Distributed Verification:
Exploring the Power of Raw Computing Power*

Luboš Brim
Faculty of Informatics, Masaryk University, Brno, Czech Republic

1 Brute-Force in Distributed Verification

With the increase in complexity of computer systems, it becomes more important
to develop formal methods for ensuring their quality and reliability. Various tech-
niques for automated and semi-automated analysis and verification have been
successfully applied to real-life computer systems. However, these techniques
are computationally hard and memory intensive in general and their applica-
ability to extremely large systems is limited. The major hampering factor is the
state space explosion problem due to which large industrial models cannot be
efficiently handled by a single state-of-the-art computer.

Much attention has been focused on the development of approaches to battle
the state space explosion problem. Many techniques, such as abstraction, state
compression, state space reduction, symbolic state representation, etc., are used
to reduce the size of the problem to be handled allowing thus a single computer to
process larger systems. There are also techniques that purely focus on increasing
the amount of available computational power. These are, for example, techniques
to fight memory limits with efficient utilisation of an external I/O device [2,
19,24,28,32], or techniques that introduce cluster-based algorithms to employ
aggregate power of network-interconnected computers.

Cluster-based algorithms perform their computation simultaneously on a
number of workstations that are allowed to communicate and synchronise them-
seves by means of message passing. Cluster-based algorithms can thus be charac-
terised as parallel algorithms performing in a distributed memory environment.
The algorithms prove their usefulness in verification of large-scale systems. They
have been successfully applied to symbolic model checking [22,23], analysis of
problems [7,9,21].

The idea of parallel verification appeared already in the very early years of
the formal verification era. However, inaccessibility of cheap parallel comput-
ers together with negative theoretical complexity results excluded this approach
from the main stream in formal verification. The situation changed dramatically
during the past several years. Computer progress over the past two decades has
measured several orders of magnitude with respect to various physical param-
eters such as computing power, memory size at all hierarchy levels from caches
to disk, power consumption, physical size and cost. In particular, the focus of

* Supported by the Grant Agency of Czech Republic grant No. 201/06/1338
novel computer architectures in parallel and distributed computing has shifted away from unique massively parallel systems competing for world records towards smaller and more cost effective systems built from personal computer parts. In addition, recent shift in the emphasis of research on parallel algorithms to pragmatic issues provided practically efficient algorithms for solving computationally hard problems. As a matter of fact, interest in parallel verification has been revived.

Asymptotic complexity analysis has turned out to be a surprisingly effective technique to predict the performance of algorithms, to classify some of the problems as computationally hard. In complexity theory, the class NC (for "Nick's Class") is the set of problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors. Just as the class P can be thought of as the tractable problems, so NC can be thought of as the problems that can be efficiently solved on a parallel computer. NC is a subset of P because parallel computers can be simulated by sequential ones. It is unknown whether NC = P, but most researchers suspect this to be false, meaning that there are some tractable problems which are probably "inherently sequential" and cannot be sped up significantly by using parallelism. Just as the class NP-complete can be thought of as "probably intractable", so the class P-complete can be thought of as "probably not amenable to efficient parallelisation".

Many efficient parallel algorithm can be found for selected subclasses of P-complete problems. Most problems of practical importance call for computer-intensive methods. Accordingly, the attitude towards "inherently sequential" began to change. Not all "inherently sequential" problems are equally difficult to parallelise. There are many problems where the average, or typical, instance is relatively easy to solve, and only the worst-case instances are computationally very demanding. For example, several efficient and highly parallel algorithms are known for solving the maximum flow problem, which is P-complete. The standard complexity theory for P-complete problems asserts the non-existence of algorithms that are efficiently parallelised across an entire problem class, but ignores the possibility that many instances, perhaps including those of interest, can be solved efficiently.

