Monte Carlo Simulation, Schuëller & Spanos (eds) © 2001 Balkema, Rotterdam, ISBN 90 5809 188 0

Parallel inversive congruential generators: Software and field-programmable gate array implementations

Michael Mascagni
Department of Computer Science, Florida State University, Tallahasse, Fla., USA
Shahram Rahimi
Program in Scientific Computing, University of Southern Mississippi, Hattiesburg, Miss., USA

ABSTRACT: Monte Carlo methods (MCMs) are extremely effective algorithms for solving a wide variety of problems. In addition, since MCMs are based on the random sampling they are usually very easy to run on parallel computers. To this end, considerable work has been done on the construction of high quality pseudorandom number generators (PRNGs) for parallel and distributed computing. PRNGs are typically based on simple, linear recursions over the integers. There are many reasons for this, chief among them being the extra computational cost associated with nonlinear recursions. Here, we discuss the optimization of the implicit and explicit inversive congruential generator (ICG) for use in the popular Scalable Parallel Random Number Generators (SPRNG) library. ICGs are nonlinear variants of the popular linear congruential generators (LCGs) and are of interest due to certain properties they possess that lead to improved quality over linear generators. However, their major drawback is the extra cost required to generate numbers with ICGs. If the cost of generating a number with an LCG modulo m is a unit of computational work, then an ICG requires $O(\log_2 m)$ computational work. The extra cost incurred in the calculation of a number via the ICG is due to the cost of computing the multiplicative inverse modulo m. In this paper, we present an optimal algorithm for computing this modular inverse along with a Field Programmable Gate Array (FPGA) version of modular inversion. Both methods we discuss here are based on simple variants of the extended Euclidean algorithm. We compare the costs of the two implementations and discuss their incorporation into the SPRNG library and their future use on a new, hybrid, supercomputing architecture . The overall improvement in the FPGA implementation over the optimal software implementation was a factor ranging from a maximum of 10.4 to a minimum of 4.6 with a reasonable sized modulus, $m = 2^{31} - 1$.

1 INTRODUCTION

Monte Carlo methods (MCMs) are extremely effective algorithms for solving a wide variety of problems. In addition, since MCMs are based on the random sampling they are usually very easy to run on parallel computers. To this end, considerable work has been done on the construction of high quality pseudorandom number generators (PRNGs) for parallel and distributed computing. Among one of the most popular of these systems is the Scalable Parallel Random Number Generators (SPRNG) library, (Mascagni, Ceperley, Mitas, Saied, & Srinivasan 1998). SPRNG is a library consisting of several different random number generators. In this article we present the optimizations used and the implementation details of SPRNG's newest random number generator family, the implicit and explicit inversive congruential generator (ICG). In particular, an optimal software implementation and a "hardware" implementation of modular inversion, the most computationally costly part of computing the ICG, are presented, compared, and discussed.

The plan of the paper is as follows. In §2, we give a brief introduction to the ICG. Here we describe what one looks for in good PRNGs, and why the ICG is of particular interest for use both in serial and in parallel. Of particular note is the existence of theoretical results that indicate how to construct vectors of high quality from explicit ICGs. This forms the basis of our proposed parallel implementation. In §3 we briefly introduce the parallelization procedure we use to incorporate the ICG into SPRNG. The procedure used is both consistent with the software architecture design of SPRNG and is based on theory described in the previous section. The key deficiency in ICGs is the high computational cost of performing a necessary modular inversion. In §4 we describe an optimal software implementation of modular inversion using an optimized version of the extended Euclidean algorithm. While better than the naïve approach, software implementations cannot provide adequate performance for modular inversion. Thus, in §5, we are forced to consider a hardware acceleration technique. Our approach is the design of a **F**ield **P**rogrammable **G**ate **A**rray (FPGA) implementation for modular inversion. We describe our FPGA implementation of the extended binary Euclidean algorithm, and compare this implementation to our optimized software implementation. Finally, in §6 we summarize our results, comment on future work, and indicate a current high-performance computing architecture that may benefit from this work.

