Pergamon

Computers Math. Applic. Vol. 31, No. 3, pp. 33–42, 1996
Copyright©1996 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0898-1221/96 $15.00 + 0.00

0898-1221(95)00204-9

A Parallel Poisson Generator Using Parallel Prefix

TAN-CHUN LU, YU-SONG HOU AND RONG-JAYE CHEN*
Department of Computer Science and Information Engineering
National Chiao Tung University
1001 TA Hsueh Road
Hsinchu 300, Taiwan, R.O.C.
`rjchen@csie.nctu.edu.tw`

(Received May 1994; revised and accepted May 1995)

Abstract—In this paper, we use the renewal theory to develop a Poisson random number algorithm without restart. A parallel Poisson random number generator is designed based on this algorithm and prefix computation. This generator iteratively produces m Poisson random numbers with mean μ in average time complexity $O(\lceil m\mu/n\rceil f(n,p))$ on EREW PRAM, where $f(n,p)$ is the time for computing an n-element parallel prefix on p processors in each iteration, assuming that parallel uniform random numbers can be generated at the rate of one number per unit time per processor. If n is selected near $m\mu$, it achieves linear speedup when p is small and the average time complexity is $O(\log(m\mu))$ when p is $O(m\mu)$.

Keywords—Random number generator, Poisson distribution, Parallel prefix computation.

1. INTRODUCTION

Recently, many uniform random number generators have been presented for parallel machines. Matteis and Pagnutti partitioned a single pseudo-random sequence among the available processors in [1] and used one linear congruential generator for each processor in [2]. Deák [3] implemented General Feedback Shift Register (GFSR) for the Connection machine and the T series, while Aluru *et al.* [4] used GFSR on parallel computers where the number of processors is a power of two.

Poisson distribution represents the number of occurrences per unit time, and an event can occur at any instant of time. For example, the number of alpha particles emitted by a radioactive substance in a single second, or the number of arrivals in an $M/M/1$ Queue in a single second has a Poisson distribution. Knuth [5] introduced a simple stochastic model to generate Poisson random variables using exponential random numbers.

In this paper, we present a new algorithm to generate Poisson random numbers, based on uniform numbers. In Section 2, we describe this stochastic model, and extend it to generate many Poisson random numbers. In Section 3, we introduce a parallel Poisson random number generator. Finally, Section 4 concludes the paper with a note on the possible parallel models of the Poisson random number generator.

The authors wish to thank the referee for his detailed and valuable comments on the paper.
*Author to whom all correspondence should be sent.

2. POISSON GENERATORS

2.1. A Simple Poisson Generator

A Poisson random variable Z has probability density

$$f_Z(z) = \frac{\mu^z e^{-\mu}}{z!}, \qquad z = 0, 1 \dots$$

and represents the number of events per unit time in a Poisson process with mean μ. The interarrival time, X_i, is an exponential distribution with mean $1/\mu$. Thus, Z is the number of complete and independent exponential random variables X_i that can be fitted into a unit time interval. The following notation is used through this paper.

DEFINITION 1.

Z_i: *Poisson distributed random variable with mean* μ;

X_i: *exponentially distributed random variable with mean* $1/\mu$;

$S_i = \sum_{j=1}^{i} X_j$, $S_0 = 0$;

$N(t) = \max n \mid S_n \le t$, $N(0) = 0$;

R_i: *uniformly distributed random variable between 0 and 1.*

The relationship between exponential distribution and Poisson distribution is illustrated in Figure 1.

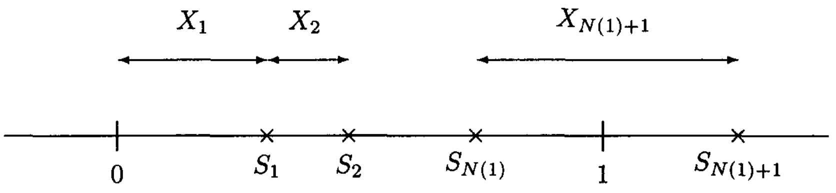

Figure 1. The relationship between exponential distribution and Poisson distribution.