Efficient parallel solution of many problems often requires invention of original, novel approaches radically different from those used to solve the same problems sequentially. Classical examples are list rankings, connected components, depth-first search in planar graphs etc. The rules of algorithm design and analysis have to be changed drastically: we still have to devise and implement clever algorithms, but complexity is not measured asymptotically in terms of the size of the problem; it is measured by actually counting operations, disk accesses, communications, and seconds. In addition, development of methods, tools, and practices for assessing and refining algorithms through experimentation is unavoidable, extensive use of various techniques for efficient implementation as known from algorithm engineering for parallel computation (see e.g.,[1]) underpins the new approach.
One of the simplest approach to parallel problem solving is brute-force parallel exploration of the state space. All possible states are generated in some systematic way and checked for the desired property. If the state space is finite, a success is guaranteed provided we can wait long enough. Although exhaustive search is conceptually simple and often effective, such an approach to problem solving is sometimes considered inelegant. The continuing increase in computing power and memory sizes has revived interest in brute-force techniques. Many “real” problems exhibit no regular structures to be exploited, and that leaves exhaustive enumeration as the only approach in sight.

Model checking finite state systems in practise is bug hunting by exhaustively searching a seemingly “chaotic” state space for failed assertions. Hence, brute-force approach to verification is in some sense quite natural and, more importantly, in fighting the practical limitations caused by the state explosion problem the exhaustive search appears to be the only viable approach. Reachability is a simple verification problem that is tractable in parallel as the exhaustive enumeration of the state space can be divided into independent subtasks. Reachability is thus one of the basic techniques for the distributed verification.

As a demonstration of the effectiveness of parallel reachability for solving theoretically intractable problems we consider parallel LTL model-checking. The state space to be explored is a directed graph with an initial state and a set of accepting states. In this state space we want to search for accepting cycles. The simplest brute-force approach is first to search for accepting states reachable from the initial one and then for each accepting vertex to test the reachability of the vertex from itself. However, we can do significantly better by using extra information.

The amount of additional information stored is a major issue for the space complexity. We need to be careful in achieving a good balance between space and speed, ideally we use storage space proportional to the logarithm of the number of states.

We present several algorithms for parallel enumerative LTL model-checking that are based on performing repeated reachability. The demonstration includes algorithms presented in [5, 10, 11, 14, 3, 4, 12, 13]. The state space is searched for states carrying a particular information, states might be re-visited as the information changes. To guide the search various data structures are used. Besides the traditional stacks and queues, more complicated data structures might be employed as well.

2 Algorithms for Accepting Cycle Detection

Although the algorithms are meant for cluster-based computing we describe the main ideas primarily as sequential, leaving thus many technical details related to distributed computation out.

The problem we consider comes out from the automata-based procedure to decide LTL model checking problem as introduced in [34]. The approach exploits the fact that every set of executions expressible by an LTL formula
is an $\omega$-regular set and can be described by a Büchi automaton. In particular, the approach suggests to express all system executions by a system automaton and all executions not satisfying the formula by a property or negative claim automaton. These automata are combined into their synchronous product in order to check for the presence of system executions that violate the property expressed by the formula. The language recognised by the product automaton is empty if and only if no system execution is invalid.

The language emptiness problem for Büchi automata can be expressed as an accepting cycle detection problem in a graph. Each Büchi automaton can be naturally identified with an automaton graph which is a directed graph $G = (V, E, s, A)$ where $V$ is the set of vertexes ($n = |V|$), $E$ is a set of edges ($m = |E|$), $s$ is an initial vertex, and $A \subseteq V$ is a set of accepting vertexes ($a = |A|$). We say that a reachable cycle in $G$ is accepting if it contains an accepting vertex. Let $A$ be a Büchi automaton and $G_A$ the corresponding automaton graph. Then $A$ recognises a nonempty language iff $G_A$ contains an accepting cycle. The LTL model-checking problem is thus reduced to the accepting cycle detection problem in automaton graphs.

The best known enumerative sequential algorithms for detection of accepting cycles are the Nested DFS algorithm [17, 27] (implemented, e.g., in the model checker SPIN [26]) and SCC-based algorithms originating in Tarjan’s algorithm for the decomposition of the graph into strongly connected components (SCCs) [33]. While Nested DFS is more space efficient, SCC-based algorithms produce shorter counterexamples in general. Here, for simplicity reasons, we will not be dealing with the counterexample generation subtask.

