Int J Softw Tools Technol Transfer (2010) 12:113–122
DOI 10.1007/s10009-010-0144-z

Distributed dynamic partial order reduction

Yu Yang · Xiaofang Chen · Ganesh Gopalakrishnan · Robert M. Kirby

Published online: 6 April 2010

Abstract While stateless model checking avoids the memory blow-up problem by not recording the search history, runtime becomes a major limiting factor. In this paper, we present distributed dynamic partial order reduction (DDPOR) which can speed up stateless model checking using computer clusters, and get the benefit of dynamic partial order reduction (DPOR). The experiments show that DDPOR can give out nearly linear (with respect to the number of CPUs) speedup on realistic multithreaded programs, comparing with sequential stateless model checking that uses DPOR.

Keywords Dynamic verification · Stateless model checking · Distributed

1 Introduction

Stateless runtime model checking, which is pioneered by Verisoft [1], is a promising verification methodology for real-world threaded software. It avoids the (implicit or explicit) overhead of modeling programs that is usually required by other model checkers [2–5]. The precision of information available at runtime allows techniques, such as dynamic partial order reduction (DPOR) [6], to dramatically cut down the number of interleavings examined.

Unfortunately, even with dynamic partial order reduction enabled, runtime is still a major limiting factor of stateless model checkers [7]. With the wide availability of computer clusters, it is desirable to use them to speed up the stateless model checking.

We observed that since stateless search does not maintain the search history, different branches of an acyclic state space can be explored concurrently with very loose synchronizations. This shows that stateless dynamic model checkers are potentially "embarrassingly parallel" for distributed verification. We implemented a parallel stateless model checker based on this observation, employing a centralized load balancer to distribute work among multiple nodes. Initially, we failed to consistently obtain the linear speedup promised by the apparent parallelism. Deeper investigation revealed the reasons. These reasons, and other features of our algorithm are now summarized:

- *Avoiding redundant computations:* despite our use of *sleep sets* [8] to avoid redundant interleavings among independent transitions, we found that redundant (and, in fact, identical) interleavings were being explored among multiple nodes. The problem was traced to the incremental way of computing backtrack sets in the DPOR algorithm (detailed in the rest of this paper), which is well suited for a sequential implementation but not a loosely synchronized distributed implementation. We have developed a heuristic technique to update backtrack sets more aggressively, as detailed in Sect. 3.4.
- *Work distribution heuristics*: numerous heuristics help achieve efficient work distribution in the parallel state-

This work was supported in part by NSF award CNS00509379, Microsoft, and SRC Contract 2005-TJ-1318.

Y. Yang (✉) · X. Chen · G. Gopalakrishnan · R. M. Kirby
School of Computing, University of Utah, Salt Lake City, UT 84112, USA
e-mail: yuyang@cs.utah.edu

X. Chen
e-mail: xiachen@cs.utah.edu

G. Gopalakrishnan
e-mail: ganesh@cs.utah.edu

R. M. Kirby
e-mail: kirby@cs.utah.edu

less model checker. These include: (i) the straightforward method of employing a single load balancing node (process) and $N-1$ worker nodes (processes); (ii) the concept of a soft limit on the number of backtrack points recorded within a worker node before that node decides to offload work to another worker; and (iii) minimizing communication by offloading work that lies deepest within the stack—points from where the largest number of program-paths are available—so that bigger chunks of work are shipped per communication.

In the rest of this paper, we present the distributed dynamic partial order reduction (DDPOR) algorithm in detail. We implemented DDPOR on top of our stateless runtime model checker `Inspect` [9]. Our experiments show almost linear speedup with increasing number of nodes (CPUs). For example, one of our benchmarks which has eight threads and requires more than 11 h to finish checking using sequential `Inspect` can be checked by the parallel `Inspect` within 11 min using 65 nodes. The parallel `Inspect` gives a speedup of 63.2 out of 65.

Roadmap: Sect. 2 presents background information on DPOR. Section 3 presents the DDPOR algorithm. Section 4 presents implementation detail, and the experiment results, Sect. 5 the related work, and Sect. 6 our concluding remarks.

