the perfect shuffle pattern to create a single RN stream and single-stream tests are used for statistical evaluation of a PRNG. A single-stream test package contains 20 different types of basic tests (which may be repeated with different parameters to create up to 150 test instances) and gives pass/fail status for each test applied to the interleaved stream. This provides a vector of pass/fail information that will be hard to use for comparisons of different PRNGs.

We tested the six generators in SPRNG using Dieharder and TestU01. All perform well with only an occasional failure for one of the tests. These tests use a few billions of RNs from the test stream for these tests. Therefore, they are not suitable to test a large number of parallel RN streams; if a billion streams are interleaved to form a single stream, then these tests only examine a few numbers from each stream, which may not be enough to assess the inter-stream correlations. On the other hand, if a billion RN streams are partitioned into several sets with each set consisting of a small number of RN streams, and single-stream tests are applied on each set, then these tests will take several 100s of hours on a desktop machine and provide multiple vectors of pass/fail information that will be hard to combine into an easy to understand quality metric.

Regarding application-based testing, the Ising model simulation codes in SPRNG are the best known and most commonly used applications PRNG evaluations.

D. Development of new statistical tests to quantify inter-stream correlation

We implemented an interstream correlation (ISC) test to evaluate the correlations among a large number of RN streams. This test requires parallel RN streams to be combined (using perfect shuffle interleaving or a biased-interleaving) into a bivariate RN stream (RNs 1, 3, … form X variates and RNs 2, 4, .. form Y variates). This bivariate RN stream is transformed into bivariate normal RN stream and the correlation coefficient, r, between the X and Y variates is computed. Several sets of RN streams are used compute several r's. Collectively, these r's are the samples that can be used to estimate ρ, the true common correlation coefficient among the parallel RN streams generated by the PRNG being evaluated. We used Donner and Rosner test method (DR-test, Applied Statistics J., vol. 29, no. 1, 1980) to combine the r's and obtain the test statistic denoted t_H, which is a standard normal random variate. This can be used to test the null hypothesis $H_0 : \rho = 0$. Large absolute values of t_H will lead to the rejection of the null hypothesis and the acceptance of the alternative hypothesis $H_1 : \rho \neq 0$. For a significance level $\alpha = 0.05$, absolute values of t_H above 1.96 leads to the rejection of the claim that parallel RN streams are independent; the probability that the rejection is erroneous is $\alpha = 0.05$. One could use different significance levels: for $\alpha = 0.02$, the absolute values of t_H above 2.33 will lead to rejection of the claim of independence of RN streams with only 0.02 probability of being wrong.

We developed a Kolmogorov-Smirnov test (KS-test) on the distribution of r's. In this test, the KS-test statistic, D_{max}, computed using the r's must be less than the critical value, $D_{\alpha,n}$, for significance level α and n, the number of r's used.

A preliminary version of this test was described in Monthly Report 3 (January 2011).

We used these ISC test with the two test metrics extensively to evaluate the correlations among the RN streams of a PRNG. This is a highly scalable test. We **tested up to 1.5 billion RN streams** with at least 100 numbers taken from each stream. To best of our knowledge, this is the first time a billion RN streams are tested simultaneously and a single figure of merit is given.

In our test process, we identified significant correlations among RN streams of MLFG, a PRNG in the SPRNG package. Both DR-test and KS-test statistics, t_H and D_{max}, give very high values leading the rejection of the claims of independence of the RN streams generated by this PRNG. MLFG fails the ISC test consistently when 15 million or more streams are considered.

We confirmed this potential problem with MLFG using a new application we developed in this project. This application simulates a discrete-time Markov chain (DTMC) with an absorbing state. (The DTMC estimates the number of packet transmissions, which are the steps or state transitions in the model, it takes for a node to suspect its next hop node of dropping its packets in a multi-hop wireless network.) Compared to the Ising model simulations, DTMC model can use

a large number of RN streams much more speedily, and the theoretical values can be calculated easily.

When 1.5 million or more streams are used, MLFG fails to match the theoretical estimation. Since the application is a simulation of the model, not the actual wireless network, using a good PRNG should lead to quick convergence of simulation estimates to match the theoretical estimates.

E. Development of new scalable PRNGs

A highly scalable context-aware parallel random number generator (CPRNG) that can provide distinct RN streams automatically for different contexts is designed. The first version is based on MLFG but with different initialization methods. This allows a large number of distinct RN streams that can be dynamically requested with very little communication cost: beyond the initial specified limit of RN streams, which do not incur any interprocess communication or thread synchronization/serialization, an application can request for new RN streams with only two interprocess communication messages or a mutex lock access. Extensive description of the design of CPRNG is given in Monthly Reports 4 and 5.

F. Preliminary implementation and evaluation of CPRNG

We implemented CPRNG in the SPRNG package. It can be used by sequential applications, multicore/multithreaded applications, MPI-based parallel applications, GPGPU based applications.

The functional correctness of the implementation for all these scenarios is tested extensively using a parallel application (denoted all_reduce) that uses multiple RN streams and multiple numbers from each stream, computes their overall sum modulo 100. We used all_reduce to test as many as 1 million RN streams and ensured that CPRNG provides consistent RNs regardless of the number processes/threads used.

We evaluated the timing costs of initialization and RN generation. The initialization cost of CPRNG is about 26,000 clock ticks, which is about the same as that of MLFG in SPRNG package. The RN generation cost is about 3 clock ticks more (23 vs. 20 ticks on a machine with Intel quad-core i7-870 CPU and 20 vs. 17 ticks on a machine with Intel Xeon E630 CPU).

We evaluated the quality of CPRNG using the ISC test, Ising model simulations, and DTMC model simulations. The results for the Ising model simulations, given in the Monthly Report 5, show that CPRNG performs about the same as that MLFG implemented in the SPRNG package. However, these simulations use at most 256 RN streams.

The ISC test is used for further evaluation. With up to 1.5 billion streams used, CPRNG performed well with test statics below the corresponding critical values in all but one instance. Even in that scenario, which used 1.5 billion RN streams, the KS-statistic was slightly higher

than the corresponding critical value, but the DR-static was well below its corresponding critical value. Further testing with the DTMC application showed that using CPRNG allows the simulation results to converge to the theoretical values much more quickly than using MLFG.

Technical details and performance data are given in a technical paper submitted as a supplement to this report.

Business and Dissemination Plan

One of ARO's objectives in supporting this research is to ensure that the PRNG software that results from this research is relevant to and used in military and commercial simulation applications. Our proposal for Phase II of this project sets forth a detailed plan to introduce the new context aware parallel random number generator (CPRNG) to the high performance computing (HPC) community and make it available to military, academic and commercial users.

Major elements of this plan include:

1. Communication to HPC User community. The first such communication will be a paper presented by Rajendra Boppana at the HPCMP Users Group Conference on June 23, 2011. The paper is entitled: *"Context-Aware Parallel Pseudorandom Number Generators for Large Parallel Computations."* Other presentation opportunities include SC11 (Seattle, November 2011, http://sc11.supercomputing.org/), SC12 (November 2012) and IDC's HPC User Forum April 2012. Please note that it might be best to introduce our commercial version of the CPRNG software through a paper/presentation at the SC12 conference.

   UTSA and Silicon Informatics will interact with and provide the prototype software to select HPC users and parallel application developers to test the usability and quality of the random numbers generated by CPRNG and to evaluate the effectiveness of the online ISC test method. These evaluations will be used to refine the prototype prior to a more general release to the HPC community.

2. Creation of a long-term sustainability plan, the product of research undertaken by Silicon Informatics, KEYW Corporation and the UTSA Center for Innovation and Technology Entrepreneurship. The plan will identify ways to reach the broadest set of military, academic and commercial users while generating sufficient revenue to ensure that availability of the CPRNG software is sustainable over the long term.

3. Release of prototype version of the CPRNG software, complete with documentation, for evaluation and implementation at US Government HPC centers, including DoD Major Shared Resource Centers.

4. Development of a website that will facilitate distribution and support of the software for military, academic and commercial users.

5. Release of a fully-robust, commercial version of the CPRNG software.

6. Granting royalty-based sublicense rights that enable the CPRNG software to be bundled and/or integrated with other applications software.

Programmatic issues: None.

2. Schedule Update

Task	Description	Month 1	Month 2	Month 3	Month 4	Month 5	Month 6	Completion Status
A	Comparison and assessment of current PPRNGs and their evaluation techniques							**100%**
B	Implementation of PPRNGs on muilti-core CPUs and GPGPUs							**100%**
C	Evaluation of PPRNGs using known statistical and application-based tests							**100%**
D	Development of new statistical tests to quantify inter-stream correlation							**100%**
E	Development of new scalable PPRNG algorithms							**100%**
F	Preliminary implementation and evaluation of new PPRNGs							**100%**
G	Phase I final report, including Phase II work plan							**100%**

3. Milestone Update

Milestones completed to date: Tasks A through G.

Milestones expected to be completed during the next reporting period: None.

Milestones expected to be missed during the next reporting period: None.

Prepared by:

Rajendra V. Boppana, Ph.D., P.I., University of Texas at San Antonio (technical section) and Robert Keller, Project Director, Silicon Informatics (Business plan)

May 23, 2011

Final Report Supplement

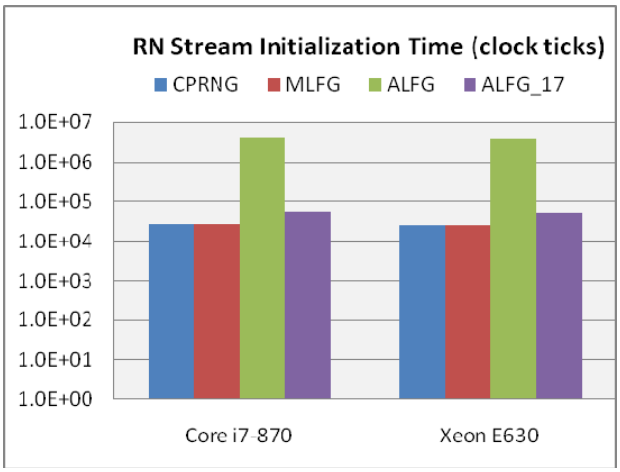
STTR Project: Random Number Generation for High Performance Computing

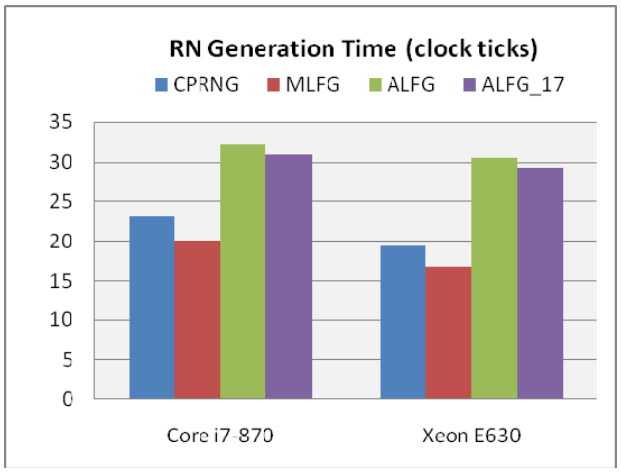
This document supplements the final report for the project by providing test data and brief explanations of the same.

1. RNG Timing Tests

CPRNG is the new random number generator (RNG) designed in this project. MLFG, ALFG, and ALFG_17 are the RNGs in SPRNG package. CPRNG is implemented in the SPRNG package. Any parallel application currently designed to use SPRNG generators can use CPRNG by changing the RNG code to 9. No additional application modifications are needed.

Two computers, a desktop computer with Intel Core i7-870 CPU and a rack server with Xeon E630 CPU, are used to estimate the time required for initialization of a random number (RN) stream and the time taken to generate a single random number from an already initialized stream. The times are given in clock ticks—2.93 ticks/ns for Core i7-870 and 2.53 ticks/ns for Xeon E630 machines. The initialization costs of CPRNG are about the same as those of MLFG, on which CPRNG is based. The cost of generating an RN is about 3 ticks higher compared to MLFG owing to the additional processing needed for context-aware RN generation. This can be easily eliminated if the application does not require contex-aware RN generation.





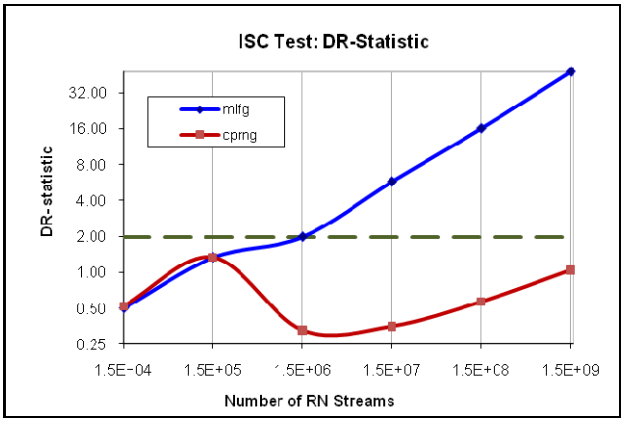
The CPRNG implementation is free of leaks, is multithread safe, and works seamlessly with MPI-based applications. The GPU version of CPRNG is implemented as a different generator (with RNG code 10) with some restrictions on features: no context-awareness, and the maximum number of streams needed by the application must be specified at the beginning of the program execution. The CPU version of CPRNG provides context-awareness, the ability to use distinct streams automatically for different contexts, and nearly unlimited number of RN distinct streams.

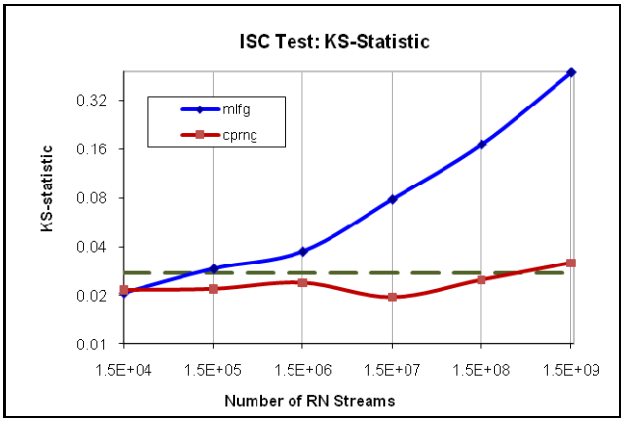
2. ISC Tests for Inter-stream Correlations

The ISC test is applied on several sets of RN streams. The RN streams in a set are interleaved using perfect shuffle or biased interleaving method. Consider three RN streams A, B and C with RNs, respectively, $a_1, a_2, a_3, \ldots$, $b_1, b_2, b_3, \ldots$, and $c_1, c_2, c_3, \ldots$. In perfect shuffle interleaving, a new stream $a_1, b_1, c_1, a_2, b_2, c_2, a_3, \ldots$ is created. In biased interleaving, $a_1, b_1, a_2, c_1, a_3, b_2, a_4, \ldots$ is created. The RNs in the odd numbered positions form the X variates and the RNs in the even numbered positions form the Y variates. These are transformed into normal bivariates using Box-Muller transform. Correlation coefficient, r, for the bivariate pairs is computed. This is repeated several times to obtain multiple r's. In our tests, we used 1500 sets of random number streams with the set size varied from 10 to 1,000,000 to obtain $r_1, r_2, \ldots, r_{1500}$.

These r's are aggregated using a well-developed test method such as Donner and Rosner test (DR-test) or Kolmogorov-Smirnov test (KS-test) and a test statistic is obtained. The statistic for DR-test is denoted as t_H and the statistic for KS-test as D_{max}. For each test, there is a critical value that is computed based on the desired significance level and the number of r's used. For DR-test at a significance level of 0.05, the critical value is 1.96 provided the number of bivariate pairs used to calculate each r is large. For KS-test, at a significance level of 0.01, the critical value is 0.0274 when the number of r's used is 1500. If test statistic is significantly above the critical value, then the RN streams generated by the PRNG are likely to have significant interstream correlations.

The two graphs below give the results of the two tests for shuffle-interleaving of RN streams. When the set size is 1,000,000, a million streams are used to obtain a single r, and a total of 1.5 billion streams are used to obtain the 1500 r's used to calculate the test statistic. The dashed line indicates the critical value for that test. Our results indicate that MLFG (the built-in random number generator in the SPRNG package) fails both DR- and KS-tests for test configurations that use 1.5 million or more streams. On the other hand, CPRNG, which is also based on the same theoretical foundation as that of MLFG, performs well; it narrowly fails the KS-test only for the largest test configurations we used.

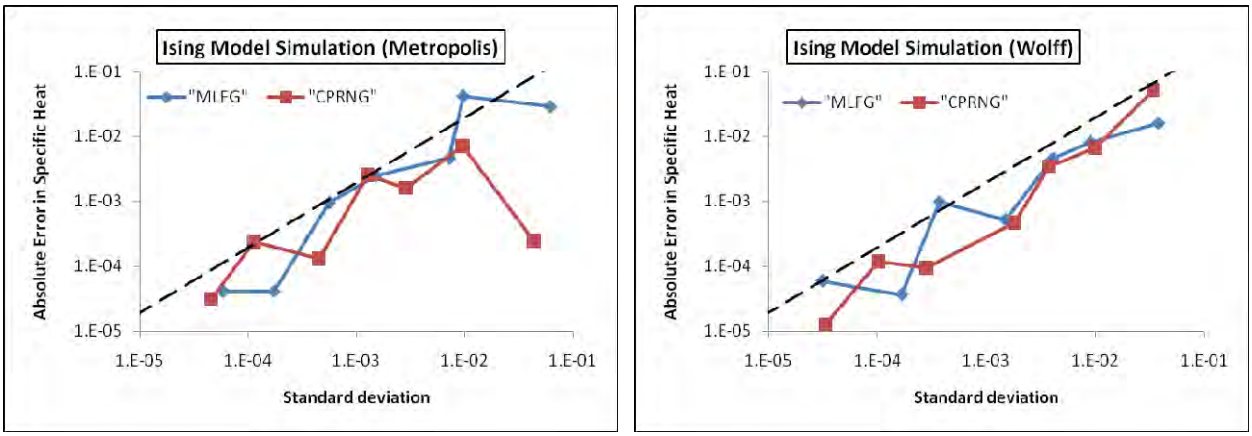




3. Application-based Tests

Ising model simulations

We have tested the prototype CPRNG with the Ising model simulations for a 16×16 lattice based on Metropolis and Wolff algorithms. A distinct RN stream is used for each lattice point. The results for absolute error in specific heat vs. the standard deviation are shown for the Metropolis and Wolff algorithms in the graphs below. We used the same parameters indicated in the paper by Srinivasan et al., "Testing parallel random number generators," Parallel Computing 2003: 16×16 lattice, 1000-word blocks, 1 million blocks with the first 100 blocks discarded. (These graphs are the revised versions of the graph presented in Monthly Report 5.)



In these simulations, the absolute error (the difference between the theoretical calculations and the simulation values) of specific heat or energy is compared to the standard error of the same metric (1.96 times the standard deviation of the simulation values) at a significance level of 0.05. That is up to 5% of the error points may be greater than the standard error and lie above the cut-off line indicated in the graph. The MLFG results in the left graph are exactly the same as the ones presented in Fig. 6 of the paper by Srinivasan et al. These results show that CPRNG is no worse than MLFG for this test.

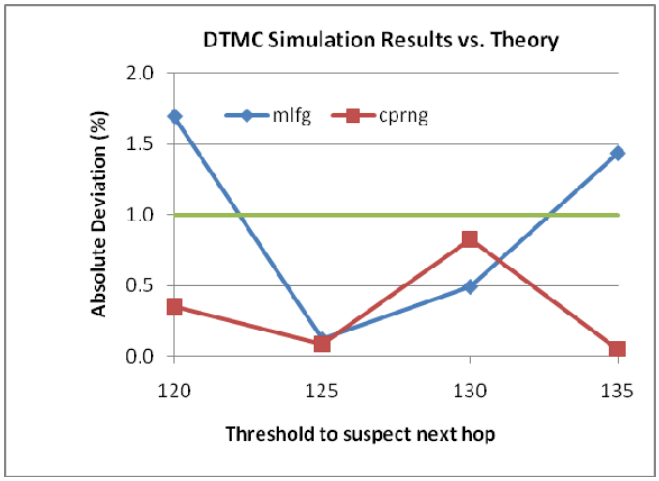
Markov model simulations

We also implemented a new test based on the simulation of a discrete-time Markov chain (DTMC) that models the time it takes a node to suspect its next hop of dropping packets based on transmission overhearing in wireless ad hoc networks. The DTMC estimates the number of packet transmissions, which are the steps or state transitions in the model, it takes for a node to suspect its next hop node of dropping its packets in a multi-hop wireless network. The DTMC has an absorbing state denoting the state in which the next hop is suspected and L transient states, where L is the threshold to suspect the next hop node. Since the application is a simulation of the model, not the simulation of the actual wireless network, using a good PRNG should lead to quick convergence of simulation estimates to match the theoretical estimates.