It is a well-known fact that computing depth-first search postorder is P-complete [31], hence probably inherently sequential. This means that none of the two algorithms can be easily adapted to work on a parallel machine. A few fundamentally different cluster-based techniques for accepting cycle detection appeared though. They typically perform repeated reachability over the graph. Unlike the postorder problem, reachability is a graph problem which can be well parallelised, hence the algorithms might be transformed to cluster-based algorithms that work with reasonable increase in time and space.

### 2.1 Maximal Accepting Predecessor (MAP)

A vertex $u$ is a predecessor of a vertex $v$ if there is a non-trivial path from $u$ to $v$. The main idea behind the algorithm is based on the fact that each accepting vertex lying on an accepting cycle is its own predecessor.

Instead of expensive computing and storing of all accepting predecessors for each (accepting) vertex, the algorithm computes a single representative accepting predecessor for each vertex. We presuppose a linear ordering $\prec$ of vertexes (given e.g. by their memory representation) and choose the maximal accepting predecessor. For a vertex $u$ we denote its maximal accepting predecessor in the graph $G$ by $map_G(u)$. Clearly, if an accepting vertex is its own maximal accepting predecessor ($map_G(u) = u$), it is its own predecessor and it lies on an accepting cycle. Unfortunately, the opposite does not hold in general. It can happen that
the maximal accepting predecessor for an accepting vertex on a cycle does not lie on the cycle. Such vertexes can be safely deleted from the set of accepting vertexes (by applying the deleting transformation) and the accepting cycle still remains in the resulting graph. Whenever the deleting transformation is applied to automaton graph $G$ with $map_{c}(v) \neq v$ for all $v \in V$, it shrinks the set of accepting vertexes by those vertexes that do not lie on any cycle.

As the set of accepting vertexes can change after the deleting transformation has been applied, maximal accepting predecessors must be recomputed. It can happen that even in the graph $del(G)$ the maximal accepting predecessor function is still not sufficient for cycle detection. However, after a finite number of applications of the deleting transformation an accepting cycle is certified. For an automaton graph without accepting cycles the repetitive application of the deleting transformation results in an automaton graph with an empty set of accepting vertexes.

Time complexity of the algorithm is $O(n^2 \cdot m)$, where $n$ is the number of accepting vertexes. Here the factor $n \cdot m$ comes from the computation of the $map$ function and the factor $a$ relates to the number of iterations.

One of the key aspects influencing the overall performance of the algorithm is the underlying ordering of vertexes used by the algorithm. In order to optimise the complexity one aims to decrease the number of iterations by choosing an appropriate vertex ordering. Ordering $\prec$ is optimal if the presence of an accepting cycle can be decided in one iteration. It can be easily shown that for every automaton graph there is an optimal ordering. Moreover, an optimal ordering can be computed in linear time.

An example of an optimal ordering is depth-first search postorder. Unfortunately, the optimal ordering problem, which is to decide for a given automaton graph and two accepting vertexes $u, v$ whether $u$ precedes $v$ in every optimal ordering of graph vertexes, is $P$-complete [12] hence unlikely to be computed effectively in a distributed environment. Therefore, several heuristics for computing a suitable vertex ordering are used. The trivial one orders vertexes lexicographically according to their bit-vector representations. The more sophisticated heuristics relate vertexes with respect to the order in which they were traversed. However, experimental evaluation has shown that none of the heuristics significantly outperforms the others. On average, the most reliable heuristic is the one based on breadth-first search followed by the one based on (random) hashing.

2.2 Eliminating bad states (OWCTY)

The accepting cycle detection problem can be directly reformulated as a question whether the automaton graph contains a nontrivial accepting strongly connected component.

A strongly connected component (SCC) of $G = (V,E,s,A)$ is a maximal (with respect to set inclusion) set of vertexes $C \subseteq V$ such that for each $u,v \in C$, the vertex $v$ is reachable from $u$ and vice versa. The quotient graph of $G$, $Q(G)$, is a graph $(W,H)$ where $W$ is the set of the SCCs of $G$ and $(C_1,C_2) \in H$ if and only if $C_1 \neq C_2$ and there exist $r \in C_1,s \in C_2$ such that $(r,s) \in E$. The height
of the graph $G$, $h(G)$, is the length of the longest path in the quotient graph of $G$ (note that the quotient graph is acyclic). A strongly connected component is **trivial** if it has no edges, **initial** if it has no predecessor in the quotient graph, and **accepting** if it contains an accepting vertex.

