A Distributed Query Structure to Explore Random Mappings in Parallel

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

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

Abstract

We explore the possibilities to organize a query data structure in the main memories or hard disks of a cluster computer. The query data structure serves to improve the performance of a parallel algorithm for the computation of the structure of a graph induced by a random function. Tradeoffs between different organizations using main memory or hard disks are developed and quantified with parameters. Thus, for concrete cluster systems with concrete parameter values, the best organization can be selected.

1. Introduction

We investigate the structure of large but finite, random looking, directed graphs where each node has outdegree exactly 1. Those graphs serve as models of state spaces with deterministic transitions. More formally, we consider an arbitrary function $f : D \rightarrow D$ and the directed graph

$$G_f = (V = D, E = \{(x, f(x)) \mid x \in D\}) \qquad (1)$$

induced by it, where D is a finite set of size n. 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 their roots, the roots being nodes 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 the size $n = |D|$ of state spaces is typically very large, the graph cannot be explicitly constructed in memory, and thus classical techniques from graph algorithms, such as pointer doubling, cannot be applied. Instead, the graph can only be explored by starting at nodes x and following paths $x, f(x), f(f(x)), f(f(f(x))), \ldots$. The size of n calls for a parallel algorithm. Our parallel machine model is a cluster of p processing units, each with a processor, a local memory of size m, a hard disk of size M, and connected by a fast message-passing network.

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 presented which computes the structure of induced graphs, and has expected runtime $O(n \log n/p)$ on a cluster with p processing units. This runtime, which considerably improves over previous algorithms, is achieved by using a data structure to store information about anchors, i.e. nodes that have already been visited, in order to stop following a path if an anchor is met. However, possible implementations of such a data structure were only briefly sketched in that paper, a gap that the present research tries to fill. In order to have as much memory available as possible, the query structure is distributed over all p processing units. Within each processing unit, either the main memory or the hard disk can be used, with several data organizations possible. The tradeoffs between the different possibilities can be quantified using parameters, so that for a concrete cluster with concrete parameter values, the best organization can be selected.

The remainder of the paper is organized as follows. In Sect. 2, we summarize our previous algorithm so that the types of queries for the current data structure can be understood. In Sect. 3, we present and compare the different possibilities for a query data structure distributed over all nodes. Section 4 concludes.

2. Parallel exploration of random mappings

We consider a finite set D of size n, and a function f from D to itself. We investigate the graph G_f induced by f, as defined in Eq. (1). Flajolet and Odlyzko [3] investigated the average case behavior of such graphs, if all functions from D to itself are equally probable. 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 $O(\sqrt{n})$ 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 $1/e$.

In the remainder of this section, we briefly summarize information from previous papers [4, 5]. In order to compute the structure of an induced graph, it can only be explored by repeated evaluation of f. The algorithm starts from each node $x \in D$, and follows the unique path x, $f(x)$, $f^2(x)$, $f^3(x), \ldots$ until either the cycle of the appropriate component is reached, or until a node is reached that has been visited before, and of which the appropriate information has been stored in memory. Such a node is called an *anchor*. By appropriate information we mean:

- the node index (a number between 0 and $n-1$),
- to which component this anchor belongs (identified by the cycle leader[1]),
- what its distance from the tree root is.

Then, when the number of steps gone so far on the path has been recorded, all relevant information about x can be computed. We will assume here that all anchors have already been placed, and that the set of anchors will not be changed for the whole exploration. The importance of the anchors becomes clear when we remember that the length of the path is expected to be $O(\sqrt{n})$, while a set of $O(\sqrt{n})$ anchors can achieve an expected number of steps of $O(\log n)$ until an anchor is reached! Hence, also the necessity to store as many anchors as possible becomes clear.

Fig. 1 depicts an example graph for $n = 16$. We see that the graph consists of two weakly connected components, of sizes 12 and 4 respectively. The larger component has a cycle of length 5, with leader 2. It also comprises two trees with roots 7 and 15. All tree nodes are shaded, the root is encircled. The tree with root 7 has height 3, the tree with root 15 has height 1. The smaller component comprises a cycle of length 3, with leader 4, and a tree with root 4 and height 1. If an anchor is placed on node 11, the information to be stored would be $(11, 2, 1)$, meaning that the node number is 11, it belongs to the component with cycle leader 2, and its distance to the tree root 7 is 1. If we start following a path in node 13, then we would visit nodes 6 and 11, and stop there after two steps because 11 is an anchor. As the distance of anchor node 11 to its root is 1, and we have gone 2 steps, the depth of node 13 in the tree is 3.

[1] There is only one cycle per component. The unique node on the cycle with the smallest number is called *cycle leader*, it serves to identify the component.