2 THE INVERSIVE CONGRUENTIAL GENERATOR

The ICG is currently one of the most attractive random number generators which is based on a nonlinear recursion. ICGs were developed as nonlinear analogs of the widely used linear PRNG, the linear congruential generator (LCG). The better known LCG is based on the following first order linear modular recursion:

$$x_n = ax_{n-1} + b \pmod{m}. \qquad (1)$$

The constants in equation (1) are the modulus, m, the multiplier, a, and the additive constant, b. In common implementations, m is chosen either to be prime or a power-of-two. In these cases, one can choose the multiplier and additive constant to get a good quality LCG with the longest possible period, (Knuth 1981). Over the years, LCGs have proven to be reliable; however, they have a defect shared by all pseudorandom numbers produced by *linear* recursions: multidimensional vectors made up of from them lie on lattices that can often be covered by a small family of parallel hyperplanes, (Marsaglia 1968). The spacing between these hyperplanes, which is computed via the spectral test, (Coveyou & MacPherson 1967), usually increases as the dimension of the vectors increases. This causes problems for Monte Carlo applications such as numerical integration, as the lattices develop larger and larger "holes" with increasing dimension.

Since the hyperplane problem is generic to linear modular recursions, interest developed in finding good quality nonlinear recursions that could be used for PRNGs. One such nonlinear generator is the implicit ICG, (Eichenauer & Lehn 1986). It is obtained by applying a single, nonlinear, transformation to an LCG. The recursion for the implicit ICG (IICG) is given by:

$$x_n = a\overline{x_{n-1}} + b \pmod{m}. \qquad (2)$$

Note that the "bar" indicates the multiplicative inverse modulo m. This operation is defined via the expression $\overline{n} \times n \equiv 1 \pmod{m}$, with zero being its own inverse, $\overline{0} = 0$. Since we have to deal with computing multiplicative inverses, it is natural to work in a finite field, and so it customary to choose the modulus, m, to be prime. More recently, (Hellekalek 1995), an ICG variant, called the explicit ICG (EICG), was found to be as useful as it's implicit cousin. The EICG is defined by an explicit formula, not a recursion:

$$x_n = \overline{an + b} \pmod{m}. \qquad (3)$$

It is important to reiterate that unlike the equation (2) for the IICG, equation (3) is a formula, not a recursion. Thus, one need not store previously computed values to compute using an EICG, and one may compute the value of an EICG at an arbitrary point in its cycle as easily as computing the next EICG point.

An important fact about ICGs is that they totally avoid the hyperplane problems seen in linear PRNGs, (Eichenauer-Herrmann 1991). This is a very important fact; however, all PRNGs based on simple recursions have some defect, and ICGs have their's as well, (Leeb & Wegenkittl 1997). None the less, ICGs offer some very important properties to the Monte Carlo community; however, at present there is no widely used random number package that provides them. For example, there arc several linear generators and a single nonlinear generator[1] that are currently provided in SPRNG.

A final property of ICGs that swayed the developers of SPRNG to include them is the following theoretical result that influences explicit ICGs for use in parallel and vector computations. Let us define a family of N EICGs as follows:

$$x_n^i = \overline{a^i n + b^i} \pmod{m},\ i = 1, 2, \ldots, N. \qquad (4)$$

Then, N-tuples of the form $(x_n^1, x_n^2, \ldots, x_n^N)$ have good statistical properties if all the N numbers $b^i\overline{a^i}$ are distinct, (Niederreiter 1991). There are several strategies for producing maximal families of EICGs that ensure that the $b^i\overline{a^i}$ are distinct, and we are still experimenting with several to discover which provides empirically better random numbers in parallel for SPRNG. One such scheme is mentioned below when we discuss incorporation of EICGs into SPRNG.

[1]The nonlinear generator is a multiplicative lagged-Fibonacci generator satisfying the recursion: $x_n = x_{n-j} \times x_{n-k} \pmod{2^\ell}$.

3 PARALLELIZATION VIA PARAMETERIZATION

The SPRNG library is designed to incorporate generators that can be parameterized, (Mascagni & Srinivasan 2000). Clearly, the above theory based on EICGs with different multipliers and additive constants provides such a parameterization provided we can satisfy the requirement that all the $b^i\overline{a^i}$ are distinct. In one approach, we set $a^i = a$, a constant multiplier, and we define $b^i = a\times(i-1)+b \pmod{m}$. Clearly we obtain distinct numbers with these definitions:

$$\begin{aligned} b^i\overline{a^i} &= (a \times (i-1) + b)\overline{a} \pmod{m}, \\ &= (i-1) + b\overline{a} \pmod{m}. \end{aligned} \tag{5}$$

Note, there are many alternative parameterizations that maintain the property that $b^i\overline{a^i}$ are distinct. However, such different methods may not all provide good parallel pseudorandom numbers. Thus, before we finalize the SPRNG implementation, a considerable set of empirical tests will be undertaken to discriminate between the different alternatives.