Based on several test runs using the DTMC application, we observed that simulations that use MLFG do not converge as rapidly as the simulations that use CPRNG.

Compared to the Ising model simulations, DTMC model simulations can use a large number of RN streams much more speedily, and the theoretical values can be calculated more easily. In fact, we implemented the code within the simulation program so that the appropriate theoretical values can be calculated based on the test parameters prior to a simulation. With appropriate choice of parameters, DTMC application can use a large number of RN streams and/or a large number of RNs from each stream. If a simulation is repeated k times, and there are L transient states, it is natural to use L distinct RN streams in each simulation run or a total of kL RN streams for the entire simulation. By changing the threshold L, the number of RNs consumed in a simulation run can be increased.

The results of simulations for various threshold values L for a scenario are given in the following graph. For each threshold value, MLFG or CPRNG is used to simulate the Markov model to determine the number of steps taken to reach the absorbing state. This is repeated 10,000 times and the average number of steps taken to reach the absorbing state is estimated. This estimate is compared with the theoretical calculations by calculating the absolute deviation as a percentage of the theoretical value. The cut-off point is 1%. If the deviation is above 1%, then the simulation is considered to have not converged. For the four tests we conducted, simulations using MLFG converge in two out of four scenarios, while the simulations using CPRNG converged in all four cases.



This application is promising, but further investigation is needed to understand its usefulness in testing the correlations among RN streams.

2011 DoD High Performance Computing Modernization Program Users Group Conference

Context-Aware Parallel Pseudorandom Number Generators for Large Parallel Computations

Rajendra V. Boppana
CS Dept. and Inst. for Cyber Security, University of Texas at San Antonio, TX
boppana@cs.utsa.edu

Abstract

Design and testing of parallel pseudorandom number generators (PRNGs) that generate millions of parallel random number (RN) streams needed for large parallel computations is a nontrivial task if: a) the number of parallel streams are not fixed at the beginning of the program execution, and they are to be generated in a distributed manner with low communication overhead; and b) the correlations among the parallel streams must be very low. Furthermore, the current PRNGs require the user to manage the number of streams and their initialization, which can be onerous if each process or thread of a parallel application consumes RNs at multiple locations and, for better randomization, distinct RN streams must be used in each instance. In this paper, both problems are addressed using context-aware PRNGs. In this approach, each request for an RN stream by a process/thread results in the allocation of a large set of RN streams, so that each distinct program statement that calls for RN generation (denoted, RN context) is served with a distinct RN stream taken from the RN streams assigned to that process. A prototype context-aware parallel random number generators (CPRNGs) based on the multiplicative lagged Fibonacci generator is implemented for automatic RN stream generation based on RN contexts. A new parallel statistical test, called the inter-stream correlation (ISC) test, is designed and implemented to assess the degree of independence among a large number of parallel RN streams and provide an easy-to-use quality metric. Preliminary results indicate that the prototype CPRNG provides high-quality RN streams, and that the ISC test promises to be a highly-effective test to assess correlations among a large number of RN streams.

1. Introduction

Many scientific computing applications, business and finance applications, and complex systems modeling and analysis techniques use random number generators[1] (RNGs) extensively for simulations of various likely scenarios and estimations of potential outcomes. Often, these applications are highly-scalable and can take advantage of the availability of thousands of computing cores on heterogeneous systems comprising multi-core processors (CPUs) and highly-parallel general-purpose graphics processing units (GPGPUs), provided that suitable parallel random number generators (PRNGs) are available to simultaneously feed thousands of computing streams with high-quality random number (RN) streams with low intra- and inter-stream correlations.

We present context-aware parallel random number generators (CPRNGs) based on a new approach to allocate and manage RN streams by parameterized random number generators that can generate virtually unlimited numbers of distinct RN Streams. Lagged Fibonacci generators (LFGs), which generate a new RN by applying an arithmetic or logic operation on two or more previously generated RNs, can provide a large number of distinct RN streams, with each stream having a large cycle—the number of RNs that can be used before the sequence repeats. A prototype CPRNG, based on multiplicative lagged Fibonacci generator (MLFG), is implemented and evaluated. CPRNG provides two new features that a basic MLFG does not provide.

- CPRNG uses the program context, in which a request for a random number is made, to automatically select and use distinct RN Streams for distinct contexts.
- A typical PRNG requires the application to declare the maximum number of RN streams used in an execution run, and the number of distinct RN streams requested to be within this limit. However, this can be a significant constraint

[1]The random number generators we consider this paper are pseudorandom number generators. For easier description, however, we simply refer to them as random number generators.

for applications that may spawn additional processes during the execution, based on the intermediate results and use unpredictable number of RN streams.

- CPRNG relaxes this constraint and allows applications to request virtually unlimited number of RN streams beyond any initially-specified RN stream limit.

Another problem addressed in this work is the evaluation of intra-stream and inter-stream correlations—i.e., the quality of the random numbers generated. Several excellent statistical tests[1,11] are available to test intra-stream correlations of a sequential RNG. Test packages such as Diehearder[16] and TestU01[15] run a battery of such tests on an RN stream and provide pass/fail results from each test. If an RN stream fails any of the tests, then additional, more detailed tests are conducted. Otherwise, it is assumed that the RN stream is free of intra-stream correlations with very high probability.

To assess the quality of a PRNG, several parallel RN streams generated by it are interleaved using the perfect shuffle pattern to create a single RN stream, and single-stream test batteries are used to evaluate the inter-stream correlations among the RN streams[7,8]. A single-stream test package contains 20 different types of basic tests (which may be repeated with different parameters to create up to 150 test instances) and gives pass/fail status for each test applied to the interleaved stream. This provides a vector of pass/fail information that will be hard to use for comparisons of different PRNGs. Furthermore, these tests use a few billions of RNs from the test stream for these tests. Therefore, they are not suitable to test a large number of parallel RN streams; if a billion streams are interleaved to form a single stream, then these tests only examine a few numbers from each stream, which may not be enough to assess the inter-stream correlations. On the other hand, if a billion RN streams are partitioned into several subsets with each subset consisting of a small number of RN streams, and single-stream test batteries are applied on each set, then these tests will take several 100's of hours on a desktop machine and provide multiple vectors of pass/fail information that will be hard to combine into an easy-to-understand quality metric.

Another approach is to use thoroughly analyzed applications to test inter-stream and long-range correlations of RNGs. For example, a physics application involving simulations of two-dimensional (2D) Ising square lattice models with periodic boundary conditions, for which the exact solutions are known, is often used to evaluate PRNGs[3,4,7]. However, application-based tests are computationally-expensive and may not be adaptable to test billions of parallel streams at a time. Therefore, faster and more informative statistical tests of parallel RN streams are needed. Currently, there are very few parallel statistical tests that do not require serialization of RN streams and have the potential to evaluate inter-stream correlations.

We present a new inter-stream correlation (ISC) test that evaluates a large number of parallel RN streams simultaneously, and provides an easy-to-use quality metric. The ISC test divides the total streams to be evaluated into subsets of streams, and computes a correlation coefficient for each subset. These correlation coefficients are aggregated using a theoretically-sound test method such as the Donner and Rosner test (DR test)[13] or Kolmogorov-Smirnov test (KS test)[14] and a test statistic is obtained. If the test statistic is too high compared to a suitably determined critical value, the claim of independent RN streams is rejected. Lack of rejection indicates that the RN streams are likely to be independent.

We present preliminary results of the implementation of a prototype CPRNG and the application of ISC test. Timing tests show that CPRNG is nearly as fast as a basic PRNG, such as MLFG, and incurs only a small amount of overhead to provide the context-awareness. The ISC test found significant correlations in the RN streams generated by multiplicative Fibonacci lagged generator (MLFG), in the widely-used SPRNG package.

The rest of the paper is organized as follows. Section 2 presents the basics of parallel random number generators. Section 3 presents the context-aware PRNGs, and compares a prototype PRNG with the widely-used MLFG in the SPRNG package. Section 4 presents the ISC test and its application to CPRNG and MLFG, with up to 1.5 billion streams analyzed. Section 5 describes the related work in PRNGs and test methods. Section 6 concludes the paper.

2. Background

We are interested in parameterized RNGs that have the capability to generate a large number of RN streams with relatively simple changes to the initialization. Lagged Fibonacci generators[8,9,10] are easy to parameterize and, with careful selection of the parameters, can be used to generate virtually unlimited number high-quality distinct RN streams easily. In particular, we are interested in the multiplicative Fibonacci lagged generator (MLFG), which uses the recurrence relation:

$$x_n = x_{n-k} \times x_{n-l} \pmod{2^m}, \quad 0 \le k < l < n, \tag{1}$$

where l and k are the lags (or indices to the older random numbers used to generate the new random number), and x_i 's are positive and odd m-bit integers. This generator produces $2^{(m-3)*(l-1)}$ different RN streams, each with a cycle length of $2^{(m-3)} \cdot (2^l-1)$. Therefore, there are $(m-3) \cdot (l-1)$ bits that need be determined uniquely for each RN stream. (One of the initial lag

words and the least significant bits of all initial lag words are specified by the canonical form and parameters specified, and are common to all RN streams with those parameters[10].)

For a 64-bit MFLG with lag 17, there are $2^{61*16}=2^{976}$ different RN streams, each with a distinct 976-bit seed value and a cycle length of $2^{61} \cdot (2^{17}-1) \approx 2^{78}$. In practice, the upper or the middle b, $b<64$, bits of x_i's are extracted and supplied as the RNs to improve the randomness, since the lower bits are often less random owing to the arithmetic operation involved. We used the SPRNG package[8] and the MLFG available from its library, to implement a prototype CPRNG.

Additive lagged Fibonacci generators (ALFGs) are obtained by replacing the multiply operation in Equation 1 with an add operation; x_i's are positive m-bit integers with at least one odd integer in the first l lags. Compared to MLFGs, ALFGs provide more distinct RN streams with longer cycles for the same bit-size. However, the intra-stream and inter-stream correlations are more significant in ALFGs. To mitigate these issues, larger lags, l=1,279 or larger, are used. SPRNG package combines two ALFGs with different lag words to provide a higher-quality ALFG. On the other hand, MLFG can be used with smaller lags, e.g., l=17. Therefore, for the most commonly used configuration in SPRNG package, ALFGs take twice as much time to initialize a new RN stream and to generate RNs as MLFG.

SPRNG library package provides init_rng() and get_rn_dbl() function calls to initialize a new RN stream and to obtain the next RN in an already initialized stream, respectively. The init_rng function is called by specifying the seed, parameter sets that specify the lags and the locations of the odd-numbered words in the initial set of lag words, maximum number of RN streams (denoted max_str) that will be requested by the application, and cur_str, the RN stream number in the range [0, max_str) that needs to be initialized. The seed, parameter set, and max_str must be common in all init_rng() calls. For most parallel applications, it is easy to allocate the RN streams to processes based on the input data and/or computations allocated to them. For example, if a computational loop is partitioned cyclically among p processes, then iteration i is executed by process *i%p* ; in this case it is natural to allocate RN streams from the set *i*,*i+p*,… to process *i*.

Each call to init_rng() results in the initialization of the RN stream specified by the stream number, cur_str, and the calling code is given a pointer to the RN stream that should be used as argument in the function call get_rn_dbl() to obtain the next RN in the stream. (SPRNG library provides several other function calls including requests for integer RNs instead of reals, but they are not of interest in this paper.)

3. Context-aware Parallel Random Number Generators

If a process uses RNs in multiple locations for different purposes, then it is generally recommended that a distinct RN stream be used for each such context. However, with the current RNG packages, this requires the application to explicitly initialize the additional RN streams needed and, more importantly, use the appropriate RN stream pointer in each context.

This puts a significant burden on the application developer to manage the RN streams and contexts. Any changes to the code that change the number of contexts require additional work by the application developer to make suitable changes to the RN stream management. While it is natural and intuitive to partition RN streams based on the partitioning of input data or computations, explicitly managing multiple RN streams based on program contexts makes the application less portable and distracts the application developer.

To address these concerns and to improve the quality of RNs used by applications, we developed the CPRNG. The design methodology is to take a parameterized random number generator that has the capability to generate a large number of RN streams with relatively simple changes to the initialization and augment it with a scalable and automatic initialization process. Our first CPRNG is based on the MLFG described in Section 2. We used the SPRNG package and the MLFG available from its library, to implement the prototype CPRNG. The design of CPRNG is elaborated below.

3.1 CPRNG Design

In CPRNG implementation, each init_rng() call allocates not just one RN stream but a set of distinct RN streams and returns a pointer, str_ptr, to the set; the streams in this set can be customized with program context without further calls to init_rng(). The RN-context, the context or the program location from which a RN number is requested, is used in addition to the stream-set pointer, str_ptr, to determine the specific RN stream to be used. The RN context is derived from a combination of the program line number in the source code, the return address of the function call to get_rn_dbl(), the process/thread numbers, and any user supplied identifiers such as the iteration number. When the application requests for a random number using the function call get_rn_dbl(str_ptr), the RN-context will be used to determine the specific RN stream to be used in the set of streams pointed by str_ptr. The appropriate RN stream is automatically initialized with the RN-context, if it is the first call from this context, and a RN from the stream is returned.

Figure 1 describes the initialization process by CPRNG with lag parameters l and k, $0 < k < l - 2$. A call to init_rng() results initialization of $l - 3$ of the lag words[2] using a sequential RNG such as the recursive with carry (RWC) generator described in the Diehard package[2] seeded with the user-specified seed. These lag words are common to the initialization of all RN streams, regardless of the process number or RN-context. One of the remaining three lag words is filled with an ID that is guaranteed to be distinct for distinct cur_str numbers specified in init_rng(). The distinct ID word is common to the set of RN streams that are allocated in response to init_rng() call. The remaining two lag words are filled with the RN-context so that distinct RN-contexts result in distinct RN streams.

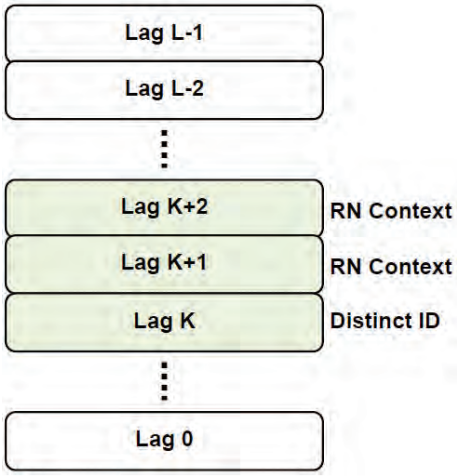


Figure 1. Initialization of RN stream lags by CPRNG. Each lag word is a 64-bit word with maximum lag L. L-3 of the lag words are filled randomly, based on the user-specified seed and a sequential RNG. These words are common to all RN streams used during the execution of the application. Lag K, K<L-2, is initialized with a unique and distinct ID that is associated with the cur_ str used in the init_rng() call. Lags K+1 and K+2 are initialized with RN-context to create a distinct RN stream fo reach distinct program context in each process.

For a CPRNG based on MLFG with maximum lag l=17 and 64-bit words, $2^{2\times61}=2^{122}$ distinct RN streams are allocated with each init_rng() call. Based on the context and str_ptr argument used in a call to get_rn_dbl(), an appropriate stream is selected, automatically initialized prior to first use, and the next RN in the stream is returned. CPRNG may be used without RN-contexts by choosing appropriate parameters to init_rng() call. If RN-contexts are not used, then the two lag words that are normally filled with RN-context are filled with the random bits generated by the sequential RWC generator; the lag word with distinct ID ensures that RN streams are distinct for distinct values of cur_str specified in the init_rng(). CPRNG will be simply a basic MLFG when used without context.

For applications that use a large and variable number of RN streams, having to specify the maximum number of streams used during an execution run is a limitation. Furthermore, certain large-scale parallel applications may spawn additional processes and threads dynamically depending on the input data and intermediate results. To accommodate such situations, CPRNG assigns 2^{10} distinct IDs for the lag word k upon a call to init_rng(), independent of any streams allocated to handle RN contexts. Typically, only one of these IDs is used by a process. However, if a process spawns threads or child processes and needs to use additional distinct RN streams without going through the initialization process, it can have them without any communication overhead by using the original initialization with the distinct ID word replaced with one of the unused IDs from its allocated IDs. This leads to faster initialization of the new RN streams on demand. If more RN streams are needed and init_rng() is called with cur_str value greater than max_str, a monotonically increasing counter is used to ensure that the lag word K is distinct. However, the access to this counter needs to be serialized by using appropriate mutex locks in threaded applications or by assigning it to a process to serve the counter-values to the other processes of the application. Only in these instances, an additional communication or serialization overhead is incurred by CPRNG, compared to the static methods used in the current packages such as SPRNG. On the other hand, CPRNG provides a virtually unlimited

[2]The initialization of the lag words is more complicated than the simpler description given here. For MLFG, all the lag words must be odd values. The two consecutive 32-bit RNs generated by the RWC generator are used form a 61-bit integer and a least significant bit determined by the canonical form and parameter set is appended to it to form a 62-bit number, say, z. The actual lag word is formed by using the operation $(-1)^y 3z \bmod 264$, where y is a randomly generated 1 or 0. However, for easier description, we omit these implementation details. See Reference 10 for the complete details.

number of RN streams on demand, and avoids depletion of the available RN streams that can occur with static partitioning of the available RN streams for applications with many levels of dynamic process/thread creation.

CPRNG is implemented in the SPRNG package as an additional PRNG. The implementation provides the same application interface as the other PRNGS in the package. Any parallel application currently designed to use SPRNG generators can use CPRNG by using an appropriate RNG code. No additional application modifications are needed. Just like the other PRNGs in the SPRNG package, CPRNG produces consistent and predictable RN streams for an application regardless of the number of processes/threads used for parallel computation. The CPRNG implementation is free of memory leaks, is multithread safe, and works seamlessly with MPI-based applications. The GPU version of CPRNG is implemented as a different generator with some restrictions on features: no context-awareness, and the maximum number of streams needed by the application must be specified at the beginning of the program execution.

3.2 Timing Tests

Two computers, a desktop computer with Intel Core i7-870 CPU and a rack server with Xeon E630 CPU, are used to estimate the time required for initialization of a random number (RN) stream by calling init_rng(),and the time taken to generate a single random number from an already initialized stream for CPRNG with lag 17 and three generators from the SPRNG package: MLFG—multiplicative lagged Fibonacci generator with lag 17, ALFG—lagged Fibonacci generator which is a combination of two additive Fibonacci generators with lag 1,279, and ALFG_17—lagged Fibonacci generator with lag 17.

We used Intel CPU time-stamp counter for the time-stamps. For RN stream initialization test, the time taken to initialize a single RN stream is subtracted from the time taken to initialize two RN streams. This is repeated 100 times and the average of the times is taken as a sample point. This experiment is repeated 10 times and the average of the 10 samples and the corresponding 95% confidence interval is calculated. For RN generation test, the time taken to generate 1,000 RNs from an already initialized stream is subtracted from the time taken to generate 1,000 RNs each from two previously initialized RN Streams. This time is divided by 1,000 to get the time taken to generate a single RN. This is repeated 100 times and the average is taken as a single sample point. This experiment is repeated 10 times and the average of the 10 sample points and the corresponding 95% confidence interval is calculated.

Figure 2 gives the times in clock-ticks—2.93 ticks/ns for the Core i7-870, and 2.53 ticks/ns for the Xeon E630 machines. The chart on the left gives the initialization time of an RN stream, while the chart on the right gives the time taken to get an RN from an initialized stream. The initialization costs of CPRNG are about the same as those of MLFG, on which CPRNG is based. The cost of generating an RN is about 3 ticks higher compared to MLFG owing to the additional processing needed for context-aware RN generation. This can be easily eliminated if the application does not require context-aware RN generation. ALFG has high initialization overhead since it uses two additive Fibonacci generators with a large amount of lag (the oldest RN used in calculating the next RN) to provide high-quality RN streams. To rule out any experimental error, we tested ALFG with lag 17 (which is not recommended), whose initialization cost is comparable to those of MLFG and CPRNG.