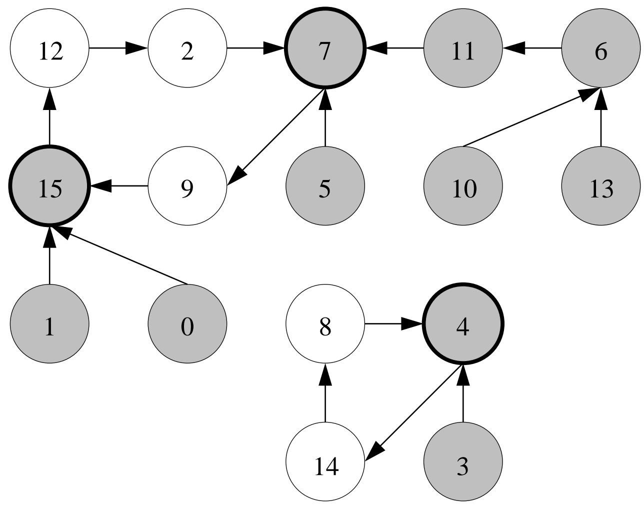

Figure 1. Example graph induced by a function.

In order to avoid to check after each step whether an anchor has been reached, only a part of the nodes can become anchors, those nodes are called *candidates*. Depending on the query time and the average path length if all nodes would be candidates, an optimum value for the fraction of nodes that are candidates can be derived. In practice, the fraction is a power of two, and a node is a candidate if an appropriate number of lowermost bits in its binary representation are zero.

Each time a candidate is reached, the query data structure is queried whether that candidate is indeed a anchor. It is obvious that the average path length is shorter if more anchors can be stored. In order to have a query structure as large as possible, the query structure is to be distributed over all p processing units. In each processing unit, it is either held within the memory or on the hard disk. In the next section, we will explore the tradeoffs between those two possibilities. We will also explain in detail the structure chosen for main memories and hard disks, respectively. While we restrict ourselves to the static case, where the query structure is initialized at program start and remains unchanged from then on, we will point out where changes are necessary if the anchor set is to be updated. That may be an option with very large graphs to adapt dynamically so that more queries are answered positively.

A major characteristic of queries to a distributed data

structure is that they involve communications and therefore take quite a long time. In order to keep the processors busy, the application therefore does the following. Each processing unit follows a large number of paths simultaneously. Each path is followed until it reaches a candidate, then the next path has its turn.. When a processing unit has reached candidates on all of its active paths, it starts a query for every candidate whether it is indeed an anchor or not. Thus, the queries can be performed in batches, i.e. each processing unit i sends packets to all other processing units. A packet from i to j contains all queries of i that can be answered by j. This serves to amortize the query time per path. For the data structure the necessity arises to deal with larger number of simultaneous queries.

3. Data organization

3.1. Main memory

An anchor requires 24 bytes if we want to be able to handle functions with n up to 2^{64}: 8 bytes for the node number, 8 bytes for the distance to the root, and 8 bytes for the component number. As the number of components will be quite low, 4 bytes or even 2 bytes would be sufficient in practice for the component number. However, 8 bytes seem preferable in order to achieve alignment with cache lines. A pointer to an anchor will require 4 bytes on a 32-bit architecture.

When using the main memory of a processing unit to store s anchors, the obvious data structures are a hash table and a binary search tree. For the static case considered here, we have simplified the search tree in the following way: we use a sorted array of the anchors, and perform a binary search on that array. For the hash table we use s entries, and handle collisions by linear lists. The hash function is simply a computation modulo s, which can be evaluated fast if s is a power of 2, as it only requires masking of the $\log_2 s$ lowermost bits in that case. This simple hash function is sufficient as we expect our queries to be random.

We see that the sorted array requires no memory overhead, while the hash table requires overhead for resolving collisions. The overhead is one pointer per entry, plus one pointer for each collision. As our experiments suggest that about one third of the elements are involved in collisions, the overhead will be $(s + s/3) \cdot 4 = 16s/3$ bytes, which is a proportion of $(16s/3)/(24s) = 2/9$. As a hash table with s anchors requires $11/9$ of the memory for a sorted array with s anchors, a sorted array requiring the same memory resources as a hash table with s anchors can store $11/9 \approx 122.2\%$ as many anchors. As long as the number of anchors is below $\sqrt{n}$ (non-saturated case), this will linearly decrease the number of steps and query rounds necessary in case of the sorted array.

