Parallel Storage Allocation for Intermediate Results During Exploration of Random Mappings

Jan Heichler[1] Jörg Keller[2] Jop F. Sibeyn[1*]

[1]Martin-Luther-Universität Halle
Institut für Informatik
Von-Seckendorff-Platz 1
06120 Halle, Germany
heichler@informatik.uni-halle.de

[2]FernUniversität in Hagen
LG Parallelität und VLSI
Postfach 940
58084 Hagen, Germany
joerg.keller@fernuni-hagen.de

Abstract

We present and analyze an improved parallel algorithm to compute the structure of a graph induced by a randomly chosen function on a finite set. The improvements center around how to use the vast memory resources of a parallel computer to store intermediate results, i.e. knowledge about graph nodes that have already been visited. Further improvements and extensions are pointed out. Experiments are presented that confirm the theoretical analysis.

1 Introduction

We consider a function $f : D \to D$ and the directed graph $G_f = (V = D, E = \{(x, f(x)) \mid x \in D\})$ induced by it. Such a graph is called a *mapping* in the literature [3]. Each weakly connected component of that graph consists of a cycle and a number of trees directed towards the root, the root being a node on the cycle. The problem to be solved is, given a function f, to determine the *structure* of the graph G_f, i.e. the number and size of the components, the length of the cycles, the heights of the trees, and so on. Because of the size of $n = |D|$, the graph cannot be explicitly constructed in memory, and thus algorithmic techniques like pointer doubling cannot be applied. The size of n calls for a parallel algorithm.

Examples of graphs induced by functions are the state spaces and their state transition functions in pseudo-random number generators or cryptographic stream cipher generators. Especially the latter shall behave like a *random* mapping, of which expected values for many properties like the size of the largest component are known from literature [3]. A widely used stream cipher is the A5/1 algorithm used for encrypting the communication between a cellular phone and the base station in the GSM network [2, Sec. 6.3.4]. If the graph induced on A5/1's $n = 2^{64}$ states by its transition function (see e.g. [1] for a description) differs considerably from what is expected, this may hint towards a weakness.

In [4], a parallel algorithm is described which computes the structure of induced graphs, and has expected runtime $O(n\sqrt{n}/p)$ on a cluster with p processors. While this algorithm achieves a good load balancing and fully exploits the computational resources available, it does not make use of the vast memory resources available. For each node v of the graph, the algorithm simply follows the path starting in v until it reaches a cycle, which uniquely determines the component. As there are n nodes to start from, and as each run has an expected length of $O(\sqrt{n})$ [3], we get the runtime bound above. As the expected size of the largest cycle and the expected depth of the largest tree is $O(\sqrt{n})$ with coefficients between 1 and 2, and the size of the largest tree is

[*]In Memoriam. Jop Sibeyn is posted as missing since a ski trip in March 2005.

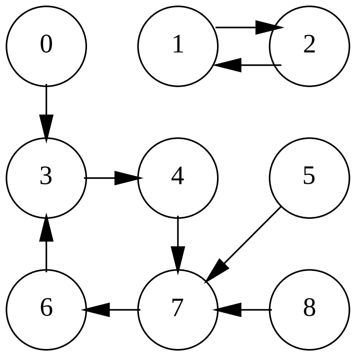


Figure 1: Example of induced graph for $n = 9$.

expected to be approximately $0.48 \cdot n$ (with a fraction of $1/e$ being leaves), we also see that most time is spent in running towards tree roots.

We consider how to exploit the memory resources of a cluster in order to reduce the runtime. We assume that the cluster consists of p processing units, each with a microprocessor, a main memory of size m and a hard disk of size M. The processing units are connected by a switched interconnection network, e.g. a switched gigabit ethernet.

In order to decrease the length of the n runs we provide tree nodes (so called *anchors*) from which we already know the number of the component, the tree root, and the distance to the root. As soon as an anchor is reached while following a path, one can stop. Those anchor nodes could be either "cached" from previous runs, or could be chosen in advance in a manner that minimizes runtime. Caching schemes have the advantage that they are completely dynamic, but they cannot guarantee optimal placement. Offline schemes could do better, but to the best of our knowledge, we do not know any good ones.

In this paper, we will present an anchor placement based on a caching scheme that is provably good, even if we do not know whether it is optimal. We will derive all parameters that influence overall runtime, and how the parameter values have to be chosen. We will also present data structures to store the anchors in the cluster. Finally we will validate our proposal with experiments.