According to this relationship, we can produce a Poisson variable Z by first generating independent exponential random variables $X_1, X_2, \dots$ with mean $1/\mu$, stopping generating as soon as $X_1 + X_2 + \cdots + X_m \ge 1$, and then $Z = m - 1$. The random variable $X_1 + X_2 + \cdots + X_m$ is a gamma random variable of order m. The probability that $X_1 + X_2 + \cdots + X_m \ge 1$ is actually $\int_\mu^\infty t^{m-1} e^{-t}\, dt/(m-1)!$, and the probability that $Z = n$ is

$$\frac{1}{n!}\int_\mu^\infty t^n e^{-t}\, dt - \frac{1}{(n-1)!}\int_\mu^\infty t^{n-1} e^{-t}\, dt = e^{-\mu}\frac{\mu^n}{n!}, \qquad n \ge 0.$$

This generating method is summed up in the following theorem [5].

THEOREM 1. *Let X_i be an exponential random variable with mean $1/\mu$, and m is the largest integer satisfying $X_1 + X_2 + \cdots + X_m \le 1$. Then $Z = m$ is a Poisson random variable with mean μ.*

An exponential random variable, X_i, can be obtained by the inversion method [6], which uses a uniform random variable R_i,

$$X_i = F_R^{-1}(R_i) = -\frac{1}{\mu}\ln(1 - R_i),$$

where $F_R(x) = \int_0^x \lambda e^{-\lambda t}\, dt = 1 - e^{-\lambda x}$ and $1 - R_i$ is also a uniform random number in $[0, 1)$, so we take $X_i = -(1/\mu)\ln(R_i)$.

From Theorem 1, given a sequence of random numbers $\{R_i\}$, we require the largest integer Z to satisfy

$$-\frac{(\ln R_1 + \ln R_2 + \cdots + \ln R_Z)}{\mu} \le 1 \quad \text{or} \quad R_1 * R_2 * \cdots * R_Z \ge e^{-\mu},$$

so we obtain the following corollary [5].

COROLLARY 1. *Let $\{R_i\}$ be a sequence of uniform random variables, Z be the largest integer to satisfy*

$$\sum_{i=1}^{z} -\frac{1}{\mu} \ln R_i \le 1 \quad \text{or} \quad \prod_{i=1}^{z} R_i \ge e^{-\mu}.$$

Then Z is a Poisson random variable with mean μ.

Using Corollary 1, we come up with a simple algorithm to generate m Poisson random variables as follows.

PROGRAM 1.

```
PROCEDURE Sequential_Poisson_with_restart($m, \mu$)
/* Procedure $Uniform(R)$ generates a randomnumber $R\tilde{}U(0,1)$ */
   FOR $I := 1$ TO $m$ DO
      $Z := 0; RR := 1;$
      WHILE ( $RR \ge e^{-\mu}$ ) DO
         $Uniform(R);$
         $RR := RR * R;$
         $Z := Z + 1$
      END WHILE
      OUTPUT( $Z - 1$ )
   END FOR
END PROCEDURE
```

This method requires generating a mean of $m(\mu + 1)$ uniform random numbers for m Poisson random variables. For large μ, it will not be efficient in time.

2.2. A New Generator without Restart

Consider two consecutive sequences of the generation of Poisson random variables, Z_1 and Z_2. By the method in 2.1, we require Z_1+1 independent uniform random numbers to generate Poisson random variable Z_1 and another Z_2+1 independent uniform random numbers to generate Poisson random variable Z_2. Indexing these two sequences of uniform random numbers together, we need $Z_1 + Z_2 + 2$ independent uniform random variables to generate Poisson random variables Z_1 and Z_2 (see Figure 2).

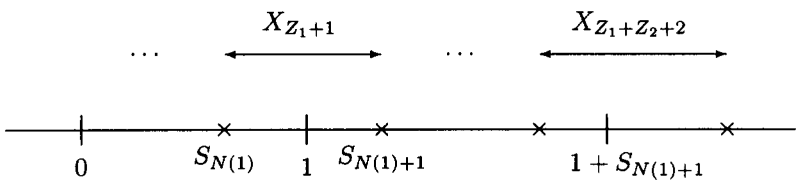

Figure 2. Two consecutive sequences of the generation of Poisson random variables Z_1 and Z_2.

