Debugging Distributed Computations by Reverse Search

Artur Andrzejak[*]
University Erlangen-Nuremberg
Department of Computer Science
Martensstr. 3, 91058 Erlangen, Germany
artur.andrzejak@gmx.de

Komei Fukuda
School of Computer Science
McGill University
Montreal, Canada H3A 2A7
fukuda@cs.mcgill.ca

ABSTRACT
We develop a memory-efficient off-line algorithm for the enumeration of global states of a distributed computation. The algorithm allows the parameterization of its memory requirements against the running time. This is particularly useful for debugging of memory-intensive parallel computations, e.g. in image processing or data warehousing. We also show how to apply our technique to evaluate in a memory-efficient way the predicate *Definitely*(Φ) defined by Cooper, Marzullo and Neiger [8, 14]. The basis for these algorithms is Reverse Search [2], a paradigm successfully applied for enumeration of a variety of geometric objects.

KEY WORDS
Distributed Computations, Global States, Debugging, Reverse Search

1 Introduction

Detecting certain conditions of a distributed computation is fundamental to solving problems related to debugging and monitoring of parallel programs, especially of the critical ones. These problems include debugging, error reporting, process control, decentralized coordination or load balancing [16, 4].

The problem of detecting such conditions by means of evaluating *global predicates* of a distributed computation is not trivial and has received a considerable amount of attention [3, 8, 9, 14]. Global predicates are evaluated over *global states*, which are essentially unions of local states of all processors.

Several approaches for evaluating of global predicates are known. For stable global predicates, *i.e.*, predicates that do not become false once they are true, the global snapshot algorithm by Chandy and Lamport [6] is used. Another research direction exploits the structure of a global predicate to identify the global states for which it might be true [11]. The major drawback of these methods is that they either apply to small classes of predicates or are predicate-specific.

On the other hand, evaluating non-specific global predicates is NP-hard, as shown in [7]. For such predicates, or if the structure of the predicate is not known a priori, the most general approach is taken - enumeration of all global states. Cooper, Marzullo and Neiger propose in [8, 14] off-line algorithms for predicates *Possibly*(Φ) and *Definitely*(Φ) based on the enumeration of all global states. The predicate *Possibly*(Φ) holds if the system could have passed through a global state satisfying the predicate Φ. *Definitely*(Φ) holds if the system definitively passed through a global state with Φ being true.

The algorithms in [8, 14] are not memory-efficient: it is possible that an exponential number of global states must be hold simultaneously in the memory of the machine evaluating the global predicates. If the distributed computation is memory intensive, e.g. in the case of image processing or database applications, already few global states might exceed the memory capacity of the enumerating system. In [12], an attempt is made to design a memory-efficient enumeration algorithm. However, the authors do not take into account memory and time requirements for reverse execution of events implicit in their approach. The necessity of reverse execution of events occurring in the distributed computation is likely for memory-efficient global state enumeration algorithms, as discussed in Section 3.

Our contributions in this paper are the following:

- We propose a new algorithm for enumeration of global states of a distributed computation based on the framework of Reverse Search by Avis and Fukuda [2]. This off-line algorithm is memory-efficient, parallelizable [5] and thus well suitable for debugging of memory-intensive distributed computations.
- We investigate the space and time cost of reverse execution of events for global state enumeration and use the results to parameterize the enumeration time against the memory usage of our algorithm.
- We show how to evaluate in a memory-efficient way the predicate *Definitely*(Φ) using our technique (the evaluation of *Possibly*(Φ) in a memory-efficient way is trivial by our approach).

[*]From 01/01/2003 at Zuse Institute Berlin (ZIB), Takustr. 7, D-14195 Berlin, Germany. This work has been done during the Post-Doctoral research studies at Hewlett-Packard Laboratories, Palo Alto, USA.