The remainder of the paper is organized as follows. In Sect. 2 we will briefly summarize the necessary facts about random functions and the structure of their induced graphs. In Sect. 3 we introduce the concept of anchors to exploit the memory in a cluster, and analyze the improvements. In Sect. 4 we present experiments conducted to validate our analysis. Section 5 presents some possible extensions of our algorithms, and issues of further research. Section 6 concludes.

2 Random Functions and Their Properties

We consider a finite set D of size n, and the set F of all functions from D to itself. Let us assume that we randomly pick a function f from F, such that each function in F is equally probable. Now we consider the graph G_f induced by f. An example of such a graph for $n = 9$ can be seen in Fig. 1. The graph consists of two components, of sizes 7 and 2, respectively, and cycle lengths 4 and 2, respectively. Flajolet and Odlyzko [3] investigated the average case behaviour of such graphs. They found that the expected number of components is $O(\log n)$ with a constant close to 1, that the expected size of the largest cycle is $\mathrm{O}(\sqrt{n})$ with a constant close to 1, that the largest component has an expected size of about $0.7 \cdot n$, that the largest tree has an expected size of about $0.5 \cdot n$ and expected depth $0.6 \cdot \sqrt{n}$, and that the expected fraction of leaves in that graph is $1/e$.

Graphs of such a structure are for example found in pseudo-random number generators (PRNG) and generators for stream cipher encryption. Here, D is the state space of the generator, f is the state transition function, and the length of the cycle in the largest component is the main period of the generator. The set of nodes of the other components is the set of unwanted start states, as they result in shorter periods, which is not wanted. If a function f is chosen to be the basis of such a generator, one would expect that it behaves as if randomly picked from the set of all functions on D, in order not to be predictable. Hence, the structure of the graph induced by f should not deviate too much from the expected values given above. Otherwise, this may hint to a weakness in the generator, which is especially undesirable if it compromises security of the application where the generator is deployed.

However, for interesting functions f, such as A5/1 stream cipher which is used to encrypt the voice data between GSM mobile phones and their base station [1, 2], it is not known how to derive the graph structure analytically. As the state spaces are often quite large (for A5/1 $n = 2^{64}$), it is also not possible to construct a representation of such a graph in memory and apply well-known graph algorithms in order to derive the structure. Hence, the only method possible is to explore the graph. By this we mean to start at a node x, and follow the edge $(x, f(x))$ that leaves x. By following the path $x, f(x), f^2(x), f^3(x), \ldots$, we finally reach the cycle of the component to which x belongs[2]. By the simple method of remembering which node was met after $2, 4, 8, \ldots$ steps, it is possible to detect that one is on a cycle, the cycle length, the cycle leader[3], the node at which the cycle was entered, and how many steps were necessary to reach the cycle, with a number of steps linear in the path length [4]. Now starting from each node in D, we find out which component it is in, and so after following n paths we have the component sizes, and the average and maximum tree heights for each component. The cycle length and cycle leader is known as soon as the component is encountered the first time.

As each path is expected to have a length of size $O(\sqrt{n})$ [3], the expected runtime of this simple algorithm is $O(n\sqrt{n})$. On a cluster computer with p processing units, each processor could follow n/p paths, thus resulting in a speedup of p, if we neglect the small overhead of merging the component informations from different processors at the end [4].

3 Placement of Anchors

The simple algorithm of the previous section does not use the vast memory resources, both main memory and hard disk memory, of a cluster computer. This memory could be used to store information about nodes that have already been visited. We will denote such nodes as *anchors*. For a node v in a tree, it suffices to store its distance d_v to the tree root and the number c of the component it belongs to. If a path from node x reaches node v after d steps, we know that it belongs to component c and has distance $d + d_v$ to the tree root, i.e. to the cycle. As a consequence, when following a path, one must check after each step whether an anchor is met, and if so, one can stop following that path.

The difficult question is, given a certain memory resource, how the anchors should be placed in order to minimize runtime. We assume that the runtime is minimized by minimizing the sum of the lengths of the paths from each node to an anchor or a cycle. We restrict our investigation to trees, as almost all nodes in an induced graph sit in trees, and almost all tree nodes belong to a small number of trees. However, as our trees are far from balanced (see next subsection), we are not aware of any algorithm that, given such a tree, could place a given number of anchors such that the sum of the path lengths is optimized. Moreover, we do not know the trees in advance. Hence, an offline algorithm is not available. Another possibility would be to place the anchors randomly on nodes of the graph. However, as a fraction of $1/e \approx 36.8\%$ of the nodes are expected to be leaves, more than one third of the anchors would be wasted, as obviously an anchor on a node is not of much help.

