JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **45,** 1–12 (1997)
ARTICLE NO. PC971363

Lagged Fibonacci Random Number Generators for Distributed Memory Parallel Computers

Srinivas Aluru[1]

Department of Computer Science, New Mexico State University, Las Cruces, New Mexico 88003-8001

To parallelize applications that require the use of random numbers, an efficient and good quality parallel random number generator is required. In this paper, we study the parallelization of lagged Fibonacci generators for distributed memory parallel computers. Two popular ways of generating a random sequence in parallel are studied: the contiguous subsequence technique and the leapfrog technique. We present a parallelization of the lagged Fibonacci plus/minus generators using the contiguous subsequence technique. For the leapfrog technique, we show that lagged Fibonacci generators with the exclusive or operator can be efficiently parallelized without any communication overhead when the number of processors is a power of 2. We also show that it is not possible to parallelize other lagged Fibonacci generators efficiently in a communication-free manner. We then present an efficient scalable parallelization of lagged Fibonacci plus/minus generators that uses communication. We discuss issues that arise in implementations of the proposed algorithms and comment on their practical efficiency. © 1997 Academic Press

1. INTRODUCTION

A number of computer applications require the use of random numbers. Techniques for the production of uniform random sequences on uniprocessors have been studied in great depth. A handful of methods are acknowledged to be appropriate sources of random numbers: linear congruential generators, lagged Fibonacci generators, combination generators, and more recently, the add with carry and subtract with borrow generators [28]. A good review of random number generators can be found in [5, 23].

With the advent of parallel architectures, the need has arisen to generate parallel streams of random numbers to feed applications that run in parallel. Two schemes have gained popularity as acceptable ways of producing parallel streams of random numbers using a given sequential generator: The contiguous subsequence technique and the leapfrog technique [6, 15]. It is easy to efficiently parallelize linear congruential generators using either of the two schemes [6]. Unfortunately, linear congruential generators exhibit regularities that make them unsuitable for certain kinds of Monte Carlo applications [24–26]. Deak [11] parallelizes lagged Fibonacci exclusive or generators using the contiguous subsequence technique. Using the leapfrog technique, Aluru *et al.* parallelize lagged Fibonacci exclusive or generators when the number of processors is a power of 2 [4].

In this paper, we investigate the parallelization of lagged Fibonacci generators without any assumption on the operator or the number of processors. A sequential random number generator generates a random number in $O(1)$ time. In theory, a parallel random number generator is efficient if it generates a random number on each processor in $O(1)$ time. For the generator to be practically efficient, the rate at which random numbers are generated on each processor should be close to the rate of the sequential generator. Due to the huge disparity in the times for unit communication and unit computation, it is important to eliminate or minimize communication required to implement a parallel random number generator in order to achieve practical efficiency. We first consider communication-free parallelization of lagged Fibonacci generators using the leapfrog technique. We show that such a parallelization is not possible except in the particular case when the operator is exclusive or and the number of processors is a power of 2. Since the way processors are connected is irrelevant for a communication-free algorithm, the result is valid for any model of parallel computation. We then present an algorithm using communication to parallelize lagged Fibonacci generators with + or −. Our algorithm is shown to be efficient for hypercubes and permutation networks. We also parallelize lagged Fibonacci + or − generators using the contiguous subsequence technique. We thoroughly discuss issues that arise in practical implementations of the algorithms described and comment on their practical efficiency.

The rest of the paper is organized as follows: In Section 2, we discuss the requirements of a parallel random number generator and describe the contiguous subsequence technique and the leapfrog technique. We discuss the issues involved in satisfactory parallelization using each of the schemes and outline how a linear congruential generator is parallelized. Section 3 contains a brief description of the lagged Fibonacci generators. In Section 4, we establish some properties of binomial coefficients modulo 2, to be used later. In Section 5, we ex-

[1]E-mail: aluru@cs.nmsu.edu.

0743-7315/97 $25.00

plore the relationship between binomial coefficients and lagged Fibonacci generators and study the parallelization of lagged Fibonacci generators using the leapfrog technique. We derive the parallelization of lagged Fibonacci with exclusive or (when the number of processor is a power of 2) as a special case of a more general equation and show that the generators cannot be parallelized efficiently in a communication-free manner in any other case. In Section 6, we study the parallelization of lagged Fibonacci generators using the leapfrog technique and allowing communication. We also study parallelization using the contiguous subsequence technique. Section 7 concludes the paper.

2. ISSUES IN PARALLEL RANDOM NUMBER GENERATION

A sequential random number generator is expected to have good randomness properties and a constant generation time per random number. The random number generator is started by specifying the initial state, called the "seed." Since a random number generator is a deterministic method, two runs of an application with the same seed produce the same result, ensuring the reproducibility of the results of applications. The same requirements naturally extend to a parallel random number generator. The following requirements are desirable for a parallel random number generator:

- The sequence generated on each processor should have good randomness properties.
- The sequences generated on any pair of processors should be free of mutual correlations [31].
- The generator should work for an arbitrary number of processors.
- The result of the application should be reproducible, irrespective of the number of processors on which it is run.
- Each processor should be able to generate its sequence independent of the other processors; i.e., there should be no communication.
- Generation time per random number on each processor should be constant, preferably the same as required for the generator on one processor.
- The amount of memory required per processor should be constant, preferably the same as required for the generator on one processor.

The small number of good sequential generators precludes the possibility of using a separate generator per processor. We are thus forced to split the sequence of random numbers generated by a sequential generator (referred to as the "original sequence" from here on) among the many parallel processors. We cannot simply use the same generator and use a different seed for each processor. This may cause significant overlap in the sequences generated depending upon the initial choice of seeds. Thus, it is important to allocate disjoint subsequences of the original sequence to the different processors. The choice of the subsequences should be such that each subsequence can be generated efficiently. In the following, we describe two popular strategies for performing such an allocation.

2.1. Contiguous Subsequence Technique

An obvious way to allocate subsequences of the original sequence to processors is to let each be a contiguous subsequence of the original sequence. Let $x_0, x_1, x_2, \ldots$ be the sequence of random numbers generated by a sequential generator. If L is the length of the subsequence allotted to each processor, processor i in this scheme should generate

$$x_{iL}, x_{iL+1}, x_{iL+2}, \ldots, x_{(i+1)L-1}.$$

Given the required number of initial elements as seed, each processor can then compute its subsequence independent of other processors. Clearly, L should be larger than the number of random numbers per processor required by the application consuming these numbers. Otherwise, two subsequences generated on different processors overlap.

In order to parallelize a generator using the contiguous subsequence technique, we should find an efficient way of generating the seeds required for the parallel generator from the seed of the sequential generator. Once this is done, each processor can generate its subsequence independent of other processors. This is the advantage in the contiguous subsequence technique. There are several disadvantages as well: It may not be possible to compute a priori the number of random numbers required on each processor. In such a case, a large value of L should be chosen so that the application is guaranteed not to use more than L numbers on any processor. This would mean that the random numbers used would depend on the number of processors on which the application is run. A second disadvantage is that it is impossible to write code in such a way that the result of the application is independent of the number of processors on which it is run, even if the number of random numbers required per processor can be predicted in advance.

As an example of parallelizing a generator using this technique, consider the linear congruential generator given by $x_k = (ax_{k-1} + b) \bmod m$. By recursive application of this equation, it can easily be seen [19] that $x_k = (A_l x_{k-l} + B_l) \bmod m$, where $A_l = a^l \bmod m$ and $B_l = b(a^l - 1)/(a - 1) \bmod m$. Using this, we can compute x_{iL} as $(A_{iL}x_0 + B_{iL}) \bmod m$. A_{iL} and B_{iL} can be found in $O(\log iL)$ time.