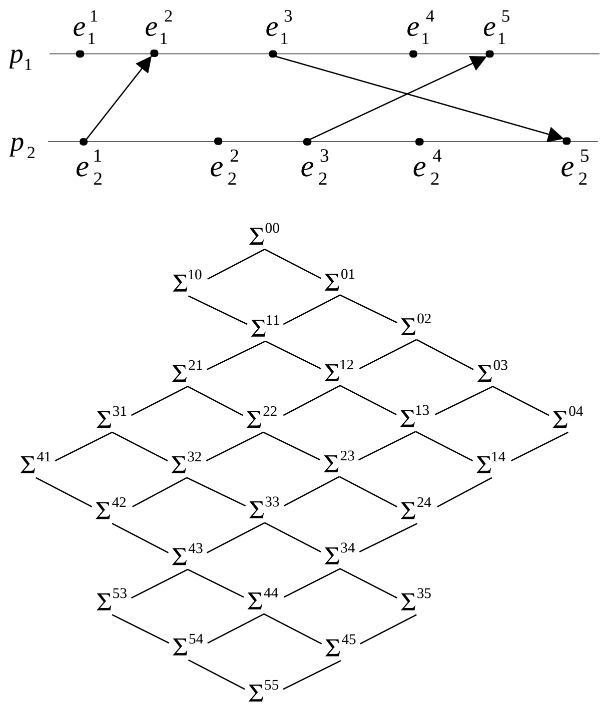
Figure 1. A distributed computation and its lattice of global states

1.1 Definitions

We assume that the reader is familiar with the definitions of a *distributed computation*, a *run* R, *local history* of a process p_i, *global history* of a distributed computation, and a *consistent cut* [3, 6].

For a process p_i, $1 \leq i \leq n$, and an integer $k \geq 0$ let σ_i^k denote the *local state* of a process immediately after having executed event e_i^k. The local state of a process may include information such as the values of local variables and the sequences of messages sent and received. The *global state* of a distributed computation is an n-tuple $\Sigma = (\sigma_1, \ldots, \sigma_n)$ of local states of all processes together with the messages still in the communication channels, see [15]. It is not hard to see that a run $R = e^1 e^2 \ldots$ results in a sequence of global states $\Sigma^0 \Sigma^1 \Sigma^2 \ldots$, where Σ^0 is the initial global state. Each global state Σ^i of the run R is obtained from the previous state Σ^{i-1} by some process executing the single event e^i. We say that Σ^{i-1} *leads to* Σ^i *in* R. The set of all consistent global states of a computation along with the leads-to relation defines a *lattice* L. Let $\Sigma^{k_1,\ldots,k_n}$ be a shorthand for the global state $(\sigma_1^{k_1}, \ldots, \sigma_n^{k_n})$ and let $\ell = k_1 + \ldots + k_n$ be its *level*. Figure 1 shows such a lattice together with the original distributed computation.

As a nonstandard definition, we say that for a global state $(\sigma_1, \ldots, \sigma_n)$, its *signature* is the tuple $(k_1, \ldots, k_n)$ of subscripts of the recently executed events $e_1^{k_1}, \ldots, e_n^{k_n}$ leading to the local states $\sigma_1, \ldots, \sigma_n$. The signature of the initial global state is $(0, \ldots, 0)$.

2 The Reverse Search Method

In this section we describe briefly elements of the Reverse Search method by Avis and Fukuda [2]. Assume that we are given a graph G whose vertices are the objects to be enumerated. In our case the objects are global states and G is the lattice L. We could possibly enumerate the vertices of G by applying the depth-search first algorithm to G. The critical problem of this approach is the requirement of storing all vertices of G at the same time in memory; for Reverse Search, we are required to store only one vertex of G in the computer memory at a time.

