Memory-Efficient and Fast Enumeration of Global States

Artur Andrzejak

Zuse-Institute-Berlin (ZIB), Takustr. 7, 14195 Berlin, Germany, Email: andrzejak@zib.de

***Abstract*—We describe a simple algorithm for level-wise enumeration of global states of a distributed computation. In addition to fast execution, it requires working memory only on the order of two global states plus a variable amount which allows trading higher speed for storage. Furthermore, we present a new caching strategy which speeds up the enumeration algorithm described in [1].**

I. Introduction

Detecting certain conditions in a distributed computation is a fundamental building block for many solutions related to control and debugging of distributed programs. Examples include error reporting, process control, decentralized coordination or load balancing [11], [3]. These conditions corresponding to *global predicates* are detected and evaluated over *global states*, which are essentially unions of local states of all processes.

Except for stable global predicates [4], *i.e.* predicates that do not become false once they are true, evaluation of global predicates is not an easy task. Most approaches exploit the structure of a global predicate to identify the global states for which a predicate might apply [7]. 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 [5]. 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 [6], [9]. Such enumeration might be prohibitively expensive due to the large number of global states. Moreover, the memory-efficiency of known approaches is very poor, including the pre-algorithms in [6], [9].

In an attempt to provide a practically feasible method for the enumeration of global states we present here a simple algorithm with fast execution and low memory requirement. In more detail, our algorithm requires working memory only on the order of two global states plus a variable amount determined by the user. This additional amount allows trading memory requirement for higher speed: essentially, doubling the additional memory (up to a certain limit) doubles the execution speed.

The new algorithm explores the lattice of global states level-wise, bearing similarities to Breadth-First Search (BFS). It complements the memory-efficient algorithm from [1], which traverses this lattice in Depth-First Search (DFS) manner; for a particular application, either BFS or DFS traversal might be a better choice.

Using the insights gained from the new approach, we also improve the strategy for caching of local states described in [1]. This improvement provides a further speed up of the DFS-based enumeration without increasing its memory requirement.

A. 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* [2], [4].

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

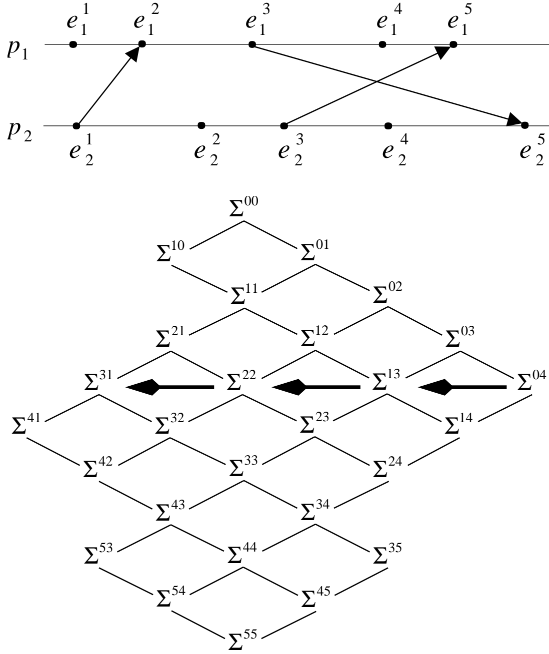

Figure 1. A distributed computation and its lattice

n-tuple $\Sigma = (\sigma_1, \ldots, \sigma_n)$ of local states of all processes together with the messages still in the communication channels, see [10]. 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 *succeeds* Σ^{i-1} or that Σ^i *is a successor of* Σ^{i-1}. Conversely, let us say that Σ^{i-1} *precedes* Σ^i or that Σ^{i-1} *is a predecessor of* Σ^i. The set of all consistent global states of a computation along with the successor 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.

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 $(0, \ldots, 0)$.

II. Reverse Execution of an Event

Computing a global state Σ from its 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 [6], [9]. 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.

On the other hand, memory-efficient enumeration algorithms strive to keep in the memory the least number of global states at a time, in extreme case only the currently visited one. This frequently leads to a situation where we must compute a predecessor of a global state, and so memory-efficient global state enumeration algorithms are likely to use reverse execution of events. Indeed, the algorithm in [1] as well as the one in this paper assumes such a mechanism.

Reverse execution is a well established concept used in debugging, fault-tolerant computing, human-computer interaction and speculative computation [8], [12]. Implementations include both hardware and software approaches.

