Implementation and Evaluation of a Parallel-External Algorithm for Cycle Structure Computation on a PC-Cluster

Latifa Boursas Jörg Keller

FB Informatik
FernUniversität Hagen
58084 Hagen, Germany
{latifa.boursas,joerg.keller}@fernuni-hagen.de

Abstract: We report on our experiences with the implementation of a parallel algorithm to compute the cycle structure of a permutation given as an oracle. As a sub-problem, the cycle structure of a modified permutation given as a table that is partitioned over N hard disks has to be computed. While a minor point during algorithm design and analysis, we spent most time to implement and tune this particular piece of code. We present the decisions taken during implementation and give preliminary performance figures.

1 Introduction

We consider the problem of computing the cycle structure of a permutation. Given is a set S of size n and a bijective function f on S, which defines a permutation on S. For every cycle of that permutation, we want to know its length and one element on the cycle. The function f is not given by a table of its values, but as an oracle, e.g. as a piece of code that we cannot inspect. If the code is executed supplied with an argument x, we get $f(x)$ in return.

This problem is of interest in cryptology, where f may be the state-transition function of a pseudo-random number generator, of which we want to guarantee a long period, but cannot do so analytically. State-transition functions in use are e.g. symmetric block-cipher encryption algorithms[1] such as the Data Encryption Algorithm (DEA), better known as the DES [Sc95]. Also, such algorithms are a worthwhile target in themselves. An encryption algorithm (given a certain key) serves to randomize the data it encrypts. Therefore, its cycle structure (possibly averaged over several keys) should be similar to the structure of a randomly chosen permutation, of which many results are known (expected number of cycles, expected length of the longest cycle, and so on, see e.g. [FO90] for details). If for several keys its cycle structure deviates substantially from the expected values, then this

[1] The key is fixed to some value.

may be a hint to some weakness or hidden trapdoor.

For the functions f we have in mind, n is quite large: 2^{32} to 2^{128}. Therefore, a construction of the directed graph $(S, \{(x, f(x)) : x \in S\}$ and the computation of the cycle structure by methods from graph algorithms, such as pointer doubling or list-ranking, is out of scope. Also, as the problem's complexity is at least linear in n, we search for parallel solutions. We have presented a parallel algorithm for a PC-cluster in [KS01], which as a sub-problem needs to solve the cycle structure problem at a much smaller size, but with a modified permutation where the function values are stored on disk. An algorithm for this sub-problem will be an external-memory algorithm [Vi99], as the sub-problem is still too large to load all function tables into main memory. During the implementation of the algorithm, we found that this sub-problem incurred most work and needed most tuning to achieve acceptable performance. We report in this paper on the algorithm used to solve this sub-problem.

The remainder of this paper is organized as follows. In Section 2 we briefly summarize the necessary knowledge about algorithms that compute the cycle structure of a permutation. In Section 3 we present the algorithm used to solve this problem in parallel with external-memory data. In Section 4 we report on the experiments we conducted, where we used as function f a DES-variant reduced to 32 bits (originally, DES is a 64-bit algorithm). In Section 5 we conclude.

2 Cycle Structure Algorithms

Consider the set $S = \{0, \ldots, n-1\}$ and a bijective function $f : S \to S$. The function f defines a permutation on the set S. Such a permutation is well-known to consist of a number of cycles. The element with the smallest index on a cycle will be called a *cycle leader* in the following.

Algorithms to compute the cycle structure of permutations typically follow a cycle from a starting point x until they either reach their starting point x again or they find that they are on a cycle which is already detected (see [KS01] and the references therein). The targeted size of n prevents us from storing a bit vector where elements are marked that have already been visited, which would render the problem trivial. Still, Knuth already in 1971 suggested an algorithm that solves our problem[2] [Kn72]: for $x = 0, 1, 2, \ldots$ do the following: starting from x follow the cycle until you either reach x again or your reach an element smaller than x. In the former case you have detected a cycle, and your starting point x is the leader of this cycle. In the latter case, you started from an element that is not the leader of the cycle it is on. Hence the leader is smaller and has already been visited. Thus the cycle already had been detected. This simple algorithm ensures that each cycle is detected and that it is reported only once. If all bijective functions f are equally likely to occur, the average runtime of this algorithm is shown to be $O(n \log n)$.

