UNIFORM GENERATION OF COMBINATORIAL OBJECTS IN PARALLEL

M.D. Atkinson and J.-R. Sack

SCS-TR-185, JANUARY 1991

School of Computer Science, Carleton University
Ottawa, Canada, KIS 5B6

Uniform generation of combinatorial objects in parallel

M.D. ATKINSON and J.-R. SACK

School of Computer Science, Carleton University, Ottawa, CANADA K1S 5B6

Abstract. Unbiased random generators for permutations and binary trees are developed for a PRAM. The generators are capable of generating a random permutation on n symbols or a binary tree on n nodes in time O(log n), space O(n), with O(n) processors; they are also capable of generating various related combinatorial objects.

1. Introduction

For many classes of combinatorial objects the number of objects of size n is at least exponential in n so it is impossible to construct the entire class unless n is very small. In such cases it may be useful to have a means of generating elements from the class uniformly at random (an *unbiased* generator). In particular, such a generator may be useful for probabilistic algorithms. For some combinatorial classes the objects can be enumerated in a systematic manner so that one can easily construct the rth element in the enumeration. If that is the case then an unbiased generator could be obtained by generating a random number r in the appropriate range and constructing the rth object. In practice however this is not as easy as it sounds because the range of values of r is exponential in n. Consequently difficulties occur because of the very large integers which have to be manipulated and with the discrimination of the random number generator. It is therefore preferable to proceed in a different way and somehow avoid numbers which are much bigger than n itself (which might be presumed to be storable in a single computer word).

To take a very simple example, suppose we wished to generate a random subset of $\{1,2,...,n\}$. There is a natural correspondence between the possible subsets and the integers in $[0,2^n-1]$. But it would be absurd to exploit this correspondence. A more satisfactory approach would be to toss an unbiased coin n times to make the decision as to which of the n elements to include in the subset.

This paper is concerned with unbiased generators for various types of combinatorial object on a PRAM. We shall begin by considering the problem of generating a random permutation of 1,2,...,n uniformly distributed over the set of all n! permutations. This result will then be applied to the problem of generating a random binary tree on n nodes and we answer a question posed in [7]. Our techniques also provide unbiased generators for combinations, well-formed bracket sequences, and triangulations of polygons. In both cases we shall present algorithms for n processors of time complexity O(log n) and space complexity O(n). A noteworthy feature of our algorithms is that they only use numbers of size O(n). It is reasonable therefore to assume that each arithmetic operation can be performed in constant time.

2. Permutation generation

If α and β are two permutations of 1,2,...,n then $\alpha\beta$ denotes the permutation whose ith component is $\alpha(\beta(i))$; in other words we consider permutation multiplication to be from right to left. Notice that, with n processors, $\alpha\beta$ can be determined in constant time.

LEMMA 2.1 Let x_i be a discrete uniform random variable with integer values in $[1,i]$, and let τ_i be the transposition which exchanges i and x_i (the identity permutation if $i=x_i$). Then the permutation $\sigma=\tau_2\tau_3....\tau_n$ is uniformly distributed in the set of all n! permutations.

Using this lemma we can define a very simple unbiased random permutation generator:

```
RANDOM PERMUTATION
Input: a positive integer n
Output: a permutation $\sigma$
        for i=2,3,...,n do
                choose $x_i$ uniformly in $1 \le x_i \le i$
                define $\tau_i$ to be the transposition which exchanges i and $x_i$
        endfor
        Compute $\sigma=\tau_2\tau_3....\tau_n$
```

The iterations of the for loop in the RANDOM PERMUTATION algorithm are independent and so this part of the algorithm can be performed with n processors in constant time. To compute $\tau_2\tau_3....\tau_n$ we shall appeal to the associativity of permutation multiplication and perform the computation indicated in the balanced binary tree of Figure 1. (To simplify the discussion it is assumed that $n=2^k+1$, for some natural number k.)