X_{Z_1+1} is the interarrival which crosses the time 1. Can we use it as an exponential random number to generate Z_2? The answer is NO because Z_1, X_{Z_1+1} is not independent by

$$\Pr\left(X_{Z_1+1} > 1 - \sum_{i=1}^{Z_1} X_i\right) = 1.$$

Furthermore, the interarrival time X_{Z_1+1} is selected by the inspection of time 1. The distribution of X_{Z_1+1} is not exponential anymore. This is an inspection paradox in renewal theory [6]. However, we can still use X_{Z_1+1} tactically to generate the next Poisson random number as discussed below. We need the following definition before the discussion.

DEFINITION 2.

$A(1)$: the time from the $N(1)^{\text{th}}$ arrival to time 1;
$E(1)$: the time between 1 and the $(N(1)+1)^{\text{th}}$ arrival.

In Figure 3, $X_{Z_1+1} = X_{N(1)+1} = A(1) + E(1)$ where

$$A(1) = 1 - \sum_{i=1}^{N(1)} X_i = 1 - S_{N(1)},$$
$$E(1) = \sum_{i=1}^{N(1)+1} X_i - 1 = S_{N(1)+1} - 1.$$

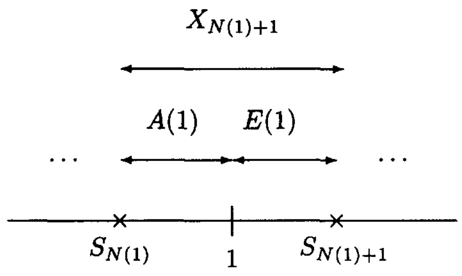

Figure 3. The interarrival time crosses the time 1.

Since X_i is the interarrival time with exponential distribution, $N(t)$, $t \geq 0$ is a Poisson process. From the memoryless property of Poisson process, the time from $t = 1$ to the next arrival is exponentially distributed and is independent of all that has previously occurred, so $A(1)$ and $E(1)$ are independent, and $E(1)$ is an exponential random variable with mean $1/\mu$. Since $E(1)$ is an exponential random variable with mean $1/\mu$, we can use $E(1)$ to generate Z_2 (see Figure 4).

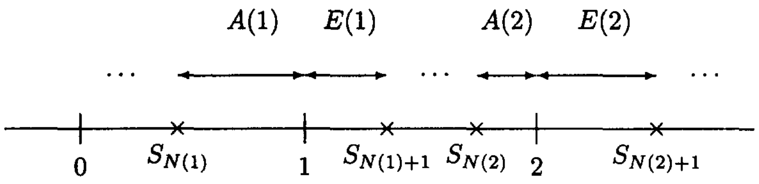

Figure 4. The memoryless property in interarrival time.

In turn, we can prove $E(n)$ is an exponential random variable with mean $1/\mu$, and it can be used to generate Poisson random variable Z_{n+1}, so we have the following theorem.

THEOREM 2. *Let*

$$E(n) = S_{N(n)+1} - n, \qquad n = 1, 2 \dots,$$
$$A(n) = n - S_{N(n)}, \qquad n = 1, 2 \dots;$$

then

1. *$E(n)$ is exponential distribution with mean $1/\mu$;*
2. *$A(n)$ and $E(n)$ are independent.*

From Theorem 2, we can improve the sequential algorithm in Section 2.1 to get $Z_1, \dots, Z_m$ in a sequence of uniform random variables $R_1, R_2, \dots, R_j$ without restart, where $j = \min\{k \mid R_1 * R_2 * \cdots * R_k \leq e^{-m\mu}\}$. The resulting algorithm is presented below.

Program 2.

```
PROCEDURE Sequential_Poisson_without_restart( $m, \mu$ )
/* Procedure $Uniform(R)$ generates a randomnumber $R\tilde{}U(0,1)$ */
    $RR := 1; Z := 0; T := e^{-\mu}; count := 0;$
    WHILE ($count < m$) DO
        WHILE ( $RR > T$) DO
            $Uniform(R);$
            $RR := RR * R;$
            $Z := Z + 1;$
        END WHILE
        WHILE ( $RR \leq T$) DO
            OUTPUT( $Z - 1$ );
            $RR := RR/T;$
            $Z := 1;$ /* From Theorem 2, $RR/T$ is a new random variable */
            $count := count + 1;$
        END WHILE
    END WHILE
END PROCEDURE
```

By Theorem 2, $E(n)$ is an exponential distribution with mean $1/\mu$, and so $N(n) - N(n-1)$ is the number of the exponential arrivals between time $n-1$ and n (see Figure 5). Thus, $Z_n = N(n) - N(n-1)$ is a Poisson distribution random variable with mean μ ($N(0) = 0$ by definition), so the following corollary is obtained.