2 Preliminaries

2.1 Background definitions

Here, we define the notations that we employ in the rest of the paper. We consider a terminating multithreaded program with a fixed number of sequential threads as a state transition system. We use $Tid = \{1, \ldots, n\}$ to denote the set of thread identities. Threads communicate with each other via *global* objects which are visible to all threads. The operations on global objects are called *visible* operations, while thread local variable updates and PC updates are *invisible* operations. A *state* of the multithreaded program consists of the state of global objects $Global$, and the local state $Local$ of each thread.

$$S \subseteq \text{Global} \times \text{Locals}$$
$$\text{Locals} = Tid \to \text{Local}$$

A transition $t : S \to S$ advances the program from one state to a subsequent state. More specifically, it starts with one visible operation, followed by a finite sequence of zero or more invisible operations of the same thread, and ends just before the next visible operation of the same thread.

Let $\mathcal{T}$ denote the set of all transitions of a multithreaded program. A transition $t \in \mathcal{T}$ is enabled in a state s if $t(s)$ is defined. If t is enabled in s and $t(s) = s'$, we use $s \xrightarrow{t} s'$ to mean that s' is the successor of s by executing transition t. We use $tid(t)$ to denote the identity of the thread that executes t. Obviously we have $tid(t) \in Tid$.

The behavior of a multithreaded program is given by a transition system $M = (S, s_0, \Gamma)$, where s_0 is the initial state, and $\Gamma \subseteq S \times S$ is the transition relation: $(s, s') \in \Gamma$ iff $t \in \mathcal{T} : s \xrightarrow{t} s'$.

2.2 Dynamic partial order reduction

Systematic state space exploration can be performed in a stateless fashion by dynamically executing the program in a depth-first search order. Instead of enumerating reachable states of the model as in classic model checkers, dynamic model checking focuses on exhaustively exploring the feasible executions. This is made possible by controlling and observing the execution of visible operations of all threads. In particular, the entire program is executed under the control of a special *scheduler*, which gives permission to, and observes the results of all visible operations. Since the scheduler has full control of every context switch, systematic exploration becomes possible. In this context, backtracking or the rollback of a partially executed sequence is implemented by re-starting the program afresh under a different thread schedule.

Given the set of enabled transitions from a state, partial order reduction algorithms attempt to explore only a (proper) subset of transitions that are enabled, and at the same time guarantee that the properties of interest will be preserved. Such a subset is called *persistent set*. Static partial order reduction algorithms compute the persistent set of a state immediately after reaching it. As for multithreaded programs, persistent sets computed statically can be excessively large because of the limitations of static analysis. For instance, if two transitions leading out of s access an array `a[]` by indexing it at locations captured by expressions `e1` and `e2` (i.e., `a[e1]` and `a[e2]`), a static analyzer may not be able to decide whether `e1=e2` (and hence whether the transitions are dependent or not).

In DPOR, given a state s, the persistent set of s is not computed immediately after reaching s; instead, DPOR populates the persistent sets of s while searching under s according to depth-first search (DFS). Figure 1 shows the DPOR algorithm. In DPOR, the scheduler maintains a *search stack* S of global states. Each state s in the stack is associated with a set $s.enabled$ of enabled transitions, a set $s.done$ of thread identities, and a *backtracking set* $s.backtrack$. In more detail, $s.done$ denotes the set of threads that have been examined at s. $s.backtrack$ refers to the sets of threads that need to be explored from s. The procedure UPDATEBACK-

```
1: Initially: S.push(s_0); DPOR(S)

2: DPOR(S) {
3:    let s = S.top;
4:    for each t ∈ s.enabled, UPDATEBACKTRACKSET(S, t);
5:    if (∃τ ∈ Tid : ∃t ∈ s.enabled : tid(t) = τ) {
6:       s.backtrack ← {τ};
7:       s.done ← ∅;
8:       while (∃q ∈ s.backtrack \ s.done)
9:          s.done ← s.done ∪ {q};
10:         s.backtrack ← s.backtrack \ {q};
11:         let t ∈ s.enabled such that tid(t) = q, and let s' be a state such that s -t-> s';
12:         S.push(s');
13:         DPOR(S);
14:         S.pop();
15:      }
16:   }
17: }

18: UPDATEBACKTRACKSETS(S, t){
19:   let T be the sequence of transitions associated with S;
20:   let t_d be the latest transition in T that is dependent and may be co-enabled with t;
21:   if (t_d = null) return;
22:   let s_d be the state in S from which t_d is executed;
23:   let E be {q ∈ s_d.enabled | tid(q) = tid(t), or q in T, q was after t_d and there is a happens-before relation for (q, t). }
24:   if (E ≠ ∅)
25:      choose any q in E, add tid(q) to s_d.backtrack;
26:   else
27:      s_d.backtrack ← s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
28: }
```