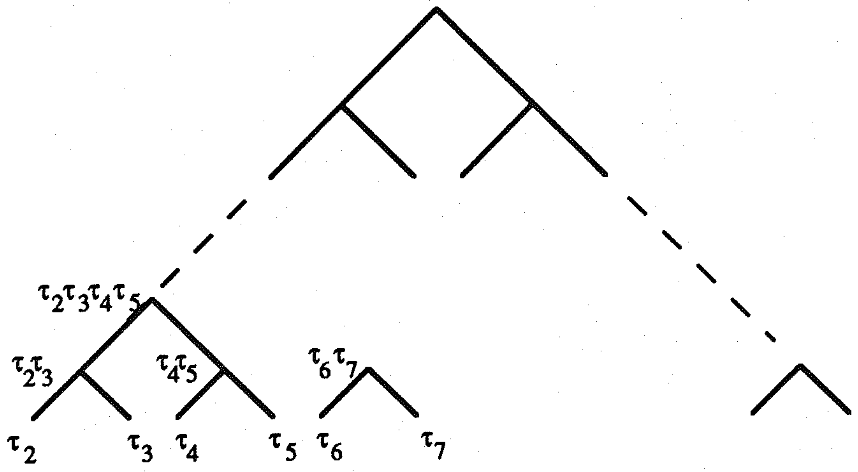

Figure 1: The tree organization of the permutation multiplication.

If there were $n^2/2$ processors we could allocate n processors to each permutation multiplication in the bottom level of the computation tree and calculate each of $\tau_2\tau_3$, $\tau_4\tau_5$,... in constant time. Each level of the tree could be processed in this way giving an overall execution time O(log n). Our main objective in this section is to perform this tree structured computation with fewer processors and prove

THEOREM 2.1 RANDOM PERMUTATION can be implemented to run in O(log n) time, using O(n) space, with O(n) processors.

COROLLARY 2.1 For fixed k and n a random combination of k objects out of n can be generated in O(log n) time, O(n) space, with O(n) processors.

Proof. An unbiased combination generator is obtained by selecting the first k elements in a random permutation.

The starting point for the proof of Theorem 2.1 is the realisation that permutations that fix all but a small number of points can be represented more concisely than arbitrary permutations if we just store their effect on the points that they move.

DEFINITION 2.1 If σ is a permutation on $\{1,2,...,n\}$ *support(σ)* is defined as $\{i \mid 1 \leq i \leq n, \sigma(i) \neq i\}$.

To describe σ it would be enough to specify support(σ) and $\sigma(i)$ for each $i \in$ support(σ). In fact, the representation of permutations that we define below is slightly different to this because it is tailored to the efficient computation of the specific product $\tau_2\tau_3....\tau_n$.

We shall suppose that 2 arrays A, C, of length 2n are available.

DEFINITION 2.2 The contiguous segments A[m+1],..., A[m+h], C[m+1],..., C[m+h] are said to define a *compact representation using indices m+1 to m+h* of a permutation σ if

(i) $A[m+1] \leq A[m+2] \leq .... \leq A[m+h]$
(ii) $\sigma(A[i]) = A[C[i]]$, $m+1 \leq i \leq m+h$
(iii) $\sigma(j) = j$ if $j \notin \{A[m+1], A[m+2], ..., A[m+h]\}$

It is evident that a compact representation of a permutation σ completely defines σ. Notice that the elements of support(σ) are included in the list A[m+1],..., A[m+h] (which may have duplicates) possibly as a proper subset.

In our computation of $\tau_2\tau_3....\tau_n$ we shall begin with a compact representation of τ_i using indices 2i-3 and 2i-2; in other words $\tau_2,\tau_3,....,\tau_n$ are stored in consecutive pairs of locations in A, C. When the product $\tau_2\tau_3....\tau_n$ is calculated in the balanced binary tree of Figure 1 the products $\tau_2\tau_3$, $\tau_4\tau_5$, $\tau_6\tau_7$, ... are calculated first (all in parallel), then $(\tau_2\tau_3)(\tau_4\tau_5)$, $(\tau_6\tau_7)(\tau_8\tau_9)$,... are calculated (also in parallel), and we continue in this way until σ has been computed. We shall arrange the computation so that whenever two permutations are multiplied their product (in a compact representation) overwrites, in the arrays A and C, the compact representations of the operands. It is apparent therefore that a typical multiplication has operands represented in adjacent segments of the arrays A and C.