2.2. Leapfrog Technique

The leapfrog technique prescribes a way of splitting the original sequence with a view towards preserving the reproducibility and randomness properties. Let $x_0, x_1, x_2, \ldots$ be the sequence of random numbers generated by a sequential generator. In the leapfrog technique, processor i generates every Nth number in the sequence starting at x_i, where

N is the number of processors: i.e., processor i generates x_i, x_{i+N}, x_{i+2N}, ... Since each processor skips (leaps) over N numbers of the original sequence, the method is called the leapfrog technique. It is easily seen that the multiple streams generated are nonoverlapping and together generate consecutive terms of the original sequence. This leaves the possibility for the user to write code in such a way that the results are identical with uniprocessor implementation and are independent of the number of processors used. This would also ensure that the arguments to support the quality of the sequential generator carry over to the parallel domain as well.

It is not straightforward to implement a parallel random number generator using the leapfrog technique and preserving all the requirements outlined before. For example, consider the linear congruential generator given by $x_k = (ax_{k-1} + b) \bmod m$, where x_0 is the seed and x_k is the kth random number. On an N-processor system, the processor computing x_k has to wait for the computation of x_{k-1}, which has to wait for the computation of x_{k-2}, and so on, essentially reducing it to a sequential generator. To satisfy the no-communication requirement, a way of computing x_k using only x_{k-mN} (the random numbers previously generated on the same processor) should be found. Furthermore, the generation of x_k should use only a constant number of operations.

Since a random number generator is an equation describing the computation of the kth random number x_k, recursive application of the same equation should be studied in order to describe all possible ways of computing x_k from previous numbers. If a way of computing x_k is possible using a constant number of operations and random numbers previously generated on the same processor, we can use it to parallelize the generator with the leapfrog technique.

As an example, consider the linear congruential generator, specified by the equation $x_k = (ax_{k-1} + b) \bmod m$. By recursive application of this equation, it can easily be seen [19] that $x_k = (Ax_{k-l} + B) \bmod m$, where $A = a^l \bmod m$ and $B = b(a^l - 1)/(a - 1) \bmod m$. By choosing $l = N$, this formula allows us to compute the next number on each processor using the current number on the same processor. Once the seeds for each processor and A and B are computed, each parallel stream can be generated at the rate of the sequential generator.

In this paper we investigate the parallelization of the lagged Fibonacci generators, attempting to satisfy the requirements outlined at the beginning of this section. Since an efficient, communication-free parallelization under the leapfrog technique satisfies all the outlined requirements, we first study such a parallelization. We show that except for the exclusive or generator when the number of processors is a power of 2, other generators cannot be parallelized in a communication-free manner. We then relax the communication restriction and present algorithms for parallelizing lagged Fibonacci generators with + or −. We present parallel generators for both the contiguous subsequence and the leapfrog techniques.

3. LAGGED FIBONACCI GENERATORS

Lagged Fibonacci generators are specified by the recurrence

$$x_k = x_{k-p} \otimes x_{k-p+q} \bmod m,$$

where $\otimes$ denotes the operation which could be any of +, −, $\times$, or $\oplus$ (exclusive or). $m = 2^l$, for generating l bit random numbers. p is called the *lag* of the generator. The seed for these generators is the first p random numbers. For the multiplicative generator, the random numbers are odd numbers mod m. For the other generators, the random numbers are integers mod m. For a detailed analysis on the quality of these generators, see [23]. We group the generators using the + or − operator under the term *additive lagged Fibonacci generators*. Thus, the equation of an additive lagged Fibonacci generator is

$$x_k = x_{k-p} \pm x_{k-p+q} \bmod m.$$

All the lagged Fibonacci generators have numerous advantages over the linear congruential generators. One distinguishing feature of these generators is their relatively large period when compared to the linear congruential generators. The maximum attainable period depends upon the particular operation used and is given in Table I. The period can be made large by choosing an appropriately large value of p, with the same word size. Therefore, the period is not limited by the word size of the machine on which the generator is implemented. In contrast, the period of linear congruential generators is 2^l for generating l-bit random numbers. Multiple occurrences of the same term within a full period are possible. This is because the sequence repeats only if p consecutive random numbers repeat. All the bits of the numbers generated are sufficiently random. Lagged Fibonacci generators with +, − and $\times$ give good results on most of the stringent statistical tests. Lagged Fibonacci generators with $\oplus$ (exclusive or) are not good for small values of p, but for large values of p (607, for example), they pass all statistical tests [23]. These generators have long been known under the name "generalized feedback shift register generators" [22]. For a thorough statistical analysis of these generators, see [29].

Linear congruential generators have been known to exhibit regularities that make them unsuitable for certain kinds of

TABLE I
Maximum Attainable Periods of Lagged Fibonacci Generators $x_k = x_{k-p} \otimes x_{k-p+q} \bmod 2^l$

Operation	Maximum attainable period
Addition, mod 2^l	$(2^p - 1)2^{l-1}$
Subtraction, mod 2^l	$(2^p - 1)2^{l-1}$
Multiplication, mod 2^l	$(2^p - 1)2^{l-3}$
Exclusive-or	$2^p - 1$

Monte Carlo applications [24–26]. Additive and multiplicative lagged Fibonacci generators give good results on statistical tests even for small values of p [23]. Example values for (p, q) include (17, 5), (31, 13), and (55, 24). For the exclusive or generator, p should be chosen as a Mersenne prime (a prime p for which $2^p - 1$ is also a prime) [11]. A list of Mersenne primes can be found in [37]. Generators based on small values of p do not perform well on the tests [23] and large values of p are recommended. Some of the suggested values include (607, 273) and (1279, 418).

Additive lagged Fibonacci generators are superior to lagged Fibonacci generators with exclusive or. They have a larger period, $(2^p - 1)2^{l-1}$, whereas the period of the exclusive-or generator is only 2^{p-1}. Thus, it is possible to choose a much smaller value of the lag to obtain the same period. This is important because the generator requires $O(p)$ memory. Furthermore, additive lagged Fibonacci generators have high quality even for small values of lag and large values of lag are required for the exclusive or generators. More recently, it has been shown that additive lagged Fibonacci generators fail certain statistical and Monte Carlo tests for small values of the lag [9, 13, 17, 35]. As a general rule, choosing a larger value of p seems to improve the quality of the generators. On the other hand, the memory requirements for the generator increase with p. This is not a serious problem in the sequential case because the memory required is linear in p and is reasonable even for large values of the lag.

4. PROPERTIES OF BINOMIAL COEFFICIENTS

Several properties of binomial coefficients modulo 2 have been known for a long time [14, 16, 18, 33, 36]. In this paper, we provide a self-contained treatment of the properties relevant to our discussion and offer independent proofs of these.

We begin with Pascal's triangle, a pictorial way of representing all binomial coefficients. The ith row of Pascal's triangle consists of all the binomial coefficients of i. A notable property of the diagram is that each number is formed by adding the two numbers above it in the previous row. This corresponds to the property that $\binom{n}{i} = \binom{n-1}{i-1} + \binom{n-1}{i}$. The first few rows of Pascal's triangle are shown in Fig. 1.

Figure 2 shows the structure obtained by representing each odd number in Pascal's triangle by a black square. The diagram between row 2^k and row 2^{k+1} consists of two copies of the diagram between row 0 and row 2^k, placed next to each other. Thus, the structure of odd numbers in the 0th row determines the entire diagram. Theorem 1 formalizes and proves this property. Since $\binom{n}{r} = \binom{n}{n-r}$, only values of $r \leq \lfloor n/2 \rfloor$ are considered.

THEOREM 1. *For all* $0 \leq l < 2^k$: $\binom{2^k+l}{i}$ *is odd,* $0 \leq i \leq \lfloor (2^k + l)/2 \rfloor \Leftrightarrow i \leq l$ *and* $\binom{l}{i}$ *is odd.*

Proof. By induction on k. If $k = 0$, $l = 0$ and both $\binom{1}{0}$ and $\binom{0}{0}$ are odd. Otherwise, for any k, consider induction on l. If $l = 0$, $\binom{2^k}{i} = (2^k/i)\binom{2^k-1}{i-1}$. If $i \neq 0$ or 2^k, i has less than k powers of 2 in its factorization and since both $\binom{2^k}{i}$ and $\binom{2^k-1}{i-1}$ are integers, $\binom{2^k}{i}$ is even. Therefore, for $i \leq \lfloor (2^k + 0)/2 \rfloor$, $\binom{2^k}{i}$ is odd only when $i = 0$, corresponding to $\binom{0}{0}$ being odd.

