A parallel algorithm to compute a permutation's cycle structure

Jörg Keller[*]
FernUniversität Hagen
LG Technische Informatik II
58084 Hagen, Germany

Abstract

We consider the parallel computation of the cycle structure, i.e. the length and a representant for each cycle, for a permutation on n elements which is given as an oracle. While sequential algorithms for this problem can be derived from algorithms for in-situ permutation of an array, those cannot be efficiently parallelized. We present a parallel algorithm with expected work $O(n \log n)$, transferring ideas from parallel list ranking. We present experimental results from an implementation, and discuss applications in the field of cryptanalysis.

1 Introduction

We consider the task of computing the cycle structure of a permutation π on n numbers. The permutation is given as an oracle, that is, when we specify x, we get back $\pi(x)$ from the oracle, but have no further knowledge about the permutation's structure. By the cycle structure we mean that for each cycle of the permutation, we get its length and one element of the cycle.

Sequential algorithms for this task can be derived from algorithms for in-situ permutation of arrays, because all known algorithms for this problem permute an array following the cycles of the permutation [1]. However, for large values of n such as $n \geq 2^{40}$, a parallel algorithm is necessary to keep the runtime in an acceptable range. We will show that parallelized versions of the algorithms for in-situ permutation of an array will only show very small speedup for randomly chosen permutations no matter how many processors we are willing to spend.

Therefore, we present a new algorithm to compute the cycle structure of a permutation. The expected runtime of a sequential version is $O(n \log n)$, as is the case for previous algorithms. The

[*]Phone/Fax +49–2331–987–376/308, Email `joerg.keller@fernuni-hagen.de`

parallel version exhibits much larger speedups than previous algorithms. The algorithms borrows ideas from parallel list ranking algorithms [5, 7]. The main difference between our algorithm and list ranking is the fact that a list to be ranked typically is stored in an array and thus changes to the list by pointer doubling are possible. In our case, the structure is only implicitly given by questioning the oracle. Therefore, changes to the permutation are not possible.

We see applications of our algorithm in the field of cryptography. First, consider pseudorandom number generators (PRNGs). The transition from the current state to the next state 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 tractable. Then, one guesses the actual period from a simplified model that can be analyzed. Computation of the cycles would reveal the real period and identify the set of unwanted start states if there are additional small cycles beside the main cycle. Second, 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. This can be tested by looking at the permutation's cycles. Sedgewick and Flajolet [6] give results on random permutations such as average length of the longest cycle, average number of cycles, and so on. If, for several keys, the permutations show a cycle structure different from that of a random permutation, then the encryption algorithm might have some weaknesses. One challenge that lies in this application field is the size of the permutation. For algorithms like DES and IDEA, $n = 2^{64}$.

The remainder of the paper is organized as follows. In Section 2, we briefly review known algorithms for in-situ permutation of arrays and show why they cannot efficiently be parallelized. In Section 3, we present a new algorithm for computing a permutation's cycle structure in sequential and parallelized versions. In Section 4, we give experimental results of an implementation. In Section 5, we discuss open problems and give a preview on further work.

2 Algorithms for In-Situ Permutation of Arrays

The task to permute an array is trivial when one is allowed to use a second array. This case is rather seldom for large n because of memory restrictions. Instead, one considers *in-situ* permutation of an array, that is, permutation without a second array. This problem has first been investigated by Knuth [4] and has been further studied by Fich et al. [1].

All known sequential algorithms that solve this problem are *cycle leader* algorithms, that is, they try to find, for each cycle of the permutation, a unique element called leader. Normally, the leader is the smallest element on the cycle. Then the algorithms permute each cycle separately, starting from the leader. The structure of the algorithms is given in Figure 1. The leaders are found by checking with a procedure `IsLeader()` all elements from 0 to $n-1$ whether they are leaders.