We now present the algorithm which, when given compact representations for σ and τ, returns a compact representation for $\sigma\tau$. The algorithm is the natural one and we give it in detail only because it is otherwise difficult to discuss its implementation and complexity analysis. To distinguish the new values in A, C from those that are overwritten we use the notation A', C'. The algorithm makes use of auxiliary functions SEARCH(i, m, h) and MERGE(m, h, k) which have the specification given below:

SEARCH(i, u, v)
Input: indices i, u, v into the array A
Output: an index p with $u \le p \le v$ and A[p]=A[i], or 0 if none of A[u], ..., A[v] have the same value as A[i].

MERGE(m, h, k)
Input: values in the arrays A and C for the indices m+1,....,m+h+k which satisfy all the requirements for being a compact representation of some permutation ρ with the possible exception of the condition A[m+h]≤A[m+h+1]
Output: a compact representation of ρ using indices m+1 to m+h+k.

(The latter function is called MERGE because it achieves its result by merging A[m+1],..., A[m+h] into A[m+h+1],..., A[m+h+k] while adjusting the array C so that properties (ii) and (iii) in the definition of a compact representation are preserved.)

COMPACT MULTIPLY(m, h, k)
Input: compact representations using indices m+1 to m+h and m+h+1 to m+h+k for permutations σ and τ
Output: a compact representation using indices m+1 to m+h+k for $\sigma\tau$

```
    for i:=m+1 to m+h do
        p := SEARCH(i, m+h+1, m+h+k)
        if p ≠ 0 then
            r := SEARCH(C[p], m+1, m+h)
            if r ≠ 0 then
                C'[i] := C[r]
            else
                C'[i] := C[p]
            endif
        else
            C'[i] := C[i]
        endif
    endfor
    for i:=m+h+1 to m+h+k do
        p := SEARCH(C[i], m+1, m+h)
        if p ≠ 0 then
            C'[i] := C[p]
        else
            C'[i] := C[i]
        endif
    endfor

    {At this point a compact representation using indices m+1 to m+h+k for στ would
    have been computed except that it may not satisfy A[m+h]≤A[m+h+1]}

    MERGE(m, h, k)

end COMPACT MULTIPLY
```

LEMMA 2.2 Just before the merging phase the values of the arrays A, C in the indices $m+1, \ldots, m+h+k$ satisfy conditions (ii) and (iii) of the definition of a compact representation of $\sigma\tau$.

Proof. Condition (iii) is valid since the values of $A[m+1], \ldots, A[m+h+k]$ are unchanged and, clearly, $\text{support}(\sigma\tau) \subseteq \text{support}(\sigma) \cup \text{support}(\tau)$. To verify condition (ii), i.e. $\sigma\tau(A[i]) = A[C'[i]]$, a number of cases have to be considered.

Suppose first that $m+1 \leq i \leq m+h$. There are 3 possibilities:

1. $A[i] = A[p]$ with $m+h+1 \leq p \leq m+h+k$, and $A[C[p]] = A[r]$ with $m+1 \leq r \leq m+h$.

   In this case C'[i] is defined to be C[r] and we have

   $\sigma\tau(A[i]) = \sigma\tau(A[p]) = \sigma(A[C[p]]) = \sigma(A[r]) = A[C[r]] = A[C'[i]]$.

2. $A[i] = A[p]$ with $m+h+1 \leq p \leq m+h+k$, and $A[C[p]]$ does not occur among $A[m+1], \ldots, A[m+h]$.

   In this case C'[i] is defined to be C[p] and we have

   $\sigma\tau(A[i]) = \sigma\tau(A[p]) = \sigma(A[C[p]]) = A[C[p]] = A[C'[i]]$.

3. $A[i]$ does not occur among $A[m+h+1], \ldots, A[m+h+k]$.

   In this case C'[i] is defined to be C[i] and we have

   $\sigma\tau(A[i]) = \sigma(A[i]) = A[C[i]] = A[C'[i]]$.

Next suppose that $m+h+1 \leq i \leq m+h+k$. There are 2 possibilities:

4. $A[C[i]] = A[p]$ with $m+1 \leq p \leq m+h$.

   Here C'[i] is defined to be C[p] and we have

   $\sigma\tau(A[i]) = \sigma(A[C[i]]) = \sigma(A[p]) = A[C[p]] = A[C'[i]]$.

5. $A[C[i]]$ does not occur among $A[m+1], \ldots, A[m+h]$.

