Parallel Exploration of an Unknown Random Forest

Jörg Keller
Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
Joerg.Keller@FernUni-Hagen.de

Patrick Eitschberger
Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
Patrick.Eitschberger@FernUni-Hagen.de

***Abstract*—We investigate how to explore with a parallel machine a random and unknown forest, of which we only know an upper bound on the total size, some leaves to start from, and the roots. The size of the forest is too large to represent it explicitly in the machine's main memory. Instead for each node, its parent node is given by an oracle, i.e. a piece of code of which no particulars may be known. We present a parallel algorithm, and use experiments to find parameter settings that influence the runtime favorably.**

***Index Terms*—Parallel graph algorithms, Multicore Computing, Performance Tuning.**

I. Introduction

Assume that you are given the leaves of a tree (or forest), and that for each node x, you can get its parent node $p(x)$ by an oracle (and thus also learn if you have reached the root), e.g. by a piece of code about which no further details are known than that its execution is done in a constant number of steps. You know that the tree is so large that only a tiny fraction of the nodes can be marked as visited, but you have no further information about the structure, regularity or depth of the tree. The task at hand is to explore the tree completely, starting at the leaves. The exploration is to be done by a parallel machine, in an efficient manner, i.e. with a reasonable speedup (we will discuss later what this can mean).

While the scenario above seems rather abstract, it has a very concrete application: the leaves are random seed states of a pseudo-random number generator with moderately large state space (i.e. of size less than 2^{100}) and a non-bijective state transition function, and the exploration serves to find some of the cycles, e.g. to find where to modify the state transition function (which is the oracle) to increase the cycle length [1].

In the present paper, we will investigate how to overcome the challenges when implementing the parallel exploration on a multicore machine. Basically, we will use the concept of distinguished points to detect that paths have met, so that only one has to be followed further, and we will explore the tradeoff involved in deciding how many paths should be followed in parallel: at least p if p hardware threads are available. More paths might be advantageous, but too many again hurt performance.

The remainder of this article is structured as follows. In Section II, we provide background information on random forests and parallel graph algorithms. In Section III, we present our implementation and how it solves the challenges at hand. In Section IV, we report preliminary experimental results, and in Section V we conclude and give an outlook to future work.

II. Background

A *tree* is a connected graph without cycles. We consider directed trees, where arcs go from child to parent nodes, and the root either has outdegree 0, or is its own parent node. A *forest* is a collection of trees. A *random* tree is a tree created by some random process. There are a number of models for random trees (cf. e.g. [2]), which differ in their properties.

By a random forest, we assume that we have a set of n nodes, and that each node randomly chooses a parent node. In such a graph, each weakly connected component consists of a cycle plus some trees attached to it via their roots (see Figure 1), or put otherwise, consists of a tree plus one back edge. We choose one node on the cycle as being the root, and its outgoing edge as the back edge. The trees in such a graph are rather ragged, i.e. the largest tree (i.e. weakly connected component) is expected to contain about 75% of all nodes, and has a depth of about $2\sqrt{n}$, of which about half is on the cycle [3]. The leaves are more or less evenly distributed over all levels of the tree. Such structures appear in practice as state transition graphs in cryptographic primitives such as stream ciphers or pseudo-random number generators, where in embedded sytems, n could be 2^{64} or 2^{80} [1]. As such, these graphs cannot be represented in memory, but with the code of the generator, the parent of a node can be found quickly.

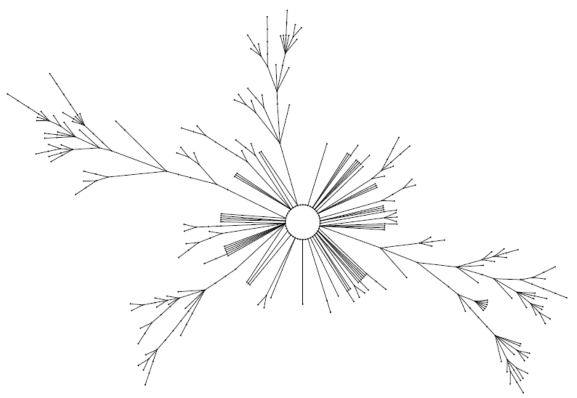
Fig. 1. Connected component of a state transition graph, taken from [4].

To explore such a graph, one can start at randomly chosen points, and follow the paths originating from there until they have reached (and surrounded) the cycle of their component. Thus, one can determine the cycle lengths of the larger components, compute lower bounds on the depths of the trees, and receive some kind of sampled spanning tree structure of the components. However, as most of these starting points will be in the same tree, the paths originating from them will merge soon and much time is wasted in following them completely.