While a straight-forward parallelization of this (or any other known) sequential algorithm

[2] Knuth actually solved the problem of permuting an array. In order to do so, he first identified the cycles, and permuted the array contents cycle by cycle.

is easy by distributing the iterations of the outer loop, the speedup that can be achieved is low for a reason which stems from the structural properties of random permutations. When the leader of the longest cycle is the starting point, then the longest cycle is tracked completely. As the expected length of the longest cycle is about $0.624 \cdot n$ [SF96], the average runtime of the parallel algorithm is $\Omega(n)$ and thus the maximum speedup is bounded by $O(\log n)$, with a small constant.

Therefore we devised a parallel algorithm which avoids the situation that one cycle has to be tracked completely by one processor [KS01]. We consider a parallel machine with P processors, each equipped with local memory of size M and a hard disk of size N. The processors communicate via message passing, i.e. the machine is a PC-cluster. The algorithm consists of four phases: In phase 1, each processor chooses N starting points, and all starting points are known to all processors[3]. For each of its starting points x, each processor follows the permutation until it meets another starting point x'. If $x' = x$, then a cycle (with exactly one starting point) has been detected. Otherwise, the triple (starting point, next starting point, distance to next starting point) is stored on the hard disk. The following pseudo-code[4] illustrates phase 1:

```
/* code for processor p, where p=0,...,P-1 */
for(i = 0;i < N;i++){
  dist = 0; xx = x = startingpoint(i,p);
  do{ x = f(x); dist++; }while(IsNotStartingpoint(x));
  if(x == xx) DetectCycle(xx,dist); else WriteToDisk(xx,x,dist);
}

/* assume n, N, P are powers of two */
/* starting points are those with only log N + log P lower
bits used, and pi applied afterwards */
#define startingpoint(i,p) (i*P+p)
/* if any bit above log N + log P is used after undoing pi,
then x is not a starting point */
#define IsNotStartingpoint(x) (x/(P*N))
```

For the cycles that contain at least one starting point, the problem is now reduced to a size at most $N \cdot P$. In phase 2, this reduced problem is solved, see next section. Phase 3 serves to detect cycles without any starting point. It uses a distributed variant of Knuth's algorithm [KS01]: The n nodes are mapped round-robin onto the processors, i.e. all nodes x with $x \bmod P = p$ are mapped onto processor p. For each node x of the $n/P - N$ nodes that are mapped to a processor and that are not starting points, this processor follows the cycle from x until it reaches either a starting-point, a node smaller than x, or x itself. In the first case, x is on a cycle with a starting-point, i.e. on a cycle that has already been detected. In the second case, x is not the leader of the cycle it is on. Only in the third case, a new cycle

[3] This can be achieved without communication by choosing starting points deterministically. This scheme can be randomized against the permutation f by using a second, randomly chosen permutation known to all processors to generate the starting points from the deterministically chosen starting points.

[4] Here the starting point xx can be computed from the processor index `p` and its index `i` in the outer loop. Therefore it would be sufficient to store only the next starting point `x` and the distance `dist` to disk.

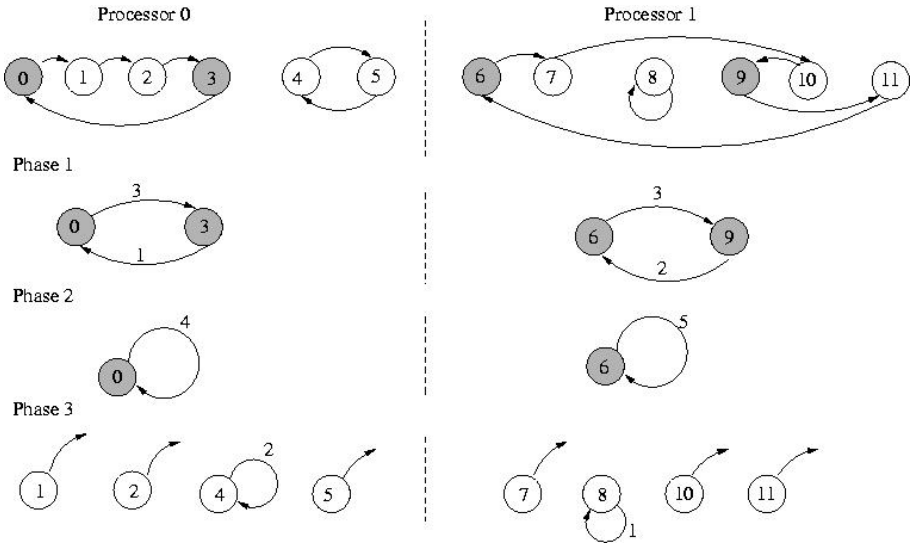

Figure 1: Algorithm from [KS01] on example permutation with $n = 12$, $N = 2$, $P = 2$.