The inspiration for the distributed SCC-based algorithm for detection of accepting cycles is taken from symbolic algorithms for cycle detection, namely from SCC hull algorithms. SCC hull algorithms compute the set of vertexes containing all accepting components. Algorithms maintain the approximation of the set and successively remove non-accepting components until they reach a fixed point. Different strategies to remove non-accepting components lead to different algorithms. An overview, taxonomy, and comparison of symbolic algorithms can be found in independent reports [20] and [30].

As the base for the enumerative algorithm presented here the One Way Catch Them Young strategy [20] has been chosen. The enumerative algorithm works on individual vertexes rather than on sets of vertexes as is the case in symbolic approach. A component is removed by removing its vertexes. The algorithm employs two rules to remove vertexes of non-accepting components:

- if a vertex is not reachable from any accepting vertex then the vertex does not belong to any accepting component and
- if a vertex has in-degree zero then the vertex does not belong to any accepting component.

Note that an alternative set of rules can be formulated as

- if no accepting vertex is reachable from a vertex then the vertex does not belong to any accepting component and
- if a vertex has out-degree zero then the vertex does not belong to any accepting component.

This second set of rules results in an algorithm which works in a backward manner and we will not describe it explicitly here.

The presented SCC-based algorithm in its forward version requires the entire automaton graph to be generated first. The same is true for the backward version. Moreover, the backward version actually needs to store the edges to be able to perform backward reachability. This is however payed out by relaxing the necessity to compute successors, which is in fact a very expensive operation in practice.

Time complexity of the algorithm is $O(h \cdot m)$ where $h = h(G)$. Here the factor $m$ comes from the computation of Reachability and Elimination functions and the factor $h$ relates to the number of external iterations. In practise, the number of external iterations is very small even for large graphs. This observation is supported by experiments in [20] with the symbolic implementation and hardware circuits problems. Similar results are communicated in [29] where heights of quotient graphs were measured for several models. As reported, 70% of the models has height smaller than 50.

A positive aspect of SCC-based algorithms is their effectiveness for weak automaton graphs. A graph is weak if each SCC component of $G$ is either fully
contained in $A$ or is disjoint with $A$. For weak graphs one iteration of the SCC-based algorithm is sufficient to decide accepting cycles. The studies of temporal properties [18, 15] reveal that verification of up to 90% of LTL properties leads to weak automaton graphs.

Last but not least, SCC-based algorithms can be effortlessly extended to automaton graphs for other types of nondeterministic word automata like generalised Büchi automata and Streett automata.

2.3 Maximal number of accepting predecessors (NEGIC)

Consider maximal number of accepting vertexes on a path from the source to a vertex, the maximum being taken over all paths. For vertexes on an accepting cycle the maximum does not exist because extending a path along the cycle adds at least one accepting vertex.

For computing the maximal number of accepting predecessors the algorithm maintains for every vertex $v$ its “distance” label $d(v)$ giving the maximal number of accepting predecessors, parent vertex $p(v)$, and status $S(v) \in \{\text{unreached}, \text{labelled}, \text{scanned} \}$. Initially, $d(v) = \infty$, $p(v) = \text{nil}$, and $S(v) = \text{unreached}$ for every vertex $v$. The method starts by setting $d(s) = 0$, $p(s) = \text{nil}$ and $S(s) = \text{labelled}$, where $s$ is the initial vertex. At every step a labelled vertex is selected and scanned. When scanning a vertex $u$, all its outgoing edges are relaxed (immediate successors are checked). Relaxation of an edge $(u, v)$ means that if $d(v)$ is an accepting vertex then $d(v)$ is set to $d(u) + 1$ and $p(v)$ is set to $u$. The status of $u$ is changed to scanned while the status of $v$ is changed to labelled. If all vertexes are either scanned or unreached then $d$ gives the maximal number of accepting predecessors. Moreover, the parent graph $G_p$ is the graph of these “maximal” paths. More precisely, the parent graph is a subgraph $G_p$ of $G$ induced by edges $(p(v), v)$ for all $v$ such that $p(v) \neq \text{nil}$.