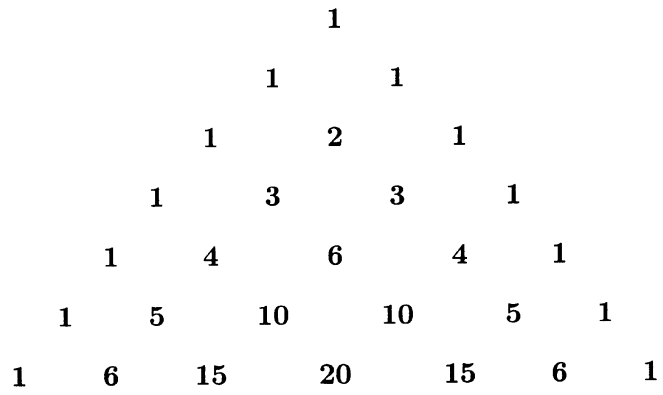

FIG. 1. First few rows of Pascal's triangle.

If $l > 0$ and $1 \leq i \leq l - 1$,

$$\binom{2^k + l}{i} = \binom{2^k + (l-1)}{i} + \binom{2^k + (l-1)}{i-1}$$
$$\binom{l}{i} = \binom{l-1}{i} + \binom{l-1}{i-1}.$$

By the induction hypothesis, $\binom{2^k+(l-1)}{i}$ and $\binom{l-1}{i}$ are both even or both odd and $\binom{2^k+(l-1)}{i-1}$ and $\binom{l-1}{i-1}$ are both even or both odd. Therefore, the results of the summation are both even or both odd. The cases where $i = 0$ or $i = l$ should be treated separately.

If $i = 0$, $\binom{2^k+l}{i}$ and $\binom{l}{i}$ are both equal to 1, an odd number. If $i = l$,

$$\binom{2^k + l}{i} = \frac{2^k + l}{l} \frac{2^k + l - 1}{l - 1} \cdots \frac{2^k + 1}{1}.$$

Since $l < 2^k$, each term in the above product is odd, making $\binom{2^k+l}{i}$ odd. $\binom{l}{i}$ is 1, which is odd. This completes the proof. ■

COROLLARY 2. *The number of odd terms in the* $(2^k + l)$*th row of the Pascal's triangle,* $0 \leq l < 2^k$*, is twice the number of odd terms in the* l*th row.*

Proof. Follows directly from Theorem 1 and an inspection of Fig. 2. ■

The following lemma estimates the number of odd binomial coefficients in a given row of the Pascal's triangle:

LEMMA 3. *Let* k *be the number of 1's in the binary representation of* n*. The number of odd terms in the* n*th row of the Pascal's triangle is* 2^k*.*

Proof. By induction on k. If $k = 0$, the statement is obviously true. Otherwise, $n = 2^{m_1} + 2^{m_2} + \cdots + 2^{m_k}$,

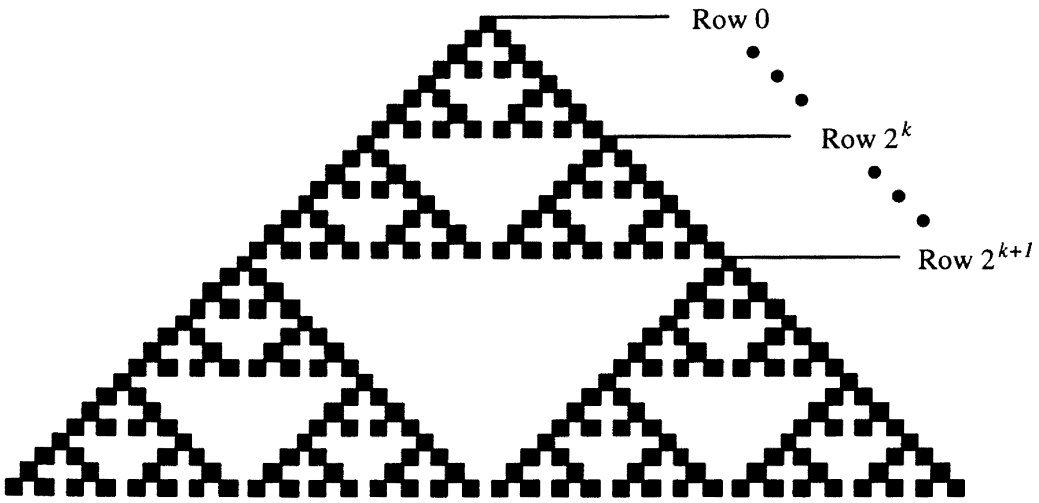

FIG. 2. The structure of odd numbers on Pascal's triangle.

$m_1 > m_2 > \cdots > m_k$. By Corollary 2, the number of odd terms in the the nth row is twice the number of odd terms in the row corresponding to $n' = 2^{m_2} + 2^{m_3} + \cdots + 2^{m_k}$, which is 2^{k-1} by induction hypothesis. Therefore, there are 2^k odd terms in the nth row. ■

The following lemma provides a simple test to determine which binomial coefficients are odd and which are even. We say that r preserves the 0's of n iff for every bit position having a 0 in n, r has a 0 in the same bit position. In other words, r can be obtained from n by changing none or more of the 1's to 0's. For example, 0110010 preserves the 0's of 1110110.

LEMMA 4. $\binom{n}{r}$ *is odd,* $r \le n \Leftrightarrow r$ *preserves the* 0*'s of* n.

Proof. For the "if" part, consider induction on the number of bits required to represent n. If $n = 0$ or 1, any r preserves the 0's of n, in accordance with the fact that all terms in the first two rows of Pascal's triangle are odd. If the number of bits in the binary representation of n is more than 1.

Case 1. $r \le \lfloor n/2 \rfloor$.

$$n = 1 \ldots$$
$$r = 0 \ldots$$

By Theorem 1, $\binom{n}{r}$ is odd if $\binom{n'}{r'}$ is odd, where n' and r' are obtained by removing the most significant bit from n and r. Clearly, r' preserves the 0's of n'. By the induction hypothesis, $\binom{n'}{r'}$ is odd, and hence $\binom{n}{r}$ is odd.

Case 2. $r > \lfloor n/2 \rfloor$.

It is easy to see that $n - r$ preserves the 0's of n and $n - r \le \lfloor n/2 \rfloor$. By Case 1, $\binom{n}{r} = \binom{n}{n-r}$ is odd.

The "only if" part is by a counting argument. Let k be the number of 1 bits in the binary representation of n. There are 2^k choices of r preserving the 0's of n. By the "if" part, all these result in odd combinations. By Lemma 3, there are only 2^k odd combinations. Hence, any r that does not preserve the 0's of n results in an even combination, completing the proof. ■

Corollary 5. $\binom{n}{r}$ *is odd* $\Leftrightarrow (n \vee r) = n$, *where* $\vee$ *is the bitwise logical or operator.*

Proof. Follows directly from Lemma 4. ■

5. COMMUNICATION-FREE PARALLELIZATION OF LAGGED FIBONACCI GENERATORS USING THE LEAPFROG TECHNIQUE

We are now ready to investigate the possibility of applying the leapfrog technique to lagged Fibonacci generators. The recurrence $x_k = x_{k-p} \otimes x_{k-p+q} \bmod m$ in its original form is of little use to generate parallel streams. We try to explore all the possible ways of computing a given term, x_k.