As the complete array must be permuted, the runtime of all calls to `PermuteCycle` adds up

```
void *array[n]; /* the array to be permuted */

extern int pi(int x); /* permutation as oracle */

extern in IsLeader(int i);

void PermuteCycle(int i)
{ int j; /* loop variable */

  j=i;
  do{ /* follow cycle */
   j=pi(j);
   exchange(i,j); /* exchange a[i] and a[j] */
  } while (j!=i);
}

void main(void)
{ int i; /* loop variable */

  for(i=0;i<n;i++)
   if(IsLeader(i)) PermuteCycle(i);
}
```

Figure 1: The structure of cycle leader permutation algorithms

to $\Theta(n)$. However, all known algorithms needing only polylogarithmic space beside storing the array need time $\Omega(n \log n)$ [1]. Hence, the majority of time is spent in the calls to `IsLeader`.

A check whether i is a leader normally proceeds along the cycle starting in i, testing i against $j = \pi^k(i)$ for $k = 1, 2, \ldots$ with procedure `TestHypo()`, until i is reached again or until the assumption that i is a leader turns out to be false, see Figure 2.

The algorithms differ in their implementations of `TestHypo(j,i)`, which give different runtime and space requirements. The simplest test is of the form $i \leq j$. If j is smaller than i, then i cannot be the smallest element on the cycle. Hence i is not a leader. This test leads to a worst case runtime of $O(n^2)$, and to an average runtime of $O(n \log n)$, if each permutation is equally likely [4]. Another algorithm uses $b \leq n$ bits to store information on whether certain elements have already been visited. It has a worst case runtime of $O(n^2/b)$ and an average runtime of $O(n \log n)$ [1]. Note that this algorithm can achieve linear runtime for $b = \Omega(n)$ but at the expense of a huge storage requirement. The algorithm by Fich et. al. constructs a hierarchy of local minima while proceeding along the cycle until it finds that either i is the global minimum, that is, the smallest element of the cycle, or that i is not a leader. This algorithm has runtime $O(n \log n)$ in the worst case.

It is also interesting to consider the space requirements of the algorithms as we target large values of n. We do not count the space used for the array as in our application, there is no array and the permutation is not stored explicitly. The latter is an important fact as for $n = 2^{64}$, storing the permutation as a list of function values would require 2^{67} bytes!

The simplest algorithm only needs a constant number of counters and indices, thus it consumes $O(\log n)$ bits beside storing the array. The second algorithm needs b additional bits and thus $O(b + \log n)$ in total. For $n = 2^{64}$, the applicability of this algorithm is restricted to $b \leq \sqrt{n} = 2^{32}$ if we set an upper bound of 512 Megabyte on memory consumption. Therefore, its worst case linear runtime cannot be used in such a setting. The algorithm by Fich et. al. uses up to $O(\log n)$ counters and indices as the hierarchy can have at most $\log n$ levels. Thus it uses at most $O(\log^2 n)$ bits.

All of these algorithms can be easily parallelized, no matter whether we think about a message-passing or a shared-memory computation model. One distributes the iterations of the `for` loop in function `main` over the available processors. Even if this distribution is static, the large number of iterations per processor promises — at first glance — a balanced load without further measures.

However, the speedup to be expected is quite small on the average, as the average length of the longest cycle in a randomly chosen permutation is quite large, namely approximately $0.63 \cdot n$ [6, p. 358]. We consider the number of evaluations of π as a measure for runtime. The runtime of the parallelized algorithm will be at least as large as the number of evaluations of π in function `IsLeader` when called with the cycle leader of the longest cycle, namely $0.63 \cdot n$. If the number of evaluations of π in the sequential implementation is $c \cdot n \cdot \log n$, then the speedup to be expected

```
extern void *array[n]; /* the array to be permuted */

extern int pi(int x); /* permutation as oracle */

int TestHypo(int j,int i)
{ return i<=j; }

int IsLeader(int i)
{ int j; /* loop variable */
  int flag;

  flag=1; /* assume i is leader */
  j=i;
  do{ /* follow cycle */
   j=pi(j);
   if(!TestHypo(j,i)){ /* until assumption turns out false */
    flag=0; break;
   }
  }while(j!=i); /* or cycle is completely traversed */

  return flag;
}
```

Figure 2: The structure of cycle leader checks

is

$$S = \frac{c \cdot n \cdot \log n}{0.63 \cdot n} = \frac{c}{0.63} \cdot \log n \ .$$

We have implemented sequential versions of the three algorithms described. The implementation of Knuth's algorithm is the fastest on the average with $c \approx 0.33$. For comparison, $c \approx 1$ for the algorithm by Fich et. al. Hence, the speedup of any parallelized version will be smaller than $0.52 \cdot \log n$, which is only 33 for $n = 2^{64}$, no matter how many processing elements we have.

