Beyond External Computing: Analysis of the Cycle Structure of Permutations

Jörg Keller[1] and Jop F. Sibeyn[2]

[1] FernUniversität, LG Technische Informatik II, 58084 Hagen, Germany
Joerg.Keller@fernuni-hagen.de
[2] Computing Science Department, Umeå University, Sweden
http://www.cs.umu.se/~jopsi/

Abstract. A parallel algorithm with super-linear speed-up for determining the structure of permutations of n elements for extremely large n is presented and analyzed. The algorithm uses sublinear space. If evaluating a randomly chosen successor oracle π costs c time, then a complete analysis can be performed in expected time $c \cdot (\ln n - \ln(P \cdot N)) \cdot n/P$, where P is the number of available processors, and N gives the size of the secondary memory of each of the processors. A simple refinement reduces this time by a factor $1 - 1/e$. The theoretical analyses are compared with experiments. At the current state of the technology values of n up to about 2^{48} might be handled. We also describe how to perform a screening of π in much less time. If $\pi^{\sqrt{n/(\ln n \cdot P)}}$ can be constructed, then such a screening can be performed in $\mathcal{O}(c \cdot \sqrt{\ln n \cdot n/P})$ time.

1 Introduction

Consider a permutation given by an oracle, that is, when we specify x, $0 \leq x < n$, we get back $\pi(x)$ from the oracle, but we have no further knowledge about the permutation's structure. How to compute the cycle structure of π for extremely large n? Here "computing the cycle structure of π," means generating a list that gives one element of each cycle of π together with its length. The memory consumption should be much smaller than n. For large values of n, we think of $n \geq 2^{40}$, an efficient parallel algorithm is necessary to obtain acceptable runtime.

We present a parallel algorithm to compute the cycle structure of a permutation π which is chosen from the set of all permutations on n numbers with uniform probability. If c is the time for evaluating π once, and N is the size of the secondary memory of a processor, then the expected runtime on a parallel computer with P processors is just over $c \cdot (1 - 1/e) \cdot (\ln n - \ln(P \cdot N)) \cdot n/P$. In comparison to earlier — sequential — algorithms we reduce the amount of work by about a factor of two. Most important, however, is the almost perfect parallelization. If, as is the case for a cluster of workstations, the PUs all come with their own hard disk of size N, then we even obtain super-linear speed-up. For example, for $n = 2^{48}$, $N = 2^{32}$ and $P = 2^8$, the speed-up lies close to $2 \cdot P$. The algorithms borrow ideas from some recent parallel list ranking algorithms [7, 10, 8, 11]. Any parallel algorithm that works by gradual reduction of the problem size, such as independent-set removal techniques [1], are ruled out because the structure of π is available only implicitly.

The considered problem is relevant in the domains of pseudo-random number generators (PRNGs) and cryptography. In a PRNG, the transitions from one state to the next can be considered as a permutation on the set of states. For a simple PRNG based on linear congruences, its period can be derived analytically. However, analysis of a complex PRNG is often not feasible; the period is estimated from a simplified model. Computing the cycles would reveal the real period and identify the set of unwanted start states if there are additional small cycles besides the main cycle.

In the second place, symmetric encryption with a fixed key can be viewed as a permutation on the set of codewords. A good encryption algorithm should behave like a randomly chosen permutation. One of the criteria that should be satisfied (and that cannot be tested by some simple random probing strategy) is that the cycle structure should not deviate too much from what is expected. The challenge, the size of the permutation, lies in these areas of application: for algorithms like DES and IDEA we have $n = 2^{64}$.

The following notation is used throughout the paper. For the parameters we also indicate some typical values. The algorithms are general.