Different strategies for selecting a labelled vertex to be scanned lead to different algorithms. When using FIFO strategy to select vertexes, the algorithm runs in $O(m + n)$ time in the worst case. For graphs with reachable accepting cycles there is no “maximal” path to the vertexes on an accepting cycle and the scanning method must be modified to recognise such cycles. The algorithm employs the walk to root strategy which traverses a parent graph. The walk to root strategy is based on the fact (see e.g. [16]) that any cycle in parent graph $G_p$ corresponds to an accepting cycle in the automaton graph.

The walk to root method tests whether $G_p$ is acyclic. Suppose the parent graph $G_p$ is acyclic and an edge $(u, v)$ is relaxed, i.e. $d(v)$ is decreased. This operation creates a cycle in $G_p$ if and only if $v$ is an ancestor of $u$ in the current $G_p$. Before applying the operation, we follow the parent pointers from $u$ until we reach either $v$ or $s$. If we stop at $v$ a cycle is detected. Otherwise, the relaxation does not create a cycle. However, since the path to the initial vertex can be long, the cost of edge relaxation becomes $O(n)$ instead of $O(1)$. In order to optimise the overall computational complexity, amortisation is used to pay the cost of checking $G_p$ for cycles. More precisely, the parent graph $G_p$ is tested only after the underlying scanning algorithm performs $\Omega(n)$ relaxations. The running time
is thus increased only by a constant factor. The worst case time complexity of the algorithm is thus $O(n \cdot m)$.

2.4 Back-Level Edges (BLEdge)

The algorithm builds on breadth-first search (BFS) exploration of the graph. BFS is typically used in graph algorithms that work with distances and distances can also be used to characterise cycles in a graph.

Distance of a vertex $u \in V$, $d(u)$, is the length of a shortest path from the initial vertex to the vertex $u$. The set of vertexes with the same distance is called level. An edge $(u, v) \in E$ is called a back-level edge if $d(u) \geq d(v)$.

The key observation connecting the cycle detection problem with the back-level edge concept is that every cycle contains at least one back-level edge. Back-level edges are therefore used as triggers which start a cycle detection. However, it is too expensive to test every back-level edge for being a part of a cycle. The algorithm therefore integrates several optimisations and heuristics to decrease the number of tested edges and speed-up the cycle test.

The BFS procedure which detects back-level edges runs in time $O(m + n)$. Each back-level edge has to be checked to be on a cycle, which requires linear time $O(m + n)$ as well. In the worst case there can be $O(m)$ back-level edges, hence the overall time complexity of the algorithm is $O(m \cdot (m + n))$. Its space complexity is $O(m + n)$.

3 Comparing the Algorithms

To compare the algorithms we can use the standard asymptotic complexity measures. In Table 1 we summarise time and space complexity of the algorithms, where $n$ is the number of vertexes, $m$ is the number of edges, $h$ is the height of the SCC tree, and $\alpha$ is the number of accepting vertexes in the graph. It seems that Nested DFS and Tarjan’s algorithm are asymptotically the best algorithms, both sharing the first place. However, Nested DFS needs only two additional bits, while Tarjan’s algorithm needs to store more data to handle DFS and completion numbers. Despite being asymptotically the same, and in fact optimal, in reality Nested DFS is more space efficient than Tarjan’s algorithm. All the other algorithms are generally worse. This is true for the worst case analysis when considering these algorithms as sequential ones.

The situation looks quite different if we want to adapt the algorithms for the distributed environment. Both, the Nested DFS and the Tarjan’s algorithm perform badly, because they rely on depth first search postorder of vertexes and in a distributed environment we need to use very expensive techniques to assure the postorder. On the other hand, all the other algorithms can be parallelised easily. The reason is that they do not use any particular order of vertexes and the graph can be explored using local information only.