3.1 Caching Scheme

A further possibility is to place anchors from time to time as we follow paths. This is at least a feasible solution, after we have decided at which distances we have to place anchors in order to minimize runtime. From time to time we may even evaluate how often anchors have been met, remove those that have not been of much use, and

[2]This means that we treat f as an oracle. By giving x, we receive $f(x)$, but have no further knowledge about f except $n = |D|$. The only exception to this rule is handled in Sect. 5.

[3]Each component can be uniquely identified by the node on its cycle with the smallest number. We call this node *leader*.

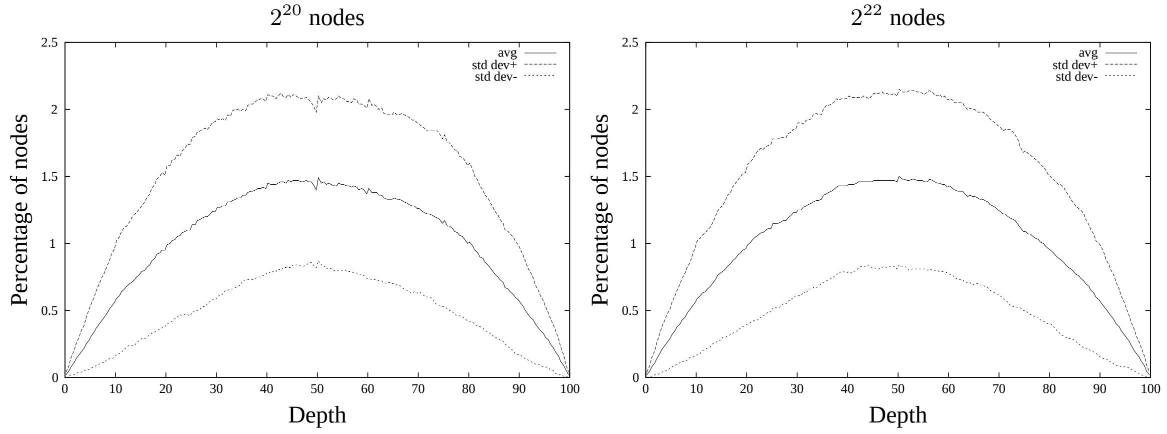


Figure 2: Distribution of nodes vs. depth in a tree.

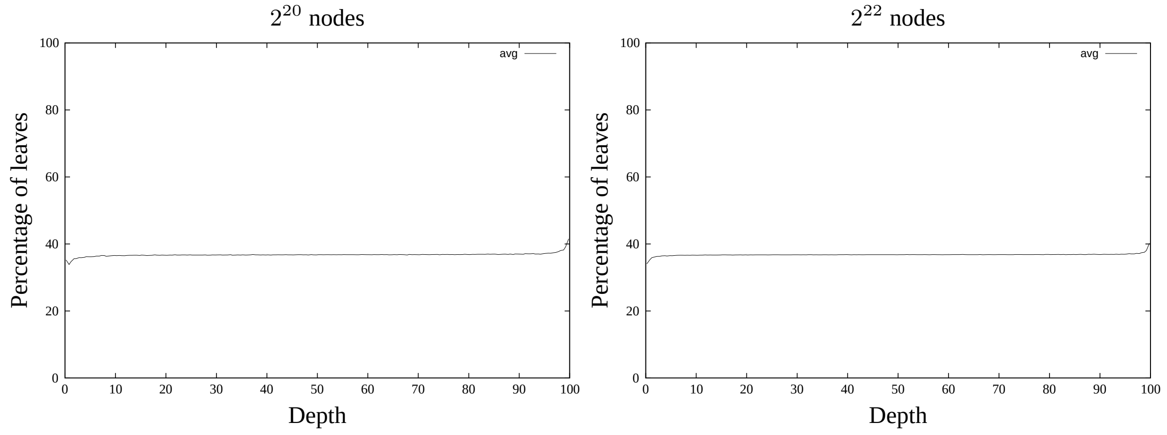


Figure 3: Fraction of leaves vs. depth in a tree.

place some more as long there is free memory. This would constitute a kind of caching scheme. Additionally, there is also a theoretical foundation behind this placement.