P=number of proc., 2^8	n=no. of elements in structure, 2^{48}
M=size of main memory per proc., 2^{26}	π=successor function
N=size of secondary mem. per proc., 2^{32}	c=time for evaluating π once

Here M and N are given in words (of size $\log n$). The processing units, *PU*s, are indexed from 0 through $P-1$. The index of element x, $0 \leq x < n$, will also be written as a pair (i, j), with $j = \lfloor x/P \rfloor$ and $i = x \bmod P$. Two such pairs are compared in a reversed lexicographical way. That is, $(i, j) < (i', j')$ when $j < j'$ or when $j = j'$ and $i < i'$. PU k, $0 \leq k < P$, is taking care of the indices (k, j), for all $0 \leq j < n/P$.

2 The Importance of Permuting

Permuting the elements of an array is trivial when it is allowed to use a second array. If space considerations make this undesirable, one should consider *in-situ* algorithms, which require only $o(n)$ additional memory. The known sequential in-situ permutation algorithms [2, 4] are *cycle leader* algorithms: first a unique *leader* element is found for every cycle of the permutation. Normally, these leaders are the smallest elements on their cycles. Once the leaders are found, performing the actual permutations takes only $\mathcal{O}(n)$ time. So the time for the whole algorithm is determined by the time for finding the leaders.

Checking whether i is a leader normally proceeds by evaluating $j = \pi^k(i)$, for $k = 1, 2, \ldots$, until $j = i$ or until additional tests indicate that i cannot be a leader. The algorithms differ in their implementations of these tests. The simplest test is $j < i$: if this happens, then i cannot be the smallest element on the cycle, so it cannot be a leader. This test does not help to reduce the worst-case running time below the $\mathcal{O}(n^2)$ achieved by the trivial algorithm, but if each permutation is equally likely then the average runtime is bounded by $\mathcal{O}(n \cdot \log n)$ [4]. The algorithm requires $\mathcal{O}(\log n)$ bits of storage. Storing b additional bits, Melville [6] reduces the worst-case time to $\mathcal{O}(n^2/b)$. For very large n, this worst-case time becomes interesting only for excessive b. In the algorithm by Fich et. al. [2] a hierarchy of local minima is constructed while proceeding along the cycle until it finds that either i is the global minimum on the cycle, or that i is not a leader. This algorithm has $\mathcal{O}(n \cdot \log n)$ worst-case runtime. The algorithm requires storage for $\mathcal{O}(\log n)$ integers, that is for $\mathcal{O}(\log^2 n)$ bits, which is a feasible amount, even for large n.

All these algorithms can be easily parallelized: the n tests whether the elements are leaders are distributed over the available processors. However, because the expected length of the longest cycle in a randomly chosen permutation is linear in n (approximately $0.624{\cdot}n$ [9, p. 358]), such straight-forward parallelizations lead to at most $\mathcal{O}(\log n)$ speed-up for any P (taking into account the leading constants of the algorithms, it can be estimated that even for $n = 2^{64}$ the speed-up can never exceed 33). To our knowledge, there is no known way to parallelize the check for a single leader, as methods like pointer doubling cannot be applied. Hence, there is no hope for an efficient parallel algorithm based on the cycle leader algorithms. Hagerup and Keller [3] give a PRAM algorithm for the insitu permutation problem with worst case runtime $\mathcal{O}(n/P + \log n \cdot \log\log(n/P))$, but, as it needs n additional bits of storage, it is not applicable to our problem.

3 Complete Analysis of Structure

We first consider the parallel algorithm for completely evaluating the cycle structure of π. Clearly any such algorithm requires that π is evaluated at least n times, which may take too long. In that case the structure can only be analyzed partially. If n evaluations are acceptable, however, then our algorithm can be performed, because in practice, applying several refinements, the total number of evaluations is bounded by about $5{\cdot}n$, perfectly distributed over the processors.

3.1 Basic Algorithm

The basic algorithm minimizes the communication. It consists of four phases. The first and the third are performed locally without communication, the second is very similar to a parallel-external list-ranking problem, for which efficient algorithms have been designed before, both in theory and in practice [5, 8, 11]. The fourth is a gather operation.