Fig. 1 Dynamic partial-order reduction

TRACKSETS(S, t) is used to *dynamically* compute the persistent sets [8] of states.

3 Algorithm

In DPOR, the thread identities recorded in the backtrack set of a state s (i.e., $s.backtrack$) help generate different (non-equivalent) executions out of s. As DPOR is implemented through stateless search, it is completely safe to explore the different transitions in the backtrack sets of states concurrently, and with no (or very little) synchronization. With the wide availability of cluster machines, the potential for distributed verification is very high.

To have multiple nodes explore multiple backtrack points concurrently, each cluster node must know: (i) the portion of the search stack from the initial state to the backtrack point; (ii) the transition sequence from the initial state to reach the backtrack point; (iii) the transition to be executed from a backtrack point. All this information is easily obtained from the search stack. To distribute work among multiple nodes, we can use a centralized load balancer to balance the workload employing very limited synchronizations.

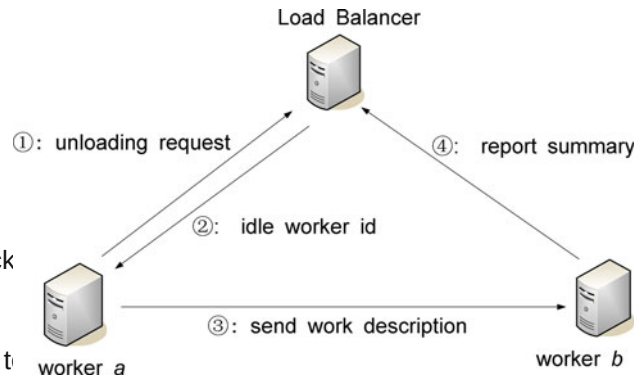


Fig. 2 The message flow among the load balancer and the workers

The computer nodes in a cluster can be classified into three categories: (i) the load balancer; (ii) *busy worker* nodes that have been assigned tasks; (iii) *idle worker* nodes that are waiting for tasks. Figure 2 illustrates how the workers and the load balancer collaborate. Let a be a busy worker node and b an idle one, with a trying to unload some work to b. First a sends a request to the load balancer. If there are idle nodes, the load balancer will return the identity of an idle node to the worker. In our example, the load bal-

```
1: Initially: W_b = ∅,
2:            W_i = { all worker nodes in the cluster };

3: LoadBalancer(){
4:   let w_0 be a worker node such that w ∈ W_i;
5:   let S be an empty program state stack;
6:   S.push(s_0);
7:   send S to w_0;
8:   W_i ← W_i \ {w_0};
9:   W_b ← {w_0};
10:  while(W_b ≠ ∅){
11:    receive event e from any worker w;
12:    if(e is work finish notification){
13:      W_b ← W_b \ {w};
14:      W_i ← W_i ∪ {w};
15:    }
16:    else if(e is new work request){
17:      if(W_i = ∅){
18:        reply "no idle workers" to w;
19:        continue;
20:      }
21:      let w' be a worker node such that in w' ∈ W_i;
22:      W_i ← W_i \ {w'};
23:      W_b ← W_b ∪ {w'};
24:      tell w that w' is idle;
25:    }
26:  }
27:  for each w ∈ W_i, send a termination message to w;
28: }
```

Fig. 3 The load balancing algorithm

```
1: Worker(){
2:   while(true){
3:     let m be a received message;
4:     if(m is a command to terminate) return;
5:     receive the search stack S;
6:     DDPOR(S);
7:     send the report to the load balancer;
8:   }
9: }
```

Fig. 4 The routine that runs on each worker

ancer tells a that b is idle, whereupon node a will send an unload message to b with all the information needed for b to start searching from an unexplored backtrack point. When b finishes the assigned work, it sends a report to the load balancer.