Recursive expansion of $x_k = x_{k-p} \otimes x_{k-p+q} \bmod m$ is shown in Fig. 3. The relationship with Pascal's triangle is clear: Each row in Fig. 3 represents a way of computing x_k. $\binom{n}{i-1}$ occurrences of $x_{k-np+(i-1)q}$ and $\binom{n}{i}$ occurrences of $x_{k-np+iq}$ in the nth row contribute to $\binom{n}{i-1} + \binom{n}{i} = \binom{n+1}{i}$ occurrences of $x_{k-(n+1)p+iq}$ in the $(n+1)$th row. It follows that, $\forall n$ s.t. $1 \le n \le \lfloor k/p \rfloor$,

$$x_k = \bigotimes_{i=0}^{n} \bigotimes_{j=1}^{\binom{n}{i}} x_{k-np+iq} \bmod m \qquad (\otimes = \times, +, \text{ or } \oplus) \quad (1)$$

$$x_k = \sum_{i=0}^{n} (-1)^i \sum_{j=1}^{\binom{n}{i}} x_{k-np+iq} \bmod m \qquad (\otimes = -), \quad (2)$$

corresponding to the nth row of Pascal's triangle. The notation $\bigotimes_{i=0}^{n}$ is much like the $\sum_{i=0}^{n}$, except that the operation involved is $\otimes$ instead of +.

Unfortunately, it is not immediately clear if this family of equations is of help in efficient parallel random number generation. The number of terms in the equation as a function of n is $\sum_{i=0}^{n} \binom{n}{i} = 2^n$.

To parallelize a generator in a communication-free manner using the leapfrog technique, the processor generating x_k

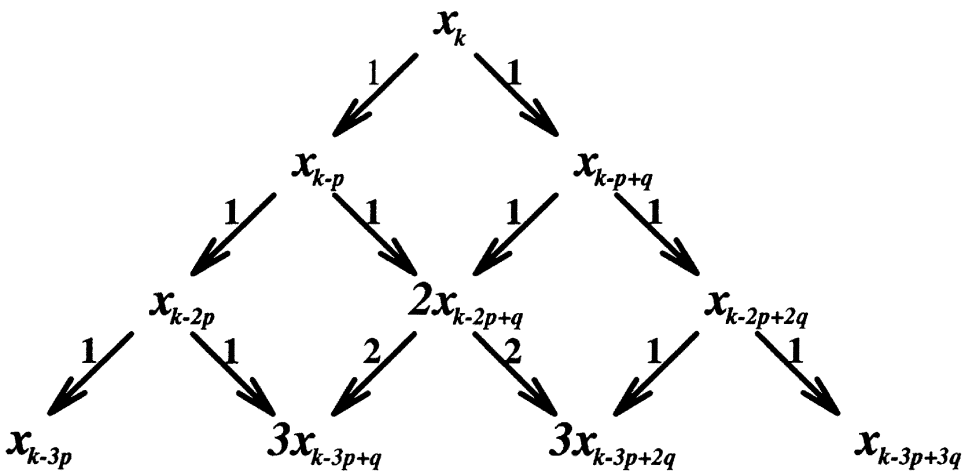

FIG. 3. Recursive expansion of the Fibonacci generator.

should be able to compute x_k using only the previously generated random numbers on the same processor, i.e., using only x_{k-mN} for $1 \leq m \leq \lfloor k/N \rfloor$. Therefore, Eq. (1) or (2) can be used for parallel generation if $x_{k-np+iq} = x_{k-(np-iq)}$ is available on the processor computing x_k for all $0 \leq i \leq n$. This implies that $(np - iq)$ should be a multiple of N for all $0 \leq i \leq n$; i.e., we require $(np - iq) \bmod N = 0$:

$$\begin{aligned} &(np - iq) \bmod N = (np - (i+1)q) \bmod N \\ &\Rightarrow q \bmod N = 0 \\ &\Rightarrow q = kN. \end{aligned}$$

It is obviously impossible to satisfy this relation since it restricts the number of processors to be a factor of q. Even otherwise, substituting $i = 0$ in $(np - iq) \bmod N = 0$ implies that $np \bmod N = 0$ or that np is a multiple of N. This can be satisfied only by taking n to be a multiple of N. Since we want n to be as small as possible, choose $n = N$. This would imply that the equation for computing x_k has 2^N terms, where N is the number of processors. This cost is prohibitive and definitely nonscalable, even if all the terms were to be found on the same processor.

There is still one avenue left open: If we can choose n such that

$$\bigotimes_{j=1}^{\binom{n}{i}} x_{k-np+iq} \bmod m = 0 \qquad (\otimes = +, -, \oplus)$$

$$\bigotimes_{j=1}^{\binom{n}{i}} x_{k-np+iq} \bmod m = 1 \qquad (\otimes = \times)$$

irrespective of $x_{k-np+iq}$ for most i, then it does not matter if $x_{k-np+iq}$ is not available on the same processor. Clearly, this cannot be done for $i = 0$ and $i = n$ since $\binom{n}{0} = \binom{n}{n} = 1$ and $x_{k-np+iq}$ is arbitrary. Therefore, we require that x_{k-np} and $x_{k-np+nq}$ be available on the same processor, which can be easily accomplished by choosing $n = N$. For the parallel generator to have the same speed per processor as the sequential generator, $\bigotimes_{j=1}^{\binom{N}{i}} x_{k-Np+iq} \bmod m$ should be equal to 0 for +, −, or ⊕ generators and 1 for a × generator, for all i except 0 and N. For such values of N, we can have a parallel lagged Fibonacci generator on N processors.

5.1. Lagged Fibonacci with ⊕

The equation $\bigotimes_{j=1}^{\binom{N}{i}} x_{k-Np+iq} \bmod m$ can be simplified in the particular case where the operation involved is exclusive or (⊕), using the properties of exclusive or and binomial coefficients modulo 2. Since for any binary vector **a** and integer l

$$\bigoplus_{i=1}^{l} \mathbf{a} = \begin{cases} \mathbf{0} & \text{even } l \\ \mathbf{a} & \text{odd } l \end{cases},$$

$$\bigoplus_{j=1}^{\binom{N}{i}} x_{k-Np+iq} = \left\{ \binom{N}{i} \bmod 2 \right\} x_{k-Np+iq}.$$

By Corollary 5, $\binom{N}{i} \bmod 2$ is 1 if $(N \vee r) = N$ and is 0 otherwise.

If N is a power of 2, by Lemma 3, there are only two possible values of i for which $\binom{N}{i}$ is odd. By Lemma 4, these values are 0 and N. The equation is further simplified to $x_k = x_{k-Np} \oplus x_{k-Np+Nq} \bmod m$.

For generating l-bit random numbers, m is chosen to be 2^l. If $a < 2^l$ and $b < 2^l$, $a \oplus b < 2^l$, allowing us to drop $\bmod m$. Thus, from $x_k = x_{k-p} \oplus x_{k-p+q}$, we obtain $x_k = x_{k-Np} \oplus x_{k-Np+Nq}$, where N is a power of 2.

Consider the task of generating N parallel streams using the leapfrog technique, where N is a power of 2. If each processor stores the last p random numbers it generated, the processor computing x_k will have $x_{k-N}, x_{k-2N}, \ldots, x_{k-(p-q)N}, \ldots, x_{k-pN}$ in its local memory. In particular, $x_{k-(p-q)N}$ and x_{k-pN} are found in the local memory using which x_k can be computed as $x_k = x_{k-(p-q)N} \oplus x_{k-pN}$.

There is a striking similarity between the sequential and parallel generators. Both generate the next random number by a simple exclusive or operation on two previous numbers found in the local memory. Both retain the last p random numbers as state information. In fact, the code is identical except for the initial generation of seeds for the parallel generator.

If N is not a power of 2, then there are at least two 1's in the binary representation of N. By Lemma 3, there are at least four odd numbers in the Nth row of the Pascal's triangle. For example, $\exists i\,(i \neq 0 \text{ and } i \neq N)$ such that $\binom{N}{i}$ is odd. For such i, $x_{k-Np+iq}$ cannot be ignored. Hence, we cannot parallelize the lagged Fibonacci $\oplus$ generator when the number of processors is not a power of 2.