has been detected. In Phase 4, each processor sends each cycle (more exactly: the leader and the length of this cycle) that it detected to a specified processor (typically processor 0) that gathers the results of phases 1 and 3. The following pseudo-code illustrates phase 3:

```
/* code for processor p, where p=0,...,P-1 */
for(i = N;i < n/P;i++){
  dist = 0; xx = x = startingpoint(i,p);
  do{ x = f(x); dist++; }while(IsNotStartingpoint(x) && (x>xx));
  if(x == xx) DetectCycle(xx,dist);
}
```

Figure 1 illustrates this algorithm[5] for $n = 12$, $N = 2$, and $P = 2$. The starting points are shaded.

The analysis in [KS01] showed that the expected runtimes are $O(n/P)$ for phase 1, $O(N)$ for phase 2, $O((\ln n - \ln(N \cdot P)) \cdot n/P)$ for phase 3, and $O(\log n)$ for phase 4, respectively. Therefore we concentrated on improving phase 3 in [KS01]. However, in practice it is often sufficient to find most cycles. In this case, phases 3 and 4 are omitted, because most cycles will be detected during phases 1 and 2, while phase 3 consumes the majority of the algorithm's runtime. This lead to a closer investigation of phases 1 and 2.

[5]Note that we used a slightly different mapping of nodes and starting-points onto processors than in the text.

3 Parallel-External Computation of Cycle Structures

Phase 1 is already highly optimized. First, there is no communication among processors. Second, assuming that the code for evaluating f fits into the cache, the phase completely runs internal to the processor except for an occasional write to the hard disk. The expected distance between two starting points is $n/(N \cdot P)$, as there are n elements in total, and there are $N \cdot P$ randomly distributed starting points. If $N \cdot P$, i.e. N in particular, is sufficiently large, then the workload will be well balanced between the processors, even without further measures. Also, a sufficiently large N will ensure that most cycles contain at least one starting point, i.e. are detected in phases 1 and 2. This indicates that N should be chosen as large as possible. N is bounded by the size of the available hard disks.

On the other hand, the larger N, the larger the problem in phase 2. On a first look, one is tempted to assume that phase 2 can be implemented by a recursive call to the algorithm from the previous section, until the problem is small enough that a sequential algorithm can be used. The difference that the distance between elements was 1 in the previous section and is an integer now can be handled in a trivial manner: in the previous section, the counter variable `dist` was incremented by 1 in the inner loop. Now it would have to be incremented by the distance to the next starting point.

However, there is another notable difference: in the previous section, we assumed that each processor could evaluate f for any argument $x \in S$, and that this evaluation consisted of executing a portion of code that fitted into the cache. Now, we consider a so called *modified* permutation f' over the set of starting points, where the function table is distributed over P files, each on a different hard disk, i.e. local to a different processor. An algorithm to solve this problem must be an external-memory algorithm, see e.g. [Vi99], as no processor will have enough main memory to load all P files into. Evaluation of f' consists of a hard disk access for starting points stored locally, and of two communications (request and response) plus a hard disk access for starting points stored on another processor's disk. The latter is the case to appear more often, because only a fraction of $1/P$ of the function table is stored locally. Even if one would bundle communications by executing one step of the inner loop for a number of iterations of the outer loop of Phase 1, the performance would be very poor, as it would still require $n/(N \cdot P)$ communication rounds, the expected number of iterations of the inner loop.

The algorithm we have in mind must use a coordinated access to the disk data, i.e. accessing the disks as seldom as possible and accessing disk blocks as large as possible. As the data reside on P hard disks, the algorithm must be parallel-external. In [KS01], we thought about adapting the peeling-off algorithm, which solves the list-ranking problem. However, during implementation (see next section) it turned out that this algorithm took much more time than phase 1, although a much smaller problem was attacked. Therefore we decided to employ a variant of the one-by-one cleaning algorithm [Si02], as explained in the remainder of this section.