We need to define a rooted spanning tree of G. Let v^* be a distinguished vertex of G (which later will be the root of the tree); in our case it is the initial global state. The spanning tree of G is implicitly defined via a *local search function* $f : V(G)\backslash\{v^*\} \longrightarrow V(G)$ (where $V(G)$ denotes the vertex set of G). For every $v \in V(G)\backslash\{v^*\}$ the pair $(v, f(v))$ must be an edge in G. Furthermore, for every $v \in V(G)\backslash\{v^*\}$ there is a finite folding $f(f(\ldots f(v)\ldots))$ which yields v^*. Intuitively, every application of the function f brings us closer to the distinguished vertex v^*. It is not hard to see that for each $v \in V(G)\backslash\{v^*\}$ the function f defines a unique path from v to v^*. The union of these paths (regarded as sets of directed edges) define the (directed) spanning tree of G we are looking for. This graph is called the *trace graph* or simply the *trace* of the local search function f. Note that the trace graph is completely determined by f.

Now an execution of an algorithm based on Reverse Search resembles an execution of the depth-first search on the trace graph of f. For a particular problem instance, the graph G is specified be the following:

- An integer δ which bounds from above the maximum degree of a vertex of G.
- An *adjacency oracle* Adj explained below.

For an integer $i \in \{1, \ldots, \delta\}$ and a vertex $v \in V(G)$ the adjacency oracle Adj(v, i) gives the i-th neighbor of v in G or *null*. The function Adj is exhaustive and injective, that is if i goes from 1 to δ, then we obtain all neighbors of v in G and each of them exactly once; see [2] for the complete description. The following procedure ReverseSearch2 taken from [2] states in detail how G is traversed.

```
procedure ReverseSearch2(Adj,$\delta$,$v^*$,$f$);
  (* $j$: neighbor counter *)
  $v := v^*$; $j := 0$;
  visit vertex $v^*$;
  **repeat**
    **while** $j < \delta$ **do**
      $j := j + 1$;
      **if** $f(\text{Adj}(v, j)) = v$ **then**
        (* reverse traverse *)
```

```
            $v$ :=Adj$(v, j)$; $j := 0$;
            visit vertex $v$;
         endif
      endwhile;
      if $v \neq v^*$ then
         (* forward traverse *)
         $w := v$; $v := f(v)$; $j := 0$;
         repeat $j := j + 1$
         until Adj$(v, j) = w$ (* restore $j$ *)
      endif
   until $v = v^*$ and $j = \delta$.
```

3 Reverse Execution of an Event

Computing a global state Σ from its direct predecessor Σ' in L is straightforward: we let a (certain) process execute its next event. Thus, having all global states of level ℓ in the memory of the enumerating machine, we can obtain all global states of level $\ell + 1$. This property is used in the algorithms described in [8, 14]. However, the number of global states of a single level can be exponential in n, and this is also a reason why the above-cited algorithms are not memory-efficient.

For memory-efficient enumeration algorithms, we may assume that at least one global state Σ is kept in the memory: the currently visited one. To derive another global state Σ'' which is not a descendant of Σ in L we could reverse execute the last event on one of the processes p_i, $1 \leq i \leq n$, obtaining a global state Σ' which leads to Σ. By possibly repeating this step (for different i's), we can reach a predecessor of Σ'' in L, and then compute Σ'' in an obvious way. If a reverse execution is not possible, we can still use the initial state Σ^0 as the predecessor of Σ''; however this might be very time-consuming.

This indicates that memory-efficient global state enumeration algorithms are likely to use reverse execution of events. Indeed, our algorithm from Section 4 assumes such a mechanism: given a global state Σ, the computation of a (certain) global state Σ' which leads to Σ is part of the approach.

Reverse execution is a well established concept used in debugging, fault-tolerant computing, human-computer interaction and speculative computation [10, 13, 17]. Implementations include both hardware and software approaches. In this paper we assume a software-based implementation of reverse execution.

3.1 Reverse execution with time versus space trade-off

In this section we discuss a reverse execution approach and describe how to parameterize the trade-off between its execution time and the memory requirements.

Assume that P is a path in a lattice L corresponding to a distributed computation, such that each global state on P leads to the next one. Given a node (global state) Σ in P, we want to obtain its predecessor Σ'. As mentioned, the only difference between both global states is that in Σ' one of the processes p_j, say, has not yet executed its next event $e_j^{k_j}$. Thus, given Σ, the goal is to "set back in time" p_j to the state before its execution of $e_j^{k_j}$.