Corollary 2. *Let Z_n be the nth Poisson random variable we generated; then $Z_n = N(n) - N(n-1)$, $n = 1, \dots, m$.*

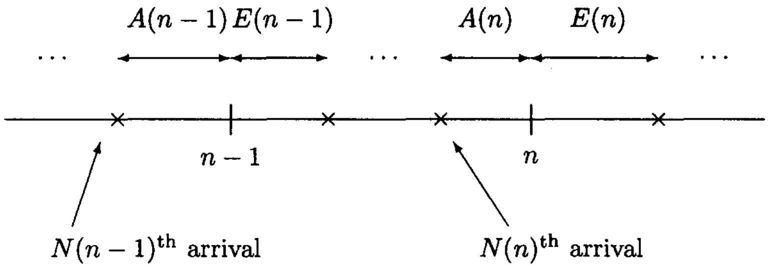

Figure 5. $E(n)$ is an exponential distribution random variable with mean $1/\mu$.

We therefore can compute $S_k = \sum_{i=1}^{k} -(1/\mu) \ln R_i$, $k = 1, \dots j$. If $i - 1 < S_k \leq i$, the kth arrival is in the time interval $(i-1, i]$. Counting the arrivals in each interval $(i-1, i]$, we can get the Poisson random variable Z_i. We develop the following algorithm using this method to get m Poisson random variables. And the parallelization of the algorithm is discussed in the next section.

Program 3.

PROCEDURE Pseudo_code_of_Poisson(m, μ)

Step 1: Generate "sufficient" uniform randomnumbers such that m Poisson numbers can be generated by using them. Assume them to be $R_1, \dots, R_j$.

Step 2: Compute $X_1, \dots, X_j$ where $X_i = -(1/\mu) \ln R_i$.

Step 3: Compute $S_k = \sum_{i=1}^{k} X_i$ for $k = 1, \dots, j$.

Step 4: Partition $\{S_k \mid k = 1, \dots, j\}$ to $\cup_{i=1}^{m} T_i$ where $T_i = \{S_k \mid i-1 < S_k \le i\}$, the cardinality of T_i is the number of arrivals in the time segment $(i-1, i]$.

Step 5: Count the cardinality of T_i for $i = 1, \dots, m$, and output them.

END PROCEDURE

3. A PARALLEL POISSON GENERATOR

We develop a parallelization for the algorithm in Program 3 on EREW PRAM model. We are concerned with the following key problems in parallelization:

1. How to generate "sufficient" uniform random numbers in parallel (Step 1).
2. How to compute $S_k = \sum_{i=1}^{k} X_i$ for $k = 1, \dots, j$ concurrently (Step 3).
3. How to compute the cardinality of T_i for $i = 1, \dots, m$ effectively (Step 5).

These three problems will be dealt with in the following sections one by one.

3.1. A Parallel Uniform Random Number Generator

In a parallel computing model, a parallel uniform random number generator (PURNG for short) is to produce a sequence of random real numbers in $[0, 1)$ on each processor. On an EREW PRAM, a common memory can be accessed by each processor. We can implement the GFSR algorithm presented in [7] on an EREW PRAM directly as follows.

The GFSR algorithm consists of two phases: the initialization phase and the random generating phase. In the first phase, a table is constructed in the common memory. In the second phase, each processor reads two words from the table, and then makes an XOR operation to obtain a uniform random number.

When the number of processors is a power of two, the GFSR algorithm can be implemented without the common memory [4]. It can avoid communication costs.