In Phase 1, every PU k, $0 \leq k < P$, performs the following steps in parallel. Next to some local variables, each PU has arrays *nxt* and *dst* for storing a remote successor of a node and the distance thereto. These arrays have length N each (assuming that secondary memory can hold $2 \cdot N$ integers). The elements (k, j) with $0 \leq k < P$ and $j \leq N$ serve as a set of "starting points". PU k, $0 \leq k < P$, follows the cycle going through starting point (k, j), for all $0 \leq j < N$, until it reaches another starting point.

Algorithm Phase 1

```
1. Set $cur_k = 0$.
2. Set $(i,j) = (k, cur_k)$ and $cnt_k = 0$.
3. Repeat $(i,j) = \pi(i,j)$ and $cnt_k = cnt_k + 1$, until $j < N$.
4. Set $nxt_k(cur_k) = (i,j)$ and $dst_k(cur_k) = cnt_k$.
5. $cur_k = cur_k + 1$. If $cur_k < N$ then goto Step 2.
```

The expected number of iterations in Step 3 is clearly $n/(P \cdot N)$ (because in a random experiment with a probability of success equal to p, the expected number of trials until the first success equals $1/p$). So, the total expected time consumption for this phase equals $c \cdot n/P$. For all but specially constructed pathological π the actual value will be only slightly larger. The experimental results in Section 5 confirm this.

As a result of the operations in Phase 1, we have constructed a set of links spanning most of the cycles. The total number of these links is $P \cdot N$. In Phase 2, we compute the lengths of these cycles.

Algorithm Phase 2

```
1. Perform the "peeling-off" algorithm from [11] to determine the minimum
   element on every cycle and the distance thereof. If such a minimum element
   is found, then the cycle statistics are updated.
2. The additional arrays are no longer needed and can be discarded.
```

All operations can be performed with $\mathcal{O}(N)$ work, paging and routing per PU. Applying the indicated optimized implementation or one of the others, the constants are small. For example, the total amount of data a PU is sending is about $8 \cdot N$. In comparison to the cost of Phase 1, and the even higher cost of Phase 3, this is negligible, even though integers cannot be pumped into the network at clock rate.

After phase 2 we have found all cycles that contain at least one starting point. Phase 3 serves to find the remaining cycles. It is similar to Phase 1, the main difference being another stopping criterion. Each PU has its own local cycle statistics which are set to zero at the beginning of Phase 3. For every PU k, $0 \leq k < P$, the following steps are performed in parallel. PU k starts from all elements (k, cur_k) which are not starting points, i.e. where $cur_k \geq N$, because they might lie on a cycle without starting point. The cycle is followed until either a starting point is reached, in which case the cycle is already detected, or until (k, cur_k) is reached again, in which case (k, cur_k) is the cycle leader, or until $(i,j) < (k, cur_k)$ is reached in which case (k, cur_k) cannot be the cycle leader.

Algorithm Phase 3

```
1. Set $cur_k = N$.
2. Set $(i,j) = (k, cur_k)$ and $cnt_k = 0$.
3. Repeat $(i,j) = \pi(i,j)$ and $cnt_k = cnt_k + 1$, until $(i,j) \leq (k, cur_k)$.
4. If $(i,j) = (k, cur_k)$, then we have found a new cycle. Update the local cycle
   statistics, using $cnt_k$.
5. $cur_k = cur_k + 1$. If $cur_k < (n/P)$ then goto Step 2.
```

At the end we have local cycle statistics, in which the PUs recorded the cycles they have discovered. In Phase 4, the PUs perform a logarithmic-depth reduction with the effect that finally the union of everything that was discovered comes to stand in PU 0, who can output the result. If π is indeed more or less random, then the time for this Phase 4 is very small, because in that case the number of cycles is bounded by $\mathcal{O}(\log n)$.

3.2 Analysis

The following analysis is not new (a slightly simpler result was proven already in [2]). We present it here for comparison purposes.