To our knowledge, there is no known way to parallelize function `IsLeader` itself as methods like pointer doubling cannot be applied. Hence, there is no hope for a more efficient parallel algorithm based on the cycle leader algorithms above. While Hagerup and Keller [2] give a PRAM algorithm for the in-situ permutation problem, their algorithm needs $b = n$ additional bits of storage and thus is not applicable to our problem.

3 A Parallel Algorithm for Cycle Structure Computation

In order to obtain an efficient parallel algorithm, it is necessary that no long cycle has to be traversed completely by one processor. A similar problem occurs in parallel list ranking. There no long part of the list may be traversed by one processor as well. Parallel list ranking algorithms therefore split the list into pieces by choosing some elements as so-called *rulers* or *anchors*. The processors start traversing the list from the anchors, and stop when they reach another anchor (or the end of the list) [5, 7], which leads to very good speedups with high probability.

We borrow this idea and obtain the following algorithm:

0. We choose r elements as anchors.

1. For each anchor a, we follow the cycle starting in a until we reach another anchor a'. We store (a, a') in a list together with the number of elements we visited between a and a'.

2. The list now represents a permutation on the a anchors, with the difference that the distance between two anchors is not 1 as in an ordinary permutation but is explicitly given as an integer. Each cycle in this permutation represents a cycle in the original permutation, and the sum of the distances equals the length of the original cycle.

   We solve this reduced problem with one of the sequential algorithms. We obtain the length and a representant of each cycle that contains at least one anchor. The representant is not necessarily the smallest element, but the smallest anchor on the cycle.

3. If the sum of the lengths of the cycles found so far equals n, then we are done. Otherwise, we need to find all cycles that do not contain an anchor. To do this, we check each non-anchor i whether it is the leader of such a cycle. We follow the cycle starting in i until we find an

anchor, an element $j < i$, or if we meet i again. In the last case, i is the cycle leader of a cycle not containing an anchor.

The code for the algorithm is shown in Figure 3.

Ideally, anchors should be chosen randomly. However, this makes detection whether an element is an anchor costly. Ranade [5] partitions the n elements into r intervals of equal size, i.e. $0, \ldots, n/r - 1$, then $n/r, \ldots, 2n/r - 1$, and so on. In each interval, one anchor is chosen randomly. This allows detection of an anchor in a constant number of steps (if each processing element possesses the complete list of the anchors). His algorithm has the same complexity as previous algorithms for parallel list ranking, only the analysis is more tricky. For a randomly chosen permutation however, even a deterministic choice of the anchors, such as multiples of n/r, should work equally well.

We now assume that we can decide in a constant number of steps whether an element is an anchor. Then, phase 0 has work at most $O(r) = O(n)$. Phase 1 has work $O(n)$ as each element is visited at most once. Phase 2 has work $O(r^2)$ in the worst case and $O(r \log r)$ on the average if we use Knuth's algorithm. Phase 3 has work $O(n^2)$ in the worst case and $O(n \log n)$ on the average, as it equals Knuth's algorithm in the worst case.

For $r = O(\sqrt{n})$, we obtain work $O(n \log n)$ on the average and work $O(n^2)$ in the worst case. Hence, the sequential algorithm has the same average case complexity as previous algorithms.

In contrast to previous algorithms, our algorithm promises more efficiency for a parallelization. If we assume deterministic choice of the anchors, then phase 0 needs constant time. The loop in phase 1 can be easily parallelized by a static mapping of iterations to processors. As no data are shared, the parallelization works equally well for shared-memory and distributed-memory machines. There is a large probability that the speedup is very good, because the probability is small that there is a cycle of length $\alpha \cdot n$ with only one anchor. The probability is

$$\frac{1}{\alpha \cdot n} \cdot \left(\frac{1}{(1-\alpha)n} \right)^{r-1}$$

Phase 3 will also show a good speedup as the probability is small that a long cycle has not been found in phase 1 and hence must be completely traversed in phase 3.