The total number of "sufficient" uniform random numbers has no upper bound. We generate and process these random numbers iteratively. The algorithm generates n uniform random numbers in each iteration with each processor sharing n/p numbers and stops when $S_{kn} > m$ after the k^{th} iteration. For the i^{th} iteration, $X_{(i-1)n+1}, \dots, X_{i \times n}$ are computed and the prefix computation is applied to obtain $S_{(i-1)n+1}, \dots, S_{i \times n}$ and the cardinalities of $T_{i'}$'s (the cardinality of $T_{i'}$ is the number of arrivals in the time segment $(i'-1, i')$). In Sections 3.2 and 3.3, we describe how to compute them in parallel.

3.2. The Parallel Prefix Algorithm

We use the parallel prefix computation to compute $S_k = \sum_{i=1}^{k} X_i$ for $k = 1, \dots, n$ concurrently. Given $X_1, \dots, X_n$, the prefix computation problem is to evaluate all products $X_1 \otimes X_2 \otimes \cdots \otimes X_i$, for $i = 1, \dots, n$, with an associative operation $\otimes$. There have been a few parallel algorithms to solve this problem under different architectures. Kruskal *et al.* [8] developed a parallel prefix algorithm on an EREW PRAM model with time complexity $O(n/p + \log p)$ where p is the number of processors. Dekel *et al.* [9] and Chen *et al.* [10] designed $O(\log n)$ algorithms on an n-leaf complete binary tree model. Moreover, Eğecioğlu *et al.* [11] presented an $O(n/p + \log p)$ algorithm in hypercube, and Chen *et al.* [12] designed an $O(n^{1/9})$ algorithm on two-dimensional mesh-connected computers with multiple broadcasting (2-MCCMB's).

The prefix algorithm in [8] is adopted to construct our Poisson number generator. Accordingly, the computation model and the syntax style to specify the generator are similar to [8]. The following syntax appears frequently in our algorithm:

```
FORALL $i \in A$ DO IN PARALLEL
          BODY.
```

This syntax indicates that the **BODY** is executed once for each processor i in the index set A.

3.3. Counting the Cardinalities of T_i's in Parallel

To count the total number of elements in each segment T_i, we define an operator $\oplus$ as follows.

DEFINITION 3.

$$(a,b) \oplus (c,d) = \begin{cases} (a, b+d) & \textit{if } a = c, \\ (c,d) & \textit{if } a < c, \\ (a,b) & \textit{if } a > c. \end{cases}$$

It is easy to check whether $\oplus$ is associative. By the definition of T_i, we have $s = i$ for s in $(i-1, i]$. We define an array TT with $TT[k] = S_k$ for $k = 1, \dots, n$; then the frequency that i appears in array TT is the cardinality of T_i. Since TT is nondecreasing, $a \le c$ is always true. Define $W[i]$ to be $(TT[i], 1)$, for $i = 1, \dots, n$, and call the procedure Parallel_Prefix_$\oplus$ to compute all prefixes of array W and store them in array U. The second coordinate of the last element in each corresponding segment of array U represents the total number of elements in this segment. The above is summarized in the following theorem.

THEOREM 3. *If TT is nondecreasing and $TT[k] \neq TT[k+1]$, i.e., $TT[k]$ is the last element of the segment $T_{TT[k]}$, then*

$$(TT[1], 1) \oplus (TT[2], 1) \oplus \cdots \oplus (TT[k], 1) = \left(TT[k], \textit{the cardinality of } T_{TT[k]}\right).$$

EXAMPLE 1. Assume

$$\{S_k \mid k = 1, \dots, 10\} = \{0.1, 0.3, 0.6, 0.8, 1.2, 1.3, 2.5, 2.8, 2.9, 3.1\};$$

then

$$\begin{aligned} T_1 &= \{0.1, 0.3, 0.6, 0.8\}, \\ T_2 &= \{1.2, 1.3\}, \\ T_3 &= \{2.5, 2.8, 2.9\}, \quad \text{and} \\ T_4 &= \{3.1\}. \end{aligned}$$

Therefore,

$$\begin{aligned} TT[1 \dots 10] &= [1,1,1,1,2,2,3,3,3,4], \quad \text{and} \\ W[1 \dots 10] &= [(1,1),(1,1),(1,1),(1,1),(2,1),(2,1),(3,1),(3,1),(3,1),(4,1)]. \end{aligned}$$

After prefix computation,

$$U[1 \dots 10] = [(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(3,1),(3,2),(3,3),(4,1)].$$

The second coordinates of $U[4]$, $U[6]$, $U[9]$, and $U[10]$ are 4, 2, 3, and 1, respectively, which are the cardinalities of T_1, T_2, T_3, and T_4.