In the rest of this section, we first present the load balancing algorithm (Sect. 3.1) and the computation of each worker (Sect. 3.2). Then the DDPOR algorithm is presented over Sects. 3.3 and 3.4.

3.1 Load balancing

For simplicity, we assign one node of an N-node cluster as the centralized load balancer and the rest of $N-1$ nodes as *workers*. The load balancer monitors the status of all workers for the purpose of partitioning the workload. That is, we have one node execute the procedure LoadBalancer (Fig. 3), and the rest of nodes executes the procedure Worker (Fig. 4).

We use W_b to denote the set of busy workers, and W_i to refer to the set of idle workers. Initially all worker nodes are idle, and the load balancer randomly picks an idle worker w_0, and send a message to w_0 to have it start checking the program (lines 4–7 of Fig. 3). After this, the local balancer adds w_0 to the busy workers set W_b, and removes w_0 from the idle workers set W_i (lines 8–9 of Fig. 3). Then it keeps waiting for messages from busy workers until no work node is busy (lines 10–26 of Fig. 3). When the load balancer finishes the while loop, all workers must have finished exploring their part of state space, which means the whole state space has been explored. At this stage, the load balancer sends a termination message to every worker to terminate them and exit.

There are two categories of messages that the load balancer can receive from the workers:

- requests from busy workers to unload some work to idle workers.
- reports from busy workers after they finish exploring the assigned state space.

While exploring the assigned state space, if a worker ends up having more than a certain number of backtrack points in its stack, it implies that too much work might have been assigned to this worker. In this situation, this worker sends a work unloading request to the load balancer. If there are idle workers available, the load balancer passes along the idle worker's information (lines 21–24 of Fig. 3). Otherwise, it tells the requester that there are no idle workers available (lines 17–20 of Fig. 3).

3.2 Worker routine

Each worker (Fig. 4) keeps passively waiting for work unloading messages, and has DDPOR-enabled depth-first search for each assigned state space (lines 5–7 of Fig. 4). The worker exits the while loop and terminates when a termination message is received (line 4 of Fig. 4).

3.3 Distributed DPOR

Figure 5 shows the DDPOR algorithm. Comparing with the original DPOR algorithm in Fig. 1, we made the following changes:

- add work unloading primitives (line 5 of Fig. 5).

Fig. 5 Distributed dynamic partial order reduction

```
1: Initially: S is received from the work assigner;

2: DDPOR(S){
3:     let s = S.top;
4:     for each t ∈ s.enabled, DDPORUPDATEBACKTRACKSETS(S, t);
5:     if (there are more than n backtrack points in the S) UNLOADWORK(S);
6:     if (∃p ∈ Tid : ∃t ∈ s.enabled : tid(t) = p) {
7:       s.backtrack ← {p};
8:       s.done ← ∅;
9:       while(∃q ∈ s.backtrack \ s.done)
10:        s.done = s.done ∪ {q};
11:        s.backtrack = s.backtrack \ {q};
12:        let t ∈ s.enabled such that tid(t) = q, and let s′ be a state such that s -t-> s′;
13:        S.push(s′);
14:        DDPOR(S);
15:        S.pop();
16:      }
17:    }
18: }

19: UNLOADWORK(S) {
20:    send a work unload request to the load balancer;
21:    receive reply r from the load balancer;
22:    if(r shows no idle node available) return;
23:    let w_i be the idle worker node that the load balancer tells this node;
24:    let s be the deepest state in the search stack S such that s.backtrack ≠ ∅;
25:    let S_s be a copy of the sequence of states from s_0 to s;
26:    let s′ be the last state in S_s (i.e. s′ is a copy of s), and let τ ∈ s′.backtrack;
27:    s′.backtrack ← {τ};
28:    s′.done ← s′.done ∪ (s.backtrack \ {τ});
29:    send S_s to w_i;
30:    s.backtrack ← s.backtrack \ {τ};
31:    s.done ← s.done ∪ {τ};
32: }
```

– to avoid the redundant exploration of the state space among multiple nodes, we use DDPORUPDATEBACKTRACKSETS to compute the backtrack points in a different way from the original DPOR algorithm. We will present the details in Sect. 3.4.