A. Reverse execution with time versus space trade-off

In the following we discuss an approach for 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 corresponding to a distributed computation. Given a node (global state) Σ in P, we want to obtain its predecessor Σ'. The only difference between both global states is that in Σ' exactly one of the processes p_i, say, has not yet executed its next event $e_i^{k_i}$. In other words, the difference between the signatures of Σ' and of Σ is that the former is less by 1 in position i; in all other positions

the signatures are identical. Thus, given Σ, the goal is to "set back in time" p_i by a single step to the local state $\sigma_i^{k_i-1}$.

Obviously the method with the highest memory usage is to cache 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 cache the path in a "differential" way, *i.e.* we store only changes in a single local state for every node of P. 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 process p_i for each $i = 1, \ldots, n$, in total $\ell = k_1 + \ldots + k_n$.

To decrease memory usage, we might also cache only *some* local states of each process. For $i = 1, \ldots, n$, let us cache every q_i-th local state of the process p_i, where q_i is a natural number of our choice. For $q_i = 1$, the retrieval of Σ' from Σ is the same as above. For $q_i > 1$ we first retrieve the cached local state of p_i with index $\lfloor (k_i - 1)/q_i \rfloor q_i$ from memory or secondary storage and "load" this state into p_i. Subsequently, if $\lfloor (k_i - 1)/q_i \rfloor q_i < k_i - 1$, we compute the local state of p_i in Σ' by "forward" executing its program code; note that we have to execute at most $q_i - 1$ events. Since the messages received by p_i are considered a part of its local state [2], we cache also those (together with the index of the sender and the event number causing this message).

B. Space and time requirements

We bound now the time and space requirements of the above approach for a single process. For i between 1 and n let us write:

- R_i for an upper bound on the time to retrieve a local state and to "load" it into process p_i,
- E_i for an upper bound on the time needed by p_i to execute an event, and
- S_i for an upper bound on memory requirement of a local state of p_i.

For $q_i > 1$, in order to compute Σ' from Σ process p_i must forward execute at most $q_i - 1$ events after retrieval of its last stored local state. Therefore, the time for computation of Σ' from Σ is at most

$$R_i + E_i(q_i - 1). \tag{1}$$

If the last global state in P has signature $(k_1, \ldots, k_n)$, then the total memory or secondary storage needed for caching of local states of process p_i is at most

$$S_i \left\lfloor \frac{k_i}{q_i} \right\rfloor . \tag{2}$$

III. Level-Wise Enumeration of Global States

In this section we describe an algorithm for level-wise enumeration of global states. In addition to memory efficiency, it achieves high speed by reducing the amount of reverse computations. The key idea is to execute only forward computations on the processes $p_1, \ldots, p_{n-1}$ and reverse computations on p_n only. As a consequence, only one local state needs to be cached for each of $p_1, \ldots, p_{n-1}$, while we apply caching techniques from Section II to speed up reverse computations of p_n. Note that choosing the "fastest" process as p_n further improves the execution speed.

A. Implementation details

Figure 1 illustrates for $n = 2$ the order in which the global states are visited (bold arrows). Assume that we have cached the local states σ_1^0, σ_2^4 and want to enumerate global states in level 4. To this aim we repeatedly reverse execute p_2 and forward execute p_1 and obtain $\Sigma^{04}, \Sigma^{13}, \Sigma^{22}, \Sigma^{31}$ in this order. (Note that this approach could be easily extended to simultaneously visit most of the global states in level 5 which occur "between" the reverse and the forward executions).

During this process we calculate the minimum index of a local state of p_1 in level 5 (which turns out to be 1) and the maximum such index for p_2 (which is 4). Thus, to prepare the algorithm for traversing of the next level, we compute by forward execution (or fetch from cache) σ_1^1 and σ_2^4.

Since the enumeration is slightly more complicated for $n > 2$, we give an example for $n = 3$

004	103	202	301
013	112	211	310
022	121	220	
031	130		
040			

TABLE I

SIGNATURES OF THE GLOBAL STATES VISITED IN THE EXAMPLE FOR $n = 3$

in Table I. The algorithm traverses here level 4 of a fictive lattice; in the beginning, local states σ_1^0, σ_2^0 and σ_3^4 are stored in a cache. The global states are visited in lexicographical order (in Table I: top to bottom, then left to right). Note that each column of the table corresponds to a sequence initiated by fetching σ_2^0 from cache and "loading" it into p_2.