Lemma 1 *For a random oracle π, the expected number of evaluations of π performed by each of the PUs in Phase 3 is bounded by $(\ln n - \ln(P \cdot N)) \cdot n/P$.*

Proof: Starting with an index (k, l), the probability to hit a node $(i, j) \leq (k, l)$ equals $P \cdot l/n$ (actually, after i hops on a path, this probability equals $P \cdot l/(n-i)$, which is slightly larger). So, for given l, writing $p = P \cdot l/n$ and $q = 1 - p$, the expected number of evaluations is given by

$$t_l = \sum_{i=0}^{\infty}(i+1) \cdot p \cdot q^i = \sum_{j=0}^{\infty}\sum_{i=j}^{\infty} p \cdot q^i = \sum_{j=0}^{\infty} p \cdot q^j/(1-q) = \sum_{j=0}^{\infty} q^j = \frac{1}{1-q} = n/(P \cdot l).$$

Hence, if cur_k is running from N to n/P, then the total expected number of evaluations of π performed by PU_k is given by

$$\sum_{l=N}^{n/P} n/(P \cdot l) \simeq (\ln(n/P) - \ln N) \cdot n/P = (\ln n - \ln(P \cdot N)) \cdot n/P.$$

$\square$

So, in order to minimize the total costs one should indeed test the maximum possible number of indices during Phase 1. For the typical values of P, N and n that were given in Section 1, the time consumption is about $6 \cdot c \cdot n/P$. This is still a very long time, but it will become feasible with foreseeable increases in N, P and processor speed. It also makes sense to look for further reductions of the time consumption.

3.3 A Refinement

With a small modification, we can gain about 40%. The idea is to keep track of the number r of indices that were not yet "found". That is, at any given time r equals n minus the sum of the lengths of all detected cycles. Clearly a new cycle cannot be longer than r. Hence, if we are searching from some index (k, cur_k), and we did not yet hit a node (i, j) with $(i, j) \leq (k, cur_k)$ after r applications of π, we may nevertheless conclude that we are on a cycle that was discovered before and stop. Thus, we should modify Step 3 of Phase 3 as follows:

3. Repeat $(i,j) = \pi(i,j)$ and $cnt_k = cnt_k + 1$, until $(i,j) \leq (k, cur_k)$ or $cnt_k = r$.

In a sequential algorithm it would be trivial to keep r up-to-date. In a parallel algorithm this requires some extra effort. Because we expect to find only few cycles, there is no need to wait: as soon as a cycle is detected by a PU, it can immediately broadcast its length to all other PUs. Such a broadcast can be performed in $\log P$ communication rounds. The involved packets have constant size. Thus, this costs $\mathcal{O}(\log P)$ time, which is negligible in comparison to $c \cdot n/P$. Thus, we may assume, as is done in the following analysis, that r is up-to-date.

Lemma 2 *During pass l of the loop in step 3, the expected value of r is about $n/(P \cdot l)$.*

Proof: From [9] we use the fact that the expected number of cycles of length m equals $1/m$. The possible discovery of such a cycle is an independent event, so the expected number of remaining cycles of length m equals $(1-m/n)^{P \cdot l}/m$. Thus, substituting $L = P \cdot l$, the expected sum $\overline{r}_l$ of the lengths of these cycles can be estimated as follows:

$$\overline{r}_l = \sum_{m=0}^{n} (1-m/n)^L \simeq \int_0^n (1-m/n)^L \,\mathrm{d}m = \frac{-n}{L+1} \cdot (1-m/n)\Big|_0^{\mathrm{n}} = \frac{n}{L+1} \simeq n/L.$$

$\square$

By an analysis similar to the proof of Lemma 1, we obtain

Theorem 1 *For a random oracle π, the expected number of evaluations of π performed by each of the PUs in the modified Phase 3 is bounded by $(1 - 1/e) \cdot (\ln n - \ln(P \cdot N)) \cdot n/P$.*

4 Other Results

Finding the Longest Cycle. For some applications it may be sufficient to know the size L and some elements of the longest cycle. For example, it may be possible to generate several oracles, with the purpose to pick the one that maximizes L. L has expected value $0.624 \cdot n$, but the actual values fluctuate a lot. It is easy to see that the probability that $L > n - m$ is at least m/n. Thus, trying r times will mostly give us an oracle with $L > n - n/r$.