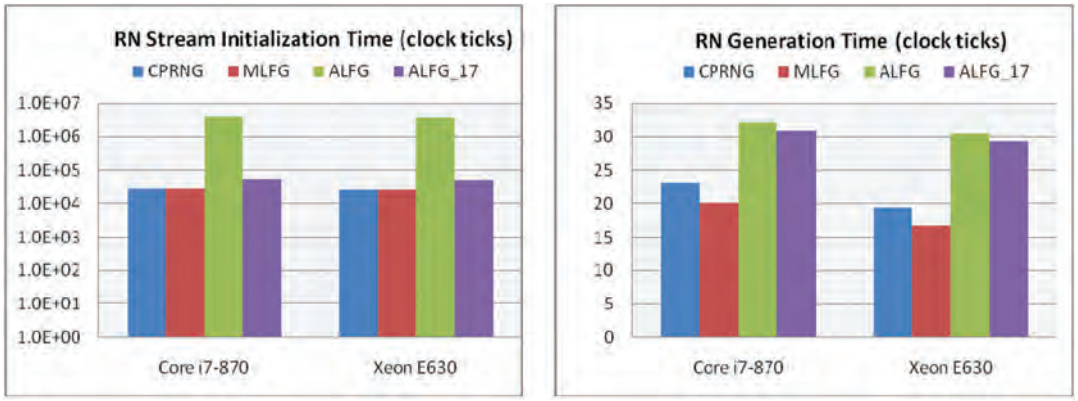


Figure 2. Time taken to initialize RN streams (left chart) and to generate RNs (right chart). A desktop computer with Intel Core i7-870 CPU and a rack server with Xeon E630 are used. The times are given in clock-ticks—2.93 ticks/ns for the Core i7-870, and 2.53 ticks/ns for the Xeon E630. The y-axis for the left chart is in log-scale. The 95% confidence intervals are ± 1% of the mean-values reported.

4. Inter-Stream Correlation Test

The inter-stream correlation (ISC) test evaluates the correlations among a large number of RN streams. The RN streams are divided into several subsets, and the streams in a subset are interleaved, using perfect shuffle or biased interleaving method. Consider three RN streams A, B and C with RNs $a_1, a_2, a_3,\ldots, b_1, b_2, b_3,\ldots, c_1, c_2, c_3,\ldots$, respectively. In perfect shuffle interleaving, a new stream $a_1, b_1, c_1, a_2, b_2, c_2, a_3,\ldots$ is created. In biased interleaving, $a_1, b_1, a_2, c_1, a_3, b_2, a_4,\ldots$ is created. The RNs in the odd-numbered positions form the X variates and the RNs in the even-numbered positions form the Y variates. These are transformed into normal bi-variates using Box-Muller transform[12]. Correlation coefficient, r, for the bi-variate pairs is computed. This is repeated several times to obtain multiple r's. Collectively, these r's are the samples that can be used to estimate ρ, the true common correlation coefficient among the parallel RN streams generated by the PRNG being evaluated.

The r's are aggregated using a theoretically-sound test method such as Donner and Rosner test (DR-test)[13] or Kolmogorov-Smirnov test (KS-test)[14] and a test statistic is obtained. The statistic for DR-test is denoted as t_H and the statistic for KS-test as D_{max}. For each test, there is a critical value that is computed based on the desired significance level and the number of r's used. For example, for DR-test at a significance level of 0.05, the critical value is 1.96 provided the number of bi-variate pairs used to calculate each r is large. If the test statistic is significantly above the critical value, then the RN streams generated by the PRNG are likely to have significant inter-stream correlations.

The DR-test combines the r's and gives the test statistic t_H, which is a standard normal variate. This can be used to test the null hypothesis $H_0 : \rho = 0$. Large absolute values of t_H will lead to the rejection of the null hypothesis and the acceptance of the alternative hypothesis H_1: $\rho \neq 0$. For the significance level α=0.05, absolute values of t_H above 1.96 lead to the rejection of the claim that parallel RN streams are independent; the probability that the rejection is erroneous is α=0.05. One could use different significance levels: for α=0.02, the absolute values of t_H above 2.33 will lead to rejection of the claim of independence of RN streams with only 0.02 probability of being wrong.

The distribution of r's is approximately normal. These r's can be converted into standard normal variates using sample variance of r's and the fact that we are testing for ρ=0. This enables us to apply the KS-test on the distribution of r's. In this test, the KS-test statistic, D_{max}, computed using the r's must be less than the critical value, $D_{a,n}$, for significance level α and n , the number of r's used. For KS-test, at a significance level of 0.01, the critical value is 0.0274 when the number of r's used is 1,500.

Figure 3 gives the results of the two tests for shuffle-interleaving of RN streams generated by CPRNG and MLFG. In our tests, we used 1,500 sets of random number streams with the set size varied from 10 to 1,000,000 to obtain $r_1, r_2,\ldots, r_{1,500}$. When the set size is 1,000,000, a million streams are used to obtain a single r, and a total of 1.5 billion streams are used to obtain the 1,500 r's used to calculate the test statistic. The dashed-lines indicate the critical values for the test statistics. Our results indicate that MLFG (the built-in random number generator in the SPRNG package) fails both DR- and KS-tests for test configurations that use 1.5 million or more streams. On the other hand, CPRNG, which is also based on the same theoretical foundation as that of MLFG, performs well; it narrowly fails the KS-test only for the largest test configurations we used.

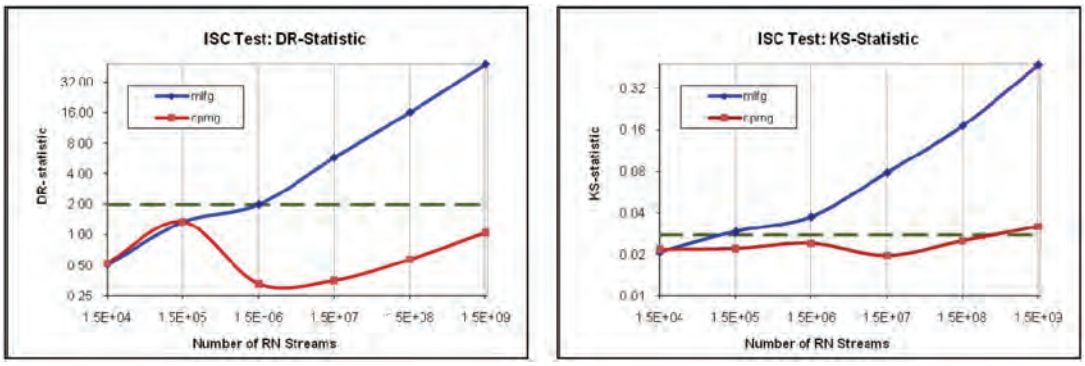


Figure 3. ISC tests for CPRNG and MLFG. The dashed-lines indicate the critical values. The y-axes for both charts are in log-scale. For both tests, MLFG's test statistic is significantly higher than the critical value, indicating that the RN streams generated by MLFG may have significant inter-stream correlations and must be tested further.

5. Related Work

The designs of sequential and parallel RNGs are extensively investigated by many researchers owing to their importance to computational science and to the elegant, mathematical nature of the problem. Knuth[1] discusses several RNGs, and many excellent single-stream test methods that are implemented in popular test batteries such as Dieharder[16] and TestU01[15]. Linear congruential generators that use a recursive equation of the form $x_n = a \cdot x_{n-1} + b \pmod{2^m}$, where a and b are constants, are commonly available as part of the C math library in a typical UNIX environment. One of 2^{19937} the most popular sequential RNGs is the Mersenne twister[5] which offers an RN stream with a cycle of length. A graphics processing unit (GPU) version of this RNG[6] is commonly used by applications designed to use GPUs.

Additive and multiplicative lagged Fibonacci generators[8–10] have been extensively investigated because of the ease with which they can be used to generate distinct RN streams. Of the two, MLFG is considered to be more robust, producing higher-quality RN streams. Both generators are implemented in the popular SPRNG package[8]. We have used the SPRNG package extensively. The prototype implementation of CPRNG is based on the MLFG implementation and supports SPRNG's application interface.

SPRNG also implements several sequential tests and provides a systematic way to interleave several streams into a single-stream and apply the sequential tests. However, owing to the availability of more exhaustive test packages, Dieharder and TestU01, we did not use the single-stream tests in SPRNG. Another important resource provided by SPRNG is the Ising model simulation codes based on Metropolis and Wolff algorithms. These applications are widely-used to evaluate sequential and parallel RNGs[3,4,7].

6. Conclusion

Context-aware parallel random number generators are based on a new approach to allocate and manage RN streams by parameterized random number generators that can generate virtually unlimited numbers of distinct RN Streams. Compared to the parallel random number generators in the current packages such as SPRNG, CPRNGs can automatically provide distinct RN streams depending on the program context to improve the quality of the RNs used. To achieve the same effect with the current PRNGs, the application needs to, explicitly, manage multiple streams and their usage based on the program context. Furthermore, CPRNGs support highly-complex parallel applications that require a large and variable number of RN streams by dynamically allocating RN streams beyond the maximum number of RN streams specified at the beginning of program execution. In contrast, the current PRNGs do not allow applications to request RN streams beyond the initially specified number of RN streams. Some implementations, e.g., SPRNG, handle this issue by partitioning the total RN streams using a binary partitioning scheme. For applications that have many levels of dynamic process/thread creation, this can result in depletion of RN streams available to dynamically-created processes/threads.

The inter-stream correlation test evaluates the correlations among a large number of RN streams. Using a well-known test method such as the Donner and Rosner test or the Kolmogorov-Smirnov test, it provides an aggregate PRNG quality metric. This test complements the existing sing-stream test batteries and application-based tests currently available. It is applied to evaluate inter-stream correlations among as many as 1.5 billion RN streams. The ISC test shows that the MLFG used in SPRNG has significant inter-stream correlations when 1.5 million or more streams are considered. In addition to providing an easy-to-use quality metric, the ISC test is fast and can be adapted to on-line testing, in which the actual RNs used by an application are fed to ISC test, and overall quality of the RNs used is provided at the end of the execution of the application.

In the future, we plan to revise the current implementation and release a CPRNG library package to the HPC community. We also plan to design and implement additional CPRNGs based on other RNGs. We will work with HPC practitioners in adapting new applications that use multi-scale simulation models to augment the current test methods.

Acknowledgments

This work was done as part of the STTR Phase I contract W911NF-11-C-0026 funded by the US Army Research Office (ARO) Program Manager, Dr. Joe Myers. The primary computer used for the development and testing was acquired with funds from National Science Foundation grant CNS-0551501. Additional computer resources were provided by UTSA Institute for Cyber Security. The contents of this paper do not necessarily represent the position or the policy of the Government and no official endorsement should be inferred. The author thanks Professors Ram Tripathi and Ravi Sandhu at UT San Antonio and Mr. Robert Keller and Mr. Hemant Trivedi at Silicon Informatics Inc. for their help and participation in this project. Prof. Tripathi identified the DR-test and helped the author implement the DR-test and the KS test. Mr. Trivedi helped with the implementation of CPRNG and the time-stamp counter used in the timing tests.

References

1. Knuth, K.E., *The Art of Computer Programming, Volume 2: Semi-Numerical Algorithms*, 3rd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1997.

2. Marsaglia, G., *Diehard software package*, available at http://stat.fsu.edu/pub/diehard, 1995.

3. Ferrenberg, A.M., D. Landau, and Y. Wong, "Monte Carlo simulations: Hidden errors from 'good'random number generators", *Phys. Rev.*, 69, 23, pp. 3382–3384, 1992.

4. Coddington, P. "Tests of random number generators using Ising model simulations", *Int. J. of Mod. Phys.*, 7, 3, pp. 295–303, 1996.

5. Matsumoto, M. and T. Nishimura, "Mersenne twister: a 623-dimensionally equi-distributed uniform pseudo-random number generator", *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, v.8 n.1, pp. 3–30, Jan. 1998.

6. Podlozhnyuk, V., *Parallel Mersenne Twister, Version 1.0*, Nvidia, 2007.

7. Srinivasan, A., M. Mascagni, and D. Ceperley, "Testing parallel random number generators", *Parallel Computing*, vol. 29, pp. 69–94, 2003.

8. Mascagni, M. and A. Srinivasan, "Algorithm 806: SPRNG: a scalable library for pseudo-random number generation", *ACM Transactions on Mathematical Software (TOMS)*, 26(3), pp. 436–461, 2000.

9. Aluru, S., "Lagged Fibonacci Random Number Generators for Distributed Memory Parallel Computers", *Journal of Parallel and Distributed Computing*, 45(1), pp. 1–12, 1997.

10. Mascagni, M. and A. Srinivasan, "Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators", *Parallel Computing*, vol. 30, pp. 899–916, 2004.

11. Marsaglia, G., "A current view of random number generators", *Computing Science and Statistics: Proceedings of the XVIth Symposium on the Interface*, pp. 3–10, 1985.

12. Box, G.E.P. and M. E. Muller, A note on the generation of random normal deviates", *The Annals of Mathematical Statistics*, 29(2), pp. 610–611, 1958.

13. Donner, A. and B. Rosner, "On inferences concerning a common correlation coefficient", *Applied Statistics*, 29(1), pp. 69–76, 1980.

14. Rao, P.V., *Statistical Research Methods in the Life Sciences*, Brooks/Cole, 1998.

15. L'Ecuyer, P. and R. Simard, "TestU01: A C library for empirical testing of random number generators", *ACM Trans. Math. Softw.*, 33, 4, August 2007.

16. Brown, R.G., D. Eddelbuettel, and D. Bauer, *Dieharder Test Package, v 3.29.0*, Duke University, online: http://www.phy.duke.edu/~rgb/General/dieharder.php, retrieved on Nov. 2010.

US008868630B1

(12) **United States Patent**
Boppana et al.

(10) **Patent No.: US 8,868,630 B1**
(45) **Date of Patent: Oct. 21, 2014**

(54) **VERIFICATION OF PSEUDORANDOM NUMBER STREAMS**

(75) Inventors: **Rajendra V. Boppana**, San Antonio, TX (US); **Ram C. Tripathi**, San Antonio, TX (US)

(73) Assignee: **Board of Regents of the University of Texas System**, Austin, TX (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 324 days.

(21) Appl. No.: **13/423,927**

(22) Filed: **Mar. 19, 2012**

Related U.S. Application Data

(60) Provisional application No. 61/454,259, filed on Mar. 18, 2011.

(51) **Int. Cl.**
G06F 7/58 (2006.01)

(52) **U.S. Cl.**
USPC **708/250**

(58) **Field of Classification Search**
None
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

8,589,460	B2 *	11/2013	Dong	708/254
2003/0065691	A1 *	4/2003	Schmidt	708/250
2003/0236803	A1 *	12/2003	Williams	708/252
2007/0162806	A1 *	7/2007	Matsumoto et al.	714/728
2008/0288566	A1 *	11/2008	Umeno et al.	708/250
2010/0235418	A1 *	9/2010	Dong	708/254

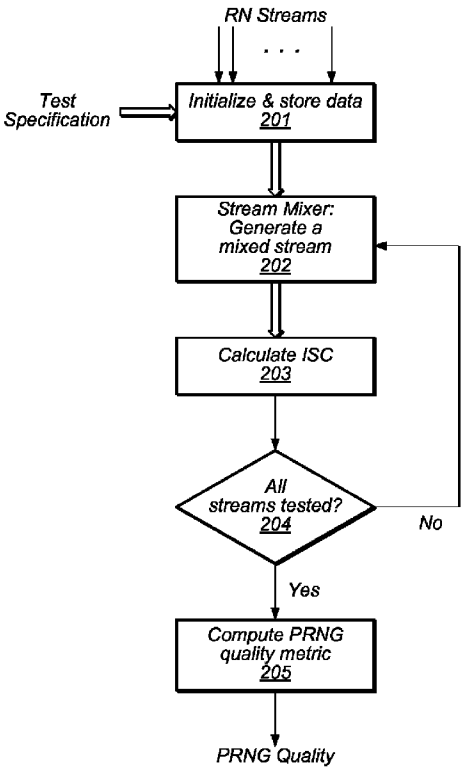
* cited by examiner

Primary Examiner — David H Malzahn
(74) *Attorney, Agent, or Firm* — Meyertons, Hood, Kivlin, Kowert & Goetzel, P.C.; Eric B. Meyertons

(57) **ABSTRACT**

A method of assessing parallel random number streams includes mixing two or more parallel random number streams. Mixing the parallel random number streams may include pairing at least one of the random number streams with other random number streams. For each mixed random number stream, an inter-stream correlation value may be computed based on a correlation among the random number steams used. A quality metric for the parallel random number streams may be determined from inter-stream correlation values for the two or more mixed streams created from the parallel random number streams. A quality metric for a single random number stream may be computed by segmenting the single random number stream into multiple substreams and applying the methods of mixing streams and computing quality metric in the case of parallel streams.

21 Claims, 3 Drawing Sheets



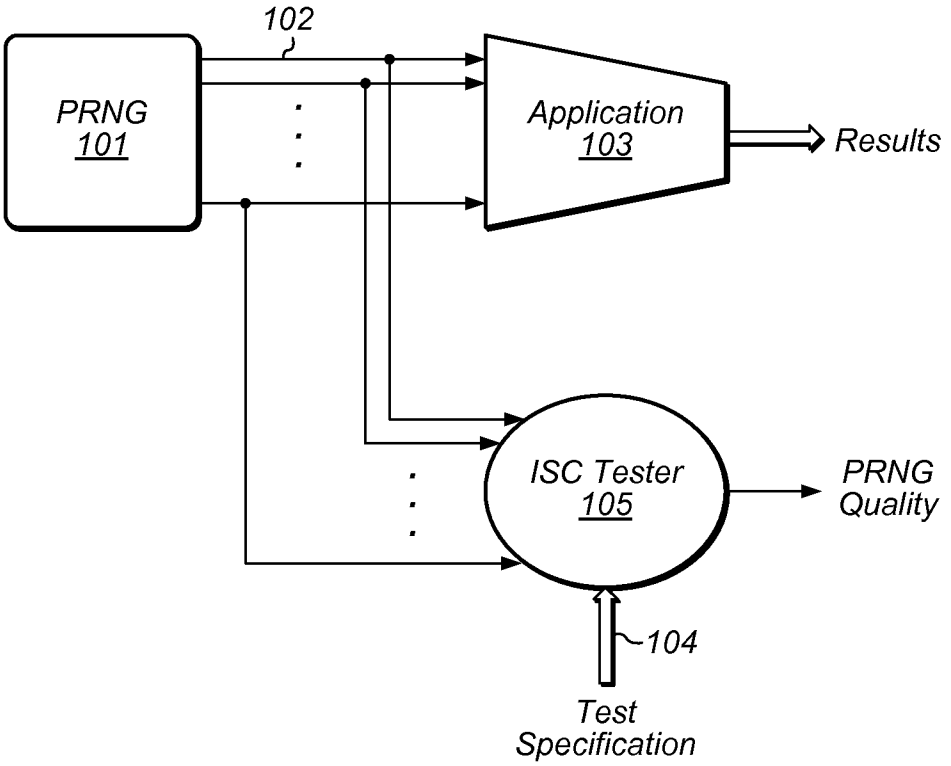

102
PRNG
101
Application
103
Results
ISC Tester
105
PRNG
Quality
104
Test
Specification

FIG. 1

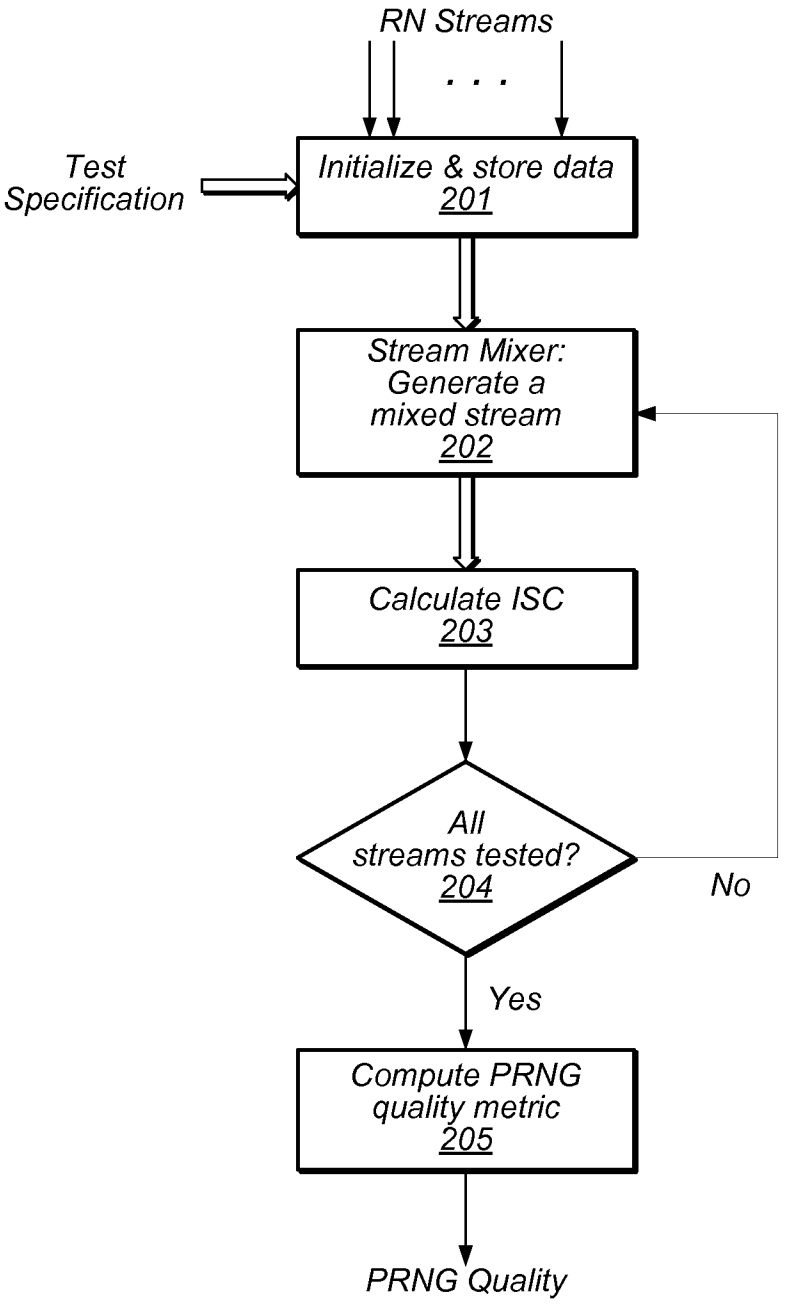

RN Streams
. . .
Test Specification
Initialize & store data 201
Stream Mixer: Generate a mixed stream 202
Calculate ISC 203
All streams tested? 204
No
Yes
Compute PRNG quality metric 205
PRNG Quality