The full implementation details are presented in the following algorithm.

```
Σ := initial global state (GS); visit Σ ;
compute F, set of signatures of successors of Σ;
M := cache with local states (LS) σ_1^{k_1}, ..., σ_{n-1}^{k_{n-1}},
     where k_i is minimal entry at position i over all
     signatures in F, i = 1 ..., n − 1;
repeat {next level}
  a := smallest signature in F;
  z := largest signature in F;
  i = (i_1, ..., i_n) := a;
  update C, cache of LSs of p_n between σ_n^{z_n} and σ_n^{a_n};
  Σ := GS with signature a (use M and C here) ;
  clear F', set of signatures of next level;
  add signatures of successors of Σ to F';
  repeat {compute and visit next GS}
    i' := lexicographical successor of i in F;
    for k := each position at which i ≠ i' do
      if k < n then {forward execution}
        using LSs in M and Σ, forward execute p_k
        until its local state has index i'_k;
      else {reverse execution}
        using C, compute LS of p_n with index i'_n ;
      end if
      update Σ with changes; visit Σ;
      add signatures of successors of Σ to F';
    end for
    i := i';
  until i' = z;
  update M by forward executions and copying LSs
  from Σ;
  F := F';
until F empty.
```

B. Running time and memory requirements

Obviously, the least cost of generating a global state is one reverse computation and one forward computation. However, sometimes a local state of one of the processes $p_1, \ldots, p_{n-1}$ must be restored from the cache. It is not hard to see that the cost of computing the first visited global state per level and the cost of updating M can be neglected. Thus, with notation from Section II, $R := \max_{i=1\ldots n-1} R_i$, $E := \max_{i=1\ldots n-1} E_i$ we obtain as an upper bound on the running time of the enumeration

$$|L| \left(R + E + R_n + E_n(q_n - 1)\right),$$

where $|L|$ is the total number of global states. Note that the time per single global state compares favorably to the cost of moving from one global state to the next in an actual distributed computation, which is on the order of E.

As for the memory requirements, the above algorithm stores the currently visited global state, the cache M, and the cache for reverse computations of p_n. While M requires no more memory than for a single global state, the cache for p_n needs memory amounting to $S_n \lfloor \frac{k_n}{q_n} \rfloor$, where k_n is the highest index of a local state in p_n. The memory requirements of F and F' are small compared to a single global state and are neglected. Summarizing, the size of the working memory equals to memory for two global states plus a variable amount determined by the user.

IV. Improved Caching of Local States

The algorithm presented in [1] enumerates global states of a distributed computation by traversing the corresponding lattice in a DFS-manner. This algorithm is memory-efficient, albeit it applies reverse computation which is responsible for the major share of its computational effort. This effort can be reduced by caching of certain local states, thus trading the running time for memory efficiency. In this section we describe an improved strategy for selecting the local states to be cached. This can significantly decrease the running time as compared

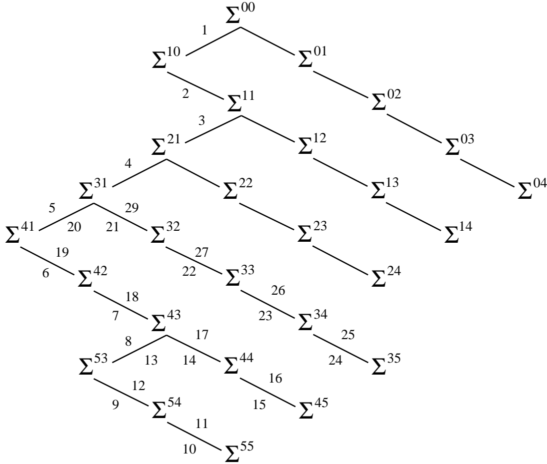

Figure 2. The spanning tree of the lattice from Figure 1 used for the DFS

to the version in [1] at the same memory requirement.

A. Depth-First Search and i-traces

Without going into implementation details, we describe in the following how the lattice of global states is traversed by the algorithm in [1]. For such a lattice let T be a spanning tree such that each edge connects a global state and its successor with lexicographically largest signature. Figure 2 shows such a spanning tree for the lattice in Figure 1. The order of enumeration of the global states is the same as the order of node visits of a Depth-First Search algorithm applied to T and starting at the initial global state. Figure 2 indicates in which order the nodes are visited by the numbers at some edges.

From the same figure we see that sometimes the traversing of the tree moves from a global state to its predecessor, e.g. from Σ^{55} to Σ^{54}. At these steps the algorithm presented in [1] incurs reverse computation. During such an operation the local state of only one of the processes p_i, say, must be reverse computed. It is not hard to see that the higher the process index, the more frequently this process is affected by reverse computation. In the following we want to explore this fact for a better caching strategy.