In DDPOR, each time after updating the backtrack points, we check whether the number of backtrack points in the search stack has exceeded a value n (line 5 of Fig. 5). Here, n is the number of backtrack points in the search stack. If so, the current node decides to unload some of this excess work to the other nodes, as captured in procedure UNLOADWORK.

The UNLOADWORK routine first checks with the load balancer to see if there are any idle nodes. If not, the routine will return immediately (line 22 of Fig. 5). Otherwise, it sends the search stack S_s to the idle node w_i (lines 24–29 of Fig. 5). The algorithm in Fig. 5 does the unload work request each time it enters the DPOR routine. This may lead to repeated failures if there are no idle nodes available for a while (not observed in our experiments). Various heuristic solutions are possible in case it arises in practice (e.g., send aggregated requests more infrequently).

To derive the most benefit per exchanged work unloading message, we observe that backtrack points situated deeper in the stack typically have larger numbers of program-paths emanating from them. Based on this heuristic, we choose the deepest state s in the search stack that satisfies $s.backtrack = \varnothing$ (line 24 of Fig. 5). After unloading a backtrack point from s, on the current node, we will put the thread id of the transition in $s.done$ to avoid it being explored by the current node (lines 29–30 of Fig. 5).

3.4 Updating the backtrack set

In dynamic partial order reduction, the persistent set of a given state is computed dynamically. Procedure UPDATEBACKTRACKSETS in Fig. 1 shows how the backtrack points are computed. One problem we encountered with the pro-

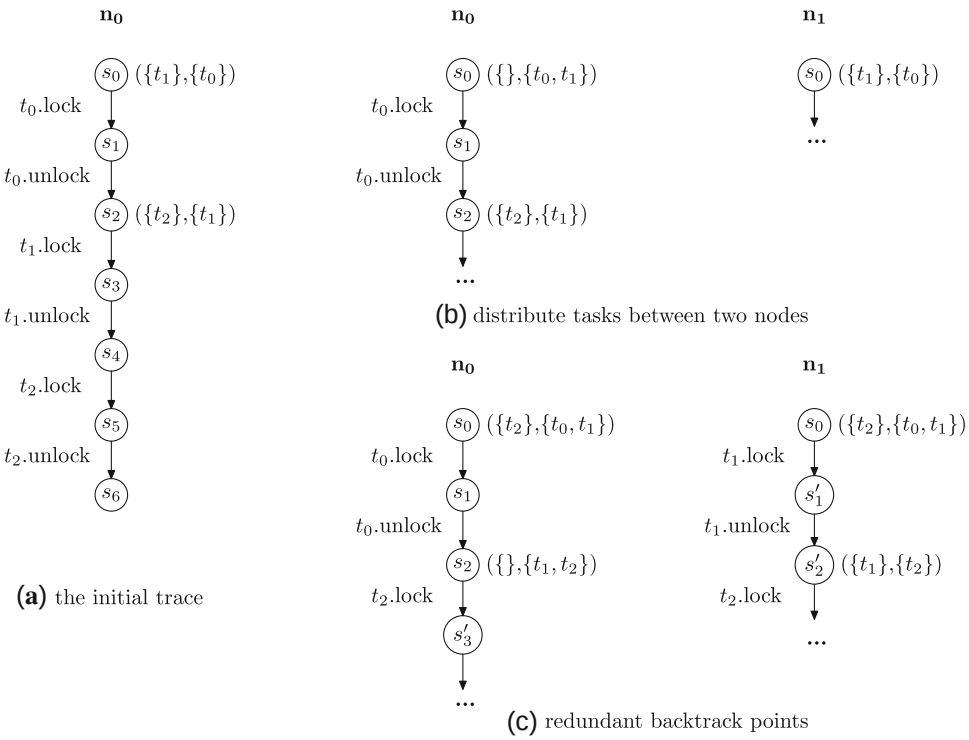
Fig. 6 A simple example

```
global: mutex  t;

                          thread t0:        thread t1:        thread t2:

lock(t);                  lock(t);          lock(t);
unlock(t);                unlock(t);        unlock(t);
```

Fig. 7 An example of redundant backtrackings. The sets maintained are $(s.backtrack, s.done)$



cedure UpdateBacktrackSets is that with more than two threads, it may result in redundancy exploration of the same branch in parallel mode.