   Here C'[i] is defined to be C[i] and we have

   $\sigma\tau(A[i]) = \sigma(A[C[i]]) = A[C[i]] = A[C'[i]]$.

This lemma justifies the correctness of the algorithm COMPACT MULTIPLY assuming that SEARCH and MERGE are implemented correctly. So the outstanding questions are how to implement SEARCH and MERGE efficiently.

In COMPACT MULTIPLY(m,h,k) the computations for each of the h+k different values of i are independent and so may be executed in parallel provided that h+k processors are available. If there are 2n processors altogether it is possible to make h+k processors available to COMPACT MULTIPLY(m,h,k) and still process all the nodes on each level of the computation tree for $\tau_2\tau_3...\tau_n$ in parallel:- the bottom level has $\lfloor n/2 \rfloor$ calls on COMPACT MULTIPLY all with h+k=4, the next level has $\lfloor n/4 \rfloor$ calls on COMPACT MULTIPLY all with h+k=8, and so on. With this allocation strategy we see that we must be able to implement SEARCH with one processor only, and MERGE(m,h,k) with h+k processors.

It would be possible to implement SEARCH using binary search and MERGE by the Batcher bitonic merge algorithm. In that case each level of the tree computation would require O(log n) time and the total time would be $O(\log^2 n)$. In order to prove Theorem 2.1 we must devise a method whereby SEARCH and MERGE can be carried out in constant time. Solutions to both of these questions involve concepts from the algorithm of Cole [3] which is capable of sorting n distinct elements in time O(log n) on n processors. This algorithm is based on a binary merge sort with clever techniques for fast merging. It is structurally similar to our tree computation of $\tau_2\tau_3...\tau_n$. Indeed the processing of the array A is exactly a binary merge sort. However the elements A[1], A[2], ..., A[2n-2] are not distinct (as required by Cole's algorithm) and we must first give them a "distinct identity". We do this by attaching to each of them their original index in the array A (which they retain as the array gets permuted). This allows us to define a total order << on the multi-set of elements represented in A[1], A[2], ..., A[2n-2] by the rule A[i] << A[j] if either

(i) A[i] < A[j], or

(ii) A[i] = A[j] and the original index of A[i] is less than that of A[j].

Implementing SEARCH in constant time

We shall consider the implementation of SEARCH under the hypothesis that certain *cross-rank functions* f and g are available to COMPACT MULTIPLY. Specifically we shall assume

(a) if $m+1 \le i \le m+h$ then we know the minimal j=f(i) in [m+h+1 .. m+h+k] such that A[i] << A[j] (if there is no such j we have f(i)=m+h+k+1), and

(b) if $m+h+1 \le i \le m+h+k$ then we know the minimal j=g(i) in [m+1 .. m+h] such that A[i] << A[j] (if there is no such j we have g(i)=m+h+1).

This assumption will be justified later. The first call on SEARCH, namely

p := SEARCH(i, m+h+1, m+h+k)

can be implemented by

```
j := f(i);
if j-1≥m+h+1 and A[j-1]=A[i] then p:=j-1
else
        if j≤m+h+k and A[j]=A[i] then p:=j
        else    p:=0
```

The second and third calls on SEARCH are similar and we consider only the second one:

r := SEARCH(C[p], m+1, m+h)

This can be implemented by

```
j := g(C[p])
if $j-1 \geq m+1$ and A[j-1]=A[i] then r:=j-1
else
        if $j \leq m+h$ and A[j]=A[i] then r:=j
        else     r:=0
```

Implementing MERGE in constant time

The cross rank functions also tell us how to perform the MERGE operation. We have to merge $A[m+1] << A[m+2] << \ldots << A[m+h]$ into $A[m+h+1] << \ldots << A[m+h+k]$ and adjust the array C so that property (ii) is preserved. If $m+1 \leq i \leq m+h$, and $j=f(i)$ so that $m+h+1 \leq j \leq m+h+k$ and $A[j-1] << A[i] << A[j]$ the rank of A[i] in $A[m+1],\ldots, A[m+h+k]$ when totally ordered by $<<$ is $i+j-m-h-1$. Similarly, using the function g, we can compute the rank of A[i] if $m+h+1 \leq i \leq m+h+k$. In other words we can define an explicit permutation π of $m+1,\ldots, m+h+k$ so that the merged result $A'[m+1], A'[m+2],\ldots, A'[m+h+k]$ is defined by $A'[i] = A[\pi(i)]$.