We have made a study about the structure of the largest tree in an induced graph, averaged over 1000 randomly chosen functions on a domain D of sizes $n = 2^{20}, 2^{22}$. Figure 2 depicts the fraction of nodes on such a tree having a certain depth, where we normalized with the height of the tree. We see that between 30% and 70% of the depth, the number of nodes in a certain depth is almost constant, if we approximate the curve by a trapezoid. We also see that a considerable fraction of the nodes in a tree is very far away from the root. Finally, we see in Fig. 3 that the ratio of leaves to nodes is constantly $1/e$ over all depths, only close to the root there are less, and in the deepest depths there are some more.

Assume that we are somewhere in the middle of a tree with respect to its height. We assume that k nodes are in depth i, $i-1$, and $i-2$, respectively, as shown in Fig. 4. Then we can assume that k/e nodes in depth $i-1$ are leaves, such that only $(1-1/e)k \approx 0.64 \cdot k$ nodes in that level have ancestors from depth i. If we now look at level $i-2$, we can conclude that $(1-1/e) \cdot k/e$ nodes have ancestors which are leaves in depth $i-1$, and that the remaining $(1-1/e)^2 \cdot k \approx 0.4 \cdot k$ non-leaf nodes in depth $i-2$ have ancestors in depth i. Thus, in level $i - 2\log_{2.5} k$, we can assume that all paths passing a node in depth i have merged into one path. One may argue that we were a bit optimistic in the step from $i-1$ to $i-2$, because ancestors in depth $i-2$ of leaves and non-leaves from depth $i-1$ may not be separate. But one can still assume that the interchange will be in

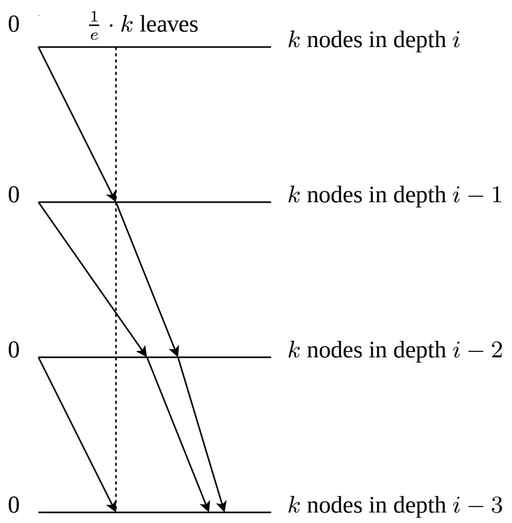


Figure 4: Subtree formation in a tree.

proportion to the numbers of those nodes, so that still a reduction of ancestors by a constant factor will occur, and the number of levels until all paths from depth i will have merged into one path is still logarithmic in k.

We can assume k to be expected less than $\sqrt{n}$, as the largest tree in the induced graph contains $0.5n$ nodes, has depth $0.6\sqrt{n}$, and has a trapezodial shape. Then at level $i - \log n$ all paths originating in nodes deeper than i will have merged into one. Now, if any node of the respective tree deeper than i has already served as the starting node of a path along which anchors were placed, we can conclude that this path will be merged too, and a path originating in any node at level i will meet the next anchor on this path. Now, when we are somewhere in the middle of a tree with respect to its height, then there is a good probability that a path originating from some node deeper in the tree already passes this point, and hence the number of steps on average will be about $\log n$, plus the distance to the next anchor. The latter will be half the distance between anchors on average. Hence, the distance between anchors should be kept constant along the path, and be in the range of $\log n$ as well.