Obviously the method with the highest memory usage is to store all global states on the path P and then retrieve the required global state from the memory or other storage as necessary. Note that we can store the path in a "differential" way, *i.e.* for the above-mentioned global states Σ' and Σ we store for Σ only the local state of the processor p_j after the execution of the event $e_j^{k_j}$. The local states of all other processes are respectively identical in Σ' and Σ. Still, if Σ is the last global state in P and has the signature $(k_1, \ldots, k_n)$, then we must store k_i local states of the processor p_i, for each $i = 1, \ldots, n$, in total $\ell = k_1 + \ldots + k_n$.

To decrease memory usage, we can store only some local states of each process. Let $q > 0$ be an integer which determines that for each process we store every q-th local state only (beginning with the initial one). For $q = 1$, the retrieval of Σ' from Σ is the same as above. However, for $q > 1$, we retrieve the stored local state of p_j with index $\lfloor \frac{k_j - 1}{q} \rfloor$ from the memory and "load" this state into p_j. Subsequently, we recompute the local states of this process just until the event $e_j^{k_j}$ by executing its program code. Since the messages received by p_j are considered as a part of its local state [3], we assume that we store for each process p_i every message sent to this process (together with the index of the sender and the event number causing this message). Thus, we can retrieve them from the local state of p_j in Σ, if they should be necessary for executing the code of this process.

3.2 Space and time requirements

Assume that R is an upper bound on the time to retrieve a local state (from memory or storage) and to "load" it into a process p_i, $1 \leq i \leq n$. Furthermore, we write E for an upper bound on the time needed by a process to execute an event. For $q > 1$, in order to compute Σ' from Σ a process p_j must execute at most $q - 1$ events after retrieval of its last stored local state. Therefore, the time for computation of Σ' from Σ is at most

$$R + E(q - 1). \tag{1}$$

Let S be the maximum storage size of a local state of a single process. It is not hard to see that if the level of the last global state Σ in the path P is $\ell = k_1 + \ldots + k_n$, then the total storage needed for P is at most

$$S\frac{\ell}{q}. \tag{2}$$

4 Enumeration of Global States

Our goal is to enumerate all (consistent) global states of a distributed computation by traversing the lattice L of global states. In this section we describe how to achieve this by applying the procedure ReverseSearch2 from Section 2. We need to specify several problem-dependent elements used by the Reverse Search method. The main challenge of this task is to find efficient implementations of these elements, especially of the local search function f.

In the following, we consider L as a (directed) graph, where the nodes are global states and (Σ^i, Σ^j) is an edge, if Σ^i leads to Σ^j. Observe that this graph is connected.

4.1 Problem-dependent elements of the Reverse Search

For the Reverse Search method to be applied, we need to specify the following problem-dependent elements:

- A local search function f and its implementation,
- An adjacency oracle Adj and its implementation,
- A distinguished vertex v^* , the root of the trace graph,
- The maximum out-degree of a global state in the lattice L.

For technical reasons, we also provide the following element:

- An implementation of the test $f(\text{Adj}(v, j)) = v$ in the reverse traverse step which does not use f nor Adj.

All three the local search function, the adjacency oracle and the test $f(\text{Adj}(v, j)) = v$ are discussed below. As a distinguished vertex v^* we set the initial global state of the computation. Finally, the maximum degree in the lattice is the number of processes n, as a global state can lead to another global state due to an execution of an event on one of them.

4.2 The Adjacency Oracle Adj

For a given global state Σ and an integer j, $1 \leq j \leq n$, we define $\text{Adj}(\Sigma, j)$ as follows. If there exists a global state Σ' such that Σ leads to Σ' due to an execution of the next event on the process j, then $\text{Adj}(\Sigma, j)$ is Σ'; otherwise $\text{Adj}(\Sigma, j)$ is *null*. It is not hard to see that Adj indeed determines the lattice L of the distributed computation.