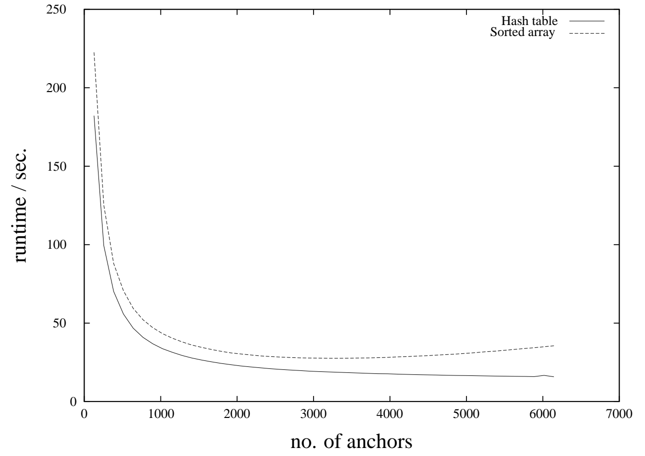

Figure 2. Hash table vs. sorted array performance.

On the other hand, the sorted array requires $\log s$ memory accesses, while the hash table will need at most about $\log s / \log \log s$ accesses (expected maximum number of balls in any box when throwing s balls into s boxes) and about 2 to 3 accesses on the average. As $s = 2^{23}$ on a processing unit with $m = 256$ MByte of main memory, the hash table still leads to shorter runtimes. This is confirmed by our experiments for 64 randomly chosen functions with $n = 2^{20}$, as Fig. 2 indicates. We see that the average runtime with a sorted array is always higher than with a hash table, no matter how many anchors we place in the structure. We even see that the runtime with a sorted array increases for numbers of anchors well beyond $\sqrt{n}$, as the linear relationship between anchor count and runtime decrease does not hold anymore (saturated case), and the increased search time cannot be compensated for by an appropriate reduction in the number of steps, i.e. path lengths.

If dynamic insertions and deletions shall be taken into account, there are two possibilities. Either, changes for single anchors happen quite frequently, or a bulk update with many anchors exchanged happens quite seldom. In both cases, the anchors have to be extended by a usage count. In the latter case, the data structures could remain as they are, as an update is a seldom event, and hence it is possible to construct an updated data structure completely anew and treat the situation as static until the next update. As the anchors get larger, the relative overhead for a hash table decreases and it performs even better compared to a sorted array. In the former case, the anchor data structure has to be further extended by a priority queue-like index structure according to the usage count, as the anchor with the lowest usage count has to be removed.

So far, we have considered each query separately, and both data structures described so far cannot take any ad-

vantage from the fact that in our application many queries arrive at once[2]. Therefore we also investigate a third data structure specifically for large batches of queries. This data structure also uses a sorted array of anchors in main memory. If the queries arrive, they get also sorted, and then a kind of set comparison is performed sequentially, i.e. it is checked which candidate is an anchor. The sorting of the r queries takes time $O(r \log r)$. As both sets are now sorted, the comparison takes time $s + r$. As r will be considerably less than s, we neglect the time to sort.

Compared to $3r$ memory accesses that a hash table needs to handle r queries on average, the performance of sequential comparison looks bad. However, the $3r$ accesses each need a full memory access time t_a, while the $s + r$ memory accesses are at adjacent addresses, and hence can be served basically from the 1st level cache. Thus, for each access in sequential comparison we need only count the transfer time from memory to cache t_c. As the ratio between transfer time and main memory access time in current computers is about $t_a/t_c \approx 10$, a sequential comparison will outperform a hash table if

$$3r \cdot t_a \geq (r + s) \cdot t_c \ ,$$

i.e. if $r \geq s/29$. To produce such a number r of queries would mean that each processing unit also would have to follow such a large number of paths simultaneously. This is unrealistic as a processing node needs memory to store the state of each path that is followed simultaneously, an overhead whose size is comparable to the overhead of the hash table, and that reduces the number of anchors that can be stored. Also, such a large number of paths may only be of advantage if n is very large. Yet, this data structure is still interesting because it can also be used for a hard disk.

The sequential comparison is not well suited for a frequent dynamic update of single anchors. However, for a seldom bulk update, it can be treated similar to the sorted array with binary search.

3.2. Hard disk

When we use hard disks, our memory resource increases by a factor of $\beta = M/m$ which ranges from 40 (M = 40 GByte, m = 1 GByte) to 640 (M = 160 GByte, $m = 256$ MByte). At the same time, the access time of hard disks is much longer than the access time to main memory, typically 10 ms versus 10 ns, i.e. a factor $f_a \approx 10^6$. Also the transfer rate to and from a hard disk is lower than that between main memory and processor, typically 20 to 60 MByte/s versus ≥ 1 GByte/s, i.e. a factor $f_t \approx 16$ to 50.