4 MODULAR INVERSION

One of the major reasons why ICGs have not become more popular is the cost of modular inversion relative to modular multiplication. If multiplication modulo m costs a unit of computer time, then modular inversion costs $O(\log_2 m)$. This fact holds whether one uses the fact that $x^{m-2} \equiv \overline{x} \pmod{m}$ to find the multiplicative inverse, or if one uses the extended Euclidean algorithm for computing the Greatest Common Divisor (GCD), (Cormen, Leiserson, & Rivest 1990). In the extended Euclidean algorithm for the computation of the GCD for inputs n and m one obtains both the GCD and the smallest numbers, x and y, that satisfy $\gcd(n,m) = x\times n + y \times m$. In our case, we desire to compute the multiplicative inverse of n modulo m. Since we always choose m to be prime, we obtain $\gcd(n,m) = 1 = x \times n + y \times m$. Reduce both sides of this equation modulo m to obtain: $x \times n \equiv 1 \pmod{m}$, and so $x \equiv \overline{n} \pmod{m}$. In fact, the optimal algorithm for modular inversion seems to be to build a table for small values for the extended Euclidean algorithm, and then to use this table instead of computation in the final stages of the algorithm. Recall that number of iterative calls to the iterative version of the extended Euclidean algorithm grows logarithmically with the size of its inputs. This means that each iterative call reduces the inputs to the next iterative call approximately by a constant factor.[2] Thus, a sizable fraction of the total calls are done on relatively small sized inputs, making it clear that tabulating small values can lead to a considerable speed up. Such an implementation for an EICG has been written by Otmar Lendl of the pLab group, and this code serves as the basis for our own software implementation, (Lendl 1997).

5 FPGA IMPLEMENTATION

An FPGA is a very general piece of hardware with many logical gates and pathways. A user can design a rather arbitrary circuit that can then be downloaded onto the chip to permit the rapid execution of the function instantiated by the circuit. In addition, at a later time, a new circuit can be downloaded allowing for new computations. FPGAs are used in many research and industrial settings as they permit one to produce a "near hardware speed" implementation of a performance critical circuit without the cost of fabricating a special purpose chip, (Rahimi & Ali 2000). This was one of many reasons why we decided to implement modular inversion on an FPGA: to obtain faithful estimates of optimal running times for modular inversion in order to understand the optimal runtime potential of ICGs.

We implemented inversion modulo $2^{31} - 1$ on a Xilinx XC4010XL-PC84CMN9749 FPGA board. Our design was written in the Very High Speed Integrated Circuit Hardware Description Language (VHDL), and was compiled and loaded onto the Xilinx board using Xilinx Foundation Series Software version 2.6.[3] The algorithm we implemented was the binary variant of the extended Euclidean algorithm, (Knuth 1981). This variant proceeds without the division step used in the modular reduction step in the ordinary extended Euclidean algorithm. Instead of dividing the numbers, manipulations that iteratively reduce their values in appropriate ways are accomplished with bit-level shifting and addition of the numbers as represented in binary. The the binary extended Euclidean algorithm not only provides us with an efficient means of computing modular inverses, but the bit manipulation aspects of the algorithm make it much easier to implement as a digital circuit. It

[2]In this case, each call to the extended Euclidean algorithms for the GCD reduces the size of the inputs by approximately the "Golden ratio," (Cormen, Leiserson, & Rivest 1990).

[3]Space constraints within this paper prevent us from including more detailed results. The optimized software implementation, the VHDL code, a circuit schematic, and more detailed timing results are available on the web at the URL: http://www.cs.fsu.edu/~mascagni/research/FPGA.

is important to note; however, that in software, the ordinary extended Euclidean algorithm is faster than it's binary counterpart. However, implementing a circuit for dividing two numbers, which is required in the ordinary extended Euclidean algorithm required far too many logic gates than were available on our FPGA. In fact, it is our opinion that an optimal FPGA implementation of the ordinary extended Euclidean algorithm will never be a viable modular inversion competitor for an FPGA implementation of the binary extended Euclidean algorithm.