Define $C'[i] = \pi^{-1}(C[\pi(i)])$. Then property (ii) still holds since $A'[C'[i]] = A[\pi(\pi^{-1}(C[\pi(i)]))] = A[C[\pi(i)]] = \sigma\tau(A[\pi(i)]) = \sigma\tau(A'[i])$.

All the computations above can be performed in parallel. The end result of this analysis is that, provided the cross rank functions f and g are available, then COMPACT MULTIPLY can be executed in constant time, space O(h+k), with O(h+k) processors.

It remains to explain how the cross rank functions can be obtained and for this we need to embed our tree computation of $\tau_2\tau_3\ldots.\tau_n$ within the framework of Cole's sorting algorithm [3]. This algorithm sorts n elements by repeated merging of sorted sublists arranged in a tree. If no additional information were available merging of two sorted sublists (each of size m) would require $\Omega(\log \log m)$ parallel time on m processors. Cole has demonstrated that with the help of an additional list (defined for each sublist), called a *sampler,* the merge operation can be performed in constant parallel time. The sampler for a list *L* contains some fraction of the elements from *L* (roughly) uniformly distributed over *L* and Cole showed how a sampler could be constructed in constant time at each step of the merge tree. He also showed (and for our algorithm this is the crucial fact) how the cross rank functions could be calculated from the sampler in constant time.

Each of our MERGE operations merges, in parallel, a pair of sorted sublists maintained in the array A and adjusts the array C appropriately. The merging of the sublists stored in portions of the array A uses Cole's merge operation in a direct manner. The effect of the permutation has been encoded in the array C. The array C has been chosen so that its adjustment reduces to cross-rank computations for the lists stored in corresponding portions of the array A.

Therefore if we carry out the procedures in Cole's algorithm concurrently with performing the tree structured computation of $\tau_2\tau_3....\tau_n$ we can be assured that the cross rank functions required by SEARCH and MERGE are always available. Our algorithm therefore operates using only the resources claimed.

The proof of Theorem 2.1 is now complete.

3. Binary tree generation

For sequential algorithms, Martin and Orr [7] were the first to overcome the problem of generating random binary trees in linear time without the very large numbers that typically arise from unranking algorithms (an improved sequential algorithm was given in [1]). They also posed the problem of finding an efficient parallel algorithm. The parallel algorithm developed in this section does not require numbers larger than n and is based on a constructive version of the Chung-Feller theorem on coin-tossing [2]. The original motivation was in the development of algorithms for generating random geometric objects; these geometric objects were generated for a comparison study of computational geometry algorithms [4, 6].

We shall consider words in the alphabet $\{\lambda, \rho\}$ where λ, ρ are to be thought of as the left and right bracket symbols. A word such as $\lambda\rho\rho\lambda\rho\lambda\lambda\lambda\rho\rho$ may be pictured as a zigzag diagram

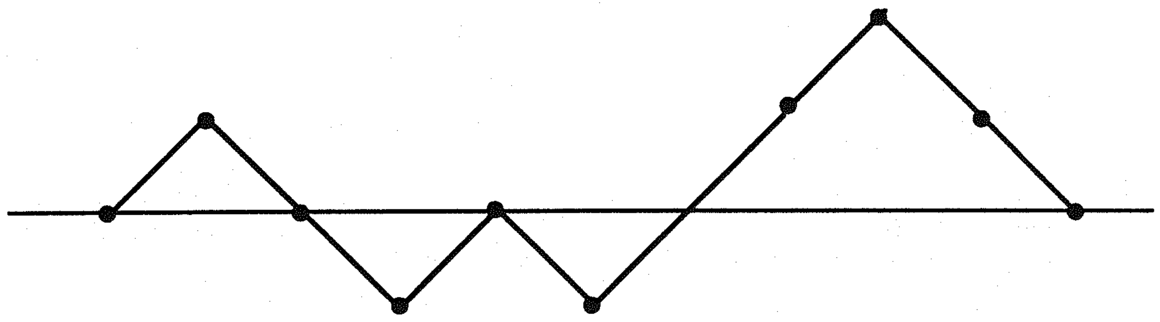

Figure 2: The zigzag diagram corresponding to the word $\lambda\rho\rho\lambda\rho\lambda\lambda\lambda\rho\rho$.

beginning on some base line where each upward edge represents λ and each downward edge represents ρ.