FIG. 2

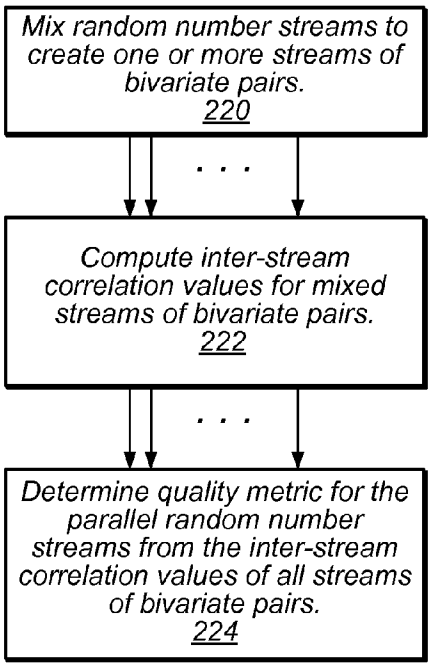

Mix random number streams to create one or more streams of bivariate pairs.
220
Compute inter-stream correlation values for mixed streams of bivariate pairs.
222
Determine quality metric for the parallel random number streams from the inter-stream correlation values of all streams of bivariate pairs.
224

FIG. 3

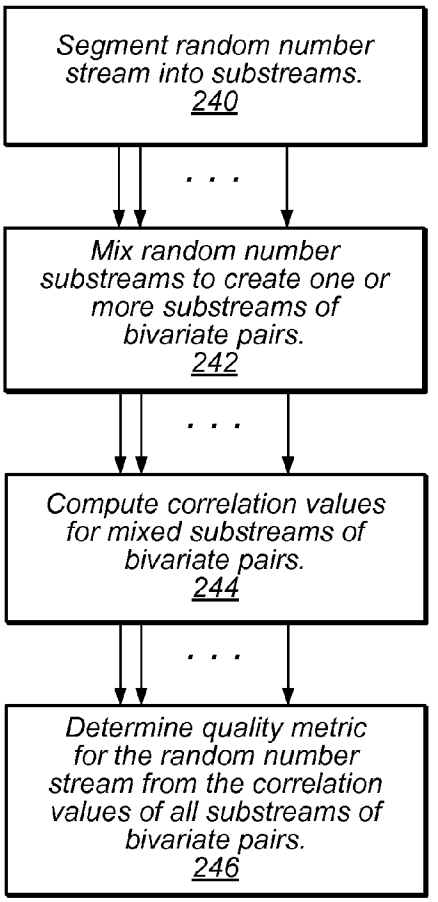

Segment random number stream into substreams.
240
Mix random number substreams to create one or more substreams of bivariate pairs.
242
Compute correlation values for mixed substreams of bivariate pairs.
244
Determine quality metric for the random number stream from the correlation values of all substreams of bivariate pairs.
246

FIG. 4

VERIFICATION OF PSEUDORANDOM NUMBER STREAMS

PRIORITY CLAIM

This application claims priority to U.S. Provisional Application No. 61/454,259 entitled "Verification of Pseudorandom Number Streams" to Boppana et al. filed Mar. 18, 2011, which is incorporated herein by reference in its entirety.

BACKGROUND

1. Field

This disclosure relates to the field of computation. More particularly, this disclosure relates to methods for assessing pseudorandom number streams.

2. Description of the Related Art

Random number generators, which generate streams of seemingly random numbers, are used in many computing applications. An application may use a single stream of random numbers or multiple streams of random numbers simultaneously. A sequential random number generator is designed to generate a single stream of random numbers, the starting point of which may be changed with the initial (seed) value. A parallel random number generator (PRNG) is designed to generate multiple, independent streams of random numbers simultaneously with a simple change in a parameter used to initialize the random number streams.

It is often useful to test a random number generator to assess the quality of the random number stream. Some single-stream statistical test batteries provide pass/fail indication for each test in the battery, since it may not be meaningful to combine the statistical computations from multiple tests to provide an overall quality metric for the RNG (random number generator) tested. Therefore, it is common to use the test results as a multi-bit vector data, with each bit representing the pass/fail status for a test. The statistical test batteries do not provide a single quantitative metric to compare the two generators. This could be a limitation if two RNGs that need to be compared fail different tests.

Single-stream tests may be ineffective for testing the correlations of random numbers among a large number (e.g., thousands to billions) of parallel random number streams since the a typical single-stream test method may operate on blocks of a few thousands of numbers at a time. Typical existing test methods may be considered off-line methods in the sense that the tests are fed with data generated by the random number generator that is being evaluated specifically for test purposes.

Parallel random number streams may be generated by a parameterized family of pseudorandom number generators, by a collection of true random number generators that generate random numbers based on environmental signals such as noise levels and temperature, computing and communication delays, events induced by computer users or other sources, or any combination of the pseudo- and true random number generators. The quality of the random numbers used may be crucial for quick and accurate results from computer-based simulations and for robust security protocols and security keys used in security protocols.

Some methods to test and assess the independence of parallel random number streams are typically based on sequential test methods that are designed to test intra-stream correlations of a single random number stream. One practice for statistical testing of PRNG quality is to generate parallel streams, interleave them to form a single stream, and apply single-stream tests to the interleaved stream. If the interleaved stream passes most or all of the single-stream tests, then the PRNG may be deemed to be of good quality and is accepted for use in applications.

SUMMARY

In an embodiment, a method of assessing parallel random number streams includes mixing two or more parallel random number streams. Mixing the parallel random number streams may include pairing one of the random number streams with one or more of the other random number streams. For each pairing of the parallel random number streams, an inter-stream correlation value may be computed based on a correlation between the two random number streams in the pair. A quality metric for the parallel random number streams is determined from inter-stream correlation values for the pairs of the parallel random number streams.

In an embodiment, a method of assessing quality of a random number stream includes segmenting the random number stream into two or more random number substreams. The random number substreams may be mixed. Mixing the random number substreams may include pairing one of the substreams with one or more of the other substreams. For each pair of the random number substreams, a correlation value may be computed based on a correlation between the random number substreams in the pair. A quality metric for the random number stream is determined from correlation values for the pairs of the random number substreams.

In various embodiments, methods, systems and apparatus are used to test a large number of parallel random number streams and to quantify interstream correlations among them so that their randomness can be assessed. Correlations may be tested among a large number (hundreds to billions) of streams and the computed correlation coefficients may be combined so that the user of a parallel random number generator can assess a priori or dynamically (during the consumption of the random numbers) the quality of random numbers used for his/her application. In some embodiments, an online test is performed of the quality of RN streams as the random numbers are generated by the PRNG for an actual application use.

In some embodiments, an interstream correlation (ISC) test evaluates a large number of parallel RN streams simultaneously and provides a quality metric. The ISC test may divide the total streams to be evaluated into subsets of streams, with at least two streams in each subset, and compute a correlation coefficient for each subset. These correlation coefficients may be combined using a theoretically sound test method such as the Donner and Rosner test (DR test) or Kolmogorov-Smirnov test (KS test), and a test statistic may be obtained. If the test statistic is higher than a suitably determined critical value, the claim of independent RN streams is rejected. A lack of rejection indicates that the RN streams are likely to be independent.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is an exemplary block diagram illustrating a parallel pseudorandom number generator test metric computation according to one embodiment.

FIG. **2** is an exemplary flow chart of the logic implemented by an inter-stream correlation test according to one embodiment.

FIG. **3** is a flow diagram illustrating one embodiment of assessing parallel random number streams.

FIG. **4** is a flow diagram illustrating one embodiment of assessing a random number stream.

While the invention is described herein by way of example for several embodiments and illustrative drawings, those skilled in the art will recognize that the invention is not limited to the embodiments or drawings described. It should be understood, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims. The headings used herein are for organizational purposes only and are not meant to be used to limit the scope of the description or the claims. As used throughout this application, the word "may" is used in a permissive sense (i.e., meaning having the potential to), rather than the mandatory sense (i.e., meaning must). Similarly, the words "include", "including", and "includes" mean including, but not limited to.

DETAILED DESCRIPTION OF EMBODIMENTS

The following abbreviations and acronyms are used herein.
RN: Random number;
RNG: pseudorandom number generator;
PRNG: parallel pseudorandom number generator;
ISC: interstream correlation;
CPU: central processing unit or processor;
GPU: graphic processing unit or graphics processor used for general purpose array computing;
MC: Monte Carlo simulations.

As used herein, "pairing", in the context of number streams, includes mixing or combining one stream with one or more other streams, or considering or assessing one stream in relation to one or more other streams (for example computing a correlation between two streams). As examples, a pairing may include: (a) pairing a selected stream with another stream, (b) pairing a selected stream with an interleaved stream of two or more other streams, and (c) interleaving a selected stream and one or more other streams.

As used herein, "random number" includes, but is not limited to, a true random number, a pseudorandom number, or a number generated from a combination of true random and pseudorandom number methods. As used herein, a "random number generator" includes, but is not limited to, a pseudorandom number generator.

FIG. **1** is an exemplary block diagram illustrating the PRNG test metric computation. In FIG. **1** PRNG **101** is the parallel random number generator that needs to be tested for the independence of its streams **102**. Each line may provide a single stream of RNs spaced in time. These RNs may be fed to the application **103** as part of the application's input data. The application **103** may be executed normally and the output of the application may be obtained.

In some embodiments, a parallel random number generator may be part of the application. In such cases, PRNG **101** and Application **103** may be described by a single block feeding ISC Tester **105**.

ISC Tester **105** may be fed with RN streams **102** and a test specification. The test specification may specify the interleaving method for mixing the streams and the statistical method that is used for computation of a quality metric.

FIG. **2** is an exemplary flow chart of logic implemented by an inter-stream correlation test according to one embodiment. ISC Tester **105** may be fed with parallel RN streams and test specification criteria. The initialization and storage unit **201** may ensure that these RNs are available for repeated use during the test method. Based on the specified interleaving, stream mixer program **202** may select a stream and mix it with the remaining streams (if the specification is biased interleaving) or with a subset of the other streams (if the specification is group, shuffled or pairwise interleaving) to create a single stream with RNs from the selected stream occupying the odd numbered positions and the RNs from the other streams occupying the even numbered positions. Stream mixer program may skip the user-specified number of initial RNs from one or more of the streams prior to mixing them. The RNs in the odd numbered positions (positions 1, 3, 5, . . .) from the resulting mixed stream may be considered as x_i's and the RNs in the even numbered positions as y_i's. Therefore, the resulting mixed stream may be considered as a sequential stream of (x_i; y_i) bivariate pairs. This mixed stream may be fed to correlation coefficient computing program **203**. Correlation coefficient computing program **203** may calculate inter-stream correlations of the two streams provided to it by the stream mixer **202**. The computed correlation coefficient is stored. A tester **204** checks if all the desired combinations of interstream correlations are computed. If there are one or more combinations remain, the stream mixer provides the next stream pair to the correlation coefficient computing program **203**. If all desired combinations of stream pairs are examined, then PRNG quality metric **205** is computed. The PRNG quality metric may be computed using, in various embodiments, an aggregation method, a goodness-of-fit method, percentile method or mean absolute deviation method. In some embodiments, the method for computing the PRNG quality metric is based on user specification. In some embodiments, the final output (which may be a p-value in statistics) may be a significance level above which the claim of independence of the parallel streams cannot be rejected. In certain embodiments, the user may specify a significance level, and the quality metric is used to determine if the PRNG meets the user-specified significance level.

FIG. **3** is a flow diagram illustrating one embodiment of assessing parallel random number streams. In some embodiments, the parallel random number streams are generated by a random number generation system for purposes of evaluating the quality of the random number generation system. This may be described as a priori or offline test. In other embodiments, the quality of parallel random number streams generated on demand by an application is assessed continually while the application is running. This may be described as dynamic, on-the-fly, or online test.

At **220**, parallel random number streams may be mixed in one or more ways to create one or more streams of bivariate pairs. Mixing the parallel random number streams may include pairing the random number streams with one another. In some embodiments, a selection of a mixing method to be used for mixing the random number streams is received from a user.

At **222**, an inter-stream correlation value may be computed for each mixed stream of bivariate pairs based on a correlation among the random number streams used to create the mixed stream. The correlation values may be, for example, a correlation coefficient computed by taking several (two or more) bivariate pairs from the mixed stream. The number of bivariate pairs used in the correlation value computation may be specified by the user.

At **224**, a quality metric for the parallel random number streams may be determined from inter-stream correlation values for the mixed streams. The quality metric may serve as a figure of merit for the parallel random number streams. The quality metric may provide a measure of the independence of the parallel number streams from one another. In some embodiments, a selection of a testing method to be used for computing a quality metric for the random number streams is

received from a user. The quality metric may be measured against a significance level specified by a user.

FIG. **4** is a flow diagram illustrating one embodiment of assessing a random number stream. In some embodiments, the random number stream is generated by a random number generation system for purposes of testing the random number generation system. In other embodiments, the quality of the random number stream is assessed during consumption of the random numbers by an application (online test).

At **240**, a random number stream is segmented into random number substreams. In one embodiment, the random number stream is segmented using a leap-frog method. In another embodiment, the random number stream is segmented using a cycle-division method.

At **242**, random number substreams may be mixed to form substreams of bivariate pairs. Mixing the random number substreams may include pairing the random number substreams with one another. In some embodiments, a selection of a mixing method to be used for mixing the random number substreams is received from a user.

At **244**, an inter-stream correlation value may be computed for each mixed substream of bivariate pairs based on a correlation between the substreams used to create the mixed substreams. The number of bivariate pairs (at least two) used in the correlation value computation may be specified by the user.

At **246**, a quality metric for the random number stream may be determined from inter-stream correlation values for the mixed substreams. The quality metric may serve as a figure of merit for the random number stream. The quality metric may serve as a figure of merit for the parallel random number streams. The quality metric may provide a measure of the independence of the parallel number streams from one another. In some embodiments, a selection of a testing method to be used for computing a quality metric for the random number streams is received from a user. The quality metric may be measured against a significance level specified by a user.

In some embodiments, inter-stream correlations are quantified among multiple parallel random number (RN) streams as a numerical factor, and a figure of merit is assigned for a PRNG. In one embodiment, a system includes three main components: stream mixer **202**, correlation coefficient calculator **203**, and PRNG quality metric calculator **205**.

Let us consider k, where $k \geq 2$, RN streams $S_1, S_2, \ldots, S_k$ for which we need to check if there is a significant inter-stream correlation (ISC) among them. To compute the correlation, we construct a bivariate sample (X,Y) given by (x_i, y_i), i=1, 2, . . . , n. (It is common to use capitalized letters for random variables and lower case letters with appropriate subscripts for the observed samples corresponding to the random variables.) A straight-forward bivariate sampling takes two RN streams at a time; but this results in

$$\frac{k(k-1)}{2}$$

possible bivariate samples, in which each bivariate sample shares one of the streams with 2(k−2) other bivariate samples, or

$$\frac{k}{2}$$

bivariate samples, in which no streams are shared among the bivariate samples. If k=10,000, then the number of bivariate samples we need to analyze to capture all possible correlations will be nearly 50 million. To reduce the computational complexity, we construct k or fewer bivariate samples in which each RN stream is checked for correlation with one or more of the other RN streams. This is explained in the following steps.

Step 1. Mix the RN Streams in one of the Following Ways

Biased Interleaving:

Use n numbers from S_1 as the n observations on the X variate, and interleave the remaining k−1 streams to provide n observations on the Y variate.

(An alternative approach is to use coarse interleaving of the k−1 streams. Let n be a large multiple of (k−1). Take the first n RNs from S_1 to form the n observations on X. Take first

$$\frac{n}{k-1}$$

RNs from S_2, the second

$$\frac{n}{k-1}$$

RNs from S_3 and so on to form n values on Y. Extensive testing showed that both methods of interleaving give statistically similar results. The first approach is oblivious to the total number of RNs to be generated by each stream, which may simplify the generation and storage of the random numbers.)

This gives (x_i, y_i), i=1, 2, . . . , n, with S_1 as the selected stream. This can be repeated with S_i, i=2, . . . , k, as the selected stream providing X values and

$$\frac{n}{k-1}$$

RNs from each of the other k−1 streams providing Y values. In this method, each (X,Y) bivariate sample shares (overlaps)

$$\frac{n(k-2)}{k-1}$$

of its Y values with each of the other bivariate samples.

Group Interleaving:

This method of mixing the RN streams extends the concept of biased interleaving to form bivariate samples with no overlap, which may be desirable for statistical test methods. In this method, the given k RN streams are grouped into groups of h streams each, where $2 \leq h \leq k$. There will be g groups, where

$$g = \left\lfloor \frac{k}{h} \right\rfloor.$$

Therefore, group interleaving uses gh streams for correlation calculations. (If h does not divide k evenly without any remainder, then $gh < k < gh+h$.) Using the streams in each group, a bivariate sample is formed as follows. One of the streams from the group is selected to provide n observations

of the X variate. The remaining h−1 streams are interleaved to provide n values for the Y variate; each of these streams provides up to

$$\left\lceil \frac{n}{h-1} \right\rceil$$

random numbers. (As indicated earlier, fine or coarse interleaving may be used to interleave the h−1 streams.) This gives g bivariate samples each with n observations. There is no sharing of random numbers among the bivariate samples.

Shuffled Interleaving:

This method is a variation of group interleaving, obtained by interleaving all streams of the group evenly and taking the values in the odd-numbered positions forming the X variate and values in the even numbered positions forming the Y variate. Shuffled interleaving also produces g different (X,Y) stream pairs with no overlapping. For the special case of h=k, there is only one group resulting in only one (X,Y) bivariate sample; this special case is the state of the art for statistical testing of interstream correlations.