5.2. Lagged Fibonacci with + or –

In this case,

$$\bigotimes_{j=1}^{\binom{N}{i}} x_{k-Np+iq} = \binom{N}{i} x_{k-Np+iq}.$$

Therefore, we require $\binom{N}{i} \bmod m$ to be 0 for all i such that $1 \leq i < N$; i.e., m should divide $\binom{N}{i}$ for all such i. Therefore, m should divide the greatest common divisor (gcd) of all $\binom{N}{i}$ $(1 \leq i < N)$. Since m should be a power of 2, we are interested in the highest power of 2 that can divide this greatest common divisor.

If N is not a power of 2, we have already seen that $\exists i\,(i \neq 0 \text{ and } i \neq N)$ such that $\binom{N}{i}$ is odd. Therefore, the gcd has to be an odd number, making it impossible to find a suitable m.

If N is a power of 2, consider the binomial coefficient

$$\binom{N}{N/2} = \frac{N}{N/2}\binom{N-1}{N/2-1} = 2\binom{N-1}{N/2-1}.$$

Since $N-1$ does not have any 0's in its binary representation, by Lemma 4, every binomial coefficient of $N-1$ is odd. In particular, $\binom{N-1}{N/2-1}$ is odd. Therefore, the highest power of 2 that can divide $\binom{N}{N/2}$ is 2. Since all binomial coefficients of N (except $\binom{N}{0}$ and $\binom{N}{N}$) are even, it follows that the highest power of 2 that divides the gcd of all binomial coefficients of N except 0 and N is 2. Thus, we are forced to choose $m = 2$.

If m is chosen to be 2, we are generating 1-bit random numbers (random strings of 0's and 1's) for which the additive generator is clearly equivalent to the 1-bit $\oplus$ generator.

From the above arguments, lagged Fibonacci generators with + or – cannot be efficiently parallelized in a communication-free manner using the leapfrog technique.

5.3. Lagged Fibonacci with ×

For the multiplicative lagged Fibonacci generator, the formula representing the computation of x_k reduces to

$$\prod_{i=0}^{N}\prod_{j=1}^{\binom{N}{i}} x_{k-Np+iq} \bmod m = \prod_{i=0}^{N} (x_{k-Np+iq})^{\binom{N}{i}} \bmod m.$$

Therefore, we want to find all N and m such that

$$(x_{k-Np+iq})^{\binom{N}{i}} \bmod m = 1$$

for arbitrary odd $x_{k-Np+iq}$, $1 \leq x_{k-Np+iq} \leq m-1$, $1 \leq i \leq N-1$.

LEMMA 6. *If m is a power of 2 and a is an odd number between 0 and m,*

$$a^n \bmod m = 1 \Leftrightarrow m \text{ divides } n.$$

Proof. For the "if" part, let $a = 2j+1$ from some j:

$$a^n = (1+2j)^n = 1 + \sum_{i=1}^{n} \binom{n}{i} (2j)^i$$
$$= 1 + \sum_{i=1}^{n} \binom{n-1}{i-1} \frac{n}{i} 2^i j^i.$$

Since i has less than i powers of 2 in its prime factorization, each term in the summation has at least as many powers of 2 as n has in the respective prime factorizations. Since m divides n and m is a power of 2, $a^n \bmod m = 1$.

For the "only if" part, suppose m does not divide n. Let $n = mp + q$ where $q < m$:

$$\begin{aligned} a^n \bmod m &= a^{mp+q} \bmod m \\ &= ((a^{mp} \bmod m)(a^q \bmod m)) \bmod m \\ &= a^q \bmod m \end{aligned}$$

since $a^{mp} \bmod m = 1$ by the "if" part.

It is enough to show that $a^q \bmod m \neq 1$ for some odd a. One can easily verify that when $a = m-1$, $a^q \bmod m = m-1$, completing the proof. ■

From Lemma 6,

$$(x_{k-Np+iq})^{\binom{N}{i}} \bmod m = 1$$

for all $\binom{N}{i}$ $(1 \leq i \leq N-1)$ iff m divides all such $\binom{N}{i}$. Therefore, m should divide the greatest common divisor of all such $\binom{N}{i}$.

This condition is the same as required for parallelizing the additive lagged Fibonacci generator. Therefore, multiplicative lagged Fibonacci generators cannot be efficiently parallelized in a communication-free manner using the leapfrog technique.

6. PARALLELIZING ADDITIVE LAGGED FIBONACCI GENERATORS

Since communication-free parallelization of arbitrary lagged Fibonacci generators using the leapfrog technique is not possible, we now consider parallelization that uses communication. We also present a parallelization using the contiguous subsequence technique. Our goal still is to generate in $O(1)$ time the next random number on each processor. Recall that under the contiguous subsequence technique each processor can compute random numbers independently of other processors and at the same rate as the sequential generator, once the required seed is supplied. However, the initial computation of the seeds is very time-consuming, as it is a function of the length of the subsequence allotted to each processor. We will describe strategies to cope with this including techniques to compute the required seeds in parallel and a way of preprocessing the generator to facilitate faster computation of seeds. For the leapfrog technique, the time for computation of the seeds is tolerable as it is a function of the number of processors alone. Computation of the random numbers must use communication (in light of the results presented in the previous section). We explore ways of minimizing the communication overhead and comment on the efficiency that can be expected in practice.

Let X_k denote the vector $[x_{k-p+1} \quad x_{k-p+2} \quad \cdots \quad x_k]^T$. Let M be a $p \times p$ matrix such that its ijth entry m_{ij} ($0 \leq i, j \leq p-1$, i is the row number and j is the column number) is 1 if $j = i+1$ or if $i = p-1$ and $j = 0$ or q. Let all other entries in the matrix be zero. The lagged Fibonacci generator with +

$$x_k = x_{k-p} + x_{k-p+q} \bmod m$$

can be written as

$$X_k = MX_{k-1}$$

with the implicit understanding that all elements of the result are reduced mod m. For example, the generator $x_k = x_{k-5} + x_{k-3}$ is written in matrix form as

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{k-5} \\ x_{k-4} \\ x_{k-3} \\ x_{k-2} \\ x_{k-1} \end{bmatrix} = \begin{bmatrix} x_{k-4} \\ x_{k-3} \\ x_{k-2} \\ x_{k-1} \\ x_k \end{bmatrix}.$$

For the lagged Fibonacci generator with −, the matrix M is defined similarly except that m_{ij} is −1 if $i = p-1$ and $j = q$. All other entries are the same as in the case of the generator with +.

Using the matrix notation, a lagged Fibonacci generator with + or − is written as

$$X_k = M^j X_{k-j}$$

for any $1 \leq j \leq k-p+1$, where M^j is the jth power of the matrix M with all elements reduced mod m. We use this formulation to parallelize the generators. Let N be the number of processors and let the processors be $P_0, P_1, \ldots, P_{N-1}$.

In our algorithms for parallelizing the random number generators, we need to communicate among the processors and the cost of such a communication depends on the topology of the interconnection network connecting the processors and the routing mechanism used. Instead of making specific assumptions about the network, we describe the algorithms using well-studied, standard communication primitives. Extensive work is done on the efficient implementation of such primitives on various topologies [21]. The running time on a specific network of interest can be easily obtained by substituting the running time of the primitives on the network. We consider three common interconnection topologies that represent most of the commercially available parallel computers—meshes (Intel Delta and Paragon), hypercubes (Intel iPSC/860 and nCUBE) and permutation networks (IBM SP-1 and SP-2). In a permutation network, the network can be configured such that a communication pattern corresponding to any permutation of the processors can be realized in parallel. The communication behavior of some parallel computers (such as the CM-5) can be closely approximated by a permutation network.