The example in Fig. 6 illustrates this problem. The program has three threads, all of which first acquire the global lock t, and then release the lock. Obviously, there are 3! = 6 different interleavings for this concurrent program with DPOR.

Assume that we use a computer cluster that has only two worker nodes. We also assume that the bound in Fig. 5 for unloading is 1. Let the two workers be n_0 and n_1, and let the three threads be t_0, t_1, and t_2. Figure 7 shows how the work would be distributed between the two nodes if we follow the UpdateBacktrackSets routine shown in Fig. 1.

Let n_0 start concretely executing the program first, and n_1 is idle. When n_0 reaches the end of its trace, we can observe the interleaving of three threads as in Fig. 7a. Here, two backtrack points at s_0 and s_2 have been recorded. When the work node n_0 detects this (i.e., more than one backtrack point in the search stack), it will send a request to the load balancer for unloading work. First the load balancer will tell n_0 that n_1 is idle. Second, n_0 will send the search stack to n_1, following the UnloadWork routine in Fig. 5. Then the work node n_1 will receive the message and be ready for exploring the state space assigned to it. The left half of Fig. 7b captures this scenario.

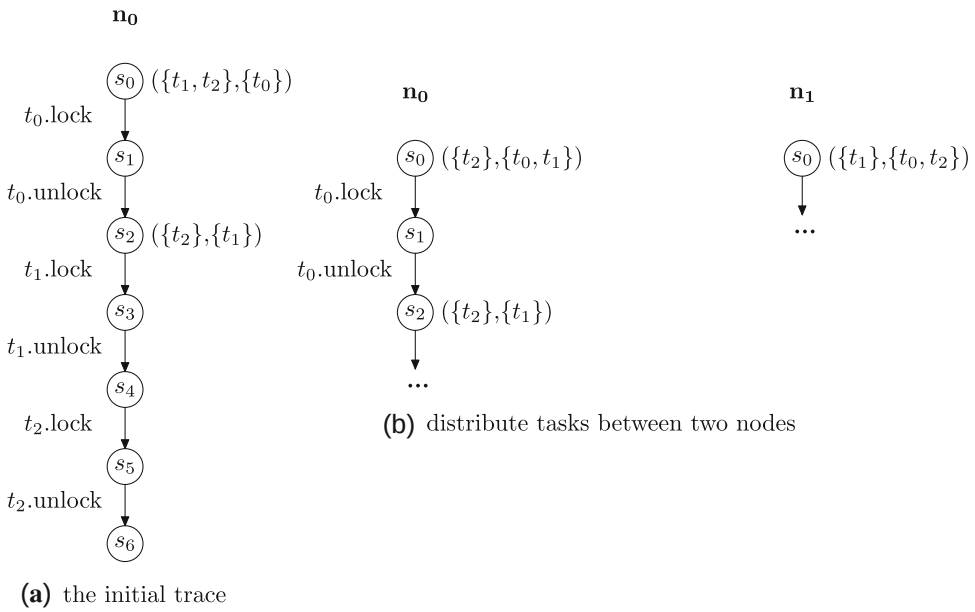
At this point, with respect to the situation in Fig. 7b, n_0 will explore transition $t_2.lock$ from the backtrack point s_2, while n_1 will explore transition $t_1.lock$ from s_0. Both nodes will update the backtrack information according to their own search stacks. The scenario in Fig. 7c results, in which both n_0 and n_1 compute and place t_2 in $s_0.backtrack$ the transition of which should be explored from s_0. This will result in redundant explorations being conducted by n_0 and n_1. In the worst case, this kind of redundancy may have all the workers explore the same interleaving, and result in little or no speedup (Our experiments shown in Section 4 confirms this).