In principle, what we try to do is to reduce cycles in a manner similar to list-ranking, which is now possible because of the expicit pointer structure, until they only consist of one point. We will use two basic operations on sets: *autoclean* and *altroclean*. By performing

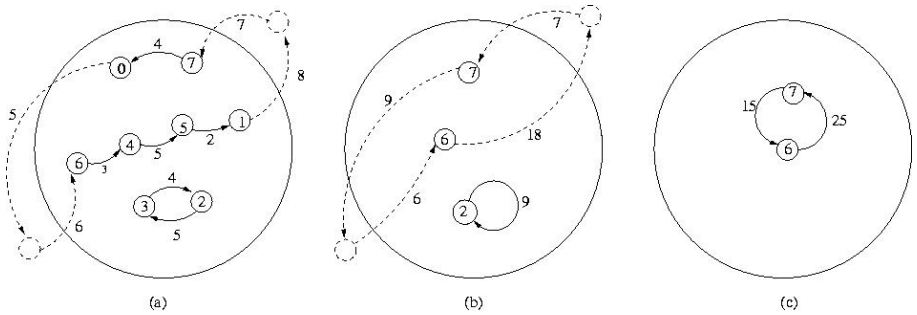

Figure 2: Examples for autoclean and altroclean.

an autoclean on a set S_i of nodes, we mean the removal of all links between nodes of the same set. Figure 2(b) depicts an example: all nodes from (a) that have a predecessor within the set are removed (such as nodes 4, 5, and 1), and a node x that has a predecessor outside the set (such as node 6) will have a successor y outside the set, with a distance which is the sum of all distances on the path from x to y. If the set fits completely into the main memory of one processor, and each node can be marked as visited, then autoclean can be performed by following each path in the set until it leaves the set, and remove the nodes inbetween. The following loop implements autoclean, with a runtime that is linear in the size of the set:

```
/* assume the set to consist of nodes 0 to r-1 */
for(x = 0;x < r;i++) if(IsNotYetVisited(x)){
  dist = dst(x); xx = x;
  while(IsInSet(x=f'(x)) && (x!=xx))do{
    SetVisited(x); dist+=dst(x);
  }
  if(x==xx){ SetVisited(x); DetectCycle(xx,dist); }
  else{ f'(xx) = x; dst(x) = dist; }
}
```

If the set S_i is distributed over several processors, then the removal of the in-between nodes leads to communications, which have to be carefully chosen. This is what the one-by-one cleaning of Sibeyn [Si02] does. It works in rounds, and the runtime is dominated by $2P - 3$ communication rounds. In each communication round, each processor sends and receives at most one packet (one-to-one communication). The sizes of these packets are bounded by the number of nodes mapped onto this processor. The set S_i is assumed to fit into the combined main memories of the P processors.

By performing an altroclean on a set S_i of nodes, we mean the removal of all links that leave the set. Assume that all nodes are partitioned into two sets S_0 and S_1, and that the set S_1 has undergone an autoclean. Now, the altroclean of the set S_0 can be performed as follows. For each node x in the set S_0 with a successor $y = f'(x)$ that is in set S_1, we pose a request to set S_1 to return the successor of y, i.e. $z = f'(y)$, together with the

distance between y and z. As the set S_1 is autoclean, each link points to a node in the set S_0, and hence z is in S_0. Figure 2(c) illustrates this, e.g. $7 = f'(f'(6))$. As the sets will be distributed over several processors, the requests and replies will involve communications between the processors. In order to minimize communication overhead, every processor first prepares all requests, and sends all requests going to one processor in one packet. Note that we only assume the set S_1 to fit into the combined main memories of the P processors. It is not required that the set S_0 fits completely in the combined main memories of the P processors. If it is larger, it can be split into subsets that fit, and the altroclean can proceed in rounds, where each round only altrocleans one subset. In one round, each processor sends and receives up to P packets (all-to-all communication). The sum of the sizes of the packets sent (received) by one processor is bounded the number of nodes from the set S_0 (S_1) that this processor hosts in this round.

Now, if we first perform an autoclean of set S_1 and then an altroclean of set S_0 with respect to set S_1, we have detected every cycle that is contained completely in S_1, and we have a set S_0 that is closed in the sense that no link points outside of S_0. Hence, from then on, we need only work with S_0.