In the previous section we have reduced the running time by a constant factor, but even $c \cdot n/P$ may be beyond feasibility. In a special case a superficial screening can be performed so fast, that it even becomes possible to test many oracles. We assume that after some (maybe quite expensive) preprocessing a new oracle $\rho = \pi^x$ can be constructed, that again can be evaluated in c time[1]. In the following we will use $x = \sqrt{n/(\ln n \cdot P)}$.

[1] This assumption is e.g. valid for permutations like $a \cdot x + b \bmod m$, but not for DES.

The algorithm is simple: Each PU randomly chooses $\ln n$ starting points. From each starting point the algorithm applies x times ρ. The indices of all nodes that are visited during these jumps are stored, this information is called a track. It is most convenient if at the end of such a batch of operations all new information is gossiped, so that all PUs have access to it locally. Then for each track, the algorithm x times applies π, while testing whether a track starting from another starting point already covered this sector. Because such a track has left traces every x nodes, this will certainly be detected. If this happens, then this track is not further extended. These operations are repeated until all tracks got stuck. The amount of double work is very small.

Theorem 2 *The described algorithm has running time $\mathcal{O}(c \cdot \sqrt{\ln n \cdot n/P})$. Afterwards, for any given seed it can be tested in $c \cdot \sqrt{n/(\ln n \cdot P)}$ time whether it lies on a sufficiently long cycle or not.*

Proof: Because the number of evaluations of π equals the number of evaluations of ρ, we only need to estimate the latter. Because each PU essentially covers only the gaps between its starting points and those that follow, the work can be estimated on the sum of the sizes of these gaps multiplied by c/x, which gives $\mathcal{O}(c \cdot n/(P \cdot x))$. We must add $\mathcal{O}(c \cdot x)$ for each starting point, thus the total time consumption for each PU is given by

$$\mathcal{O}(c \cdot n/(P \cdot x) + c \cdot x \cdot \ln n) = \mathcal{O}(c \cdot \sqrt{\ln n \cdot n/P}).$$

The length of the cycles can be found by knotting all tracks together. If this should be used as a random generator, then for a seed s it can be tested by performing $\sqrt{n/(\ln n \cdot P)}$ times π whether s lies on a sufficiently long cycle. $\square$

Very Short Cycles. Cycles with length up to l can trivially be detected with $l \cdot n/P$ evaluations of π per PU.

Tree Like Structures. More interesting are oracles that only guarantee outdegree 1, but for which a node may be the successor of several nodes. In that case, the given algorithm may not terminate, because there may be cycles of nodes with high indices to which is attached a tail containing a node with low index. The solution is to throw anchors every now and then, and stopping as soon as an anchor is reached. A possibility is to throw anchors after a number of evaluations that is doubled every time. In this way at most twice as many evaluations are performed as necessary.

Testing on Randomness. For some applications there is no need to find all cycles. In such cases a cheaper screening will suffice. Here, we consider omitting Phase 3. Then, the algorithm runs in just over $c \cdot n/P$ time. In the following we work under the assumption that π is random. If it is not, then it may either still behave as assumed, or it does not, and in that case it will be detected.

Lemma 3 *After Phase 1 and Phase 2, the expected fraction of the cycles of length m that has been found equals $1 - (1 - m/n)^{P \cdot N} \simeq 1 - e^{m \cdot P \cdot N/n}$. For $m \geq n/(P \cdot N) \cdot \ln n$ all cycles have been found, with high probability.*

Proof: For any given cycle of length m, the probability that none of its elements is among the lowest $P \cdot N$, equals $(1 - m/n)^{P \cdot N}$. Substituting $m = n/(P \cdot N) \cdot \ln n$ gives $1/n$. As, for random π there are only $\mathcal{O}(\ln n)$ cycles in total, the probability that any of them is not detected is hardly larger. $\square$

5 Experimental Results

We have implemented a sequential simulation of the basic algorithm together with the simple improvement from Section 3.3 in C. We have run this program on a computer with sufficient internal memory for the size of the experiments that we wanted to perform. So, the random permutations could be generated easily and were stored in arrays. We were not interested in time, but in the number of evaluations made by each PU. Other points of interest are the occurring fluctuations in the numbers of cycles and the load balancing among the PUs.