Our algorithm needs space $O(r \cdot \log n)$, as we need to store the permutation over the anchors explicitly.

Note that the algorithm can be accelerated with several heuristics that use knowledge about the number of elements not yet visited in function `TestHypo` to stop searches earlier [3].

The exact analysis of the algorithm's behaviour is subject to future work. For now, we support our arguments by an implementation.

```
extern int pi(int x); /* permutation as oracle */

struct Anchorlist{ /* list of anchors */
int next; /* next anchor found */
int length; /* number of elements visited */
} alist[n];

void FindNextAnchor(int i)
{ int j; /* loop variable */
  int length=0; /* counter */

  j=GetAnchorValue(i); /* deterministic: j=i*n/r */
  do{ /* follow cycle */
   j=pi(j); length++;
  } while (IsNotAnchor(j)); /* until anchor is reached */
  j=AnchorNum(j); /* deterministic: AnchorNum= j / (n/r) */
  alist[i].next=j; alist[i].length=length; /* append to list */
}

int IsAnchorLeader(int i)
{ int j; /* loop variable */
  int flag;
  int length=0; /* counter */

  flag=1; /* assume i is leader */
  j=i;
  do{ /* follow cycle */
   length+=anchorlist[j].length; /* is length++ in IsLeader() */
   j=anchorlist[j].next; /* is j=pi(j) in IsLeader() */
   if(!(i<=j)){ /* until assumption turns out false */
    flag=0; break;
   }
  }while(j!=i); /* or cycle is completely traversed */
  /* return cycle length if i is leader, 0 otherwise */
  if(flag) return length; else return 0;
}

int TestHypo(int j,int i)
{ return ((i<=j) && IsNotAnchor(j)); }

int IsLeader(int i)
{ int j; /* loop variable */
  int flag;
  int length=0; /* counter */

  flag=1; /* assume i is leader */
  j=i;
  do{ /* follow cycle */
   j=pi(j); length++;
   if(!TestHypo(j,i)){ /* until assumption turns out false */
    flag=0; break;
   }
  }while(j!=i); /* or cycle is completely traversed */
  /* return cycle length if i is leader, 0 otherwise */
  if(flag) return length; else return 0;
}

void main(void)
{ int i; /* loop variable */
  int length; /* cycle lengths */

 /* phase 0: choose r anchors, empty with deterministic choice */
 for(i=0;i<r;i++) FindNextAnchor(i); /* phase 1 */
 for(i=0;i<r;i++) /* phase 2 */
  if(length=IsAnchorLeader(i)) printf("Leader %d, length %d\n",i,length);
 for(i=0;i<n;i++) /* phase 3 */
  if(IsNotAnchor(i))
   if(length=IsLeader(i)) printf("Leader %d, length %d\n",i,length);
}
```

Figure 3: New algorithm to compute a permutation’s cycle structure

4 Experimental Results

We implemented our algorithm on a workstation with a Pentium II processor running at 333 MHz, with FreeBSD operating system and GNU C compiler. We tested on the one hand randomly chosen permutations for $n = 2^{18}, \ldots, 2^{26}$, and on the other hand used the `lrand48` pseudorandom number generator function as permutation on $n = 2^{48}$ elements. This permutation has only one cycle, and hence the algorithm terminates after phase 2. In all cases, we used $r = \sqrt{n}$ anchors.

The randomly chosen permutations were stored as an array. Hence, the available main memory gave an upper bound on n. We created the permutations with the procedure given in [6, Prog. 6.1] for randomly permuting an array. Here, the array was initialized with `pi[i]=i`. We used `lrand48` as pseudo random number generator and used multiples of 100 as seeds.

To measure execution time, we used the number of calls to `pi` instead of wall-clock time in order to abstract from unwanted influences on execution time such as cache behaviour, which might have significant impact depending on the permutation. We also felt that neglecting the number of instructions executed in a loop iteration beside evaluation of the permutation does not hurt, as the evaluation of permutation π will often dominate the time spent in one iteration of the loops in phases 1 and 3.