The other choice we could have made for implementing inversion on the FPGA is to implement modular multiplication, modulo m. With modular multiplication and the "square and multiply" algorithm, one could obtain a fast inversion implementation that required only $O(\log_2 m)$ modular multiplies due to the fact that $x^{m-2} \equiv \overline{x} \pmod{m}$. This is asymptotically the same cost as with the Euclidean algorithm; however, implementing modular multiplication on an FPGA poses certain problems of its own. If the prime modulus, m is fixed, then one can "hardwire" an efficient algorithm for modular multiplication that does not require integer division. For example, for moduli close to powers of two, multiplication may be implemented with only binary shifts and adds, and so in these cases an FPGA implementation will be relatively easy to design, and will be expected to be comparable in efficiency to the extended binary Euclidean algorithm FPGA implementation. However, to implement a general inversion routine using an arbitrary, but bounded, modulus requires implementing integer division. Above, we rejected using integer division due to its complexity in an FPGA design, and so we will not even attempt a comparison with this inversion method with our FPGA implementation of the extended binary Euclidean algorithm. In addition, it is important to comment that even though we chose to use a Mersenne prime modulus, $m = 2^{31} - 1$, one of the most efficient in terms of shift and add modular multiplication, as our modulus, our extended binary Euclidean algorithm implementation allows inversion with respect to any prime modulus that fits in the initial 31-bit register. Indeed, if we modified our design to include a 64-bit register for m and x, then we would have a general FPGA inversion implementation for moduli up to $2^{65} - 1$. Thus, using the extended binary Euclidean algorithm for inversion provides a general FPGA solution for the EICG for arbitrary prime moduli.

The maximal delay of the last (31st) bit of the inversion through our circuit ranged from 240 nanoseconds to 410 nanoseconds. This was compared to a range of 1.9 μseconds to 2.5 μseconds for the optimal software implementation running on a 550 megahertz Pentium III workstation running Linux. The FPGA result was obtained through direct measurement of delays in the Xilinx FPGA system; however, as yet we do not have a complete software/FPGA implementation of the full EICG available for measurement. We hope to have such a system in place in the very near future, and will then make total EICG generation timings available on our web site. Thus, the range of speedups that we see in our FPGA implementation over the optimal software implementation run from a factor of 4.6 to a factor of 10.4 over the optimal software implementation. This is already a substantial speedup; however, it is important to mention that the particular Xilinx chip that we used has a clock speed of only 24 megahertz. While it is typical that current FPGAs have clock speeds much slower than the contemporary, high-end, microprocessors, the factor is usually smaller than the approximate factor of 23 seen here. Typically, FPGAs have clock speeds a factor of 10 smaller than current, high-end, microprocessors. Thus, we expect that our current VHDL design will find a stable FPGA implementation that is a factor of 10 to 20 faster than an optimized software implementation.

6 CONCLUSIONS AND REMARKS ON FUTURE WORK

We have discussed the implementation of modular inversion on an FPGA system to improve the performance of the EICG. The overall simulated performance increase of our FPGA implementation is a factor of 4 to 10 over that of an optimal software implementation using the modulus $m = 2^{31} - 1$. Moreover, there is the expectation that with more current FPGA hardware, a factor of 10 to 20 improvement may be seen over the software performance executed on state-of-the-art microprocessors.

The motivation for our project is many-fold. First, we have an intrinsic desire to improve the EICG to make it a more acceptable PRNG for general Monte Carlo use. Second, we wish to complete the EICG's inclusion in the SPRNG library. As mentioned above, we currently have only one PRNG in SPRNG based on a nonlinear modular recursion. In addition, one of the design criteria for SPRNG is the exclusive use of parameterized generators. Another very important principle in SPRNG is that SPRNG purposely provides the user with a wide variety of qualitatively distinct PRNGs. The reason for this is due to the fact that Monte Carlo computations are numerical experiments with stochastic components that cannot be completely controlled in the sense of experimental science, and that there is no single PRNG that provides correlation free numbers to all Monte Carlo applications. Thus, with a wide variety of qualitatively distinct PRNGs a user can redo a computation to rule out empirically that an aberrant numerical result is due to an unfortunate interaction between their Monte Carlo application and their particular PRNG. Having more generators with distinct qualitative and quantitative properties provides the user with more choices for PRNG controls in their Monte Carlo experimentation. We feel that this is our most important reason for wanting to include the EICG in SPRNG.