We use cut-through routing to model the communication cost, which is the routing mechanism used in most parallel computers including all of the computers mentioned in the previous paragraph. In a cut-through routed network, a message transfer of size m between two processors that are l links apart takes $t_s + t_w m + t_h l$ time, where t_s is the set-up time, t_w is the transfer time per word, and t_h is the time for the header of the message to traverse one link. The startup time t_s is often large, and can be several hundred machine cycles or more. In comparison, t_h is quite small and hence the component $t_h l$ can often be subsumed into the startup time t_s without significant loss of accuracy. This is because the diameter of the network, which is the maximum of the distance between any pair of processors, is relatively small for most practical sized machines.

The following is a description of the primitives used and the running times of these primitives on various network topologies are summarized in Table II. We do not describe how the primitives are implemented and the interested reader is referred to [21].

1. *Parallel Prefix.* Suppose that $x_0, x_1, \ldots, x_{N-1}$ are N data elements with processor P_i containing x_i. Let $\otimes$ be a binary associative operation, which can be applied in constant time. The Parallel Prefix operation stores the value of $x_0 \otimes x_1 \otimes \cdots \otimes x_i$ on processor P_i.

2. *All-to-All Broadcast.* In an all-to-all broadcast, each processor broadcasts a message of size m to every other processor.

3. *All-to-All Communication.* In this operation, each processor sends a distinct message of size m to every processor.

TABLE II
Running Times of Some Parallel Communication Primitives on Various Network Topologies Using Cut-Through Routed Networks with N Processors

Topology	Parallel prefix	All-to-all broadcast	All-to-all communication
Mesh	$O((t_s + t_w)\log N + t_h\sqrt{N})$	$O(t_s\sqrt{N} + t_w mN)$	$O((t_s + t_w mN)\sqrt{N})$
Hypercube	$O((t_s + t_w)\log N)$	$O(t_s \log N + t_w mN)$	$O((t_s + t_w m)N + t_h N \log N)$
Permutation network	$O((t_s + t_w)\log N)$	$O(t_s \log N + t_w mN)$	$O((t_s + t_w m)N)$

6.1. Contiguous Subsequence Technique

In this method, processor P_i should generate

$$x_{iL},\ x_{iL+1},\ x_{iL+2},\ \ldots,\ x_{(i+1)L-1}$$

where L is the length of the subsequence allocated to each processor. If we compute the first p random numbers to be generated on processor P_i, the remaining numbers can be generated using the formula of the sequential generator and independent of the other processors. Therefore, we need to compute X_{iL+p-1} on processor P_i. We use

$$X_{iL+p-1} = M^{iL} X_{p-1},$$

where X_{p-1} (the first p random numbers $x_0,\ x_1,\ \ldots,\ x_{p-1}$) is given as the seed to this generator. We can compute M^{iL} in $O(p^3 \log iL)$ time using matrix multiplication. Note that processor P_i computes M^{iL} independent of any other processor. This is the preprocessing cost involved in the algorithm. Since i can be as high as $N-1$, the time required for preprocessing is $O(p^3 \log NL)$.

It may not be possible to determine the number of random numbers required by each processor a priori. In such a case, L is chosen to be as large as possible. The largest L can be is $\lfloor P/N \rfloor$, where P is the period of the generator. Therefore, the worst-case preprocessing time is $O(p^3 \log P)$, which is $O(p^3(p+l))$, since $P = (2^p - 1)2^{l-1}$.

Once the first p random numbers of the contiguous subsequence assigned to each processor are computed, the formula for the sequential generator can be used to generate the rest of the subsequence independent of other processors. There is a trade-off in choosing the value of the lag. A large value of the lag should be chosen for quality considerations. Initializing the generator takes $O(p^3 \log NL)$ time, which can be prohibitively large for large values of the lag p and may render this method impractical. For example, if p is 607 and we are using 256 processors, allocating a subsequence of length 2^{20} to each, initialization requires approximately 6.26×10^9 calculations. Depending on the speed of the processor, this may take anywhere from minutes to hours.

We sketch two strategies that can be used to get around this problem. Note that the matrix M is fixed once the generator (i.e., p and q) is fixed. Also, M^{iL} depends on the choice of L but not on the seed. Therefore, one can precompute the matrices and store them which can be retrieved on demand and used. If N is the maximum number of processors in the parallel computer, precompute M^{iL} for every $1 \le i \le N-1$. Storing these matrices requires $O(p^2 N)$ space. If an application is run on $N' \le N$ processors, matrix M^{iL} is retrieved and used by processor P_i for $1 \le i \le N' - 1$.

Another strategy to reduce the time required for initialization is to compute the matrix M^{iL} itself in parallel. Suppose that the number of processors, $N \le p^2$. This is perhaps the case even for a small value of p such as 31. Two $p \times p$ matrices can be multiplied using N processors in optimal $O(p^3/N)$ time on a mesh, hypercube or a permutation network using a standard parallel matrix multiplication algorithm [21]. In the following analysis, we limit ourselves to the computation part of the run time for two reasons: the communication terms are of a lower order, typically lower by a factor of p. Besides, they depend upon the choice of the matrix multiplication algorithm. We first compute M^L in parallel using $O(\log L)$ matrix multiplications each taking $O(p^3/N)$ time. Using an all-to-all broadcast, the matrix M^L is distributed to every processor. We can now compute M^{iL} on P_i with a parallel prefix operation using the matrix multiplication operator. Computing the parallel prefix takes $O(\log N)$ steps each of which now takes $O(p^3)$ time as it involves sequential multiplication of two $p \times p$ matrices. Thus, the time for initialization is reduced from $O(p^3 \log NL)$ to $O(p^3 \log N + p^3 \log L/N)$.

It is possible to further reduce this time to $O(p^3 + p^3 \log L/N)$. In the parallel prefix operation, each processor can have a potentially different element. In the above algorithm, each processor has the same matrix $M_1 = M^L$. The idea is to pool together all the processors that compute the same matrix and compute the matrix in parallel and share the result. For example, in the first step of the parallel prefix algorithm, the matrices at processors P_i and P_{i-1} are multiplied (for every even i). As all the initial matrices are the same, the N processors can be pooled together to compute M_1^2 and share the result. This works since the resulting matrix has size $O(p^2)$ where as the work to compute it increases as $O(p^3)$. An easy analysis shows that this strategy reduces the time for parallel prefix from $O(p^3 \log N)$ to $O(p^3)$.

A similar reduction in the time for initialization can be achieved when $N > p^2$. In this case, we divide the processors into groups such that each group contains p^2 processors. In each group, M^L is computed using all the p^2 processors and is then distributed to all the processors. This takes $O(p \log L)$ time. The parallel prefix is done as before. The time for

initialization is $O(p^3 + p \log L)$. Therefore, the time required for initialization on an arbitrary number N of processors is $O(p^3 + p^3 \log L / \min\{p^2, N\})$.

Of course, one can combine the two strategies and precompute the matrices in parallel and store for future use.

6.2. Leapfrog Technique

To generate random numbers according to the leapfrog technique, we use

$$X_k = M^N X_{k-N},$$

where M^N is the Nth power of the matrix M with all elements reduced mod m (recall that N is the number of processors). Once M^N is precomputed, the above equation can be used to generate successive random numbers on each processor without any communication. But, this requires p^2 multiplications and $p(p-1)$ additions to compute one random number on each processor while the sequential generator requires just one addition (or subtraction) to compute the next random number.

The vector X_k contains p consecutive random numbers, $x_{k-p+1}, x_{k-p+2}, \ldots, x_k$. Therefore, once X_k is computed, the equation of the sequential lagged Fibonacci generator can be used to generate successive random numbers $x_{k+1}, x_{k+2}, \ldots$ etc. Thus, the $O(p^2)$ cost of computing X_k can be amortized to give $O(1)$ generation time per random number if at least $\Omega(p^2)$ random numbers after x_k are computed. Since these numbers are to be generated on other processors, they will have to be distributed efficiently to the appropriate processors.