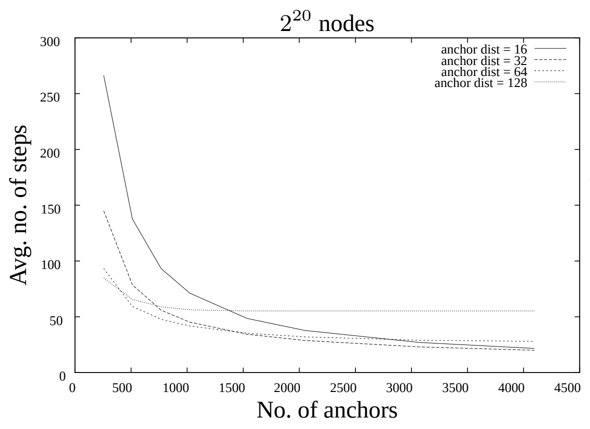
3.2 Quotas

If we introduce anchors, the time for following a path will increase in each step. Besides the time t_s to evaluate $f(x)$, we now have to check whether $f(x)$ is an anchor, taking time t_c. No matter which data structure we choose, the data structure will be large because of the number of anchors, and therefore the time t_c may well dominate t_s. In order to minimize the runtime, we refrain from checking for an anchor in each step, and search for the optimum frequency of checks. This is achieved by allowing only a fraction $1/d$ of the nodes to be candidates for anchors, meaning that on average, we will meet such a candidate every d steps, and only then have to check whether such a candidate is really an anchor. We will denote d as *quota*. On average, we will increase the number of steps by $d/2$ by this. Therefore, the average runtime for following a path will be

$$T(d) = \left(w + \frac{d}{2}\right) \cdot \left(t_s + \frac{t_c}{d}\right) \ , \qquad (1)$$

where w is the average number of steps on a path for a quota of $d = 1$. Now the miminum runtime is achieved by differentiating $T(d)$ and searching for d_{opt} with $T'(d_{opt}) = 0$. We find that

$$d_{opt} = \sqrt{\frac{2wt_c}{t_s}} \ .$$



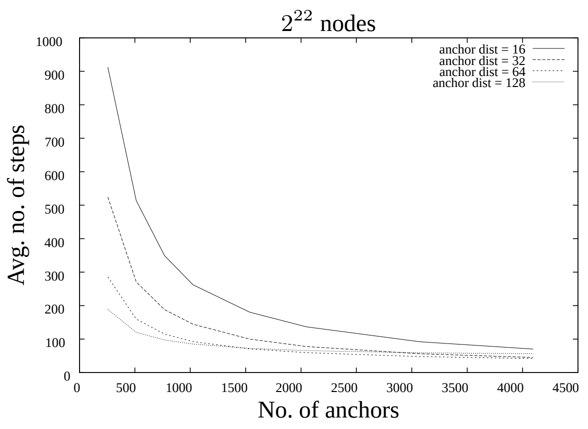


Figure 5: Average number of steps in a path.

Note that the domain of $T(d)$ is the interval $[1 : n/s]$, where s is the total number of anchors that can be stored. If $d = 1$, then there is no quota, i.e. every node is a candidate. If $d = n/s$ then every candidate will become an anchor, we have a static placement of anchors. If d_{opt} is outside that interval, one should choose the closer border of the interval.

In practice, one will choose d to be a power of 2, and determine candidates by a property that can be computed fast, such as the lowermost $\log d$ bits being zero.

3.3 Data Structure

To store the anchors, several possibilities are possible. The simplest ones would be to store in each processing unit either the same set of anchors, or an independent set of anchors. Then no communication would be necessary at all. However, the number of anchors is limited by the main memory or hard disk capacity of a single processing unit. Another possibility would be to distribute the anchors over all p processing units, which allows a number of anchors p times as large as before. However, the check whether a candidate is an anchor now requires a communication between the processing node doing the check and the processing node that would store the anchor if that candidate really is an anchor. In practice, the mapping of candidates to processing units will be static and easy to compute, such as the $\log p$ middle bits of the candidate giving the number of the processing unit where it will be stored as an anchor.

In order to keep the check time t_c per anchor low, we pursue the following scheme. Each processing unit follows r paths simultaneously, each path until the next candidate is reached. Then all the candidates are sorted according to the processing unit they will be stored at (if they are anchors), and an all-to-all communication takes place. Thus, on average only a fraction of r/p of the communication time contributes to each check. Each processing unit now receives check requests from all other processing units. It treats them all, and another all-to-all communication is used to return the answers to the checks, i.e. for each candidate whether it is an anchor, and if so, the number of component, and its distance to the tree root.

The question of data structure chosen in each processing unit depends on the size of r. If r is quite small, then a small number of requests reach a processing unit at one time, and the best way to store the anchors is a hash table in main memory. However, if r is large, then a large number of requests reach a processing unit at one time, and it might be wiser to keep the anchors in a sorted list, sort all requests, and treat them all together by a kind of merge procedure.