This means, if we use the hard disk instead of the main memory to store the anchors, the performance will change accordingly. For a hash table (data structure 1), the performance will be worse by a factor of $\alpha_1 = f_a$, as long as the collision chains do not increase significantly. For a sorted array with binary search (data structure 2), the performance will be slightly worse, as each access (except the last 8 or 9, where we remain within one disk page) is slower by a factor of f_a, and the number of accesses needed increases by $\log \beta = \log(M/m)$. Hence, the performance is slower by

$$\begin{aligned} \alpha_2 &= \frac{(\log(\beta \cdot s) - 9) \cdot f_a + 9}{\log s} \\ &\approx f_a \cdot \left(1 + \frac{\log(\beta) - 9}{\log s}\right) \ . \end{aligned}$$

The performance of a sequential comparison (data structure 3) is slower by a factor of $\alpha_3 = f_t \cdot \beta$, as the number of comparisons increase by a factor of β, and the access to the data is slower by a factor of f_t.

In the case that the total number of anchors on hard disks, i.e. $s_{tot} = p \cdot s \cdot M/m$, is still less than $\sqrt{n}$, i.e. one is not yet in a saturation, then the number of steps and thus the number of query rounds will reduce by a factor of $\beta = M/m$. We assume that a fraction γ of the runtime t_{tot} has been spent with the query processing when using main memory to store anchors. When we switch over to using the hard disks with data structure i, the time spent to handle queries increases by a factor of α_i. Hence the new runtime t'_{tot} will be

$$t'_{tot} = \gamma \cdot \frac{t_{tot}}{\beta} \cdot \alpha_i + (1 - \gamma) \cdot \frac{t_{tot}}{\beta} \ ,$$

and the runtime changes by a factor

$$q = \frac{t'_{tot}}{t_{tot}} = \frac{1 + \gamma \cdot (\alpha_i - 1)}{\beta} \ .$$

The runtime decreases if $q < 1$, i.e. if

$$\alpha_i \leq \frac{\beta - 1}{\gamma} + 1 \ . \qquad (2)$$

A lower bound on the right hand side of Eq. (2) is reached when γ approaches 1, i.e. if almost all time is spent with query processing. In that case, Eq. (2) reduces to

$$\alpha_i \leq \beta \ .$$

This could only be achieved by data structures 1 and 2, if $f_a < \beta$, which typically is not the case.

Eq. 2 also reveals that in general, a runtime advantage by using hard disks can be achieved for

$$\gamma < \frac{\beta - 1}{\alpha_i - 1} \approx \frac{\beta}{\alpha_i} \ .$$

For data structures 1 and 2, this is achieved if $\gamma < \beta/f_a$. For data structure 3 this is achieved if $\gamma < 1/f_t$. The right

[2] They can only take advantage if two paths happen to reach the same candidate, an event that will be quite unlikely.

hand side in the first term is about $1/25,000$ for $\beta = 40$ and $f_a = 10^6$, the right hand side of the second term is about $1/16$ for $f_t = 16$.

Under this prospect, the sequential comparison is the only organization with a chance for hard disks to outperform main memory. However, this is only worthwhile if n is sufficiently large, i.e. if $n \geq s_{tot}^2$. For $p = 2^5$ and $s = 2^{25}$ (fits in $m = 1$ GByte of main memory), n would have to be larger than 2^{87} to be in the non-saturated case.

4. Conclusions

We have presented a study on different organizations of a distributed data structure suitable to handle large numbers of simultaneous queries. We have pointed out how these organizations would have to be extended to support dynamic insertions and deletions. The parameterized comparison allows to select the appropriate organization given a concrete cluster computer and a concrete problem size. Further work will center around implementations for hard-disk-based organizations and extended experimental work. For example, using a B^+ tree[3] organization could be an alternative to sequential search, as the number of accesses is reduced to $\log_k s$, and after each access a k-ary decision is done instead of a binary decision. As the disk pages are large enough, each node still fits into one page so that the time for each access remains constant, while the number of accesses reduces by a factor of

$$\frac{\log_2(s) - 9}{\log_k s} = \log_2 k - \frac{9}{\log_k s} .$$

E.g., for $s = 2^{24}$ and $k = 16$, the number of disk accesses reduces from 15 to 8.

Acknowledgements

We are deeply indebted to Jop Sibeyn. The investigation of different organisations was suggested by him. He is posted as missing since a ski trip in March 2005.

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. Heichler, J. Keller, J. F. Sibeyn. Parallel Storage Allocation for Intermediate Results During Exploration of Random Mappings. In Proc. *PARS 2005*, Lübeck, June 2005.

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

[3] A data structure where each internal tree node has $k \geq 3$ children. The depth for a tree with s leaves is $\log_k s$, compared to $\log_2 s$ in a binary tree.