Let $r = \max(N, p^2)$. We generate rN consecutive numbers of the random sequence collectively on N processors and redistribute the numbers such that each processor has the next r numbers it should generate according to the leapfrog technique. Initially, processor P_i computes the vector X_{ri+p-1} and the matrix M^{rN}. This is a one time pre-processing cost and requires $O(p^3(\log r + \log N)) = O(p^3 \log r)$ time. We generate a total of rN numbers at a time. At a stage when a total of jrN (for some j) random numbers are generated, P_i will have the vector $X_{(j-1)rN+ri+p-1}$. Our goal is to generate the next r numbers on P_i according to the leapfrog technique, i.e., to generate x_{jrN+i}, $x_{jrN+i+N}$, $x_{jrN+i+2N}, \ldots, x_{jrN+i+(r-1)N}$.

First, P_i computes $X_{jrN+ri+p-1}$ using

$$X_{jrN+ri+p-1} = M^{rN} X_{(j-1)rN+ri+p-1}$$

in $O(p^2)$ time. P_i now has the random numbers x_{jrN+ri}, $x_{jrN+ri+1}, \ldots, x_{jrN+ri+p-1}$. Using these p consecutive numbers and the formula for the sequential generator, P_i computes an additional $(r - p)$ numbers to have x_{jrN+ri}, $x_{jrN+ri+1}, \ldots, x_{jrN+ri+r-1}$. The total time required is only $O(r)$ since $r \geq p^2$.

Each processor has thus generated r numbers and they should be redistributed such that every processor has r numbers according to the leapfrog ordering. Each processor splits its list of r random numbers into N lists such that the ith list contains numbers that should be sent to P_i. Finding which processor a particular random number should be sent to is easy: x_k should be sent to processor $P_{k \bmod N}$. Once the first random number is put in the appropriate list, successive random numbers go into the successive lists with wraparound. Each processor has $\lfloor r/N \rfloor$ or $\lceil r/N \rceil$ random numbers to send to every processor. At the end of the communication, each processor has the next r numbers according to the leapfrog ordering by simply appending the lists it received from $P_0, P_1, \ldots, P_{N-1}$.

The communication pattern where each processor sends a distinct message of size m to every processor is known as All-to-All Communication and is a well-studied problem in parallel computing. The required redistribution of random numbers can be performed using such an all-to-all communication. In the random number redistribution of interest to us, each processor has $\lfloor r/N \rfloor$ or $\lceil r/N \rceil$ numbers to send to every other processor. Therefore, the redistribution takes $O((t_s + t_w r)\sqrt{N})$ time on a mesh, $O(t_s N + t_w r + t_h N \log N)$ time on a hypercube, and $O(t_s N + t_w r)$ time on a permutation network (see Table II). The $O(r)$ time spent in computing r consecutive random numbers on each processor prior to the redistribution should be added to this to find the time for generating r numbers on each processor according to the leapfrog ordering.

Recall that $r = \max(N, p^2)$. For efficient parallel random number generation ($O(1)$ generation time per random number on each processor), the redistribution should be accomplished in $O(r)$ time. This is indeed the case for a permutation network. On a mesh, redistribution requires $O(r\sqrt{N})$ time, which is clearly unacceptable.

The time for generating r random numbers according to the leapfrog ordering on the hypercube is $O(t_c r + t_s N + t_w r + t_h N \log N)$, where t_c is the time to do a unit computation. Since the set-up time t_s is typically two orders of magnitude larger than t_h, the term $t_h N \log N$ is dominated by $t_s N$ for practical values of N. With this approximation, the next r random numbers on each processor can be generated in $O(r)$ time.

It is only necessary to choose r to be greater than or equal to $\max(N, p^2)$. We require $r \geq p^2$ in order to amortize the $O(p^2)$ initial cost of generating p numbers. We require $r \geq N$ to ensure that each processor has at least one number to send to every processor. The only cost in choosing a large value of r is that we need $O(r)$ memory on each processor to store the random numbers. By choosing $r = \max(N \log N, p^2)$, the hypercube running time of $O(t_c r + t_s N + t_w r + t_h N \log N)$ reduces to $O(r)$ without making the approximation of ignoring the term involving t_h. However, this may not be necessary for practical values of N.

In order to reduce the overhead due to communication, we can consider the following strategies when appropriate: Since set-up time for communication is large, it is more economical to send a single message of large size than to send several short message of equivalent total size. Therefore, we can choose r

to be as large as possible within the constraints of available memory. Another approach can be used if the application using the random numbers itself requires frequent all-to-all communication. In such a case, we can piggyback the random numbers to the messages sent in the application and completely eliminate the overhead due to set-up times.

Even though the overhead due to set-up times can be brought under control using one of the above strategies, the cost of generating a random number is still proportional to $(t_c + t_w)$. For the additive lagged Fibonacci generator, t_c is the time to compute an addition or subtraction followed by a mod operation. t_w, the per-word transfer time, is typically an order of magnitude larger or more. Thus, even though the parallel algorithm has ideal speed up theoretically, the speed-up in practice can be expected to be no more than

$$\frac{t_c}{t_c + t_w} N.$$

It is important to carefully consider the effect of the communication parameters on the practical speedup that can be attained by the algorithm. The algorithm should at the least perform better than a simple algorithm in which each processor generates the entire sequence and selects only those numbers that it should generate according to the leapfrog technique. This happens only when the number of processors $N > ((t_c + t_w)/t_c = 1 + t_w/t_c$. The ratio t_w/t_c for a variety of machines is shown in Table III. The data is colleted from [8] in which t_c is chosen to be the cost of one floating point operation. In our case, the required computation is one integer addition or subtraction followed by a mod operation.

Notice that some of the machines for which the ratio t_w/t_c is presented are meshes (Delta, Paragon) even though our algorithm is not efficient on a mesh but is useful only for hypercubes and permutation networks. However, the data is still meaningful as the ratio depends on network and processor hardware characteristics but is independent of the topology. From the data, the number of processors at which the presented algorithm starts to outperform the naive algorithm (of generating the entire sequence on every processor) can be readily observed to range from low single digit numbers to high double digit numbers. For some of the machines, this is well within the range of practical configurations. In any case, the presented algorithm is scalable and can outperform the naive algorithm and provide linear speed-up for large configurations.

TABLE III
Ratio of the Per-Word Transfer Time t_w to the Unit Computation Time t_c for a Variety of Machines (t_c is Taken to Be the Time for One Floating Point Operation)

Machine	CM5	Delta	Paragon	SP1	T3D
t_w/t_c	4	87	9	50	9

Note that the initialization required for this parallelization is to compute M^{rN} and store it in every processor. Also, P_i needs X_{ri+p-1}, which can be computed from X_{p-1} using M^{ri}. Clearly, the required M^{ri}'s can be computed in the same way as M^{iL}'s are computed in Section 6.1. By multiplying $M^{r(N-1)}$ by M^r in parallel or by directly parallelizing the computation of M^{rN} itself, M^{rN} can be computed and distributed to all the processors. Using an analysis similar to the one presented in Section 6.1, the initialization cost can be reduced from $O(p^3(\log r + \log N))$ to $O(p^3 + p^3 \log r / \min\{p^2, N\})$.

If the initialization time is deemed to be expensive, the required matrices can be precomputed, stored and retrieved and used as needed.

7. CONCLUSIONS

In order to parallelize a sequential random number generator, disjoint subsequences of the sequence generated by the sequential generator should be allocated to processors and an algorithm to generate each subsequence efficiently should be designed. The contiguous subsequence technique allocates contiguous subsequences to processors where as the leapfrog technique allocates interleaved but disjoint subsequences to processors. Computing the seeds (initial random numbers) for each subsequence of the parallel generator is time-consuming under the contiguous subsequence technique. Once the seeds are computed, each individual subsequence can be generated at the same rate as the sequential generator. Under the leapfrog technique, there is no straightforward way of generating the individual subsequences. If an efficient way of computation could be found, the leapfrog technique would be preferable.