As for the implementation, we assume that the current global state Σ is "loaded" on the n processes $p_1, \ldots, p_n$. The value of $\text{Adj}(\Sigma, j)$ is computed as follows. First we test, whether the process p_j is waiting for a message and cannot execute the next event. If this is the case, then $\text{Adj}(\Sigma, j)$ returns *null*. Otherwise, we let p_j execute the next event. As a consequence the new global state becomes exactly $\text{Adj}(\Sigma, j)$.

Note that our definition of Adj implies that its values are only the "outgoing" global states in respect to the first argument of Adj (i.e. global states with level one higher than the argument). However, it is not hard to see that the correctness of the Reverse Search method is not influenced by this fact.

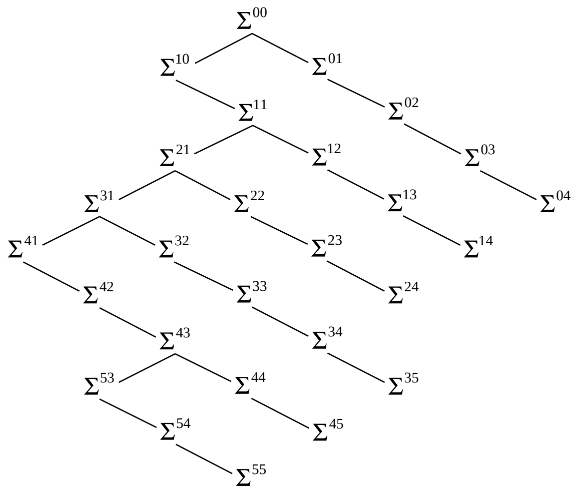

Figure 2. A trace induced by the local search f in the lattice shown in Figure 1

4.3 The local search function f

A possible and quite natural realization of a local search function is the following one. For a global state Σ of level ℓ we define $f(\Sigma)$ to be the global state of level $\ell - 1$ which leads to Σ and has lexicographically smallest signature. Figure 2 shows the trace of the local search for the lattice from Figure 1.

The question is how to implement f, since the computation of $f(\Sigma)$ from Σ is a "time-reversed" execution of an event of one of the processes. We propose two approaches. The first one is very memory-efficient and can be used for both the execution of f in the test $f(\text{Adj}(v, j)) = v$ and the execution of f in the forward traverse step. However, it has a large computational overhead. The second approach works only for the forward traverse case, but it allows us to trade the computational time at a cost of additional memory usage.

Both approaches assume the case that the current global state - the argument of f - is "loaded" on the n processes $p_1, \ldots p_n$.

4.3.1 Universal implementation of f

The following implementation has small memory usage - in the order of memory requirement of one global state. On the other hand, its computational overhead is large. It can be applied both for the reverse traverse and the forward traverse cases.

Let $(k_1, \ldots, k_n)$ be the signature of the current global state Σ. We want to find the smallest j, $1 \leq j \leq n$, such that the global state with signature $(k_1, \ldots, k_j - 1, \ldots)$ exists. Starting with $j = 1$, we attempt to compute the local state $\sigma_j^{k_j - 1}$ of the process p_j after the event $e_j^{k_j - 1}$ as follows. First we save the current local state of the process p_j. Then we initialize p_j with its initial local state and let the process p_j replay all those among the events $e_j^1 \ldots, e_j^{k_j - 1}$, which did not occur "after" the global state with signature $(k_1, \ldots, k_j - 1, \ldots)$. More precisely, if p_j requires a message m for replaying an event, m must be sent by a process p_i due to an event e_i^k with $k \leq k_i$ for $1 \leq i \leq n$, $i \neq j$. If p_j cannot reach the local state $\sigma_j^{k_j - 1}$, then there is no global state with signature $(k_1, \ldots, k_j - 1, \ldots)$ (or $(k_j - 1, \ldots)$ for $j = 1$). In this case we restore the local state of the process p_j and repeat the local state computation for $j+1$, $j+2, \ldots$, until success. Because we are testing the existence global states with lexicographically increasing signatures, the first found global state will be the correct value of $f(\Sigma)$. Note that we have as a result the global state $f(\Sigma)$, not only its signature.