This problem is caused by the fact that the algorithm shown in Fig. 1 computes $s.backtrack$ incrementally with respect to state s. Let s be a state and $s.saturated$ be the set of enabled transitions that have been explored from

```
1: DdporUpdateBacktrackSets(S, t) {
2:   let T be the transition sequence associated with S;
3:   for each(t_d ∈ T that is dependent and may be co-enabled with t){
4:     let s_d be the state in S from which t_d is executed;
5:     let E be {q ∈ s_d.enabled | tid(q) = tid(t), or q in T, q was after t_d and there is a happens-before relation for (q, t). }
6:     if (E ≠ ∅)
7:       choose any q in E, add tid(q) to s_d.backtrack;
8:     else
9:       s_d.backtrack = s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
10:  }
11: }
```

Fig. 8 Updating the backtrack sets of states in the distributed context

Fig. 9 With DdporUpdateBacktrackSets



(a) the initial trace

(b) distribute tasks between two nodes

when the search algorithm backtracks from s. We say that $s.backtrack$ is saturated if $s.backtrack = s.saturated$. In the distributed context, if we compute the backtrack sets of states in the same way as DdporUpdateBacktrackSets of Fig. 1, when a worker unloads the search task for a sub-state-space rooted with s to some idle node (Fig. 3), it is possible that $s.backtrack$ is only a pure subset of $s.saturated$. In this example, as $s_0.backtrack$ is not saturated when the copy of the stack is passed along, each node will independently add the same backtrack point in its search stack. This ends up in a redundant search among multiple nodes.

To solve this problem, we need to make $s.backtrack$ saturated as fast as possible while updating the backtrack set of s. We propose a solution in Fig. 8. This solution aggressively updates the backtrack sets of states. For each to be executed transition, the new routine DdporUpdateBacktrackSets checks the stack to find all states from which a dependent and may be co-enabled transition was executed (Line 5 of Fig. 8), and updates the correspondent backtrack set. As this routine adds multiple backtrack points to multiple states in a search stack, $s.backtrack$ does not need to wait until the last backtrack from s to saturate. Hence, it can speed up the saturation of $s.backtrack$. With the new routine, we will get the distributed scenario as shown in Fig. 9.

Note that as a heuristic, our solution does not provide guarantee that $s.backtrack = s.saturate$ at line 24 of Fig. 3 when UnloadWork is called. In general, we have not found a way to retain loose synchronizations between the threads and still avoid this redundancy.

Correctness:
The soundness of the DDPOR algorithm follows from the fact that the parallel algorithm is guaranteed to compute at least all the backtrack set entries computed by the sequential algorithm for every state. We alter only where this information is computed.

Table 1 Checking time with the sequential `Inspect`

Benchmark	Threads	Runs	Check using sequential `Inspect`(s)
fsbench	26	8192	29132
indexer	16	32768	118873
aget	6	113400	566296
bbuf	8	1938816	3971043

4 Implementation and experiments

We implemented DDPOR on top of `Inspect` [9] using MPI [10,11]. `Inspect` is a stateless runtime model checker for multithreaded C programs. `Inspect` uses DPOR to prune the state space. MPI (Message Passing Interface) is a message-passing library specification, designed to ease the use of message passing by end users. It is supported by virtually all supercomputers and clusters, and is the de facto standard of high-performance computing.

One interesting problem we encountered while we implemented the parallel `Inspect` is that the cluster's network file system can be a bottleneck for a parallel runtime checker if there are disk-write operations in the program under test. This problem can be easily avoided by using the local disks.

We conducted our experiments on a 72-node cluster with 2 GB memory and two 2.4 GHz Intel XEON processors on each node. We compiled the program with gcc-4.1.0 and -O3 option. We used LAM-MPI 7.1.1 [2] as the message passing interface. The runtimes that we report are the average runtimes calculated over three runs.

Table 1 shows some benchmarks we have used to test the parallel `Inspect`. In Table 1, the second column is the number of threads in each benchmark, the third column shows the number of runs needed for runtime checking the program, and the last column shows the time that the sequential `Inspect` needs for checking the program.