We apply this scheme to our data organization in the following manner. The set S_1 consists of the last M elements in each file, just enough that the set can be loaded into the main memory of the P processors. The set S_0 consists of the first $N - M$ elements of each file. We apply the one-by-one cleaning algorithm of Sibeyn [Si02] to autoclean the set S_1. Then we do an altroclean of the set S_0 in $N/M - 1$ rounds. In each round, M elements of S_0 are loaded into the main memory of each processor, and this subset of S_0 is altrocleaned and the updated function tables of S_0 written back to the disks. Then the complete scheme is recursively repeated for the set S_0, so that there are N/M rounds of autoclean/altroclean alternations, where the size of the set S_0 is continually shrinking. We see that each disk access involves reading or writing a block of M nodes. As disk accesses occur only at the beginning of the autoclean routine, and the beginning and end of each round in the altroclean, we see that the access to data is very regular.

The layout of the data is as follows: each file consists of N nodes. For each node, we store its successor and the distance to this successor. Therefore, we need to store 8 bytes per nodes as long as we restrict n to at most 2^{32}. The node index itself need not be stored, because it can be computed from the position within the file. At any time, a processor stores at most M nodes from the sets S_0 and S_1, and up to M requests or replies. Hence, the processor needs a main memory capacity of $24 \cdot M$ bytes.

We now analyze the time complexity of the proposed method. We will concentrate on the communication and disk access times and ignore local computation, because its complexity is linear in the block sizes. We use the term $t_{disk}(M)$ for the time to read/write a block of M nodes from/to a disk. The time for the autoclean is

$$t_{auto} = t_{disk}(M) + (2P - 3) \cdot t_{1-1}(M) \ , \tag{1}$$

where $t_{1-1}(M)$ is the time to perform a one-to-one routing with packets of size at most M nodes. The time to altroclean one M-node piece (p-altro) of the set S_0 is

$$t_{p-altro} = 2 \cdot t_{disk}(M) + 2 \cdot t_{all-all}(M) \ , \tag{2}$$

where $t_{all-all}(M)$ is the time to perform an all-to-all communication where each processor sends and receives at most M nodes. The complete scheme consists of $N/M - 1$ rounds, where round i ($1 \leq i < N/M$) consists of one autoclean and i altroclean pieces. The total time is

$$\begin{aligned} t &= \sum_{i=1}^{N/M-1} (t_{auto} + i \cdot t_{p-altro}) \\ &= ((N/M)^2 - 1) \cdot t_{disk}(M) + (N/M - 1)(2P - 3) \cdot t_{1-1}(M) \\ &\quad + (N/M) \cdot (N/M - 1) \cdot t_{all-all}(M) \ . \end{aligned} \tag{3}$$

To simplify equation (3), we assume $t_{all-all}(M) \approx \log P \cdot t_{1-1}(M)$ and $t_{1-1}(M) \approx t_{disk}(M)$. Furthermore, we set $N = c_1 \cdot M$ and $t_{disk}(M) = c_2 \cdot M$, where $c_1, c_2 > 1$ are reals. We then obtain

$$t \approx (c_1(\log P + 1) + 2P - 2)(N - M) \cdot c_2 \ . \tag{4}$$

Therefore, $t = O(N \cdot P)$, if we assume c_1 and c_2 to be constants.

4 Experiments

We implemented an optimal sequential algorithm that uses a bit vector of length n to mark which elements have already been visited by the algorithm. This algorithm has runtime $O(n)$. We also implemented our parallel algorithm in C. Communication was done with the MPI library. For phase 2, we implemented both the peeling-off algorithm and the adapted one-by-one-cleaning algorithm, as described in the previous section. As function f, we used a DES variant where all data paths have been reduced from 64 bits to 32 bits. We used $N = 2^{24}$ and $M = 2^{20}$ for the performance measurements. We tested all three implementations on PCs with AMD Athlon processors with 800 MHz and 512 MByte of main memory, under the Linux operating system. The PCs communicated over switched fast ethernet. Each processor had a file storing N nodes. Each node occupied 12 bytes, as we stored the successor in 8 bytes[6] and the distance to this successor in 4 bytes.