Consider the sequence of global states in the order as visited by the algorithm. For $i = 1, \ldots, n$ let us define an *i-trace* as a longest subsequence of it such that the process i was subject to reverse computation only. Every such a subsequence can be determined by a path in T. In Figure 2, the only one 1-trace is determined by the path from Σ^{55} to Σ^{00} at the left "edge" of the lattice, and e.g. Σ^{21}, Σ^{22}, Σ^{23}, Σ^{24} determines one of the multiple 2-traces.

To bound the number of i-traces in T consider a *complete lattice* which has m levels, and all global states except those with the highest level have exactly n successors (recall that n is the number of processes). It is not hard to see that the number of i-traces in such a lattice is maximal (over all lattices) for $i = 1, \ldots, n$. Especially, it has only one 1-trace. Furthermore, every global state Σ with signature $(k_1, \ldots, k_{i-1}, 0, \ldots, 0)$ where $k_1 + \ldots + k_{i-1} = \ell < m$ gives rise to a unique i-trace. This is due to the fact that at some point (exactly when it encountered Σ after a reverse execution of p_{i-1}) the algorithm will visit all global states with signatures starting with $k_1, \ldots, k_{i-1}$. By doing so, it first only forward executes p_i ($m - \ell$ many times) and then reverse executes it until the global state with signature $(k_1, \ldots, k_{i-1}, 0, \ldots, 0)$ is encountered again (the second part corresponds to our i-trace). Of course, during this process the processes $p_{i+1}, \ldots, p_n$ are forward and reverse executed. It follows by induction that in the complete lattice we have for i between 2 and n exactly

$$tr_i := (m-1)(m-2)\ldots(m-i+1)$$

i-traces (and $tr_1 := 1$), which is also an upper bound on the number of i-traces in any lattice of depth m.

B. Minimizing the running time

We assume that the total time spent for reverse execution of process p_i is proportional to the number of i-traces. This assumption is not completely accurate, since not every local state

of p_i will occur in equally many i-traces; e.g. in the complete lattice the local state of p_2 with index j will occur in exactly $m - j$ of the $m - 1$ 2-traces. However, since these frequencies depend on the form of the lattice and so on the distributed computation we do not consider them.

Using the notation and results from Section II, the total time for the reverse computations is obviously proportional to

$$t_{rev} = \sum_{i=1}^{n} tr_i \left(R_i + E_i(q_i - 1)\right). \quad (3)$$

We can minimize this number by choosing the values for $q_1, \ldots, q_n$ subject to the following constraint that the total memory or secondary storage devoted for caching does not exceed S_{total} :

$$S_{total} \geq \sum_{i=1}^{n} S_i \lfloor \frac{k_i}{q_i} \rfloor, \quad (4)$$

where $k_i \leq m$ is the highest index of the local state of process p_i.

Since the above optimization problem requires non-linear integer programming, finding an optimal solution is likely to be NP-complete. Thus, heuristics such as genetic algorithms seem to be a good choice. Another approach worth noting is to try to balance out the summands in the r.h.s. of (3). This can be achieved by setting e.g. $q_i = \lceil (tr_{n-i} + 1)/E_i \rceil$ (under the assumption that $R_i = 0$) for $i = 1, \ldots, n$.

V. Conclusion

The most universal approach to evaluating global predicates is the enumeration of global states. Although such an enumeration might generate a huge number of global states to be checked, a real obstacle of this approach is the anticipated memory usage.

We addressed this problem by designing a memory-efficient enumeration algorithm which traverses the lattice of global states level-wise. As a minimum, it has a memory requirement of two global states. Additional memory devoted to it by the user is used to speed up the reverse computations utilized in the algorithm. If all local states of a single process can be cached, the algorithm computes a single global state at the speed comparable to the original computation.

Using the insights gained from the development of the new approach we showed how the strategy for selecting local states to be cached applied in [1] can be improved. This implies a higher execution speed of the algorithm from [1] without increased memory requirement.

References

[1] A. Andrzejak and K. Fukuda. Debugging Distributed Computations by Reverse Search. *Applied Informatics 2003*, Innsbruck, February 2003.

[2] Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. J. Mullender, editor, *Distributed Systems*, pages 55-96, Addison Wesley, NY, 1994.

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

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

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

[6] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163-173, Santa Cruz, May 1991.

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

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

[9] K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG-91)*, pages 254-272, Delphi, October 1991.

[10] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215-226, Chateu de Bonas, October 1988.

[11] N. Mittal and V. K. Garg. Debugging distributed programs using controlled re-execution. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 239-248, 2000.

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