3.4. The Parallel Algorithm

The parallel algorithm developed from Sections 3.1–3.3 is shown in Program 4. First, each processor generates n/p uniform random numbers, stores them in array R, and computes array X by assigning $-(1/\mu)\ln R[i]$ to $X[i]$. Next, accumulate the $S[n]$ value (variable *oldS*) in the last iteration, and call the procedure Parallel_Prefix_+ to compute prefix sums (will be stored in array S) in array X based on addition operation.

We then compute array TT and W, which were discussed in Section 3.3. Before the procedure Parallel_Prefix_$\oplus$ is called, the U value is accumulated in the last iteration (variable *oldU*). In the next step, we will look for the last element of each segment. This can be done by checking whether the TT value of an element differs from its next element. We then use array V to denote these last elements, which are finally output, and save the residual values in this iteration.

PROGRAM 4. A parallel Poisson number generator.

```
PROCEDURE Parallel_Poisson($m$, $\mu$)
/* The procedure produces $m$ Poisson numbers with mean $\mu$ */
/* Procedure Init_Uniform() constructs the initial table that is used in GFSR algorithm. Procedure Uniform($R$) generates a random number $R \sim U(0,1)$; Procedure Parallel_Prefix_$\odot(n,p,X,S)$ compute $S[k] = \sum_{i=1}^{k} X[i]$ for $k = 1,\dots,n$ under operator $\odot$ on $p$ processors EREW PRAM. */
/* Initialization */
  Init_Uniform();
  $oldS := 0$;
  $a := n/p$;
/* Begin iteration */
  WHILE ($oldS \le m$) DO
/* Step 1 and Step 2 */
    FORALL $i \in \{1,\dots,p\}$ DO IN PARALLEL
      FOR $j := 1$ TO $a$ DO
        Uniform($R[i]$);
        $X[(i-1)*a+j] := -\ln R[i]/\mu$
      END FOR
    END FORALL
/* Accumulate the $S$ value in the last iteration */
    $X[1] := X[1] + oldS$;
/* Step 3 */
    Parallel_Prefix_+$(n,p,X,S)$;
/* Output or accumulate the $U$ value in the last iteration */
    FORALL $i \in \{1,\dots,p\}$ DO IN PARALLEL
      FOR $j := 1$ TO $a$ DO
        $TT[(i-1)*a+j] := S[(i-1)*a+j]$;
        $W[(i-1)*a+j] := (TT[(i-1)*a+j], 1)$
      END FOR
    END FORALL
    IF ($oldS \ne 0$ and $oldTT \ne TT[1]$) THEN
      OUTPUT($oldU$)
    END IF
    IF ($oldS \ne 0$ and $oldTT = TT[1]$) THEN
      $W[1] := (TT[1], 1+$ second_coordinate of $oldU)$
    END IF
/* Step 5 */
    Parallel_Prefix_$\oplus(n,p,W,U)$;
    FORALL $i \in \{1,\dots,p\}$ DO IN PARALLEL
```

```
        FOR $j := 1$ TO $a$ DO
          $V[(i-1)*a+j] := 0$
        END FOR
      END FORALL
/* Find the last element of each segment */
      FORALL $i \in \{1,\ldots,p\}$ DO IN PARALLEL
        FOR $j := 1$ TO $a-1$ DO
          IF $(TT[(i-1)*a+j] \neq TT[(i-1)*a+j+1])$ THEN
            $V[(i-1)*a+j] := 1$
          END IF
        END FOR
      END FORALL
      FORALL $i \in \{1,\ldots,p-1\}$ DO IN PARALLEL
        IF $(TT[i*a] \neq TT[i*a+1])$ THEN
          $V[i*a] := 1$
        END IF
      END FORALL
/* Output U */
      FOR $j := 1$ TO $a$ DO
        FORALL $i \in \{1,\ldots,p\}$ DO IN PARALLEL
          IF $(V[(j-1)*p+i] = 1$ and $TT[(j-1)*p+i] \leq m)$ THEN
            OUTPUT($U[(j-1)*p+i]$)
          END IF
        END FORALL
      END FOR
/* Save values in this iteration */
      $oldS := X[n]$;
      $oldTT := TT[n]$;
      $oldU := U[n]$
    END WHILE
END PROCEDURE
```