Pairwise Interleaving:

A special case of group interleaving (and shuffled interleaving) is the pairwise interleaving, which is obtained by choosing h=2; each group is a pair of streams. Therefore, pairwise interleaving uses n RNs from stream S_1 as the n observations of the X variate and n RNs from S_2 as the observations of the Y variate from the first group. This again gives (x_i,y_i), i=1, 2, . . . , n. This can be repeated to obtain up to

$$\left\lfloor \frac{k}{2} \right\rfloor - 1$$

additional pairs with stream

$$S_i,\ i = 2, 4, \ldots, 2\left\lfloor \frac{k}{2} \right\rfloor - 1,$$

providing X values and stream S_{i+1} providing Y values.

Step 2. Calculate Correlation of X,Y Streams

Consider a pair of values (x_i,y_i), i=1, 2, . . . , n, taken one each from the two streams. If the RNs are integers in the range [0, m−1], then they are converted to reals in the range (0,1] using the conversion

$$1 - \frac{RN}{m},$$

where RN is an integer random number. If the RNs are from uniform [0, 1), then they are converted to (0, 1] range using the conversion 1−RN. If the RNs are from uniform (0,1) distribution, no additional preprocessing is needed. Let the resulting random variates be denoted ux_i and uy_i. The Box-Muller transform given by the following equations is applied to convert RNs to normal random variates, zx_i and zy_i. (All logarithms are to the base e.)

$$r^2 = -2\log(ux_i) \quad (1)$$

$$\theta = 2\pi uy_i \quad (2)$$

$$zx_i = r\cos\theta \quad (3)$$

$$zy_i = r\sin\theta \quad (4)$$

The correlation coefficient of the bivariate normal pairs (zx_i, zy_i), i=1, 2, . . . , n, is computed.

The Box-Muller transform is not symmetric in the sense that switching (X,Y) ordering yields a different correlation coefficient value. In particular, Box-Muller transform is sensitive to the RN streams used for Y variates and amplifies the correlations among the RN streams used for Y variates to calculate different θ's. If the selected stream is used to draw observations for X and the interleaved stream is used to draw observations for Y with biased interleaving, then Box-Muller transform correctly amplifies the correlation among the different versions of the interleaved streams used for Y. Any pair of interleaved streams formed by biased-interleaving share

$$\frac{n(k-2)}{k-1}$$

values, and the quality metric computed in the next step is dominated by the correlation among the interleaved streams. To avoid this, since the purpose of ISC test is to find correlations among different individual streams, the interleaved stream should be used for the observations of X and the selected stream for the observations of Y when biased interleaving is used to mix RN streams. For group, shuffled, and pairwise interleaving the order of the streams is not an issue since all streams used for X and Y variates are independent.

Correlation coefficients from several pairs of streams generated using the biased interleaving are obtained. Let these coefficients be denoted $r_1, r_2, \ldots, r_k$. Each r_i gives the interstream correlations from a selected stream to the rest of the streams.

If group or shuffled interleaving is used, $r_1, r_2, \ldots, r_g$, where

$$g = \left\lfloor \frac{k}{h} \right\rfloor$$

and h is the group size, are the interstream correlations with r_i representing the correlation coefficient between streams S_{ih}, $S_{ih+1}, \ldots, S_{ih+h-1}$. For the special case of pairwise-interleaving,

$$r_1, r_2, \ldots, r_{\lfloor \frac{k}{2} \rfloor}$$

are the interstream correlations, where r_i represents the correlation coefficient between streams S_{2i} and S_{2i+1}.

(Alternatively, the polar transform may be used to convert (x_i, y_i) pairs to normal random variate pairs. First, x_i and y_i are converted to reals in the range (−1, 1). If the RNs are integers, they can be converted into reals in the range (−1,1). If the RNs are from uniform (0,1) distribution, then the numbers are extended to (−1,1) range. Let these be denoted ux_i and uy_i. If $ux_i^2+uy_i^2 \geq 1$, the (x_i, y_i) pair is rejected and another pair from the streams is chosen and tested for suitability. This is repeated until a suitable pair is found. The processed values ux_i and uy_i of (x_i, y_i) pair that is found suitable are used to compute the corresponding normal random variates pair using the following equations.

$$s = \sqrt{ux_i^2+uy_i^2} \quad (5)$$

$$zx_i = ux_i\sqrt{\frac{-2\log(s)}{s}} \tag{6}$$

$$zy_i = uy_i\sqrt{\frac{-2\log(s)}{s}} \tag{7}$$

Since it rejects RN pairs that are simultaneously too large or too small, ISC testing based on the polar transform may result in the underestimation of the actual inter-stream correlations. Therefore, polar transform is not recommended for ISC testing and the computation of PRNG quality metric. However, the polar transform may be used to reduce the correlations between a given pair of RN streams by removing RN pairs that result in s≥1.)

Step 3. Compute the Overall Interstream Correlation Metric

The sequence of r's obtained in the previous step denote

$$k\left(\text{or } \frac{k}{2} \text{ if pairwise-interleaving is used}\right)$$

if pairwise-interleaving is used) estimates of the actual correlation coefficient ρ among the streams converted using the Box-Muller transform. The RNG quality metric may be obtained by converting the r's to normal variates using Fisher's z-transformation and using one of the following correlation-coefficient combining methods described below.

3.1. Aggregation Method

Let r_i, i=1, . . . , k, be a correlation coefficient based on n_i bivariate pairs. In the present disclosure, $n_1=n_2=\ldots=n_k=n$ Let N=kn.

Define

$$Z_i = \frac{1}{2}\log\left(\frac{1+r_i}{1-r_i}\right). \tag{8}$$

Let

$$\overline{Z}_W = \frac{\sum_{i=1}^{k}(n_i-3)Z_i}{\sum_{i=1}^{k}(n_i-3)}. \tag{9}$$

An estimate of the common correlation ρ is

$$r_F = \tanh(\overline{Z}_W) = \frac{e^{2Z_W}-1}{e^{2Z_W}+1}. \tag{10}$$

An alternative expression for r_F in terms of the r_i is

$$r_F = \frac{\prod_{i=1}^{k}(1+r_i)^{c_i} - \prod_{i=1}^{k}(1-r_i)^{c_i}}{\prod_{i=1}^{k}(1+r_i)^{c_i} + \prod_{i=1}^{k}(1-r_i)^{c_i}}, \tag{11}$$

with

$$c_i = \frac{n_i-3}{N-3k},\ i=1,2,\ldots,k.$$

For the case of equal sample size,

$$c_i = \frac{1}{k},$$

and the following bias-corrected transform

$$\overline{Z}_H = \overline{Z}_W - \frac{r_F}{2n-\frac{9}{2}} \tag{12}$$

may be used to estimate ρ by

$$r_H = \tanh(\overline{Z}_H) = \frac{e^{2Z_H}-1}{e^{2Z_H}+1}. \tag{13}$$

We can use the statistic $t_H=\overline{Z}_H\sqrt{N-3k}$ to test the hypothesis: H_0: ρ=0. Under the null hypothesis H_0, t_H has an asymptotic standard normal distribution. This gives a significance level above which the null hypothesis cannot be rejected. This significance level can be used to determine the quality of the PRNG.

3.2. Percentile Method

To compute the quality metric, a significance level α is chosen and $r_h=r_{1-\alpha/2}$ and $r_l=r_{\alpha/2}$ quantile values are taken from the sorted sequence of r's. The Fisher's z-transformation given by the following equation is applied to both quantiles to obtain Z_h and Z_l.

$$Z_i = \frac{0.5\log\left[\frac{1+r_i}{1-r_i}\right]}{\left(1/\sqrt{n-3}\right)},\ i \text{ is } h \text{ or } l. \tag{14}$$

The quality of the PRNG is given by the significance level at which Z_h<2.33 and Z_l>−2.33, where 2.33 is the 99th percentile (0.99 quantile) for the standard normal random variable.

Alternatively, the significance level for the selection of r quantiles may be fixed and the significance level at which Z_h and Z_l satisfy the corresponding Z-quantiles may be taken as a PRNG quality metric.

3.3. Goodness-of-Fit Method

Kolmogorov-Smirnov (KS) test is a goodness-of-fit test method that may be used instead of the aggregate method to determine the correlation among the RN streams in consid-

eration. The method is applied as follows. Each r_i, $1 \le i \le k$, is converted to standard normal variates using the Fisher's z-transform described above and sorted in ascending order to obtain z_i, i=1, . . . , k. For each z_i, the corresponding cumulative probability, f_i, is computed. If r_i's are normally distributed, then the cumulative probabilities will be uniformly spaced in the interval [0, 1]. The KS test statistic, D, the maximum deviation of f_i, i=1, . . . , k from a true uniform distribution, is computed as follows.

$$D = \underset{1 \le i \le k}{\text{MAX}}\left\{f_i - \frac{i-1}{k}, \frac{i}{k} - f_i\right\} \tag{15}$$

If D is below the critical value for a given significance level, then the hypothesis that r_i's are normally distributed cannot be rejected at that significance level. The critical values for KS test precomputed for various significance levels are given in most standard books on statistics.

3.4. Mean Absolute Deviation Method

Let r_q be the q-quantile value in the sorted sequence of r_i's. Also, let $r_{q1}, r_{q2}, \ldots, r_{qm}$ be m r_i's selected at quantiles q1, . . . , qm, from this sequence. Using Fisher's z-transform above, the corresponding standard normal values $z_{q1}, z_{q2}, \ldots, z_{qm}$ are computed. From these, the corresponding cumulative probabilities for the z values are computed; let they be $f_{q1}, f_{q2}, \ldots, f_{qm}$. The mean absolute deviation is computed using the following equation.

$$E = \sum_{1 \le i \le m} \frac{|f_{qi} - qi|}{m} \tag{16}$$

There is no critical value against which E can be compared. The lower the value E, the better. Though KS test requires more computations, it is a more thorough test and should be preferred to the mean absolute deviation test. On the other hand, for on-the-fly testing of very long RN streams, the mean absolute deviation method may be more practical to implement.

Application of ISC Test to a Single Stream

In some embodiments, an ISC test may be used to determine intra-stream correlations as follows. A single stream may be segmented into k substreams by leap-frog or cycle-division methods, or by any other method. In the leap-frog method, substream i, $1 \le i \le k$ consists of RNs in positions i, k+i, 2k+i . . . of the stream. In the cycle-division method, k pairwise disjoint subsets, each containing n consecutive RNs of the original single RN stream are picked. An ISC test can be applied on the substreams to obtain the quality metric as in the case of parallel RN streams. In this case, however, the ISC test gives the quality metric based on the intrastream correlations.

In some embodiments, an interstream correlation (ISC) test evaluates a large number of parallel RN streams simultaneously and provides a quality metric. The ISC test may divide the total streams to be evaluated into subsets of streams, and compute a correlation coefficient for each subset. These correlation coefficients may be combined using a theoretically sound test method such as the Donner and Rosner test (DR test) or Kolmogorov-Smirnov test (KS test), and a test statistic may be obtained. If the test statistic is higher than a suitably determined critical value, the claim of independent RN streams is rejected. A lack of rejection indicates that the RN streams are likely to be independent.

In some embodiments, an interstream correlation test evaluates correlations among a large number of RN streams. Using a test method such as the Donner and Rosner test or the Kolmogorov-Smimov test, the interstream correlation test may provide an overall PRNG quality metric. In some embodiments, results of an interstream correlation test are used in conjunction with other single-stream test batteries and application-based tests. The test may be used to evaluate interstream correlations among billions of RN streams.

In an embodiment, an interstream correlation test evaluates the correlations among a large number of subsets. The subsets may be interleaved using shuffled or biased interleaving method. As one example, three RN streams A, B and C may be considered with RNs $a_1, a_2, a_3, \ldots, b_1, b_2, b_3, \ldots$, and $c_1, c_2, c_3, \ldots$, respectively. In shuffled interleaving (also called perfect shuffle interleaving), a new stream $a_1, b_1, c_1, a_2, b_2, c_2, a_3, \ldots$ is created. In biased interleaving, $a_1, b_1, a_2, c_1, a_3, b_2, a_4, \ldots$ is created. The RNs in the odd numbered positions form the X variates and the RNs in the even numbered positions form the Y variates to create bivariate pairs. These may be transformed into bivariate normal pairs using Box-Muller transform. Correlation coefficient, r, for the bivariate normal pairs is computed. This may be repeated several times to obtain multiple r's. Collectively, these r's are the samples that can be used to estimate ρ, the true common correlation coefficient among the parallel RN streams generated by the PRNG being evaluated.

The r's may be combined using a theoretically sound test method such as Donner and Rosner test (DR-test) or Kolmogorov-Smirnov test (KS-test). Based on the test data, a test statistic may be obtained. For purposes of this example, the statistic for DR-test is denoted as t_H and the statistic for KS-test as D_{max}. For each test, there may be a critical value that is computed based on the desired significance level and the number of r's used. For example, for DR-test at a significance level of 0.05, the critical value may be 1.96 provided the number of bivariate pairs used to calculate each r is large and the number of r's is more than 2. If test statistic is above the critical value, then the RN streams generated by the PRNG are likely to have significant interstream correlations.

In this example, the DR-test combines the r's and gives the test statistic t_H, which is a standard normal variate. This can be used to test the null hypothesis H_0:ρ=0. Large absolute values of t_H will lead to the rejection of the null hypothesis and the acceptance of the alternative hypothesis H_1:ρ≠0. For the significance level α=0.05, absolute values of t_H above 1.96 lead to the rejection of the claim that parallel RN streams are independent. The probability that the rejection is erroneous is α=0.05. One could use different significance levels: for α=0.02, the absolute values of t_H above 2.33 will lead to rejection of the claim of independence of RN streams with only 0.02 probability of being wrong.

The distribution of r's may be approximately normal. These r's can be converted into standard normal variates using sample variance of r's, testing for ρ=0. The KS test may be applied on the distribution of r's. In this case, the KS-test statistic, D_{max}, computed using the r's is to be less than the critical value, $D_{\alpha,n}$, for significance level α and n, the number of r's used. For KS-test, at a significance level of 0.01, the critical value may be 0.0274 when the number of r's used is 1500.

In some embodiments, r's may be combined using other computationally more complex tests such as Anderson-Darling or Shapiro-Wilk tests.

In some embodiments, r's may be combined using computationally simpler tests such as the percentile method and mean absolute deviation method. The simpler methods may

be preferred for online tests to reduce the use of computing resources used for quality metric computations, whereas the more complex methods may be preferred for offline tests.

Systems and methods described herein may be used in a variety of applications. Examples of applications for systems and methods as described herein include (a) simulation-based solutions to large scientific and engineering problems, (b) parameterized Monte Carlo simulations of scientific, engineering, and finance problems, (c) distributed computing, and (d) protocols and keys used for information assurance and security.

Systems and methods described herein, such as the ISC tester described above relative to FIG. **1**, may be implemented in hardware including field programmable gate arrays (FPGAs) and application specific integrated circuit (ASIC) chips, or a suitable combination of hardware and software and which can be one or more software systems on a general purpose processor (CPU) or graphics processing unit (GPU).

Computer systems may, in various embodiments, include components such as a CPU with an associated memory medium such as Compact Disc Read-Only Memory (CD-ROM). The memory medium may store program instructions for computer programs. The program instructions may be executable by the CPU. Computer systems may further include a display device such as monitor, an alphanumeric input device such as keyboard, a directional input device such as mouse, a voice recognition system to dictate text and issue commands for processing, and a touch screen that may serve as a keyboard or mouse. Computer systems may be operable to execute the computer programs to implement computer-implemented systems and methods. A computer system may allow access to users by way of any browser or operating system.

Embodiments of a subset or all (and portions or all) of the above may be implemented by program instructions stored in a memory medium or carrier medium and executed by a processor. A memory medium may include any of various types of memory devices or storage devices. The term "memory medium" is intended to include an installation medium, e.g., a Compact Disc Read Only Memory (CD-ROM), floppy disks, or tape device; a computer system memory or random access memory such as Dynamic Random Access Memory (DRAM), Double Data Rate Random Access Memory (DDR RAM), Static Random Access Memory (SRAM), Extended Data Out Random Access Memory (EDO RAM), Rambus Random Access Memory (RAM), etc.; or a non-volatile memory such as a magnetic media, e.g., a hard drive (which may be a disk or solid state), or optical storage. The memory medium may comprise other types of memory as well, or combinations thereof. In addition, the memory medium may be located in a first computer in which the programs are executed, or may be located in a second different computer that connects to the first computer over a network, such as the Internet. In the latter instance, the second computer may provide program instructions to the first computer for execution. The term "memory medium" may include two or more memory mediums that may reside in different locations, e.g., in different computers that are connected over a network. In some embodiments, a computer system at a respective participant location may include a memory medium(s) on which one or more computer programs or software components according to one embodiment may be stored. For example, the memory medium may store one or more programs that are executable to perform the methods described herein. The memory medium may also store operating system software, as well as other software for operation of the computer system.

The memory medium may store a software program or programs operable to implement embodiments as described herein. The software program(s) may be implemented in various ways, including, but not limited to, procedure-based techniques, component-based techniques, and/or object-oriented techniques, among others. For example, the software programs may be implemented using ActiveX controls, C++ objects, as a library or standalone programs in a programming language such as C, C++, Java or in a scripting language such as Bash, Perl, Python, or AWK, JavaBeans, Microsoft Foundation Classes (MFC), browser-based applications (e.g., Java applets), traditional programs, or other technologies or methodologies, as desired. A CPU executing code and data from the memory medium may include a means for creating and executing the software program or programs according to the embodiments described herein.

The ISC Tester may be embedded in an application or may be combined with a random number generator.

Further modifications and alternative embodiments of various aspects of the invention may be apparent to those skilled in the art in view of this description. Accordingly, this description is to be construed as illustrative only and is for the purpose of teaching those skilled in the art the general manner of carrying out the invention. It is to be understood that the forms of the invention shown and described herein are to be taken as embodiments. Elements and materials may be substituted for those illustrated and described herein, parts and processes may be reversed, and certain features of the invention may be utilized independently, all as would be apparent to one skilled in the art after having the benefit of this description of the invention. Methods may be implemented manually, in software, in hardware, or a combination thereof. The order of any method may be changed, and various elements may be added, reordered, combined, omitted, modified, etc. Changes may be made in the elements described herein without departing from the spirit and scope of the invention as described in the following claims.

What is claimed is:

1. A method of assessing parallel random number streams, comprising:

creating mixed random number streams by mixing two or more parallel random number streams, wherein mixing the two or more parallel random number streams comprises pairing at least one of the random number streams with at least one other of the random number streams;

computing, by a computer system, for each of the mixed random number streams, an inter-stream correlation value based on a correlation between the bivariate pairs constructed from the mixed stream; and

determining, from inter-stream correlation values for two or more mixed random number streams, a quality metric for the parallel random number streams.

2. The method of claim **1**, wherein determining the quality metric comprises off-line testing of the two or more parallel random number streams, wherein the two or more parallel random number streams are generated by a random number generation system for purposes of testing the random number generation system.

3. The method of claim **1**, wherein determining the quality metric comprises on-line testing of the two or more parallel random number streams during consumption of the random numbers by an application.

4. The method of claim **1**, wherein determining the quality metric comprises combining inter-stream correlation values for at least two random number streams.

5. The method of claim 1, wherein mixing two or more parallel random number streams comprises receiving a user selection of a mixing approach.

6. The method of claim 1, wherein the set of all streams may be mixed.

7. The method of claim 1, wherein the set of all streams may be grouped into subsets.

8. The method of claim 1, wherein mixing a set or subset of three or more parallel random number streams comprises biased interleaving of a stream with the remaining streams in the set or subset.

9. The method of claim 1, wherein mixing a set or subset of two or more parallel random number streams comprises shuffled interleaving of all streams in the set or subset.

10. The method of claim 1, wherein mixing a set or subset of two parallel random number streams comprises pair-wise interleaving of at the two streams.

11. The method of claim 1, wherein determining the quality metric comprises receiving a user selection of a test method.

12. The method of claim 1, wherein the quality metric comprises a significance level, wherein the significance level comprises a level above which a claim of independence cannot be rejected.