To prevent this, each path records from time to time a node that it has passed, and queries from time to time if it has reached a node recorded from another path. However, as querying such a data structure in every step would be slow, the nodes to be recorded should have a certain property, and only when a node with such property is reached, the data structure must be queried. Checking if a node has this property should be much faster than querying the data structure. This concept has been applied repeatedly under different names such as distinguished points, anchors, or candidates [4]–[7]. As the above mentioned property, typically a certain bit pattern in the binary representation of the node index is used. As the trees are considered random, choosing a fixed bit pattern is still considered independent enough to be — in practice — similar to assigning this property to a random subset of nodes. Thus, if i bits must have a certain value in the pattern, the average distance between candidate nodes is assumed to be 2^i. Yet, we note that it is not guaranteed that a path will meet any candidate: if e.g. a tree contains only nodes with even index, and the bit pattern requires that the lowermost bit is set to 1, then no candidate will be encountered. Hence, this situation must be detected, and such paths must be handled separately. This can e.g. be done by additionally recording a node reached after 2^j steps, where $j = 1, 2, 3, \ldots$, and checking in each step if the last recorded node has been reached. However, we will assume that at least the tree root is a candidate node.

III. Parallel Algorithm

Assume that we are given s starting points, and have a shared-memory machine with p processors to follow the paths starting from there. Then, as typically $s \gg p$, the question arises if to proceed in a DFS-like or BFS-like manner. By DFS-like manner, we understand starting p paths, follow them until they terminate by either reaching the root or a candidate that was already visited previously, and only add another path if one path terminates. By BFS-like manner, we understand to start following all s paths immediately and have each of the p threads follow s/p paths simultaneously.

By following paths, we mean that each thread follows its assigned path or paths until they have reached the next candidate. We will call this a *round* in the sequel. Then, a search data structure that contains all candidates reached is queried to find out if one of the candidates reached has been reached before, in which case the respective path can terminate (and be replaced by another in the DFS-like approach). The search data structure is updated with the newly reached candidates, and the next round starts. This continues until all paths have been followed completely.

Both approaches have their merits. If an initial path is rather long, then many of the paths started later can terminate quickly because they soon will merge into the long path already pursued. This calls for a DFS-like approach. In contrast, the time to reach the next candidate can differ widely between threads, as the number of steps necessary follows a geometric distribution with success probability $P = 2^{-i}$, if we assume that each node is a candidate with probability 2^{-i}. Thus, while the expected number of steps is $1/P = 2^i$ for every thread, the standard deviation is $\sqrt{1-P}/P$, i.e. close to the expectation as P is small. Hence, following multiple threads to their next candidate simultaneously balances the round workloads of the different threads, and can reduce the thread idle time. This calls for a BFS-like approach.

We will generalize from these two extremal viewpoints by having each thread follow k paths simultaneously, where $1 \leq k \leq s/p$, cf. Alg. 1. We will determine advantageous values for k by experiments in the next section.

Algorithm 1 Parallel algorithm to compute one round.

Precondition: $C = \{n_1, \ldots, n_{pk}\}$ set of unvisited candidates, ISCAND checks if state is candidate.

```
1: function ROUND($C$)
2:     for all $i \leftarrow 0$ to $p-1$ do                ▷ Parallel Loop
3:         for $j \leftarrow 1$ to $k$ do
4:             $p_i \leftarrow n_{ik+j} \in C$
5:             repeat
6:                 $p_i \leftarrow$ TRANS($p_i$)
7:             until ISCAND($p_i$)       ▷ Reached next candidate
8:             $C_{new} \leftarrow C_{new} \cup \{p_i\}$
9:     return $C_{new}$
```

Splitting the algorithm (cf. Alg. 2) in rounds where first paths are followed in parallel, and then the search data structure (which is a hashmap in our case) is accessed sequentially, is due to our intention to run this algorithm later on an accelerator (like a GPU), where control-flow intensive parts like search might be difficult to implement. However, to find out relevant parameter settings like the choice of k first, we have first implemented it on a multicore CPU with OpenMP.

Although we assume (see previous section) that each path will ultimately reach a candidate, for reasons of load balance we still use a timeout, i.e. when a path does not meet a candidate after a large number of steps (such as $10 \cdot 2^i$), it is interrupted, and further handled in a separate way, e.g. completey on the host instead of the accelerator. Also, the number of paths might be less than $p \cdot k$ in the last rounds, which we did not specifically address in Alg.s 1 and 2, but which is handled in our implementation.

Please note that the situation is different if the tree structure is known before hand. There, at least in the BFS-approach each node might be labeled with the minimum distance from a starting point. Along a path, the distances increase. As soon

Algorithm 2 Algorithm to split starting points into rounds.