$\log n$	$\log(P \cdot N)$	*tests*	*Cm*	*Ca*	*CM*	*Ta*	*TM*	*ta*	*tM*
16	4	10000	2	11.66	26	309647	535040	463808	913248
16	8	10000	2	11.67	26	223927	395357	257792	452864
16	12	100000	1	11.67	28	134846	240286	142400	252576
20	4	2000	5	14.49	27	6451451	10409239	8947232	14887648
20	8	2000	3	14.41	27	5023912	8647220	5561552	9275280
20	12	10000	3	14.41	30	3587918	6467906	3714624	6593576
20	16	10000	4	14.47	30	2157791	3855698	2187328	3884932
24	8	1000	6	17.27	32	103595973	170821183	112231808	183887984
24	12	1000	5	17.39	31	90363923	135110679	82525232	137614384
24	16	1000	6	17.12	30	57065471	98106800	57659024	98444288
24	20	1000	4	17.21	34	34271603	58610751	34389440	58816288
28	12	100	11	19.31	31	1579220642	2371955835	1608459888	2413276912
28	16	100	10	20.35	29	1270724358	1978054028	1278394896	1985230560
28	20	100	9	19.84	30	880985077	1468056382	882947248	1470623392
28	24	500	7	19.50	32	542089349	971220637	542567680	972277488

Table 1. Results of experiments. The columns give $\log_2 n$, $\log_2(P \cdot N)$, the number of tests, the minimum number of cycles that was found, the average number and the maximum number. *Ta* is the average value of the total number of evaluations divided by P. *TM* gives the corresponding maximum number. *ta* is the average of the maximum number of evaluations performed by any of the (virtual) PUs. *tM* gives the corresponding maximum.

We have tested for $n = 2^{16}, 2^{20}, 2^{24}, 2^{28}$. For larger n the main problem is that the time consumption does not allow to run sufficiently many tests to draw reliable conclusions. All these n have been investigated for $P \cdot N = n/2^4, n/2^8, n/2^{12}, n/2^{16}$. We have set $P = 16$, this has no implications for the number of evaluations. All results are given in Table 1. We have performed a least square adaptation for determining the values of β and γ in the number T of evaluations per PU:

$$T = (\beta + \gamma \cdot (\ln n - \ln(P \cdot N)) \cdot n/P.$$

The best fit is obtained with $\beta = 0.710$ and $\gamma = 0.486$. These values are smaller than expected, which might be due to the positive effect of stopping when $r = 0$. For all instances with $n \leq 2^{24}$, the deviation is less than 1%. For $n = 2^{28}$, the deviation is slightly larger, due to an insufficient number of experiments.

For each n, the average number of cycles lies close to the value it should have: the harmonic numbers $H_n = \sum_{i=1}^{n} 1/i$. The deviation is at most a few percent and due to an insufficient number of tests.

The maximum number of cycles is bounded by about twice the expected number, though the deviation appears to decrease for larger n. The minimum number of cycles may be very small. For larger n it increases. This is due to two effects: for larger n the probability on very few cycles decreases considerably (the probability that a permutation consists of exactly one cycle equals $1/n$); for large n we could not perform many tests. The experiments do not allow to draw any final conclusion, but it appears that the distribution around the expected value is more or less symmetrical. For example, for $n = 2^{28}$, the expected value is close to 20, and permutations with less than 10 cycles are about equally frequent as permutations with more than 30 cycles (1 or 2 on every 100 permutations).