4.3.2 Implementation of f with parameterized running time

For the current global state Σ let $P(\Sigma)$ be the (reversed) unique path induced by the local search function f on the lattice L of the distributed computation. We maintain an ordered list of signatures of the global states on $P(\Sigma)$ in the following way. Each time when the reverse traverse step of the algorithm takes place (i.e. the test $f(\text{Adj}(v, j)) = v$ is successful), we store the signature of the newly determined global state in the list. Since the signatures have very small memory usage compared to global states, the additional memory requirement for the list can be neglected.

Furthermore, we apply the technique from Section 3 and store a subset of the local states of each process on the path $P(\Sigma)$. The size of the subset is controlled by the parameter q defined there.

Now it is not hard to see that if Σ is the currently visited global state, then $f(\Sigma)$ is the before-last element of $P(\Sigma)$. Thus, given the information in our signature list and the stored local states, we can compute $f(\Sigma)$ as described in Section 3. The maximum computation time for f is then given by (1), and the maximum memory usage for $P(\Sigma)$ by (2).

Note that this approach cannot be used for the test $f(\text{Adj}(v, j)) = v$ in the reverse traverse step, since $\text{Adj}(\Sigma, j)$ is not in $P(\Sigma)$. Thus, the implementation of f presented here can be only used in conjunction with the implementation of the test $f(\text{Adj}(v, j)) = v$ described in the next section.

4.4 Implementation of the test $f(\textbf{Adj}(v, j)) = v$

We first describe the conditions under which the test $f(\text{Adj}(v, j)) = v$ is true, and turn then our attention to its implementation.

Assume that the current global state Σ has the signature $(k_1, \ldots, k_n)$. For an integer j, $1 \leq j \leq n$, the global state $\text{Adj}(\Sigma, j)$ (if exists) has the signature $(k_1, \ldots, k_j + 1, k_{j+1}, \ldots)$ (or $(k_1 + 1, k_2, \ldots)$ for $j = 1$). By the definition of f either $j = 1$ and so $f(\text{Adj}(\Sigma, j)) = \Sigma$, or for $j > 1$ we find $f(\text{Adj}(\Sigma, j))$ by successively testing the existence of the global states with the signatures $(k_1 - 1, k_2, \ldots, k_j + 1, \ldots)$, $(k_1, k_2 - 1, k_3, \ldots, k_j + 1, \ldots)$ till $(k_1, \ldots, k_j, \ldots)$, respectively. Thus, the test returns true if $\text{Adj}(\Sigma, j)$ exists and $j = 1$ or $f(\text{Adj}(\Sigma, j))$ has the signature $(k_1, \ldots, k_j, \ldots)$ (the last inspected).

In the actual implementation, we first check whether $\text{Adj}(\Sigma, j)$ exists as described in Section 4.2; if not, the test fails. Otherwise the test is immediately successful in case $j = 1$. For $j > 1$, we have to check successively the existence of the global states with signatures $(k_1 - 1, k_2, \ldots, k_j + 1, \ldots)$ until $(k_1, \ldots, k_{j-1} - 1, k_j, \ldots)$ by attempting to recompute these global states. If one of them exists, the test fails. Otherwise it is true since then $f(\text{Adj}(\Sigma, j))$ has the signature $(k_1, \ldots, k_j, \ldots)$. We recompute each of these global states by the technique described in Section 3. This approach can be applied since only one of the processes p_i, $1 \leq i < j$, needs to re-execute its computation. We use for this aim the data of the local states stored for the computation of f from Section 4.3.2.