Precondition: $S = \{c_1, \ldots, c_n\}$ set of starting points.

```
1:  function COMPUTE($S$)
2:      $C \leftarrow \emptyset$
3:      repeat
4:          $C \leftarrow C \cup \{pk - |C|$ elements from $S\}$
5:          $S \leftarrow S \setminus C$
6:          $C \leftarrow$ ROUND($C$)
7:          for each $c \in C$ do
8:              if $c$ is already visited then
9:                  $C \leftarrow C \setminus \{c\}$
10:             else
11:                 mark $c$ as visited
12:     until $S = \emptyset$                    ▷ All paths processed
```

as a path reaches a parent node with a smaller distance label than its child, the path can be terminated, as the parent has already been visited previously on another path. Thus the tree can be partitioned into paths of known length, and the problem turns into a question of load balancing, where the task is to assign each processor a number of paths such that the total workload of any processor is minimized.

IV. Experiments

All experiments presented in this section were executed on a machine with four AMD Opteron 6328 processors, i.e. with 32 cores in total. The cores have a maximum operating frequency of 3,200 MHz (boost frequency 3,800 MHz). The main memory has a size of 512 GiByte. The machine runs under the Fedora 23 operating system. The programs are compiled with `gcc` 4.7.2 with options `-O3` and `-fopenmp`. The time measurements are done with the `time` command, using the `real` time.

The first set of experiments is done with $s = 16,384$ randomly chosen candidate starting points. Candidates are defined as nodes with bits 8 to 31 set to 1. Each run uses the same set S_1 of starting points. As transition function, the index of a graph node is interpreted as a `double` value in the range $[0;1]$, and $f(x) = a \cdot x \cdot (1-x)$ is used to compute the successor node, where $a = 3.99$. For $a \leq 4$, this function has a range of $[0;1]$, and for values of a close to 4 exhibits a chaotic behaviour on the reals, i.e. will also look quite "random" on the finite number representations used here. The doubles in the range $[0;1]$ correspond to 62 bit unsigned integers[1], as the sign bit and the uppermost bit of the biased exponent are 0.

We first execute our algorithm sequentially, and get a runtime of 168 seconds, which only increases for very large values of k. We then execute our algorithm with 32 threads for different values of k. Each run was repeated 3 times, but the variation in runtime is below 2%. Figure 2 depicts the runtime achieved. We see that the runtime is minimized for k around 24. The speedup compared to the sequential version is around 16, which can be explained by the sequential part to update the set of already visited candidates after each round, and the insufficient parallelism in the last rounds, when only a few active paths are left. Applying Amdahl's law assigns the sequential part about 3% of the total work, so most work still is done in parallel.

The position of the minimum can be explained by the fact that for small k, the chance of exploring a long path within the first starting points is small, so that the paths starting from the next starting points cannot profit from meeting this path soon. On the other hand, for large k, all paths progress simultaneously, and the long path does not progress fast enough. Please note that the reduction in runtime from $k = 256$ to $k = 512$ for set S_1 has been confirmed in several runs, but cannot be explained presently.

Figure 2 also depicts the runtimes for another set S_2 of randomly chosen starting points. The behaviour is similar, thus we assume that this behaviour is independent of the concrete set of starting points. Figure 2 finally depicts the runtimes when the transition function is changed by using $a = 3.98$ (set of starting points S_3). Also here, the runtime dependence on k is similar, although the position of the minimum is slightly shifted.

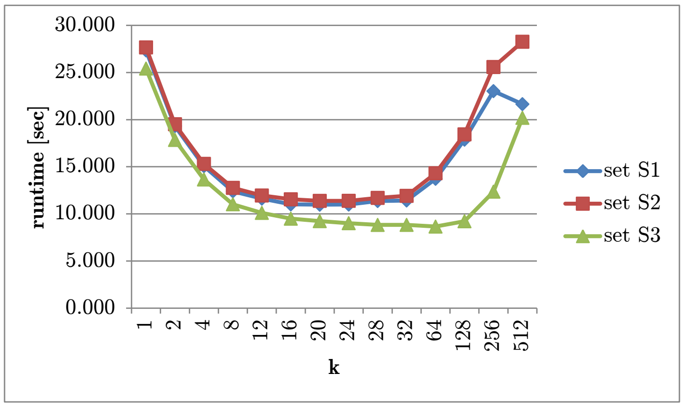

Fig. 2. Runtime for different sets of starting points and transition functions, depending on parameter k.

In a second set of experiments, we investigated the influence of the candidate definition. For the set S_1 of starting points, and $k = 24$, we modified the candidates by defining them as bits 8 to i set, where i was decreased from 31 to 24. Please note that the starting points in S_1 are candidates under each of these definitions. The runtimes are shown in Figure 3. When the candidate is defined by one bit less, then the number of candidates doubles and their average distance is halved. One would expect that having candidates with smaller distance is favorable, as it can be detected earlier when two paths have merged into one. However, putting candidates too close seems to increase the sequential part of the program, as the rounds, where paths are followed to the next candidate node,

[1]To be more exact, 62 bit unsigned integers correspond to the doubles in the range $[0;2)$, but if we choose a starting point > 1, we divide by 2, and application of the transition function only produces values in $[0,1]$. Thus, a fraction of 2^{-7} of the 62 bit unsigned integers is not used, assuming that denormalized floating point representations are supported and used.

get shorter. Earlier detection might happen, but is not certain. If e.g. candidates are d steps apart, and paths merge $3d/4$ steps after the last candidate, and $d/4$ steps before meeting the next candidate, then halving the distance is no advantage. Thus, the overhead at some point gets higher than the gain.