In this paper, we sought to parallelize lagged Fibonacci generators for distributed memory parallel computers. We first considered efficient parallelization using the leapfrog technique. There does not appear to be any way to parallelize the lagged Fibonacci generators in a communication-free manner except for the restrictive case where the operation is $\oplus$ and the number of processors is a power of 2. We then presented a parallelization of lagged Fibonacci generators with + or − that is efficient for hypercubes and permutation networks. Parallelization of lagged Fibonacci multiplicative generators is still open. We also presented a way of parallelizing lagged Fibonacci plus or minus generators using the contiguous subsequence technique. In both cases, preprocessing and/or parallel computation can be used to reduce the start-up cost of the generators.

It is known that a combination of independent, uniformly distributed random sequences will have a more (or at least equally) uniform distribution than either of its component sequences and the terms of the resulting sequence tend to be more statistically independent. This observation led to the design of combination generators which are formed by combining two or more generators. An example of such a generator is

$$z_i = x_i \pm y_i \bmod m,$$

where x_i and y_i are the ith random numbers produced by two random number generators, preferably of different type. The parallelization of lagged Fibonacci generators presented in this paper can be potentially useful in designing some parallel combination generators.

ACKNOWLEDGMENTS

The author thanks the anonymous referees for valuable comments and for drawing the author's attention to analyzing the practical efficiency of the algorithms presented.

REFERENCES

1. Aluru, S. Parallel additive lagged Fibonacci random number generators. *Proc. 10th International Conference on Supercomputing '96.* 1996, pp. 102–108.
2. Aluru, S. Properties of binomial coefficients and implications to parallelizing lagged Fibonacci random number generators. *Proc. Intl. Conf. Par. Proc. '95.* 1995, Vol. III, pp. 25–28.
3. Aluru, S. Random number generators for parallel computers. M.S. thesis, Iowa State University, 1991.
4. Aluru, S., Prabhu, G. M., and Gustafson, J. A random number generator for parallel computers. *Parallel Comput.* **18** (1992), 839–847.
5. Anderson, S. L. Random number generators on vector supercomputers and other advanced architectures. *SIAM Rev.* **32,** 2 (June 1990), 221–251.
6. Bowman, K. O., and Robinson, M. T. Studies of random number generators for parallel processing. *Proc. Second Conference on Hypercube Multiprocessors* (M. T. Heath, Ed.). SIAM, Philadelphia, 1987, pp. 445–453.
7. Brent, R. P. Uniform random number generators for supercomputers. *Proc. 5th Australian Supercomputer Conference.* 1992, pp. 95–104.
8. Chakrabarti, S., Demmel, J., and Yelick, K. Modeling the benefits of mixed data and task parallelism. *Proc. Symposium on Parallel Algorithms and Architectures.* 1995, pp. 74–83.
9. Coddington, P. D. Analysis of random number generators using Monte Carlo simulation. *Int. J. Mod. Phys.* **C5** (1994), 547.
10. Collings, B. J., and Hembree, G. B. Initializing generalized feedback shift register pseudorandom number generators. *J. Assoc. Comput. Mach.* **33** (1986), pp. 706–711.
11. Deak, I. Uniform random number generators for parallel computers. *Parallel Comput.* **15** (1990), 155–164.
12. Evans W., and Sugla, B. Parallel random number generation. *Proc. Fourth Conference on Hypercubes.* [In *Concurrent Comput. Appl.* **1** (1989), 415–420]
13. Ferrenberg, A. M., Landau, D. P., and Wong, Y. J. Monte Carlo simulations: Hidden errors from "good" random number generators. *Phys. Rev. Lett.* **69** (1992), 3382.
14. Fine, N. J. Binomial coefficients modulo a prime. *Amer. Math. Monthly* **54** (1947), 589.
15. Fox, G. *et al. Solving Problems on Concurrent Processors. Vol. I. General Techniques and Regular Problems.* Prentice–Hall, Englewood Cliffs, NJ, 1988.
16. Glaisher, J. W. L. On the residue of a binomial-theorem coefficient with respect to a prime modulus. *Quart. J. Math.* **30** (1899), 150.
17. Grassberger, P. On correlations in "good" random number generators. *Phys. Rev. Lett.* **181** (1993), 43.
18. Kimball, S. H., *et al.* Odd binomial coefficients. *Amer. Math. Monthly* **65** (1958), 368.
19. Knuth, D. E. *The Art of Computer Programming. Vol. 2. Seminumerical Algorithms,* 2nd ed. Addison–Wesley, Reading, MA, 1981.
20. Knuth, D. E. *The Art of Computer Programming. Vol. 1. Fundamental Algorithms,* 2nd ed. Addison–Wesley, Reading, MA, 1973.
21. Kumar, V., Grama, A., Gupta, A., and Karypis, G. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin–Cummings, Redwood City, CA, 1994.
22. Lewis, T. G., and Payne, W. H. Generalized feedback shift register pseudorandom number algorithm. *J. Assoc. Comput. Mach.* **20** (1973), 456–468.
23. Marsaglia, G. A current view of random number generators. *XVIth Conference on Computer Science and Statistics: The Interface* (L. Billard, Ed.). 1985, pp. 3–10.
24. Marsaglia, G. The structure of linear congruential sequences. *In* Zaremba, Z. K. (Ed.). *Applications of Number Theory to Numerical Analysis.* Academic Press, New York, 1972.
25. Marsaglia, G. Regularities in congruential random number generators. *Numer. Math.* **16** (1970), 8–10.
26. Marsaglia, G. Random numbers fall mainly in the planes. *Proc. Nat. Acad. Sci.* **61** (1968), 25–28.
27. Marsaglia, G., and Tsay, L. H. Matrices and the structure of random number sequences. *Linear Algebra Appl.* **67** (1985), 147–156.
28. Marsaglia, G., and Zaman, A. A new class of random number generators. *Ann. Appl. Probab. I* **3** (1991), 462–480.
29. Niederreiter, H. A statistical analysis of generalized feedback shift register pseudorandom number generators. *SIAM J. Sci. Statist. Comput.* **8** (1987), 1035–1051.
30. Park, S. K., and Miller, K. W. Random number generators: Good ones are hard to find. *Comm. ACM* **31** (1988), 1192–1201.
31. Percus, O. E., and Kalos, M. H. Random number generators for MIMD parallel processors. *J. Parallel Distributed Comput.* **6** (1987), 477–497.
32. Pryor, D. V., Cuccaro, S. A., Mascagni, M., and Robinson, M. L. Implementation of a portable and reproducible pseudorandom number generator. *Proc. Supercomputing '94.* 1994, pp. 311–319.
33. Roberts, J. B. On binomial coefficient residues. *Can. J. Math.* **9** (1957), 363.
34. Sharp, H. F., III, and Still, C. H. Random number generation in the parallel environment. *Proc. Fifth Conference on Distributed Memory Computing.* 1990, Vol. 1, pp. 378–381.
35. Vattulainen, I., Ala-Nissila, T., and Kankaala, K. Physical tests for random numbers in simulations. *Phys. Rev. Lett.* **73** (1994), 2513.
36. Wolfram, S. Geometry of binomial coefficients. *Amer. Math. Monthly* **91** (1984), 566–571.
37. Zierler, N. Primitive trinomials whose degree is a Mersenne exponent. *Inform. and Control* **15** (1969), 67–69.
38. Zierler, N., and Brillhart, J. On primitive trinomials (mod 2). *Inform. and Control* **13** (1968), 541–554.

SRINIVAS ALURU received his B.Tech. in computer science and engineering from the Indian Institute of Technology, Madras, in 1989 and his M.S. and Ph.D. in computer science from Iowa State University in 1991 and 1994. From 1994 to 1996, he worked as a visiting assistant professor in the School of Computer and Information Science at Syracuse University. He is currently an assistant professor in the Department of Computer Science at New Mexico State University. His research interests include parallel algorithms and applications, scientific computing, computational geometry, and randomized algorithms. He is a member of ACM, IEEE, and the IEEE Computer Society.

Received November 27, 1995; revised June 30, 1997; accepted July 1, 1997