13. The method of claim 1, wherein the quality metric is tested against a user-specified significance level.

14. The method of claim 1, wherein the quality metric is determined based on an aggregate method.

15. The method of claim 1, wherein the quality metric is determined based on a goodness-of-fit method.

16. The method of claim 1, wherein the quality metric is determined based on a percentile method.

17. The method of claim 1, wherein the quality metric is determined based on a mean absolute deviation method.

18. The method of claim 1, further comprising applying a polar transform to remove some bivariate pairs from a mixed random number stream from the determination of the quality metric, wherein removing the one or more bivariate pairs reduces correlations among the random number streams used in creating the mixed random number stream.

19. The method of claim 1, further comprising determining whether the quality metric for the two or more parallel random number streams meets a user-specified significance level.

20. A system, comprising:

a processor;

a memory coupled to the processor, wherein the memory comprises program instructions executable by the processor to implement:

creating mixed random number streams by mixing two or more parallel random number streams, wherein mixing the two or more parallel random number streams comprises pairing at least one of the random number streams with at least one other of the random number streams;

computing, for each of the mixed random number streams, an inter-stream correlation value based on a correlation between the bivariate pairs constructed from the mixed stream; and

determining, from inter-stream correlation values for two or more mixed random number streams, a quality metric for the parallel random number streams.

21. A non-transitory, computer-readable storage medium comprising program instructions stored thereon, wherein the program instructions are configured to implement:

creating mixed random number streams by mixing two or more parallel random number streams, wherein mixing the two or more parallel random number streams comprises pairing at least one of the random number streams with at least one other of the random number streams;

computing, for each of the mixed random number streams, an inter-stream correlation value based on a correlation between the bivariate pairs constructed from the mixed stream; and

determining, from inter-stream correlation values for two or more mixed random number streams, a quality metric for the parallel random number streams.

* * * * *

Appendix D: CPRNG Patent

PATENT
5660-14400

GENERATION OF DISTINCT PSEUDORANDOM NUMBER STREAMS BASED ON PROGRAM CONTEXT

By:

Rajendra V. Boppana

Atty. Dkt. No.: 5660-14400

Customer No. 35690
Eric B. Meyertons
Meyertons, Hood, Kivlin, Kowert & Goetzel, P.C.
P.O. Box 398
Austin, TX 78767-0398
Ph: (512) 853-8800

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

NOTICE OF ALLOWANCE AND FEE(S) DUE

35690 7590 06/10/2015

MEYERTONS, HOOD, KIVLIN, KOWERT & GOETZEL, P.C.
P.O. BOX 398
AUSTIN, TX 78767-0398

EXAMINER
YAARY, MICHAEL D

ART UNIT	PAPER NUMBER
2193	

DATE MAILED: 06/10/2015

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
13/426,028	03/21/2012	Rajendra V. Boppana	6693-14400	2562

TITLE OF INVENTION: GENERATION OF DISTINCT PSEUDORANDOM NUMBER STREAMS BASED ON PROGRAM CONTEXT

APPLN. TYPE	ENTITY STATUS	ISSUE FEE DUE	PUBLICATION FEE DUE	PREV. PAID ISSUE FEE	TOTAL FEE(S) DUE	DATE DUE
nonprovisional	SMALL	$480	$0	$0	$480	09/10/2015

THE APPLICATION IDENTIFIED ABOVE HAS BEEN EXAMINED AND IS ALLOWED FOR ISSUANCE AS A PATENT. PROSECUTION ON THE MERITS IS CLOSED. THIS NOTICE OF ALLOWANCE IS NOT A GRANT OF PATENT RIGHTS. THIS APPLICATION IS SUBJECT TO WITHDRAWAL FROM ISSUE AT THE INITIATIVE OF THE OFFICE OR UPON PETITION BY THE APPLICANT. SEE 37 CFR 1.313 AND MPEP 1308.

THE ISSUE FEE AND PUBLICATION FEE (IF REQUIRED) MUST BE PAID WITHIN THREE MONTHS FROM THE MAILING DATE OF THIS NOTICE OR THIS APPLICATION SHALL BE REGARDED AS ABANDONED. THIS STATUTORY PERIOD CANNOT BE EXTENDED. SEE 35 U.S.C. 151. THE ISSUE FEE DUE INDICATED ABOVE DOES NOT REFLECT A CREDIT FOR ANY PREVIOUSLY PAID ISSUE FEE IN THIS APPLICATION. IF AN ISSUE FEE HAS PREVIOUSLY BEEN PAID IN THIS APPLICATION (AS SHOWN ABOVE), THE RETURN OF PART B OF THIS FORM WILL BE CONSIDERED A REQUEST TO REAPPLY THE PREVIOUSLY PAID ISSUE FEE TOWARD THE ISSUE FEE NOW DUE.

HOW TO REPLY TO THIS NOTICE:

I. Review the ENTITY STATUS shown above. If the ENTITY STATUS is shown as SMALL or MICRO, verify whether entitlement to that entity status still applies.

If the ENTITY STATUS is the same as shown above, pay the TOTAL FEE(S) DUE shown above.

If the ENTITY STATUS is changed from that shown above, on PART B - FEE(S) TRANSMITTAL, complete section number 5 titled "Change in Entity Status (from status indicated above)".

For purposes of this notice, small entity fees are 1/2 the amount of undiscounted fees, and micro entity fees are 1/2 the amount of small entity fees.

II. PART B - FEE(S) TRANSMITTAL, or its equivalent, must be completed and returned to the United States Patent and Trademark Office (USPTO) with your ISSUE FEE and PUBLICATION FEE (if required). If you are charging the fee(s) to your deposit account, section "4b" of Part B - Fee(s) Transmittal should be completed and an extra copy of the form should be submitted. If an equivalent of Part B is filed, a request to reapply a previously paid issue fee must be clearly made, and delays in processing may occur due to the difficulty in recognizing the paper as an equivalent of Part B.

III. All communications regarding this application must give the application number. Please direct all communications prior to issuance to Mail Stop ISSUE FEE unless advised to the contrary.

IMPORTANT REMINDER: Utility patents issuing on applications filed on or after Dec. 12, 1980 may require payment of maintenance fees. It is patentee's responsibility to ensure timely payment of maintenance fees when due.

PRIORITY CLAIM

[0001] This application claims priority to U.S. Provisional Application No. 61/454,856 entitled "GENERATION OF DISTINCT PSEUDORANDOM NUMBER STREAMS BASED ON PROGRAM CONTEXT" to Boppana filed March 21, 2011, which is incorporated herein by reference in its entirety.

BACKGROUND

Field

[0002] This disclosure is generally related to parallel computing applications, simulation codes and protocols that use pseudorandom numbers and more specifically to algorithms and methods to generate pseudorandom numbers.

Description of the Related Art

[0003] Many important scientific computing applications, business and finance applications, and complex systems modeling and analysis techniques use pseudorandom number generators ("RNGs"). These applications may take advantage of the availability of thousands of computing cores on heterogeneous systems comprising multi-core processors ("CPUs") and highly parallel general purpose graphics processing units ("GPUs"), provided that suitable parallel pseudorandom number generators ("PRNGs") are available to simultaneously feed thousands of computing streams with high quality random number ("RN") streams with low intra- and inter-stream correlations (inter-stream correlations may be referred to herein as "ISCs").

[0004] A parallel or distributed application has the computational task that may be divided into several thousands or millions of subtasks, with each subtask executed by a separate thread or process (henceforth, process). Each process has

distinct ID that is usually logically numbered within the context of the application execution.

[0005] For an iterative parallel application, each process may execute some of the iterations. For example, for a large lattice structure simulation, each process may simulate the working of a few of the lattice points. Therefore, processes often cycle through computing and communication mode. In the computing mode, a process may use the available data to perform new calculations needed to make progress toward the solution. In the communication mode, a process may send its data or receive other process' data.

[0006] It is common to use the single-program multiple data (SPMD) programming method to code parallel applications, in which each of the processes receives the same computer code but has explicit instructions that specify based on the process's ID its portion of the task.

[0007] If an SPMD-based parallel application code that uses random numbers is executed, all or some of the processes (spawned for the execution of the application code) request random numbers from the same program locations or contexts.

[0008] In some applications, all required processes may be spawned statically at the start of the code execution. In other applications, some of the processes are spawned initially and any additional processes are spawned dynamically by the existing processes based on the application data and the coded algorithm or model. In highly complex simulation codes, the initial processes may need to spawn additional processes, dynamically, during the execution. However, with SPMD programming method, all processes use the same application code with the task for each process specified by conditional statements based on the data and the process ID.

[0009] In some systems, to distinguish requests for random numbers from different processes, an application is coded such that each process uses a RN stream identifier to explicitly identify a distinct stream allocated to it. The stream allocated to

a process may be initialized by a special function call prior to generating or using any RNs from that stream.

[0010] A large application code that uses RNs may be executed by dividing the computing task among multiple processes. Typically, each process is allocated at least one distinct RN stream to provide the RNs needed during its computations. To improve randomness and to improve the reproducibility of results, an application may be coded such that each portion of computing workload, for example, each small subset of the iterations of a large iterative code, may be assigned a distinct RN stream identifier so that each workload may use a distinct RN stream for the necessary RNs in its execution. In such cases, especially for efficiency reasons, each process may be assigned one or more of the computing workloads, and thus, one or more of the distinct RN stream identifiers. It is computationally inefficient, hard to reproduce results, or both to code an application so that an RN stream is shared by multiple processes.

[0011] The RN streams to processes may be allocated based on the input data and/or computations allocated to them. For example, if a computational loop is partitioned cyclically among p processes, then iteration i may be executed by process $i\%p$; if each iteration is to use a separate RN stream, then the number of iterations is smaller than the maximum of RN streams and it may be natural to allocate RN streams $i, i+p, \ldots$ from the set of all RN streams to process i.

[0012] One way to ensure that distinct RN streams are used is to allocate distinct RN stream identifiers and to use a PRNG that ensures that distinct RN stream identifiers result in initialization of distinct RN streams, which for a well-designed PRNG, may have low or undetectable—based on the currently available statistical and other tests—interstream correlations.

[0013] If the application requires each process or computational workload to request random numbers from multiple program locations or contexts, then there may be two options. One option is to use the same RN stream for all contexts within a

process. The same contexts in two different processes will still use distinct RN streams provided distinct stream identifiers are allocated and initialized for different processes.

[0014] A second option is to use multiple distinct streams for multiple contexts in each process, potentially one distinct RN stream for each distinct program context. This second option may be desirable for better randomness properties. In such a case, the application code is explicitly written to manage these multiple streams. If the number of distinct streams needed for an application is not known in advance, the maximum number of streams needed per process is estimated and the same are allocated to each process.

[0015] If the estimation is too small, then a program error is generated and execution is halted. In this case, the user needs to revise the estimate for the number of streams needed and resubmit the application for execution.

[0016] If the estimation is too large, then the program may run out of distinct RN streams for processes spawned after some point. This is especially true for parallel applications that are tuned and run on large clusters of computers with a large number of processes are run on even larger clusters of computers with even more processes, by a simple change in compile-time or runtime options without application recoding, to take advantage of the additional performance offered by the larger hardware.

[0017] To further control the generation of RN streams, an application may provide a single-seed value, typically by a designated master process (usually process 0) to a PRNG. The single-seed value is typically a 32- or 64-bit number, often an integer, specified by the user as part of the application's input data. By keeping all other input data the same and changing only the seed value, the user can run multiple instances of the same scenario, average the results and obtain potential simulation error estimations (also called, confidence intervals in statistics).

[0018] The quality of the random numbers used may be crucial for quick and accurate solutions to simulation-based computer solutions and for robust security protocols and security keys used in security protocols. It may be desirable to use distinct parallel RN streams if an application code calls for RNs from multiple distinct locations so that, within a process, multiple calls for RNs from the same location (also called, program context) are satisfied by providing RNs from a specific stream, while the calls for RNs from different locations of the program within the same computing iteration will be satisfied by providing RNs from different streams. Distinct RN streams across different processes may be ensured by the use of distinct RN stream identifiers to initialize the RN streams. To use distinct RN streams for distinct contexts within a process or computational workload, the application has to be coded specifically to use distinct RN stream identifiers for each such program context. Such an approach may, however, provide an unreasonable burden on the application designer and make revisions to application code, which may change the number of program contexts from which RNs are requested, cumbersome and potentially error-prone.

[0019] In some parameterized PRNGs, each process is given one RN stream with appropriately parameterized seed or iteration function. Two main approaches to design PRNGs are (a) splitting a sequential RN stream into multiple substreams, with each substream treated as a distinct RN stream for application execution purposes, and (b) parameterization of the initialization (seed) state of an RNG with multiple random number cycles or the parameterization of the iteration function of the initialization of an RNG. The leap-frog technique which splits a sequential RN stream in an interleaved manner — if a sequential stream consisting of $x_1, x_2, x_3, \ldots$ needs to be split into k streams, then stream i consists of RNs $x_i, x_{k+i}, x_{2k+i}, \ldots, 1 \leq i \leq k$ —received extensive attention. But it is inherently not scalable owing to initialization cost—a large multiple of k RNs must be generated first to initialize each processor/process—and potentially increased intra-stream correlations.

[0020] The Mersenne twister (MT) is a variant of feedback shift register-based random number generator. The original generator MT19937, which generates a single RN stream with a very long cycle of length 2^{19937} (that is, the sequence of RNs repeats after generating this many RNs), is very popular and is widely implemented in various software packages (including Gnu Scientific Library, gsl package). SFMT19937, a parallel 128-bit version, and MTGP, a GPU version as part of NVIDIA CUDA library, are also available. Using MT to generate multiple parallel RN streams often requires splitting its sequential RN stream. This is largely an ad hoc process since the maximum number of RNs needed in each segment needs to be estimated. This also may compromise the randomness quality since segmenting the stream and using the segments changes the correlations among the RNs used. Direct parallelization by changing the parameters of MT is computationally expensive and may not be suitable for dynamic generation of random number streams in a high-performance simulation code.

SUMMARY

[0021] In an embodiment, a method of providing random number streams to a process includes determining one or more program contexts within a process. Each of the program contexts may include code that calls for one or more random numbers. For each of at least two of the program contexts, a random number stream is provided to the process. The random number stream for each program context is based on the determined program context and is distinct from the random number stream for the other program contexts in the process.

[0022] In an embodiment, a method of providing random numbers streams to processes performing a parallel computation includes determining program contexts within one process of a parallel computation. Each of the program contexts may include code that calls for one or more random numbers. A random number stream is provided to the process for each of the program contexts. The random number stream provided is based in part on the determined program context and based in part on which of the two or more processes the program context is in.

[0023] In an embodiment, a method of providing random numbers streams to processes performing a parallel computation includes receiving a call for one or more random numbers from a program context in a process of a parallel computation. A random number stream is used to provide a random number for each such call. The random number stream provided is based at least in part on the determined program context.

[0024] In some embodiments, a context-aware parallel pseudorandom number generator uses the program context in which a request for a random number is made to automatically select and use distinct random number streams for distinct contexts.

BRIEF DESCRIPTION OF THE DRAWINGS

[0025] FIG. 1 is a block diagram illustrating a random number generator that provides distinct random number streams to different program contexts of a parallel computation.

[0026] FIG. 2 is a block diagram illustrating a random number generator that can provide distinct random number streams to different program contexts and different processes of a parallel computation based on program context and other information.

[0027] FIG. 3 illustrates providing random number streams to a process based on a determined program context.

[0028] FIG. 4 illustrates one embodiment of the initialization process by a context-aware random number generator.

[0029] While the invention is described herein by way of example for several embodiments and illustrative drawings, those skilled in the art will recognize that the invention is not limited to the embodiments or drawings described. It should be understood, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims. The headings used herein are for organizational purposes only and are not meant to be used to limit the scope of the description or the claims. As used throughout this application, the word "may" is used in a permissive sense (i.e., meaning having the potential to), rather than the mandatory sense (i.e., meaning must). Similarly, the words “include”, “including”, and “includes” mean including, but not limited to.

DETAILED DESCRIPTION OF EMBODIMENTS

[0030] As used herein, "random number" includes a pseudorandom number. As used herein, a "random number generator" includes a pseudorandom number generator.

[0031] As used herein, a "context-aware parallel pseudorandom number generator" means a parallel pseudorandom number generator which generates one or more random number streams and provides random numbers based on information relating to a program context for requesting random numbers.

[0032] As used herein, the phrase "primitive process", or simply "process", is used to represent a thread or process assigned to execute one computational workload. In some cases, a thread or process used in an execution of the application may perform the work of multiple primitive processes.

[0033] In some embodiments, distinct random number streams are assigned to different program contexts. The streams may be assigned such that no two processes cooperatively working on a parallel computation use the same random number stream. In some embodiments, the use of program context enables context-aware parallel pseudorandom number generators to generate distinct random number streams even for processes that use only one stream identifier by call for random numbers from multiple locations.

[0034] In some embodiments, a collection of random number streams is given to each process so that each distinct statement (denoted, random number context) that calls for a random number is served with a distinct generator taken from the PRNGs assigned to that process. To ensure that each process of the parallel computation that executes the same code uses distinct random number streams, the streams may, in certain embodiments, be further initialized with distinct RN stream identifiers supplied by the application code. This RN stream identifier may be used to

determine a distinct identifier, in 64- or more bits, generated by a special library module.

[0035] In some embodiments, random number context (RN-context) is used in conjunction with the RN stream identifier to determine the RN stream to be used. The RN context may be derived from the return address of the function call to the random number generator.

[0036] FIG. 1 is a block diagram illustrating a random number generator that provides distinct random number streams to different program contexts of a parallel computation. Parallel computation 100 includes processes 102. In some embodiments, processes 102 each include SPMD-based parallel application code for carrying out parallel computation 100. Contexts 104 may correspond to a location in the code of one of processes 102. Processes 102 include contexts 104.

[0037] Random number generator 106 may provide random number streams to contexts 104 in processes 102. Each of contexts 104 may make calls 108 requesting random numbers. In response, random number generator 106 may generate a random number stream 110 to the context. In some embodiments, each random number stream 110 is generated from, or retrieved from, one of library modules 114.

[0038] In some embodiments, a distinct stream is provided to each random number context. For example, the random number stream provided to context A of process 1 may be distinct from the random number streams provided to context B of process 1, which may be different from the random number stream provided to context C of process 1, and so on.

[0039] Each of processes 102 may include multiple iterations 112. Each of iterations 112 may be associated with an iteration number. For each of iterations 112 of processes 102, context 104 may separately call for a random number stream.

[0040] In some embodiments, random number context (RN-context) is used with other information to determine an RN stream to be used for a computation. The RN context may be derived from the return address of the function call to the random number generator, a process number or thread number, an iteration number (if appropriate), any user supplied stream identifier, or a combination of one or more of these elements. A user supplied stream identifier may be, for example, an index to RN stream contexts or a pointer to a data structure containing the RN stream context.

[0041] FIG. 2 is a block diagram illustrating a random number generator that can provide distinct random number streams to different program contexts and different processes of a parallel computation based on program context and other information. An application's request for a random number may provide user-specified stream ID 120 to library module 114. A process ID 122 may be associated with each of processes 102. An iteration number 124 may be associated with each iteration of a process. User-specified stream ID 120, process ID 122, and iteration number 124 may be accessed by random number generator 106. In some embodiments, random number generator 106 uses one or more of user-specified stream ID 120, process ID 122, and iteration number 124, in combination context information associated with one of contexts 104, to determine the random number stream to be used to provide one or more random numbers to the context. The random number stream may be initialized if it is not already initialized, as in the case of the first call to this stream.

[0042] Each of processes 102 may have unique process ID 122. Random number generator 106 may provide a distinct stream to each program context and process. Thus, for example, random number stream 115 supplied to Context A of process 2 in response to call 113 may be distinct from random number stream 110 supplied to Context A of process 1 in response to call 108.

[0043] In one embodiment, context-aware parallel pseudorandom number generators are implemented as library modules that can be linked to application codes

at the compile time. Random numbers may be retrieved from the CPRNG library using function calls at the run time.

[0044] FIG. 3 illustrates providing random number streams to a process based on a determined program context. At 200, a program context is determined for program contexts within a process. Each of the program contexts may include code that calls for one or more random numbers. For example, referring to FIG. 1, process 1 includes program context A, Context B, and Context C.

[0045] At 202, a random number stream is provided for each of the program contexts based on the determined program context. For example, referring to FIG. 1, random number generator 106 may provide a distinct random number stream to each of Context A, Context B, and Context C in process 1. For example, random number stream 111 provided to Context B in response to call 109 may be distinct from random number stream 110 provided to Context A in response to call 108.