Definition 3.1 A word is *balanced* if it contains equal numbers of λ's and ρ's.

Definition 3.2 A balanced word has *defect i* if its diagram has precisely 2i links below its base line. Defect 0 words are called *well-formed.*

It is well known that well-formed words correspond to well-nested bracket sequences. Observe that the defect of a word is easily found by a summation: we scan the word from left to right regarding each λ as +1, each ρ as -1, and computing the partial sums; the final sum is zero and the number of negative interim odd sums is the defect. We call this calculation *partial summation.*

Let B_n denote the set of $\binom{2n}{n}$ balanced words of length 2n, and let B_{ni} denote the subset of balanced words of defect i. Clearly B_n is the disjoint union of $B_{n0}, B_{n1}, \ldots, B_{nn}$. The Chung-Feller Theorem, see [2, Theorem 2A and 5, p.94], states that these subsets all have the same size. Our algorithm uses a constructive version of this theorem exploiting explicit 1-1 correspondences between these subsets.

Our algorithm is based on the following scheme:

```
RANDOM BRACKET SEQUENCE
    Generate a uniformly random combination L of n integers from $\{1,2,\ldots,2n\}$
    Define a uniformly random member $x=(x_1x_2\ldots x_{2n})$ of $B_{2n}$ by the rule $x_i=\lambda$ if
    $i\in L$, $x_i=\rho$ if $i\notin L$
    Return the well-formed member of $B_{2n}$ to which x corresponds.
```

To implement this algorithm on a PRAM we have to know how to generate a random combination of n integers from $\{1,2,\ldots,2n\}$ and then how to compute the well-formed bracket sequence to which it corresponds. The former is accomplished by Corollary 2.1. For the latter we shall define a surjective mapping $\Psi_n:B_n\rightarrow B_{n0}$ which is bijective when restricted to any defect class; consequently since x is uniformly distributed in B_n, $\Psi_n(x)$ will be uniformly distributed in B_{n0}.

If $w \in B_n$ then λw may be written as $\lambda_0u_0\lambda_1u_1\ldots\lambda_nu_n$ where each $\lambda_i=\lambda$ and each u_i is a (possibly empty) string of ρ's. We shall regard this string as being arranged cyclically. Then there are precisely n+1 circular readings of λw which begin with a symbol λ. Let these be $\lambda w_0,\lambda w_1,\ldots,\lambda w_n$. It follows from the discussion in [8, Chapter 3, pp. 69-70] that $w=w_0,w_1,\ldots,w_n$ lie in distinct defect classes. We can therefore define $\Psi_n(w)$ to be the unique w_i which lies in B_{n0}. It follows that Ψ_n maps each of $w_0,w_1,\ldots,w_n$ onto the same member of B_{n0} and maps no other element of B_n to this well-formed sequence. Thus Ψ_n is indeed bijective on each B_{ni}. As we shall now show, $\Psi_n(w)$ can be computed in O(log n) parallel steps using n processors.

Let s_i be the partial sum of λw corresponding to the substring $\lambda_0u_0\lambda_1u_1\ldots\lambda_iu_i$, let $m=\min\{s_i|0\le i\le n\}$, and let $k=\max\{i|s_i=m\}$. It is easy to check that the circular reading of λw which begins with λ_{k+1} has w_{k+1} well-formed, so that $\Psi_n(w)=w_{k+1}$. We can compute all the partial sums of λw in O(log n) steps using n processors by a standard divide and conquer technique, compute the right-most minimum in the same way, and thereby identify k to obtain $\Psi_n(w)$ as required.

THEOREM 3.1 RANDOM BRACKET SEQUENCE is an unbiased generator for a random well formed bracket sequence with n pairs of brackets running in O(log n) time, using O(n) space and O(n) processors.

The set of well-formed bracket sequences using n bracket pairs is in one-to-one correspondence with many other sets of combinatorial objects: binary trees on n nodes, rooted (ordered) trees with n branches, triangulations of a convex (n+2)-gon, and lattice paths from (0,0) to (n,n) which do not cross the diagonal. The one-to-one correspondences are explicit and efficient to compute and so a well-formed bracket sequence can be regarded as an encoding of any of these combinatorial objects; thus a uniform random generator for well-formed bracket sequences gives rise to a uniform random generator for all these other objects, and vice versa.