So far, we tested 11 different keys which were randomly selected. The runtime of the sequential algorithm was an average of 11 hours, with at most $+/-1$ hour deviation depending on the key. Of course, to rule out the effect of the bit vector, we would have to multiply the sequential runtime by $(2/3) \log n \approx 21$ (obtaining 231 hours), because the optimal algorithm performs n evaluations of f, while Knuth's basic algorithm performs $(2/3)n \log n$ evaluations of f [Ke02].

The runtimes for the parallel algorithm, averaged over the keys, are shown in Table 1. They show that the processor time product for phase 1, where at most n evaluations of f

[6] Instead of storing the index of the successor, we stored the processor where that index is mapped to, and the local index, in 4 bytes each.

Phase	$P = 4$	$P = 8$	$P = 16$
1	3.90	2.00	1.10
2 (peeling-off)	14.00	12.00	—
2 (one-by-one)	0.88	1.55	3.71
3	14.80	6.40	3.16
total (one-by-one)	19.58	9.95	7.97

Table 1: Average parallel runtimes in hours

are performed, is only slightly larger than the runtime of the optimal sequential algorithm. It also shows that the runtime of phase 2 with the peeling-off variant was much longer than phase 1, although a much smaller problem was attacked. This justifies our implementation of the one-by-one cleaning algorithm. As expected from equation (4), the runtime of phase 2 grows about linearly with P. Also notable is that phase 3 (albeit with several optimizations) takes much longer than phases 1 and 2 together, which will in many situations lead to its omission.

In order to validate our analysis of the previous section, we also tested the influence of N on the runtime. For $M = 2^{20}$ and $P = 8$, we tested also $N = 2^{25}$, and $N = 2.5 \cdot 2^{25}$, and obtained runtimes of 5.51 and 29.91 hours, respectively, compared to 1.55 hours for $N = 2^{24}$ (see Table 1). Equation (4) suggests an almost quadratic increase with N, as c_1 grows linearly with N if M is fixed. The increase in runtime is surely more than linear.

Note, that after the end of our experiments, we found out that we had used a sub-optimal implementation of all-to-all communication. After correcting this, the runtime of phase 2 shrinked considerably to 65% to 79% of the previous values, depending on the packet sizes.

5 Conclusions

We have presented our experiences with the implementation of a parallel external-memory algorithm, which was done as part of a much larger implementation. While the subproblem tackled here seemed to be a minor one during algorithm design and analysis, it turned out a major effort during implementation and tuning.

We have given our rationale for choosing a particular combination of algorithms and also a runtime analysis. The experiments we made support this analysis.

Acknowledgements

This work would not have been possible without access to the cluster computers of the computer architecture group at FernUniversität Hagen and of the computer engineering

group at University of Lübeck. We also thank Vidit Jain and Jop Sibeyn of University of Halle for the possibility to build upon their implementation of the cycle-structure algorithm. Finally, we would like to thank the anonymous reviewers for their helpful comments.

References

[FO90] Flajolet, P. und Odlyzko, A. M.: Random mapping statistics. In: *Proc. Eurocrypt '89*. LNCS 434. Pages 329–354. Springer Verlag. 1990.

[Ke02] Keller, J.: A heuristic to accelerate in-situ permutation algorithms. *Information Processing Letters*. 81(3):119–125. 2002.

[Kn72] Knuth, D. E.: Mathematical analysis of algorithms. In: *Proc. IFIP Congress 1971, Information Processing 71*. Pages 19–27. North-Holland Publ. Co. 1972.

[KS01] Keller, J. und Sibeyn, J.: Beyond external computing: Analysis of the cycle structure of permutations. In: *Proc. Euro-Par 2001, Manchester, UK, Aug. 2001*. LNCS 2150. Pages 333–342. Springer Verlag. 2001.

[Sc95] Schneier, B.: *Applied Cryptography*. John Wiley & Sons. 2nd ed. 1995.

[SF96] Sedgewick, R. und Flajolet, P.: *An Introduction to the Analysis of Algorithms*. Addison Wesley. 1996.

[Si02] Sibeyn, J.: One-by-one cleaning for practical parallel list ranking. *Algorithmica*. 32:345–363. 2002.

[Vi99] Vitter, J. S.: External memory algorithms and data structures. In: Abello, J. und Vitter, J. S. (Ed.), *External Memory Algorithms and Visualization*. Pages 1–38. American Mathematical Society Press. Providence, RI. 1999.