[0046] In some embodiments, random number streams are generated for two or more processes in a parallel computation. The random numbers streams may be provided such that the random number streams used by one process are distinct from those of other processes. In certain embodiments, streams are generated such that the corresponding contexts of different parallel processes are provided with distinct random number streams. For example, random number generator 106 may provide a random number stream to context A of process 1 that is distinct from the random number stream provided to context A of process 2.

[0047] In some embodiments, a parameterized pseudorandom number generator (RNG) is used to generate a large number of random number (RN) streams. The RNG may be augmented with a scalable and automatic initialization process. Parameterized PRNGs that may be used in some embodiments of a context-aware random number generator include an additive lagged Fibonacci generator (ALFG) or a multiplicative lagged Fibonacci generator (MLFG).

[0048] An additive lagged Fibonacci generator (ALFG) uses an addition-based recursion:

[0049] $x_n = x_{n-k} + x_{n-l} \pmod{2^m}, \ 0 < k < l < n,$

[0050] where l and k are the lags (or indices to the older numbers used to generate the new number), n, l, k are positive integers, and x_i's are m-bit random numbers. The values $l = 17$ and $k = 5$ are commonly used to generate multiple distinct streams of 32- or 64-bit RNs. However, to pass very stringent intra-stream correlations tests, the lag, l, needs to be very high, over 1000.

[0051] A drawback of ALFG may be the initialization cost of l words before generating any RNs that can be used by the application code.

[0052] An advantage of ALFG may be that it has a large number of independent and long cycles of RNs. For a b-bit, r lagged ALFG, there are $2^{(b-1)(l-1)}$ cycles, each of length $(2^l - 1)2^{b-1}$.

[0053] A multiplicative lagged Fibonacci generator (MLFG) is similar to ALFG except that multiplication instead of addition is used in the recursion. MLFG has only one-fourth as many cycles, and each of only one-fourth as long as those in ALFG. MLFGs may be suitable in many embodiments of a CPRNG, since even with a small lag of 17, it may be feasible to generate RN streams that pass many of the stringent tests.

[0054] The multiplicative Fibonacci lagged generator (MLFG) uses the recurrence relation

[0055] $x_n = x_{n-k} \times x_{n-l} \pmod{2^m}, \ 0 < k < l < n,$

[0056] Where m is the random integer size in bits, l and k are the lags or offsets to the stream of previously generated random numbers, and x_i, $i > l$, are the random numbers generated. RNs $x_1, x_2, \ldots, x_l$ form the initialization (seed)

sequence or state and the initial words of a RN stream. The state of RN stream is always given by its most recent l words. Theoretical results show that each distinct combination of certain $(l-3)\times(m-1)$ of the $l\times m$ bits in the seed gives a distinct RN stream for a total of $2^{(l-3)(m-1)}$ streams, each with a cycle of $2^{(l-3)}\times(2^m-1)\approx 2^{m+l-3}$ RNs. Therefore, there are $(m-3)\times(l-1)$ bits that may need to be determined uniquely for each RN stream initialization (seed) sequence.

[0057] A 64-bit MFLG with lag 17 may be implemented in one example. With 64-bit integers and a lag of 17, there are $2^{61\times16}=2^{976}\approx 6\times10^{293}$ different RN streams, each with distinct 976-bit seed value and a cycle length of $2^{61}\cdot(2^{17}-1)\approx 2^{78}\approx 3\times10^{23}$. A few of the lower bits of x_i's may be discarded and remaining bits of x_i's are used to supply the RNs to improve the randomness since the lower bits are often less random owing to the arithmetic operation involved. The random numbers may be provided as integers or as real numbers in the range [0,1) by computing the fractions resulting from the division of the integer x_i's with $1+\max\_rn$, where $\max\_rn$ is the maximum value an x_i may take. In one embodiment, a PRNG package called SPRNG and the MLFG available from its library are used to implement a CPRNG.

[0058] In one implementation of context-aware random number generation, a SPRNG library package provides init_rng() and get_rn_dbl() function calls to initialize a new RN stream and to obtain the next RN in an already initialized stream, respectively. The init_rng function is called by specifying the seed, parameters set that specify the lags and the locations of the odd numbered words in the initial set of lag words, maximum number of RN streams (denoted max_str) that will be requested by the application, and cur_str, the RN stream number in the range 0,1,…,max_str-1 that needs to be initialized. The seed, parameter set, and max_str may be common in all init_rng() calls. Each call to init_rng function returns a pointer to one RN stream.

[0059] In one embodiment of a CPRNG implementation, each init_rng() call allocates not just one RN stream but a set of distinct RN streams and returns a pointer, str_ptr, to the set; the streams in this set can be customized with program context without further calls to init_rng(). The RN-context, the context or the program location from which a RN number is requested, is used in addition to the stream-set pointer, str_ptr, to determine the specific RN stream to be used. The RN context may be derived from a combination of the program line number in the source code, the return address of the function call to get_rn_dbl(), the process/thread numbers, and any user supplied identifiers such as the iteration number. When the application requests for a random number using the function call get_rn_dbl(str_ptr), the RN-context is used to determine the specific RN stream to be used in the set of streams pointed by str_ptr. The appropriate RN stream may be automatically initialized with the RN-context, if it is the first call from this context, and a RN from the stream is returned.

[0060] Each call to init_rng() may result in the initialization of the RN stream specified by the stream number, cur_str, and the calling code is given a pointer to the RN stream that should be used as argument in the function call get_rn_dbl() to obtain the next RN in the stream.

[0061] In this example embodiment, CPRNG differs from the MLFG in the SPRNG package in several ways: (a) automatically generating distinct RN streams based on program context for the same str_ptr value; (b) initialization method used to seed RN streams to improve the randomness and also to ensure that RN context can be added to dynamically create distinct RN streams without requiring additional init_rng() calls; (c) the distinct ID field that allocates distinct values for a portion of the seed sequence statically (when the cur_str value is less than max_str value in the function call init_rng()) and additional seed sequences dynamically beyond the max_str limit in case the application requires more RN streams than originally estimated. Extensive statistical tests are used to show that CPRNG implementation of MLFG generates billions of RN streams with low interstream correlations while the

implementation of the same theoretical generator in SPRNG exhibits statistically significant correlations for more than a million streams. The specification of max_str limits the maximum number of cur_str values that can be used to call init_rng() in SPRNG implementation, whereas max_str is a threshold to determine whether the initialization sequences are allocated statically or dynamically. Static allocation of the seed sequences improves repeatability of the computations when rerun with the same input data and dynamic allocation of seed sequences relieves the burden of specifying the maximum number of stream allocations needed a priori. Context-awareness provides distinct RN streams for distinct program contexts even when str_ptr used in the calls to get_rn_dbl() is the same. In SPRNG implementation, the application needs to be coded explicitly to use different str_ptr in calling get_rn_dbl() to achieve the same functionality. In this example embodiment, CPRNG may avoid such application coding and automate the management of distinct streams for distinct contexts.

[0062] FIG. 4 illustrates one embodiment of the initialization process by CPRNG. In this example shown in FIG. 4, the initialization may be based on lag parameters l and k, $0 < k < l - 1$. A call to init_rng() results initialization of $l - 3$ of the lag words using a sequential RNG such as the recursion with carry (RWC) generator, a 32-bit generator, initialized with the user specified seed integer. In this example, these lag words are common to the initialization of all RN streams regardless of the process number or RN-context. One of the remaining three lag words is filled with an ID that is guaranteed to be distinct for distinct cur_str numbers specified in init_rng(). The distinct ID word is common to the set of RN streams that are allocated based on different RN contexts but have the same cur_str number. The remaining two lag words are filled with the RN-context so that distinct RN-contexts result in distinct RN streams.

[0063] In the embodiment shown in FIG. 4, initialization of RN stream state by CPRNG. In this example, the state consists of l lag words. Each lag word is a 32-bit or, more typically, 64-bit word with maximum lag l, $l - 3$ of the lag words is

filled randomly based on the user specified seed and a sequential RNG. In this example, these words are common to all RN streams used during the execution of the application. Lag k, $k < l-1$, is initialized with a unique and distinct ID that is associated with the cur_str used in the init_rng() call. Lags $k+1$ and $k+2$ are initialized with RN-context to create a distinct RN stream for each distinct program context in each process.

[0064] For MLFG, all the lag words are odd values. Therefore, for each lag word, only $(m-3)$ of each lag word in an m -bit MLFG are determined uniquely, and a least significant bit determined by the canonical form and parameter set is appended to it to form an $(m-2)$-bit number, say, z. The actual lag word may be formed by using the operation $(-1)^y 3^z \bmod 2^m$, where y is a randomly generated 1 or 0. Henceforth, the discussion of a lag word initialization pertains to the generation of the $(m-3)$ bits since every initial lag word will be transformed using the operation $(-1)^y 3^z \bmod 2^m$. For a 64-bit MLFG, two consecutive 32-bit RNs generated by the RWC generator may be used form a 61-bit integer for the lag words filled by it. Similarly, only 61 bits of each of the lag words used for distinct ID word and the RN context words need to be determined uniquely.

[0065] In some embodiments, the number of bits used for distinct ID may be more or fewer than $m-3$ bits, and more than one lag word or only a portion of a lag word may be used. Up to $l-2$ lag words are available for distinct ID specification. Similarly, the number of bits used RN context may be more or fewer than $2(m-3)$ bits used in the example embodiment in FIG. 4. Furthermore, the positions of distinct ID bits and RN context bits can be anywhere in the $(m-3)\times(l-1)$ bits available to seed distinct RN streams. Any bits not used for distinct ID and RN context fields will be randomly filled with the RWC or some other good sequential random number generator initialized with user supplied 32-bit or 64-bit single-seed value

[0066] For a CPRNG based on MLFG with maximum lag $l = 17$ and 64-bit words, $2^{2\times 61} = 2^{122}$ distinct RN streams may be allocated with each init_rng() call. Based on the context and str_ptr argument used in a call to get_rn_dbl(), an appropriate stream is selected, automatically initialized prior to first use, and the next RN in the stream is returned. CPRNG may be used without RN-contexts by choosing appropriate parameters to init_rng() call. If RN-contexts are not used, then the two lag words that are normally filled with RN-context are filled with the random bits generated by the sequential RWC generator. The lag word with distinct ID may be used to ensure that RN streams are distinct for distinct values of cur_str specified in the init_rng(). CPRNG may be simply a basic MLFG when used without context.

[0067] For applications that use a large and variable number of RN streams, having to specify the maximum number of streams used during an execution run is a limitation. Furthermore, certain large-scale parallel applications may spawn additional processes and threads dynamically depending on the input data and intermediate results. To accommodate such situations, CPRNG may assign several (2^{10} in the example embodiment) consecutive distinct IDs for the lag word k upon a call to init_rng(), independent of any streams allocated to handle RN contexts. Therefore, CPRNG may allocate multiple initialization (seed) sequences, which can be used to initialize distinct RN streams by simply initializing the distinct ID lag word based on the unused distinct IDs allocated and keeping the other initialization words the same, to the calling process. Typically, only one of these IDs is used by a process. However, if a process spawns threads or child processes and needs to use additional distinct RN streams without going through the initialization process, it can have them without any communication overhead by using the original initialization with the distinct ID word replaced with one of the unused IDs from its allocated IDs. This leads to faster initialization of the new RN streams on demand. If more RN streams are needed and init_rng() is called with cur_str value greater than max_str, a monotonically increasing counter is used to ensure that the lag word K is distinct. However, the access to this counter may need to be serialized by using appropriate mutex locks in threaded applications or by assigning it to a process to serve the

counter values to the other processes of the application. In these instances, an additional communication or serialization overhead may be incurred by CPRNG compared to the static methods used in some packages. On the other hand, CPRNG provides virtually unlimited number of RN streams on demand, limited only by the number of bits used for the distinct ID, and avoids depletion of the available RN streams that can occur with static partitioning of the available RN streams for applications with many levels of dynamic process/thread creation.

[0068] In some existing parallel random number generators (PRNG), only the user supplied stream identifier is used to determine the RN stream, thus leaving the burden of managing multiple RN streams to the user. This can be onerous, especially if the application is iterative and RNs are consumed at multiple locations in each iteration. Use of a CPRNG may relieve a user from managing multiple streams for each thread or process. In some embodiments, the use of process/thread numbers may be used in addition to context information. The option of using process/thread number to determine RN contexts may be selected by a user at a compile-time or runtime. Use of a process/thread number in determining the RN context may reduce reproducibility of results.

[0069] In some embodiments, once a unique RN-context is determined, RN-context information may be embedded into a seed sequence to initialize an RN stream. The seed sequence may be, for example, a 976-bit sequence for a 64-bit MLFG with lag 17. In some cases, it may be sufficient to limit the RN-context size to, for example, two lag words (122 bits; only 61 bits of each 64-bit lag word are determined, and the remaining three bits are determined by a canonical form used to initialize the lag words). The RN-context may be concatenated with an additional deterministically generated distinct ID (one lag word or 61 bits) to further distinguish the initialization of RN streams. The remaining bits may be filled randomly using a good sequential RNG, such as a recursion with carry (RWC) generator using a user-supplied seed integer. These random bits may be common to the initialization of all RN streams.

[0070] In some embodiments, a CPRNG implements a scalable initialization of RN streams. In one embodiment, the CPRNG initializes RN streams using a return address, any user supplied identifier, seed information, and additional information that is generated by a CPRNG library. This additional information may be generated in different ways depending, for example, on the application code.

[0071] If the application is an MPI-based parallel program using single-program multiple data (SPMD) program model, then a special CPRNG module may be associated with process 0. The user may be neither aware of this nor expected to modify the application code. This CPRNG module may allocate several, for example, 2^{10}, consecutive distinct 64-bit IDs in response to each initialization request. Each RN context may be augmented with one of the distinct IDs.

[0072] Some MPI processes dynamically spawn processes/threads that use RN streams. In some embodiments, a process supplies its unused IDs to its child processes to automatically ensure that RN streams are distinct. If a process runs out of its allocated distinct IDs, then the CPRNG module may allocates additional distinct IDs. (In such instances, an additional communication overhead may be incurred by CPRNG compared to the static methods used in the some packages.) Such an approach may require very low communication among the processes for RN stream initialization.

[0073] For parametric studies based on Monte Carlo simulations, the RN streams used for each instance of simulation can be ensured to be distinct by specifying the specific IDs (fore example, 64-bit IDs) to be used as additional input file that will be used by the CPRNG library. A script (such as a Python script) may partition ID space and generate the additional input files.

[0074] In SPRNG and other works, the initialization for an RN stream may be determined based on a user-supplied stream identifier and a seed integer. The seed integer may be, for example, a 32-bit or a 64-bit integer. To handle the issue of new RN streams for additional processes/threads spawned dynamically, the RN

stream initialization space may be partitioned statically using a binary partitioning scheme to ensure initialization without any communication among processes. This can result in depletion of the initialization sequences quickly for applications with many levels of dynamic process/thread creation.

[0075] Although certain of the embodiments described above relate to simulations, systems and methods described herein may be used in a variety of applications. Examples of applications systems and methods described herein include (a) simulation-based solutions to large scientific and engineering problems, (b) parameterized Monte Carlo simulations of scientific, engineering, and finance problems, (c) distributed computing, and (d) protocols and keys used for information assurance and security.

[0076] Systems and methods described herein may be implemented in hardware including field programmable gate arrays (FPGAs) and application specific integrated circuit (ASIC) chips, or a suitable combination of hardware and software and which can be one or more software systems on a general purpose processor (CPU) or graphics processing unit (GPU).

[0077] Computer systems may, in various embodiments, include components such as a CPU with an associated memory medium such as Compact Disc Read-Only Memory (CD-ROM). The memory medium may store program instructions for computer programs. The program instructions may be executable by the CPU. Computer systems may further include a display device such as monitor, an alphanumeric input device such as keyboard, a directional input device such as mouse, a voice recognition system to dictate text and issue commands for processing, and a touch screen that may serve as a keyboard or mouse. Computer systems may be operable to execute the computer programs to implement computer-implemented systems and methods. A computer system may allow access to users by way of any browser or operating system.

[0078] Embodiments of a subset or all (and portions or all) of CPRNG may be implemented and executed in a computer and the random number streams and random numbers so generated are accessed via computer network by at least one other computer executing the application requesting random numbers.

[0079] Embodiments of a subset of all (and portions or all) of code and data needed for CPRNG operation—initialize and maintain random number streams and provide random numbers from these streams—may be stored on a remote computer, which, in turn, provides the said instructions and data via a computer network to at least one other computer, which executes uses the received instructions and data to initialize and maintain random numbers and provide random numbers for applications requesting the same.

[0080] Embodiments of a subset or all (and portions or all) of the above may be implemented by program instructions stored in a memory medium or carrier medium and executed by a processor. A memory medium may include any of various types of memory devices or storage devices. The term "memory medium" is intended to include an installation medium, e.g., a Compact Disc Read Only Memory (CD-ROM), floppy disks, or tape device; a computer system memory or random access memory such as Dynamic Random Access Memory (DRAM), Double Data Rate Random Access Memory (DDR RAM), Static Random Access Memory (SRAM), Extended Data Out Random Access Memory (EDO RAM), Rambus Random Access Memory (RAM), etc.; or a non-volatile memory such as a magnetic media, e.g., a hard drive, or optical storage. The memory medium may comprise other types of memory as well, or combinations thereof. In addition, the memory medium may be located in a first computer in which the programs are executed, or may be located in a second different computer that connects to the first computer over a network, such as the Internet. In the latter instance, the second computer may provide program instructions to the first computer for execution. The term "memory medium" may include two or more memory mediums that may reside in different locations, e.g., in different computers that are connected over a network. In some embodiments, a

computer system at a respective participant location may include a memory medium(s) on which one or more computer programs or software components according to one embodiment may be stored. For example, the memory medium may store one or more programs that are executable to perform the methods described herein. The memory medium may also store operating system software, as well as other software for operation of the computer system.

[0081] The memory medium may store a software program or programs operable to implement embodiments as described herein. The software program(s) may be implemented in various ways, including, but not limited to, procedure-based techniques, component-based techniques, and/or object-oriented techniques, among others. For example, the software programs may be implemented using ActiveX controls, C++ objects, as a library or standalone programs in a programming language such as C, C++, Java or in a scripting language such as Bash, Perl, Python, or AWK, JavaBeans, Microsoft Foundation Classes (MFC), browser-based applications (e.g., Java applets), traditional programs, or other technologies or methodologies, as desired. A CPU executing code and data from the memory medium may include a means for creating and executing the software program or programs according to the embodiments described herein.

[0082] Further modifications and alternative embodiments of various aspects of the invention may be apparent to those skilled in the art in view of this description. Accordingly, this description is to be construed as illustrative only and is for the purpose of teaching those skilled in the art the general manner of carrying out the invention. It is to be understood that the forms of the invention shown and described herein are to be taken as embodiments. Elements and materials may be substituted for those illustrated and described herein, parts and processes may be reversed, and certain features of the invention may be utilized independently, all as would be apparent to one skilled in the art after having the benefit of this description of the invention. Methods may be implemented manually, in software, in hardware, or a combination thereof. The order of any method may be changed, and various

elements may be added, reordered, combined, omitted, modified, etc. Changes may be made in the elements described herein without departing from the spirit and scope of the invention as described in the following claims.

WHAT IS CLAIMED IS:

1. A method of providing random number streams to a process, comprising:

determining one or more program contexts within a process, wherein at least one of the one or more program contexts comprises code that calls for one or more random numbers; and

providing, for each of at least one of the program contexts, a random number stream to the process, wherein the random number stream provided for at least one of the program contexts is based at least in part on the determined program context, and wherein the random number stream provided for at least one of the program contexts is distinct from the random number stream for at least one other of the program contexts.

2. The method of claim 1, wherein each of the program contexts is provided a random number stream that is distinct from the random number stream for any of the other program contexts in the process.

3. The method of claim 1, wherein providing the random number stream to the process for each of at least one of the two or more program contexts comprises providing a set of distinct random number streams in response to a call from one of the program contexts.

4. The method of claim 1, wherein providing the random number stream to the process for each of at least one of the two or more program contexts comprises initializing the states of the random number streams, wherein the states are used to generate distinct random number streams for at least two of the program contexts.