COROLLARY 3.1 For any of the following combinatorial objects there is an unbiased generator with time complexity O(log n), space complexity O(n) and O(n) processors: binary trees on n nodes, rooted (ordered) trees with n branches, triangulations of a convex (n+2)-gon, and lattice paths from (0,0) to (n,n) which do not cross the diagonal.

Acknowledgement We thank Frank Dehne for several useful discussions.

References

[1] Atkinson, M.D. and Sack, J.-R.: Generating bracket sequences at random, SCS-TR-184, School of Computer Science, Carleton University, December 1990.

[2] Chung, K.L. and Feller, W.: Fluctuations in coin-tossing, Proc. Nat. Acad. Sci. USA 35 (1949), 605-608.

[3] Cole, R.: Parallel merge sort, Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science (1986), 511-516 and SIAM J. Comput. 17 (1988), 770-784.

[4] Epstein, P., Knight, A., May, J., Nguyen, T., and Sack, J.-R.: A workbench for computational geometry, SCS-TR-180, September 1990 (summary in: Proc. ACM Conf. on Computational Geometry, Berkeley (June 1990), p. 370).

[5] Feller, W.: An Introduction to Probability Theory and its Applications, vol. 1 (3rd edition), John Wiley (New York - London - Sydney), 1968.

[6] Knight A. and Sack, J.-R. : Generating Jordan sequences and comparing Jordan sorting algorithms, preprint, Computer Science, Carleton University, October 1990.

[7] Martin, H.W. and Orr, B.J.: A random binary tree generator, Computing Trends in the 1990's, ACM Seventeenth Computer Science Conference, Louisville, Kentucky (February 1989), ACM Press, 33-38.

[8] Mohanty, S.G.: Lattice Path Counting and Applications, Academic Press (New York - London), 1979.

School of Computer Science, Carleton University
List of Recent Technical Reports

SCS-TR-142	**An Algorithm for Distributed Mutual Exclusion on Arbitrary Networks** H. Glenn Brauen and John E. Neilson, September 1988
	SCS-TR-143 to 146 are unavailable.
SCS-TR-147	**On Transparently Modifying Users' Query Distributions** B.J. Oommen and D.T.H. Ng, November 1988
SCS-TR-148	**An O(N Log N) Algorithm for Computing a Link Center in a Simple Polygon** H.N. Djidjev, A. Lingas and J.-R. Sack, July 1988, revised full version Sept. 1990 Published in Proc. 6th Annual Symposium on Theoretical Aspects of Computer Science, Paderborn, FRG, February 16-18, 1989, Lecture Notes in Computer Science, Springer-Verlag No. 349, pp 96-107.
SCS-TR-149	**Smallscript: A User Programmable Framework Based on Smalltalk and Postscript** Kevin Haaland and Dave Thomas, November 1988
SCS-TR-150	**A General Design Methodology for Dictionary Machines** Frank Dehne and Nicola Santoro, February 1989
SCS-TR-151	**On Doubly Linked List ReOrganizing Heuristics** D.T.H. Ng and B. John Oommen, February 1989
SCS-TR-152	**Implementing Data Structures on a Hypercube Multiprocessor, and Applications in Parallel Computational Geometry** Frank Dehne and Andrew Rau-Chaplin, March 1989
SCS-TR-153	**The Use of Chi-Squared Statistics in Determining Dependence Trees** R.S. Valiveti and B.J. Oommen, March 1989
SCS-TR-154	**Ideal List Organization for Stationary Environments** B. John Oommen and David T.H. Ng, March 1989
SCS-TR-155	**Hot-Spot Contention in Binary Hypercube Networks** Sivarama P. Dandamudi and Derek L. Eager, April 89
SCS-TR-156	**Some Issues in Hierarchical Interconnection Network Design** Sivarama P. Dandamudi and Derek L. Eager, April 1989
SCS-TR-157	**Discretized Pursuit Linear Reward-Inaction Automata** B.J. Oommen and Joseph K. Lanctot, April 1989
SCS-TR-158 (revised)	**Parallel Fractional Cascading on a Hypercube Multiprocessor** Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, May 1989 (Revised April 1990)
SCS-TR-159	**Epsilon-Optimal Stubborn Learning Mechanisms** J.P.R. Christensen and B.J. Oommen, June 1989
SCS-TR-160	**Disassembling Two-Dimensional Composite Parts Via Translations** Doron Nussbaum and Jörg-R. Sack, June 1989
SCS-TR-161 (revised)	**Recognizing Sources of Random Strings** R.S. Valiveti and B.J. Oommen, January 1990 Revised version of SCS-TR-161 "On the Data Analysis of Random Permutations and its Application to Source Recognition", published June 1989
SCS-TR-162	**An Adaptive Learning Solution to the Keyboard Optimization Problem** B.J. Oommen, R.S. Valiveti and J. Zgierski, October 1989
SCS-TR-163	**Finding a Central Link Segment of a Simple Polygon in O(N Log N) Time** L.G. Alexandrov, H.N. Djidjev, J.-R. Sack, October 1989
SCS-TR-164	**A Survey of Algorithms for Handling Permutation Groups** M.D. Atkinson, January 1990