Another motivation for the implementation of an FPGA accelerated version of the EICG is our access to very particular new high-performance computing architecture. An advanced prototype of the SRC Computers, Incorporated *SRC 6* machine, (SRC Computers 2000) is currently installed at the Center for Computational Science at the Oak Ridge National Laboratory in Tennessee, USA.[4] The most important architectural feature of the *SRC 6* machine, from our point of view, is that it has Lucent Technologies FPGA chips installed as coprocessors within its otherwise conventional distributed-memory multiprocessor design. The reasons for FPGAs being incorporated into the *SRC 6* machine are partly based on the expected customers for this machine. Since the cost of developing and testing an FPGA function is very considerable, it is expected that the typical FPGA-accelerated application will compile a complicated function that is used in almost every pass of a long inner loop. This is exactly the computational context of our EICG application. Thus, we feel that this PRNG application is very appropriate to test the capabilities of this machine.

It is important to note that even though we have developed our digital design for modular inversion on a Xilinx development system for use on Xilinx chips, the design is completely specified in VHDL. Thus, the actual design should be totally portable to the particular Lucent FPGA chips used on the SRC 6 prototype. The only issue, that we have not had the opportunity to investigate, is whether or not the spatial, logic, gate, and wire constraints of our Xilinx implementation, that will be implicitly enforced in the Xilinx development system, will be totally compatible with the development software and hardware for the Lucent FPGA.

The project described here is, in fact, part of a larger interest of one of the authors (SR) in developing the hardware and software infrastructure for FPGA acceleration in personal computers and computer workstations, (Rahimi & Ali 2000). Thus, we expect to not only provide a unified implementation of the EICG within SPRNG that can use available FPGA hardware, but we will run this version of SPRNG both on the Oak Ridge *SRC 6* machine as well as on machines that have been fitted with special FPGA accelerator boards. Besides helping popularize the EICG, we also hope to help popularize the generic use of FPGA acceleration in general scientific computing.

[4]The name, SRC Computers, is taken from the initials of the company's late founder, Seymour R. Cray.

REFERENCES

Cormen, T. H., C. E. Leiserson, & R. L. Rivest (1990). *Introduction To Algorithms.* Cambridge, MA: MIT Press.

Coveyou, R. R. & R. D. MacPherson (1967). Fourier analysis of uniform random number generators. *J. of the ACM* 14:100–119.

Eichenauer, J. & J. Lehn (1986). A nonlinear congruential pseudorandom number generator. *Statist. Hefte* 37:315–326.

Eichenauer-Herrmann, J. (1991). Inversive congruential pseudorandom numbers avoid the planes. *Math. Comput.* 56:297–301.

Hellekalek, P. (1995). Inversive pseudorandom number generators: concepts, results, and links. In C. Alexopoulos, K. Kang, W. Lilegdon, & D. Goldsman (eds.), *Proceedings of the 1995 Winter Simulation Conference*, pp. 255–262.

Knuth, D. E. (1981). *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms* (Third ed.). Reading, Massachusetts: Addison-Wesley.

Leeb, H. & S. Wegenkittl (1997). Inversive and linear congruential pseudorandom number generators in selected empirical tests*ACM Transactions on Modeling and Computer Simulation* **7**(2).

Lendl, O. (1997). URL: `http://random.mat.sbg.ac.at/ftp/pub/software/gen/`.

Marsaglia, G. (1968). Random numbers fall mainly in the planes. *Proc. Nat. Acad. Sci. U.S.A.* 62:25–28.

Mascagni, M., D. Ceperley, L. Mitas, F. Saied, & A. Srinivasan (1998). URL: `http://sprng.cs.fsu.edu`.

Mascagni, M. & A. Srinivasan (2000). SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*. In the press.

Niederreiter, H. (1991). Recent trends in random number and random vector generation. *Ann. Oper. Res.* 31:323–346.

Rahimi, S. & D. Ali (2000). Asal architecture: A combination of a processor with a reconfigurable co-processor. *Journal of Mathematical Modeling and Scientific Computing* 10.

SRC Computers, I. (2000). `http://www.srccomp.com`.