In the worst case, we need to compute $n - 1$ global states, and each computation needs at most the time specified by (1). Together with the upper bound E on the time required to execute Adj, the whole test needs no longer than

$$E + (n - 1)(R + E(q - 1)). \tag{3}$$

4.5 Optimizing the forward traverse part of Reverse Search

By taking a closer look at the Reverse Search method we notice that for restoring the value of j (in the forward traverse part) the adjacency oracle Adj does not need to compute a new global state $\text{Adj}(\Sigma, j)$ since only the signature of $\text{Adj}(\Sigma, j)$ is needed. The computation of such a signature is trivial, since if a global state Σ has signature $(k_1, \ldots, k_n)$, then $\text{Adj}(\Sigma, j)$ has signature $(k_1, \ldots, k_j +$

$1, \ldots, k_n)$. We can then rewrite the forward traverse part of the procedure ReverseSearch2 from Section 2 as follows. Let $Adj_{sig}(\Sigma, j)$ be a function which computes only the signature of $\text{Adj}(\Sigma, j)$.

```
if $v \neq v^*$ then
   (* forward traverse *)
   $w :=$ signature of $v$;
   $v := f(v)$; $j := 0$;
   repeat $j := j + 1$
   until $\text{Adj}_{sig}(v, j) = w$ (* restore $j$ *)
endif
```

By this change, the time for restoring the value of j becomes constant.

4.6 Analysis of running time

In this section we bound from above the running time of the evaluation algorithm for the case of the solution with parameterized running time, *i.e.* if the implementations from Section 4.3.2 and Section 4.4 are used. If the implementation from Section 4.3.1 is used, no bound of the running time can be given, since then the execution time of f depends strongly on global state taken as f's argument.

We apply Theorem 2.4 from [2] by putting the symbols from [2] into the context of this paper:

- $t(\text{Adj})$, the execution time for Adj is at most E (Section 4.2),
- δ, the maximum (out-)degree in the lattice L is n,
- $t^R(\text{Adj}, f)$, the worst-case time for the test $f(\text{Adj}(v, j)) = v$ is given by (3),
- $t(f)$, the worst-case time for f is given by (1),
- $t^F(\text{Adj}, f)$, the time needed to restore j is constant (Section 4.5),
- $|L|$ is the total number of global states in the lattice.

Theorem [2, Theorem 2.4]. *The time complexity of* $ReverseSearch2$ *is*

$$O((t(Adj) + \delta t^R(Adj, f) + t(f) + t^F(Adj, f))|L|).$$

Using notation from Section 3 and assuming that E, R and q are non-constant, we have then:

Corollary 1. *Suppose that the implementations from Section 4.3.2 and from Section 4.4 is used. Then the running time of the enumeration of all global states is*

$$O(n^2(Eq + R)|L|),$$

and the memory requirement of the algorithm is at most

$$S(n + \frac{\ell_{max}}{q}),$$

where ℓ_{max} is the maximum level in the lattice.

5 Memory-Efficient Detection of *Definitely(Φ)*

As noted in the introduction, the algorithms given in [8] for detecting of *Possibly*(Φ) and *Definitely*(Φ) are not memory-efficient. In worst case, each algorithm holds in memory all global states of a single level of L. This number is exponential in n.

We can apply our memory-efficient algorithm for detecting of *Possibly*(Φ) in a straightforward way: just enumerate all global states until Φ applies or the enumeration terminates.

For the detection of *Definitely*(Φ) some more effort is needed. Assume that we remove from the lattice L all global states for which Φ applies, obtaining a graph L'. The idea is to run our enumeration algorithm on L' instead of L. Note that *Definitely*(Φ) does *not* hold exactly if there is a path from the initial state to a terminal global state in L'. Thus, our enumeration will reach a terminal state exactly in this case. Since we traverse L' in a depth-first search manner, the case that *Definitely*(Φ) is not true is detected relatively fast. On the other hand, we might have to evaluate almost all global states to reach the conclusion that *Definitely*(Φ) is true.

We need only slight changes of f and Adj to enumerate the global states of L' instead of L. At each computation of Adj we evaluate Φ for the new created global state; if Φ applies, we simply return *null*. The local search f is modified in an analogous way.

6 Conclusions

The only known way of detecting non-specific global predicates is the enumeration of global states. Although such an enumeration might generate a huge number of global states to be tested, a real obstacle of this approach is the anticipated memory usage of the enumerating system.