SCS-TR-165	**Key Exchange Using Chebychev Polynomials** M.D. Atkinson and Vincenzo Acciaro, January 1990
SCS-TR-166	**Efficient Concurrency Control Protocols for B-tree Indexes** Ekow J. Otoo, January 1990
SCS-TR-167	**A Hierarchical Stochastic Automaton Solution to the Object Partitioning Problem** B.J. Oommen, January 1990
SCS-TR-168	**Adaptive List Organizing for Non-stationary Query Distributions. Part I: The Move-to-Front Rule** R.S. Valiveti and B.J. Oommen, January 1990
SCS-TR-169	**Trade-Offs in Non-Reversing Diameter** Hans L. Bodlaender, Gerard Tel and Nicola Santoro, February 1990
SCS-TR-170	**A Massively Parallel Knowledge-Base Server using a Hypercube Multiprocessor** Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, April 1990
SCS-TR-171	**Parallel Processing of Quad Trees on the Hypercube (and PRAM)** Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, April 1990
SCS-TR-172	**A Note on the Load Balancing Problem for Coarse Grained Hypercube Dictionary Machines** Frank Dehne and Michel Gastaldo, May 1990
SCS-TR-173	**Self-Organizing Doubly-Linked Lists** R.S. Valiveti and B.J. Oommen, May 1990
SCS-TR-174	**A Presortedness Metric for Ensembles of Data Sequences** R.S. Valiveti and B.J. Oommen, May 1990
SCS-TR-175	**Separation of Graphs of Bounded Genus** Ljudmil G. Aleksandrov and Hristo N. Djidjev, May 1990
SCS-TR-176	**Edge Separators of Planar and Outerplanar Graphs with Applications** Krzystof Diks, Hristo N. Djidjev, Ondrej Sykora and Imrich Vrto, May 1990
SCS-TR-177	**Representing Partial Orders by Polygons and Circles in the Plane** Jeffrey B. Sidney and Stuart J. Sidney, July 1990
SCS-TR-178	**Determining Stochastic Dependence for Normally Distributed Vectors Using the Chi-squared Metric** R.S. Valivetei and B.J. Oommen, July 1990
SCS-TR-179	**Parallel Algorithms for Determining K-width- Connectivity in Binary Images** Frank Dehne and Susanne E. Hambrusch, September 1990
SCS-TR-180	**A Workbench for Computational Geometry (WOCG)** P. Epstein, A. Knight, J. May, T. Nguyen, and J.-R. Sack, September 1990
SCS-TR-181	**Adaptive Linear List Reorganization under a Generalized Query System** R.S. Valiveti, B.J. Oommen and J.R. Zgierski, October 1990
SCS-TR-182	**Breaking Substitution Cyphers using Stochastic Automata** B.J. Oommen and J.R. Zgierski, October 1990
SCS-TR-183	**A New Algorithm for Testing the Regularity of a Permutation Group** V. Acciaro and M.D. Atkinson, November 1990
SCS-TR-184	**Generating Binary Trees at Random** M.D. Atkinson and J.-R. Sack, December 1990
SCS-TR-185	**Uniform Generation of Combinatorial Objects in Parallel** M.D. Atkinson and J.-R. Sack, January 1991