We found that the average number of evaluations was about $0.35 \cdot n \cdot \log n$, over all n, in accordance with our arguments of the previous section.

We found that the speedups in phases 1 and 3 were 350 and 838, respectively, as the average over 5 permutations, when we simulated 1,024 processors. To do this, we ran the algorithm on the workstation, and counted the evaluations for each loop iteration. Then we used a cyclic static mapping of iterations to processors to compute how many evaluations each processor would have done, and how many were there in total. From these numbers, we computed the speedups. The speedup for phase 1 is much lower than for phase 3 as we counted the number of evaluations `j=alist[j].next` in phase 2 as evaluations that are mapped to processor 0. To speed up our implementation, we had replaced Knuth's algorithm in phase 2 by the faster algorithm using $b = r$ bits and thus needing only $O(r)$ evaluations.

We found that the average length of the longest cycle found was $0.84 \cdot n$ over all n, which is even larger than the number given in [6]. The average number of cycles grows slowly with n and is about 15 for $n = 2^{26}$. Note also that the average number of elements not visited in phase 1 is only about 3,500 for $n = 2^{24}$. Thus, if we are not interested in the "micro"-structure of the permutation, we can stop after phase 2 and omit phase 3.

For the `lrand48` application, we found that our sequential implementation executed about 2^{29} evaluations per minute. Hence, to complete phase 1, it would need 2^{19} minutes or 364 days. On a workstation cluster with 2^7 processors and a 35% percent efficiency, phase 1 would be completed in 8 days. Phase 2, solving the problem for 2^{24} anchors needs less than 30 minutes. It seems that following a list instead of evaluating a permutation needs more time, as the 2^{24} evaluations of π would need less than a minute. We re-implemented the application on a Compaq XP1000 workstation with an Alpha 21264 processor running at 500 MHz. After setting a few compiler switches, we obtained a performance increase by a factor of 5 compared to the Pentium-based workstation. We attribute this to the fact that the Alpha is a 64-bit microprocessor and is running at a higher frequency. For a multiprocessor such as a Cray T3E with $2^9 = 512$ Alpha microprocessors running at 750 MHz, we guess that it would take less than 7 hours to complete phases 1 and 2.

5 Open Problems and Further Work

An exact analysis of the parallel complexity of our algorithms remains to be done. Furthermore, it is not clear how many processors can be successfully employed, if the processor count reaches or even goes beyond the number of anchors.

Future work will be concentrating 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 by a factor of 2^{16} larger than in the `lrand48` case. Also, evaluation of the DES permutation will take much longer than evaluating the `lrand48` permutation. Therefore, an implementation without further optimization could lead to a runtime of more than a hundred years.

Hence, an interesting field of research will be the investigation of the performance improvement possible through the use of 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) and are used to accelerate evaluation of the permutation. 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 regularly. 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 specialized FPGA-based machine to solve the cycle-structure problem.

Acknowledgements

We are very thankful to Torben Hagerup and Jop Sibeyn for inspiring the use of anchors.

References

[1] Faith E. Fich, J. Ian Munro, and Patricio V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, April 1995.

[2] Torben Hagerup and Jörg Keller. Fast parallel permutation algorithms. *Parallel Processing Letters*, 5(2):139–148, 1995.

[3] Jörg Keller. A heuristic to accelerate in-situ permutation algorithms. Informatik-Berichte 274, FernUniversität Hagen, Germany, September 2000.

[4] Donald E. Knuth. Mathematical analysis of algorithms. In *Proc. of IFIP Congress 1971*, Information Processing 71, pages 19–27. North-Holland Publ. Co., 1972.

[5] Abhiram Ranade. A simple optimal list ranking algorithm. In *Proceedings of the 5th High Performance Computing Conference*. Tata McGraw-Hill Publ. Company, 1998.

[6] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison Wesley, Reading, Mass., 1996.

[7] Jop F. Sibeyn. Ultimate parallel list ranking? Research Report MPI-I-1999-1-005, Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany, September 1999.