From the difference between the average and maximum number of evaluations we see that the actual running times may fluctuate a lot (the maximum is up to 70% higher than the average, and exceeds the minimum even by a factor three). There is some correlation with the number of cycles (if there are many cycles, then probably r will be large during the experiment and the last cycle will be found late), but we have seen many exceptions.

Comparing the total number of evaluations divided by P and the maximum number of evaluations performed by any PU, we see that there may be a considerable difference for the smallest n, which are not relevant in real applications. For the larger n, this difference decreases to 1% and less. This is so little, that there is no need for additional load balancing.

In order to estimate c, we have tested the pseudo-random number generator function `lrand48`. On a workstation with a Pentium II processor running at 333 MHz, FreeBSD operating system and GNU C compiler, we found that about 2^{29} evaluations were performed per minute. Hence, for $n = 2^{48}$, n evaluations would take about 2^{19} minutes or 364 days of processing time. This means, that with sufficiently many processors (which by now are several times faster), a problem of this size may be tackled in a few days.

6 Open Problems and Further Work

Future work will focus on a message-passing implementation for a workstation cluster or multiprocessor to validate our preliminary results. Another permutation to be tested will be the DES encryption algorithm with a fixed key. The number of elements in this permutation is larger than in the `lrand48` case by a factor of 2^{16}. Also, evaluation of the DES permutation will take much longer than evaluating the `lrand48` permutation. It appears that with the sketched improvements we have fully exploited the available information and memory. Only by storing data more compactly some further gains are possible, but this will not change things dramatically. Hence, there is a need for faster evaluation. A very interesting possibility is to use programmable hardware such as field-programmable gate arrays (FPGAs). In the simplest case, these devices are added to the processors as boards in the cluster scenario. In the case of DES, this could lead to a performance improvement by more than a magnitude. Also, the control flow in our algorithm is quite regular. Thus in a single FPGA, several processing instances to solve the problem could be implemented and multiple FPGAs could be used. It could thus be possible to construct a dedicated FPGA-based machine to solve the cycle-structure problem.

References

1. Cole, R., U. Vishkin, 'Approximate Parallel Scheduling, Part I: the Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time,' *SIAM Journal on Computing*, 17(1), pp. 128–142, 1988.
2. Fich, F.E., J.I. Munro, P.V. Poblete, 'Permuting in Place,' *SIAM Journal on Computing*, 24(2), pp. 266–278, 1995.
3. Hagerup, T., J. Keller, 'Fast Parallel Permutation Algorithms,' *Parallel Processing Letters*, 5(2), pp. 139–148, 1995.
4. Knuth, D.E., 'Mathematical Analysis of Algorithms,' *Proc. of IFIP Congress 1971*, Information Processing 71, pp. 19–27, North-Holland Publ. Co., 1972.
5. Lambert, O., J.F. Sibeyn, 'Parallel and External List Ranking and Connected Components on a Cluster of Workstations,' *Proc. 11th International Conference Parallel and Distributed Computing and Systems*, pp. 454–460, IASTED, 1999.
6. Melville, R.C., 'A Time-Space Tradeoff for In-Place Array Permutation,' *Journal of Algorithms*, 2(2), pp. 139–143, 1981.
7. Reid-Miller, M., 'List Ranking and List Scan on the Cray C-90,' *Journal of Computer and System Sciences*, 53(3), pp. 344–356, 1996.
8. Ranade, A., 'A Simple Optimal List Ranking Algorithm,' *Proc. of 5th High Performance Computing*, Tata McGraw-Hill Publishing Company, 1998.
9. Sedgewick, R., Ph. Flajolet, *An Introduction to the Analysis of Algorithms.* Addison Wesley, Reading, Mass., 1996.
10. Sibeyn, J.F., 'One-by-One Cleaning for Practical Parallel List Ranking,' to appear in *Algorithmica.* Preliminary version in *Proc. 9th Symposium on Parallel Algorithms and Architectures*, pp. 221-230, ACM, 1997.
11. Sibeyn, J.F., 'Ultimate Parallel List Ranking,' *Techn. Rep. MPI-I-99-1005*, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1999.