We are aware that the optimal choice of k might depend on the candidate definition, which is a topic of further research.

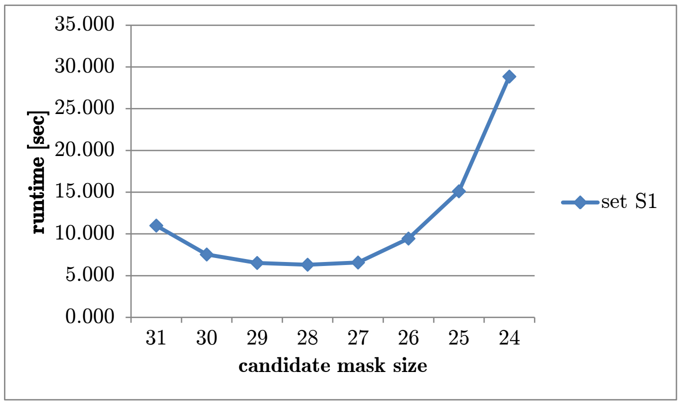

Fig. 3. Runtime for starting points S_1 and $k = 24$ for different candidate definitions.

Finally, we have investigated how to handle timeout. The straightforward solution is to apply a fixed maximum number of steps that a path is followed without reaching a candidate. This maximum distance should be larger than the average distance d between candidates. We have used a factor c, i.e. stop following a path after $c \cdot d$ steps. Choosing c too large however might hurt performance, as some threads might already have completed finishing their k paths to the next candidate node, while some might use most of the maximum of $k \cdot c \cdot d$ steps. Hence, we also tried to balance between paths, by applying a bound of

$$b_j = \frac{k + 2j}{3} \cdot c \cdot d - \sum_{l<j} r_l$$

for the j-th path that a thread follows, where $r_l \leq b_l$ is the number of steps taken by the l-th path (with $r_0 = b_0 = 0$). In total, the number of steps is still $k \cdot c \cdot d$ at most, but some paths could use more than cd steps without taking away too much of the others. The goal is to be able to apply a smaller value of c without introducing additional paths that do not reach a candidate before their timeout.

Table I depicts the number of paths that did not reach a candidate before time out in the first two rounds, in an experiment with 1,024 starting points and $k = 8$ paths per thread. The results indicate that the more flexible timeout allows to use a smaller c.

TABLE I
Number of paths with timeout for fixed and variable timeout bounds.

c	fixed	variable
2	81	5
4	14	0
6	1	0
8	0	0

V. Conclusions

We have presented an algorithm to explore a large, random tree or forest of unknown structure in parallel. We have discussed implementation details and have explored parameter settings that reduce the parallel runtime. In our future work, we plan to investigate a GPU-based implementation, which involves further parameters, and to apply the algorithm in the security domain to explore state spaces of pseudo-random number generators.

Acknowledgment

We are very grateful to Christoph Kessler for discussions about exploration of trees of known and unknown structure. We also thank Torsten Linss for the possibility to use a machine owned by FernUniversität's numerical analysis group.

References

[1] G. Spenger and J. Keller, "Tweaking cryptographic primitives with moderate state space by direct manipulation," in *IEEE International Conference on Communications (ICC'17)*, 2017.

[2] M. Drmota, *Random Trees, An Interplay between Combinatorics and Probability*. Springer, 2009.

[3] P. Flajolet and A. M. Odlyzko, "Random mapping statistics," in *Advances in Cryptology*. Springer, 1990, pp. 329–354.

[4] A. Beckmann, J. Fedorowicz, J. Keller, and U. Meyer, "A structural analysis of the A5/1 state transition graph," in *Proc. First Workshop on GRAPH Inspection and Traversal Engineering*, ser. Electronic Proceedings in Theoretical Computer Science, vol. 99. Open Publishing Association, 2012, pp. 5–19.

[5] D. Denning, *Cryptography and Data Security*. Addison-Wesley, 1982.

[6] J. Hong, G. W. Lee, and D. Ma, "Analysis of the parallel distinguished point tradeoff," in *Progress in Cryptology – INDOCRYPT 2011*, D. J. Bernstein and S. Chatterjee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 161–180.

[7] J. Heichler and J. Keller, "A distributed query structure to explore random mappings in parallel," in *Proc. 14th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2006, pp. 173–177.