We addressed this problem by designing a memory-efficient enumeration algorithm. The algorithm can be also used to evaluate the predicates *Possibly*(Φ) and *Definitely*(Φ) introduced in [8, 14]. We argued that every memory-efficient algorithm is likely to use reverse execution and discussed the time/space trade-off of this technique in our context. As a consequence, we could parameterize the memory usage of our algorithm versus its running time. Worst-case bounds on both quantities have been derived.

The disadvantage of our approach is the need for instrumentation of user executable and a likely large enumeration time. However, our technique allows parallelization of the enumeration to a stronger degree than the original computation, which provides a partial remedy to the last problem.

7 Acknowledgments

The first author would like to thank Professor Friedemann Mattern, ETH Zurich, for valuable suggestions.

References

[1] A. Andrzejak and K. Fukuda, Optimization over k-set polytopes and efficient k-set enumeration, *Proc. 6th International Workshop on Algorithms and Data Structures (WADS'99)*, LNCS 1663, Vancouver, Canada, 1999, 1–12.

[2] D. Avis and K. Fukuda, Reverse search for enumeration. *Disc. Applied Math.*, 65, 1994, 21–46.

[3] Ö. Babaoğlu and K. Marzullo, Consistent global states of distributed systems: Fundamental concepts and mechanisms, in S. J. Mullender (Ed.), *Distributed Systems*, (New York: Addison Wesley, 1994) 55–96.

[4] Ö. Babaoğlu and M. Raynal, Specification and Verification of Behavioral Patterns in Distributed Computations, *Proc. of Fourth IFIP Working Conference on Dependable Computing for Critical Applications*, San Diego, USA, 1994.

[5] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt, The parallel search bench zram and its applications, *Annals of Operations Research*, 90, 1999, 45–65.

[6] K. M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, 3(1), 1985, 63–75.

[7] C. Chase and V. K. Garg, Detection of Global Predicates: Techniques and their Limitations, *Distributed Computing*, 11(4), 1998, 191-201.

[8] R. Cooper and K. Marzullo, Consistent detection of global predicates, *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, USA, 1991, 163–173.

[9] C. Diehl, C. Jard, and J. X. Rampon, Reachability analysis on distributed executions, in J.-P. Jouannaud, M.-C. Gaudel (Ed.) *Proc. TAP-SOFT*, LNCS 668 (New York: Springer-Verlag, 1993) 629–643.

[10] M. Frumkin, R. Hood, L. Lopez, Trace-Driven Debugging of Message Passing Programs, *Proc. IPPS/SPDP'98*, Orlando, USA, 1998, 753-762.

[11] V. K. Garg and B. Waldecker, Detection of strong unstable predicates in distributed programs, *IEEE Trans. on Parallel and Distributed Systems*, 7(12), 1996, 1323–1333.

[12] R. Jégou, R. Medina, and L. Nourine, Linear space algorithm for on-line detection of global predicates, *Proc. STRICT '95*, Berlin, Germany, 1995, 175–189.

[13] T. J. LeBlanc and J. M. Mellor-Crummey, Debugging parallel programs with Instant Replay, *IEEE Transactions on Computers*, C-36(4), 1987, 471–482.

[14] K. Marzullo and G. Neiger, Detection of global state predicates, *Proc. 5th International Workshop on Distributed Algorithms (WDAG-91)*, Delphi, Greece, 1991, 254–272.

[15] F. Mattern, Virtual Time and Global States of Distributed Systems, *Proc. International Workshop on Parallel and Distributed Algorithms*, Chateu de Bonas, France, 1988, 215–226.

[16] N. Mittal and V. K. Garg, Debugging distributed programs using controlled re-execution, *Proc. Symposium on Principles of Distributed Computing*, 2000, 239–248.

[17] R. Sosič, History Cache: Hardware Support for Reverse Execution, *Computer Architecture News*, 22(5), 1994, 11–12.