While a hash table structure is not very suitable for an implementation on a hard disk, a sorted list and a merge procedure may well be implemented. The use of the hard disks to store the anchors increases the number of anchors that can be stored by at least one magnitude, but also slows down the check considerably. Hence, the exact tradeoffs require further research.

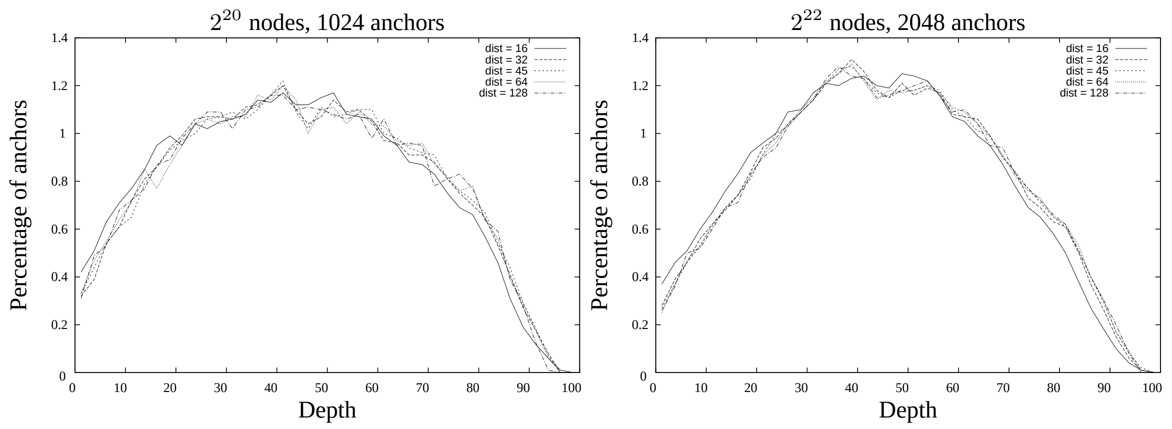


Figure 6: Ratio of anchors to nodes vs. depth.

4 Experiments

In all of the following experiments, anchors were placed on paths from randomly chosen starting points, with the distance between them as indicated, until the maximum number of anchors has been reached. No update on anchors has been performed.

Fig. 5 depicts, for $n = 2^{20}, 2^{22}$, and $\sqrt{n}$ anchors, averaged over 96 randomly chosen functions, the average number of steps while following a path, until an anchor is reached, for different distances between anchors and different numbers of anchors. We see that for fixed n, the number of steps converge if enough anchors are placed. We also see that if the ratio of n and the number of anchors is fixed, so that an equal coverage is reached, the number of steps for different n deviate only slightly. The deviating behavior of distance 128 in $n = 2^{20}$ can be easily explained: one cannot hope to get to a number of steps below half the distance between anchors!

Fig. 6 depicts, for $n = 2^{20}, 2^{22}$, and $\sqrt{n}$ anchors, the ratio of anchors to nodes in the largest tree, over the depth. An average was taken over 96 randomly chosen functions f. We see that the anchor distribution is as intended: few anchors near the leaves at the end of the tree, the majority in the middle of the tree where most nodes are.

Fig. 7 depicts, for $n = 2^{22}, 2^{24}$, the relation between quota and runtime for different numbers of anchors. For the previous experiments, we took randomly chosen functions, with the function table stored in main memory. Hence, evaluation of function f required a slow access to main memory, because hit ratio on the function table is close to zero. In order to concentrate on the influence of the quota, we seek to minimize t_s, and therefore took a variant of the A5/1 algorithm as function f. We see the optimum values of d, that differ depending on the size of the graph, the number of steps in a path, and the anchor check time. Yet, the image is somehow distorted as we restricted ourselves to values of d that were powers of 2. The real optimum values would have been much closer.

5 Extensions

So far, we have treated f as a kind of black box. However, there are cases where f is *efficiently invertible*, i.e. where the set $f^{-1}(y) = \{x \mid f(x) = y\}$ can be efficiently computed for any given $y \in D$. The algorithm A5/1 belongs to this category. Now, the problem can be solved once a weakly connected component is detected: if the component is known, then also the cycle of the component is known. On the cycle, all the tree roots are known, they are exactly those nodes y on the cycle with $|f^{-1}(y)| \geq 2$. If a tree root is known, then the tree can be completely explored by a depth first search. So the time complexity is the time to find the components, plus $O(n)$ to explore all trees. However, the only known way to find all the components is to select starting