The time complexity of the algorithm is not deterministic. The algorithm uses $O(1)$ preprocessing time if the initial time of the parallel uniform random number generator is not considered. In average, to generate m Poisson numbers needs $m\mu$ uniform random numbers and $\lceil m\mu/n \rceil$ iterations. Since each iteration takes $O(n/p + \log p)$ time, the average time complexity of the parallel Poisson algorithm is $O(\lceil m\mu/n \rceil (n/p + \log p))$. It is not time-effective when $n > m\mu$ because it is worse than the case where $n = m\mu$. If n is near $m\mu$, the time complexity is $O(m\mu/p + \log p)$. This algorithm achieves linear speedup for small p (the sequential algorithm takes $O(m\mu)$ time in average), and takes $O(\log m\mu)$ time where $p = O(m\mu)$. The space complexity is clearly $O(n)$.

EXAMPLE 2. Given $m = 6$, $\mu = 10/3$, $n = 12$. The following table shows the process of this algorithm:

R	X	S	TT	U	V
0.010559	1.365222	1.365222	2	$(2,1)$	1
0.003967	1.658893	3.024115	4	$(4,1)$	0
0.335154	0.327949	3.352064	4	$(4,2)$	1
0.033265	1.020973	4.373037	5	$(5,1)$	0
0.355724	0.310080	4.683117	5	$(5,2)$	1
0.217200	0.458081	5.141198	6	$(6,1)$	0
0.536973	0.186542	5.327740	6	$(6,2)$	0

The output sequence is $(2,1),(4,2),(5,2),(6,5)$. It represents the Poisson number sequence: $0,1,0,2,2,5$.

4. CONCLUSION

In this paper, we combine the memoryless property of Poisson process with the prefix computation to parallelize a simple sequential Poisson number generator. It seems to be hard to develop parallel generators without the memoryless property.

The parallel algorithm developed in this paper is based on EREW PRAM, and we can easily modify the generator on popular architectures such as the binary tree, hypercube and 2-MCCMB.

REFERENCES

1. A. De Matteis and S. Pagnutti, Parallelization of random number generators and long-range correlations, *Numer. Math.* **53**, 595–608 (1988).
2. A. De Matteis and S. Pagnutti, A class of parallel random number generators, *Parall. Comput.* **13** (2), 193–198 (1990).
3. I. De, Uniform random number generators for parallel computers, *Parall. Comput.* **15** (1–3), 155–164 (1990).
4. S. Aluru, G.M. Prabhu and J. Gustafson, A random number generator for parallel computers, *Parall. Comput.* **18** (8), 839–847 (1992).
5. D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd edition, Addison-Wesley, New York, (1981).
6. S.M. Ross, *Introduction to Probability Models*, Academic Press, London, (1985).
7. T.G. Lewis and W.H. Payne, Generalized feedback shift register pseudorandom number algorithm, *J. ACM* **20** (3), 456–468 (1973).
8. C.P. Kruskal, L. Rudolph and M. Snir, The power of parallel prefix, *IEEE Tran. Comput.* **C-34** (10), 965–968 (1985).
9. E. Dekel and S. Sahni, Binary trees and parallel scheduling algorithms, *IEEE Tran. Comput.* **C-32** (3), 307–315 (1983).
10. R.J. Chen and Y.S. Hou, Non-associative parallel prefix computation, *Inform. Process. Lett.* **44** (2), 91–94 (1992).
11. Ö. Eğecioğlu, Ç.K. Koç and A.J. Laub, A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors, *J. Comput. Applic. Math.* **27** (1/2), 95–108 (1989).
12. Y.C. Chen, W.T. Chen, G.H. Chen and J.P. Sheu, Designing efficient parallel algorithms on mesh-connected computers with multiple broadcasting, *IEEE Tran. Parall. Distrib. Sys.* **1** (2), 241–246 (1990).