5. The method of claim 1, wherein the random-number context is determined based, at least in part, on the return address of a function call to obtain a random number.

6. The method of claim 1, wherein providing the random number stream to the process for each of at least one of the two or more program contexts comprises embedding context information into a seed sequence to initialize the random number stream.

7. The method of claim 1, wherein the process is one of two or more processes in a parallel process computation, wherein the random number stream provided for at least one of the program contexts is based in part on a process identifier for the process, wherein the random number stream is distinct from the random number stream provided for program contexts in at least one other process of the two or more processes in the parallel process computation.

8. The method of claim 1, wherein the random number stream provided for at least one of the program contexts is based in part on a user-supplied stream identifier for program context.

9. The method of claim 1, wherein providing the random number stream to the process for each of at least one of the two or more program contexts comprises receiving a stream identifier stored in, or generated from, a library module.

10. The method of claim 1, wherein the random number stream provided for at least one of the program contexts is based in part on an iteration number.

11. The method of claim 1, wherein the random number stream provided for at least one of the program contexts is based in part on a user-specified seed value.

12. The method of claim 1, wherein the process is a dynamically spawned process, wherein a random number stream allocated to it is based in part on unused initialization sequences from the random number streams originally allocated to the parent process from which the process was spawned.

13. A system, comprising:

a processor;

a memory coupled to the processor, wherein the memory comprises program instructions executable by the processor to implement:

determining one or more program contexts within a process, wherein at least one of the one or more program contexts comprises code that calls for one or more random numbers; and

providing, for each of at least one of the program contexts, a random number stream to the process, wherein the random number stream provided for at least one of the program contexts is based at least in part on the determined program context, and wherein the random number stream provided for at least one of the program contexts is distinct from the random number stream for at least one other of the program contexts.

14. The system of claim 13, further comprising:

a network of systems in which one or more systems may store portions or all of code and data needed for CPRNG and compute or provide instructions or data needed to use CPRNG or the random numbers to at least one or more other systems by way of the computer network.

15. The system of claim 13, wherein each of the program contexts is provided a random number stream that is distinct from the random number stream for any of the other program contexts in the process.

16. A non-transitory, computer-readable storage medium comprising program instructions stored thereon, wherein the program instructions are configured to implement:

determining one or more program contexts within a process, wherein at least one of the one or more program contexts comprises code that calls for one or more random numbers; and

providing, for each of at least one of the program contexts, a random number stream to the process, wherein the random number stream provided for at least one of the program contexts is based at least in part on the determined program context, and wherein the random number stream provided for at least one of the program contexts is distinct from the random number stream for at least one other of the program contexts.

17. The computer-readable storage medium of claim 16, wherein the program instructions further comprise:

CPRNG code and data in the storage medium of one computer accessed by way of a computer network by another computer to initialize and maintain random number streams and generate random numbers.

18. The computer-readable storage medium of claim 16, wherein each of the program contexts is provided a random number stream that is distinct from the random number stream for any of the other program contexts in the process.

19. A method of providing random numbers streams to processes performing a parallel computation, comprising:

determining one or more program contexts within one process of a parallel computation, wherein the parallel computation includes two or more

processes performed in parallel, wherein each of the one or more program contexts comprises code that calls for one or more random numbers; and

providing a random number stream to the one process for each of at least one of the one or more program contexts, wherein the random number stream provided is based in part on the determined program context and based in part on which of the two or more processes the program context is in.

20. The method of claim 19, wherein each of the program contexts is provided a random number stream that is distinct from the random number stream for any of the other program contexts in the process.

21. The method of claim 19, further comprising:

determining one or more program contexts within a second process of the parallel computation; and

providing a random number stream to the second process for each of at least one of the one or more program contexts,

wherein the random number stream to the second process is determined based in part on the determined program context and based in part on which of the two or more processes the program context is in,

wherein the random number stream provided for a program context is distinct from the random number stream provided for the program contexts in at least one other process of the two or more processes in the parallel computation.

22. The method of claim 19, wherein the random number stream is distinct from the random number stream provided for a corresponding program context in at least one other process of the two or more processes in the parallel computation.

23. The method of claim 19, wherein the random number stream provided for at least one of the program contexts is based in part on a process identifier for the process.

24. The method of claim 19, wherein the random number stream is distinct from the random number stream provided for program contexts in at least one other process of the two or more processes in the parallel computation.

25. The method of claim 19, wherein providing a random number stream to the one process for each of at least one of the two or more program contexts comprises providing a random number stream for each of at least two of the two or more program contexts,

wherein the random number stream provided for the program contexts is based at least in part on the determined program context, and

wherein the random number stream provided for at least one of the program contexts is distinct from the random number stream for at least one other of the program contexts.

26. The method of claim 19, wherein the random-number context is based, at least in part, on the return address of a function call to obtain a random number.

27. A method of providing random numbers streams to processes performing a parallel computation, comprising:

receiving a call for one or more random numbers from a program context in a process one process of a parallel computation, wherein one process is one of two or more processes performed in a parallel computation; and

providing a random number stream to the one process for the program contexts, wherein the random number stream provided is based at least in part on the determined program context.

28. The method of claim 28, wherein based in part on which of the two or more processes the program context is in.

29. The method of claim 28, wherein the random-number context is based, at least in part, on the return address of a function call to obtain a random number.

Amendments to the Claims

This listing of claims will replace all prior versions, and listings, of claims in the above-captioned application:

1. (Currently Amended): A method of dynamically providing random number streams to a process, comprising:

determining, by a processing device, a plurality of program contexts within the process, wherein each program context comprises calls for one or more random numbers; and

providing automatically, for each program context, a distinct random number stream, wherein the random number stream provided for one of the program contexts is based at least in part on the determined program context, and wherein the random number stream provided for one of the program contexts is distinct from the random number stream for at least one other of the program contexts.

2. (Original): The method of claim 1, wherein each of the program contexts is provided a random number stream that is distinct from the random number stream for any of the other program contexts in the process.

3. (Original): The method of claim 1, wherein providing the random number stream to the process for each of at least one of the two or more program contexts comprises providing a set of distinct random number streams in response to a call from one of the program contexts.

4. (Original): The method of claim 1, wherein providing the random number stream to the process for each of at least one of the two or more program contexts comprises initializing the

states of the random number streams, wherein the states are used to generate distinct random number streams for at least two of the program contexts.

5. (Currently amended): The method of claim 1, wherein one or more of the program contexts includes one or more random-number context~~[s]~~ and each of the random number contexts is determined based, at least in part, on a return address of a function call to obtain a random number.

6. (Original): The method of claim 1, wherein providing the random number stream to the process for each of at least one of the two or more program contexts comprises embedding context information into a seed sequence to initialize the random number stream.

7. (Original): The method of claim 1, wherein the process is one of two or more processes in a parallel process computation, wherein the random number stream provided for at least one of the program contexts is based in part on a process identifier for the process, wherein the random number stream is distinct from the random number stream provided for program contexts in at least one other process of the two or more processes in the parallel process computation.

8. (Original): The method of claim 1, wherein the random number stream provided for at least one of the program contexts is based in part on a user-supplied stream identifier for program context.

9. (Original): The method of claim 1, wherein providing the random number stream to the process for each of at least one of the two or more program contexts comprises receiving a stream identifier stored in, or generated from, a library module.

10. (Original): The method of claim 1, wherein the random number stream provided for at least one of the program contexts is based in part on an iteration number.

11. (Original): The method of claim 1, wherein the random number stream provided for at least one of the program contexts is based in part on a user-specified seed value.

12. (Original): The method of claim 1, wherein the process is a dynamically spawned process, wherein a random number stream allocated to it is based in part on unused initialization sequences from the random number streams originally allocated to the parent process from which the process was spawned.

13. (Currently Amended): A system, comprising:

a processor;

a memory coupled to the processor, wherein the memory comprises program instructions executable by the processor to implement:

determining, using the processor, a plurality of program contexts within a process, wherein each program context comprises calls for one or more random numbers; and

providing automatically, for each program context, a distinct random number stream, wherein the random number stream provided for one of the program contexts is based at least in part on the determined program context, and wherein the random number stream provided for one of the program contexts is distinct from the random number stream for at least one other of the program contexts.

14. (Original): The system of claim 13, further comprising:

a network of systems in which one or more systems may store portions or all of code and data needed for CPRNG and compute or provide instructions or data needed to

use CPRNG or the random numbers to at least one or more other systems by way of the computer network.

15. (Original): The system of claim 13, wherein each of the program contexts is provided a random number stream that is distinct from the random number stream for any of the other program contexts in the process.

16. (Currently amended): A non-transitory, computer-readable storage medium comprising program instructions stored thereon, wherein the program instructions are configured to implement:

determining one or more program contexts within a process, wherein at least one of the one or more program contexts comprises code that calls for one or more random numbers; and

providing automatically, for each of at least one of the program contexts, a random number stream to the process, wherein the random number stream provided for at least one of the program contexts is based at least in part on the determined program context, and wherein the random number stream provided for at least one of the program contexts is distinct from the random number stream for at least one other of the program contexts.

17. (Original): The computer-readable storage medium of claim 16, wherein the program instructions further comprise:

CPRNG code and data in the storage medium of one computer accessed by way of a computer network by another computer to initialize and maintain random number streams and generate random numbers.

18. (Original): The computer-readable storage medium of claim 16, wherein each of the program contexts is provided a random number stream that is distinct from the random number stream for any of the other program contexts in the process.

19-29. (Canceled)

ABSTRACT

A method of providing random number streams to a process includes determining two or more program contexts within a process. Each of the program contexts may include code that calls for one or more random numbers. For each of at least two of the program contexts, a random number stream is provided to the process. The random number stream for each program context is based on the determined program context and is distinct from the random number stream for the other program contexts in the process.

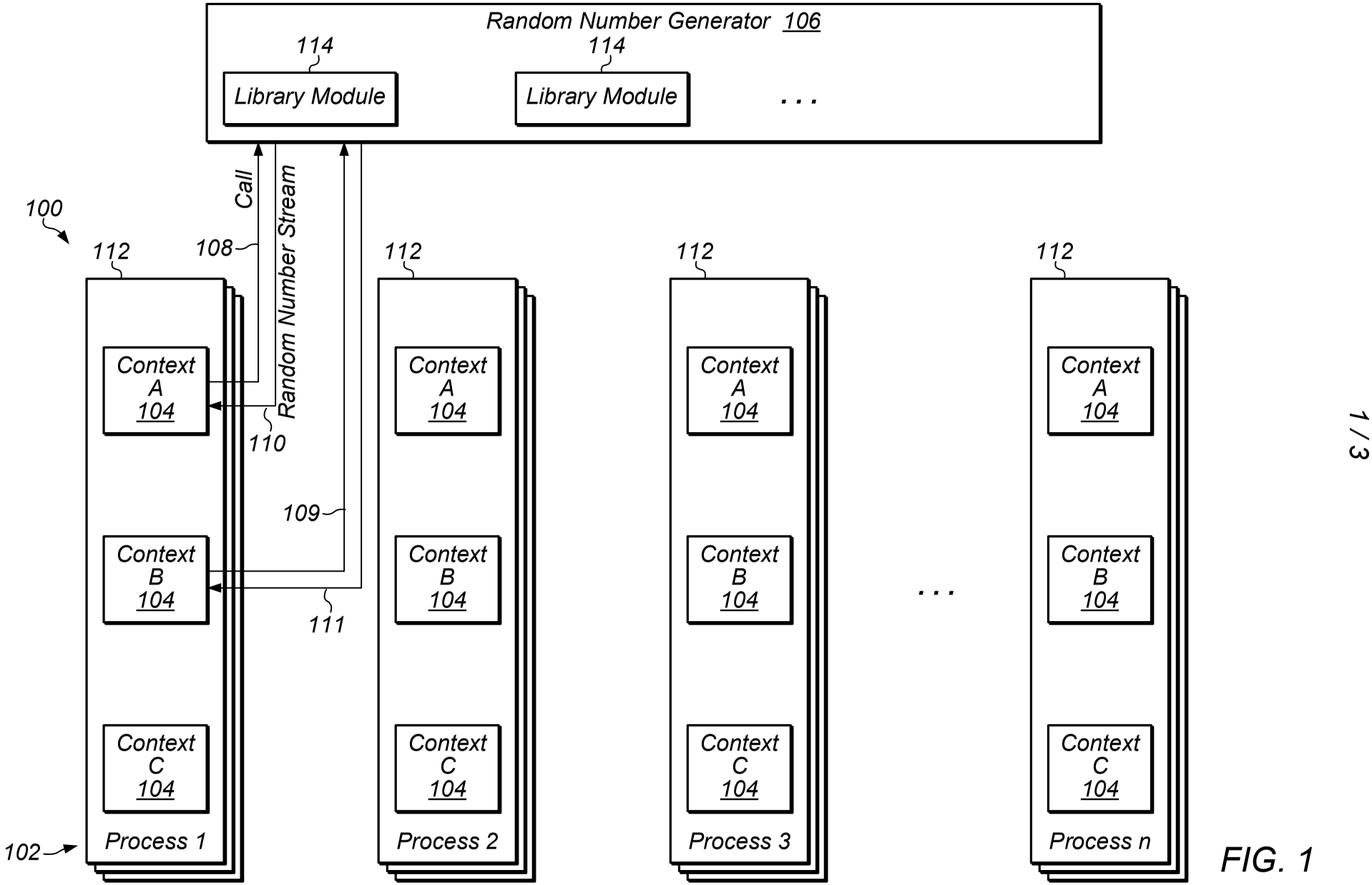

Random Number Generator 106
114
Library Module
114
Library Module
. . .
100
Call
108
Random Number Stream
110
109
111
112
Context A 104
Context B 104
Context C 104
Process 1
Process 2
Process 3
Process n
. . .
102

FIG. 1

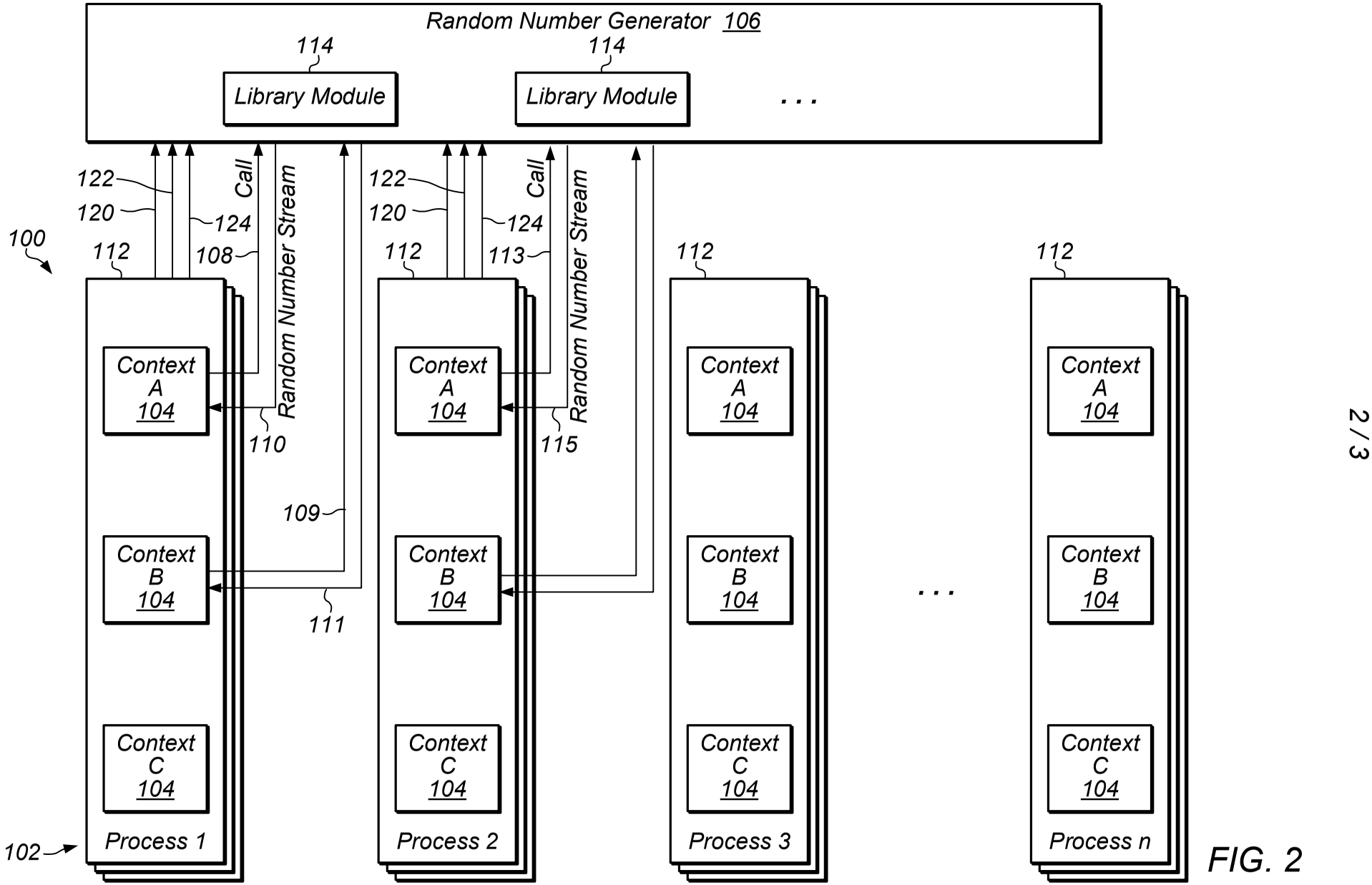

Random Number Generator 106
114
Library Module
114
Library Module
. . .
122
120
124
108
Call
Random Number Stream
100
112
Context A 104
Context B 104
Context C 104
110
109
111
Process 1
102
122
120
124
113
Call
Random Number Stream
112
Context A 104
Context B 104
Context C 104
115
Process 2
112
Context A 104
Context B 104
Context C 104
Process 3
. . .
112
Context A 104
Context B 104
Context C 104
Process n

FIG. 2

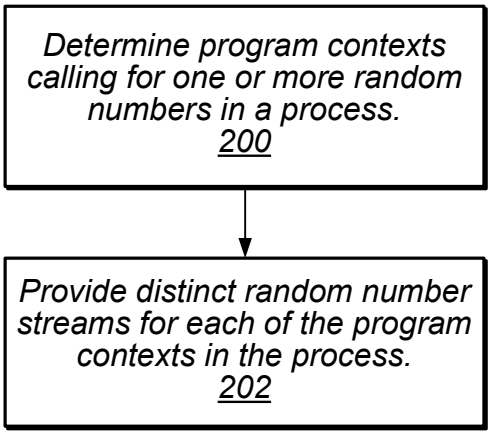

Determine program contexts calling for one or more random numbers in a process.
200
Provide distinct random number streams for each of the program contexts in the process.
202

FIG. 3

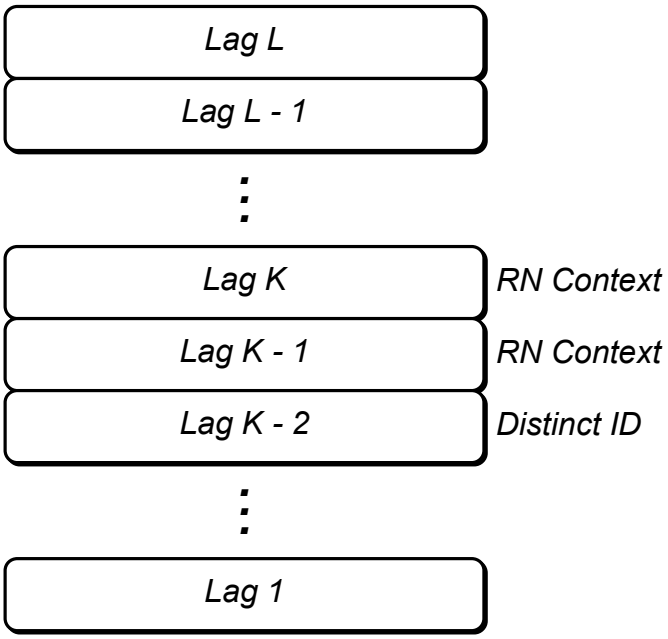

Lag L
Lag L - 1
Lag K
RN Context
Lag K - 1
RN Context
Lag K - 2
Distinct ID
Lag 1

FIG. 4