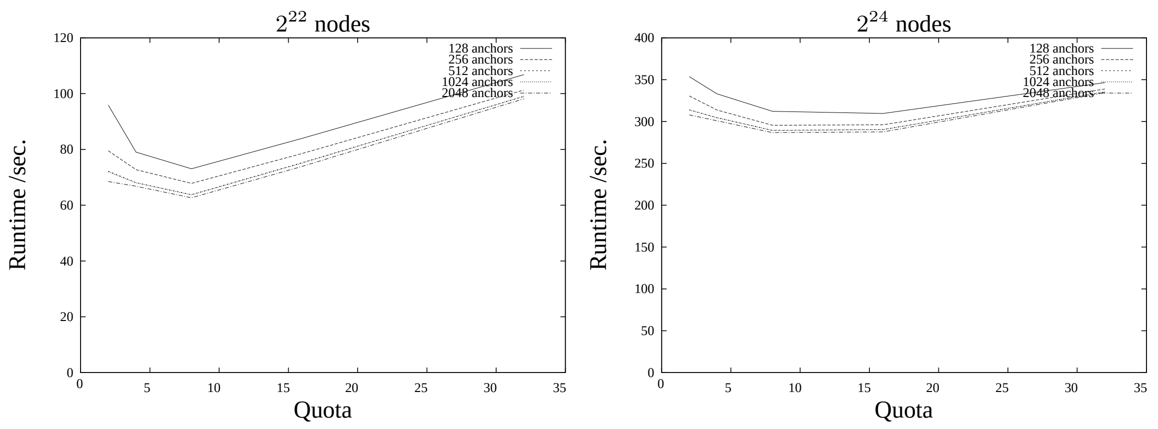


Figure 7: Quota vs. runtime.

nodes v and follow the path starting in v until one can decide whether the component is known or not, i.e. when the cycle has been reached. Hence, while the structure analysis itself is simpler than in the general case, the detection of the components remains almost as hard as in the general case. It is only almost as hard as we can discard starting points v with $|f^{-1}(v)| \geq 1$, i.e. if v is not a leaf. This reduces the number of starting points by a factor of e on the average.

The task at hand gets simpler for both categories of functions, if we do not insist on computing the complete structure, but are satisfied with an approximation of the structure. This scenario is quite likely for very large n, as even an algorithm with linear complexity and perfect parallelization may take too long for $n \geq 2^{64}$. It is still to be expected that the larger components will be detected fast, i.e. with a sample of starting points with a feasible size. If a component is detected, the length of its cycle is known exactly. The sizes of the components can be approximated from the distribution of starting points onto components: if a fraction α of the starting points from the sample belong to one component, we can approximate the size of that component by αn. The tree heights of the components can be approximated as the tree heights seen when handling the sample.

For a function that is efficiently invertible, we could get exact results of the components that are detected, as explained above. However, if even a linear time algorithm takes too long in the case of very large n, then one would either stick to the general case, or have to explore efficient ways to replace a depth first search by an approximation with less than linear complexity.

6 Conclusions

We have presented an improved parallel algorithm to compute the structure of a graph induced by a random function. The improvements centered around using the vast memory resources of a cluster computer to store intermediate results of our computation, i.e. knowledge about some nodes that were already visited. We presented several variants that serve to further decrease runtime, and also pointed out possible extensions and issues for further research. Finally, we reported about some preliminary experiments that confirm our theoretical analysis of the algorithm.

Acknowledgements

We would like to thank Thomas Schürger for mentioning the possibility of placing anchors along paths, and the anonymous reviewers for their suggestions and their trust.

References

[1] A. Birykov, A. Shamir, D. Wagner. Real Time Cryptanalysis of A5/1 on a PC. Presented at the Fast Software Encryption Workshop, April 10-12, 2000, New York, NY. Available at `http://cryptome.org/a51-bsw.htm`

[2] J. Eberspächer, H.-J. Vögel, C. Bettstetter. GSM — Global System for Mobile Communication. 3rd Edition, Teubner-Verlag 2001.

[3] P. Flajolet, A. M. Odlyzko. Random mapping statistics. In Proc. EUROCRYPT'89, LNCS 434, Springer-Verlag, 1990, pp. 329–354.

[4] J. Keller. Parallel Exploration of the Structure of Random Functions. In Proc. PASA 2002, Karlsruhe, April 2002, pp. 233-236, VDE Verlag 2002.