Still, despite the purely theoretical asymptotic worst case differences, there are many other, often more practical, aspects that may make a difference. Typical
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
<th>Space complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested DFS</td>
<td>(O(m + n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Tarjan's algorithm</td>
<td>(O(m + n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>MAP</td>
<td>(O(a^5 \cdot m))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>BLEV</td>
<td>(O(m \cdot (m + n)))</td>
<td>(O(m + n))</td>
</tr>
<tr>
<td>NEGOC</td>
<td>(O(m \cdot n))</td>
<td>(O(m + n))</td>
</tr>
<tr>
<td>OWCTY</td>
<td>(O(h \cdot (m + n)))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

Table 1. Asymptotic time and space complexity

example is, when we consider some particular instances of the problem, like graphs with or without accepting cycles. Consider the algorithm MAP. If there is no accepting cycle in the graph, the number of iterations is typically very small in comparison to the size of the graph (up to 40-50). Thus, the algorithm exhibits nearly linear performance. Some other points have been also made when describing the algorithms.

Often we do not know the problem belongs to the specific instance class. However, sometimes we might be able to classify the problem instance in advance and use the most appropriate algorithm to solve the problem.

In the case of automata-based approach to LTL model checking the product automaton that originates from synchronous product of the property and system automata. Hence, vertexes of product automaton graph are ordered pairs. An interesting observation is that every cycle in a product automaton graph emerges from cycles in system and property automaton graphs.

As the property automaton originates from the LTL formula to be verified, it is typically quite small and can be pre-analysed. In particular, it is possible to identify all strongly connected components of the property automaton graph. Cluster-based algorithms use a partition function which distributes vertexes of the graph among the participating workstations so that every workstation maintains only a part of the graph. Respecting strongly connected components of the property automaton, a partition function preserving cycle locality can be defined. The partitioning strategy is to assign all vertexes that project to the same strongly connected component of the property automaton graph to the same workstation. Since no cycle is split it is possible to employ localised Nested DFS algorithm to perform local accepting cycle detection simultaneously on all participating workstations.

Yet another interesting information can be drawn from the property automaton graph decomposition. Maximal strongly connected components can be classified into three categories:

**Type F: (Fully Accepting)** Any cycle within the component contains at least one accepting vertex. (There is no non-accepting cycle within the component.)
Type P: (Partially Accepting) There is at least one accepting cycle and one non-accepting cycle within the component.

Type N: (Non-Accepting) There is no accepting cycle within the component.

Realising that a vertex of a product automaton graph is accepting only if the corresponding vertex in the property automaton graph is accepting it is possible to characterise types of strongly connected components of product automaton graph according to types of components in the property automaton graph. Classification of components into types $N$, $F$, and $P$ is useful in other cluster-based algorithms presented in this paper.

It is evident, and we have already explicitly highlighted this with the algorithm presentation on several places, that performing empirical studies for comparing actual relative performance of algorithms so as to study their amenability for use in LTL model checking is of crucial importance and in some sense even more important than the traditional asymptotic view. This may lead to the discovery of problem instances for which the performances of solving algorithms are clearly different. Other important results of empirical investigations include assessing heuristics for hard problems, characterising the asymptotic behaviour of complex algorithms, discovering the speed-up achieved by parallel algorithms and studying the effects of the memory hierarchy and of communication on real machines, thus helping in predicting performance and finding bottlenecks in real systems. Experiments can thus help measure many practical indicators that may be extremely difficult to predict theoretically.

A careful tuning of the code, as well as the addition of ad-hoc heuristics and local hacks, may dramatically improve the performances of some algorithms, although the theoretical asymptotic behaviour may be not affected. Unfortunately, it may be sometimes difficult to draw general conclusions about algorithms from experiments. One of the common pitfalls is the irreproducibility of experimental results for distributed algorithms.

4 Conclusions

Distributed verification is a new emerging field. Extending the techniques as known from the sequential world adds significant complications and often requires entirely new approaches. In designing practical parallel solutions for distributed verification we need to change our attitude. The key steps for their effective deployment in industry and real applications is to forget about asymptotics, use algorithm engineering techniques and experimental algorithmics, consider often overlooked, yet practically important issues such as hidden constant factors, effects of the memory hierarchy, implications of communication complexity, numerical precision, and use of heuristics. The new demand for distributed verification algorithms that are of practical utility has raised the need to refine and reinforce the traditional theoretical approach.
References