The first two benchmarks, *indexer* and *fsbench*, are from [6]. *Indexer* captures the scenarios in which multiple threads insert messages into a hash table concurrently. *fsbench* is an abstraction of the synchronization idiom in Frangipani file system. The third benchmark *aget* [13] is an ftp client in which multiple threads are used to download different segments of a large file concurrently. The last benchmark *bbuf* is an implementation of a bounded buffer with four producers and four consumers that have put/get operations on it.

Indexer and *fsbench* are relatively small benchmarks. Using one node in the cluster, the sequential `Inspect` takes about 25 min to check *indexer*, and 5 min to check *fsbench*. Using parallel `Inspect` and at most 65 nodes (one node as the load balancer and 64 worker nodes), we can check both of them within 40 s.

Figure 10 shows the speedup we got using the parallel `Inspect` against the sequential `Inspect` on *indexer* and *fsbench*. As the performance of using DDPORUPDATEBACKTRACKSETS in Fig. 8 does not differ significantly from using the original UPDATEBACKTRACKSETS in Fig. 1, we do not show the comparison in Fig. 10.

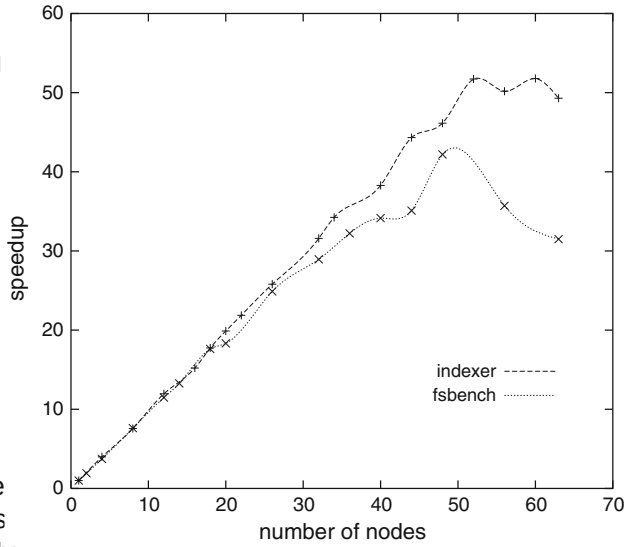


Fig. 10 The speedup of the two benchmarks, *indexer* and *fsbench* from [6]. As the state spaces of these two benchmarks are relatively small, with the number of worker nodes increasing, the communication overhead increases more rapidly than the time reduction we get from distributing the work to more nodes. As a result, we see a degradation of speedup when we use more than 52 nodes to do parallel checking for *indexer*, and more than 48 nodes for *fsbench*.

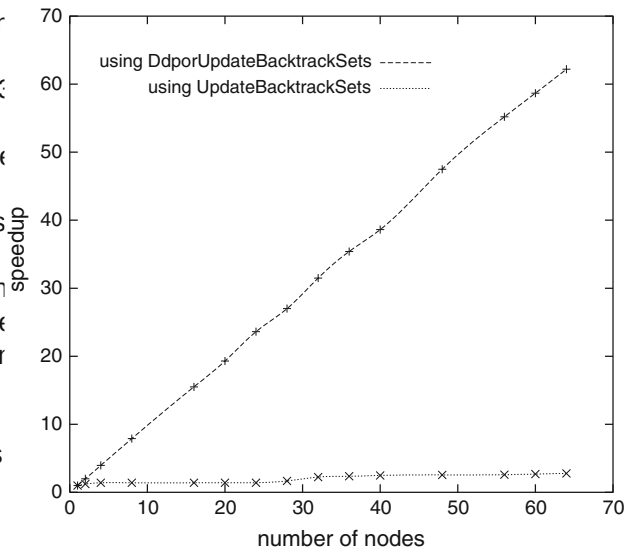


Fig. 11 Speedups on the bounded buffer example

Figure 11 shows the speedup we got using the parallel `Inspect` on *bbuf*. The sequential `Inspect` needs more than 11 hours to finish checking the program. During this period of time, `Inspect` needs to re-run the program for more than 19 million times. As shown in Fig. 11, the par-

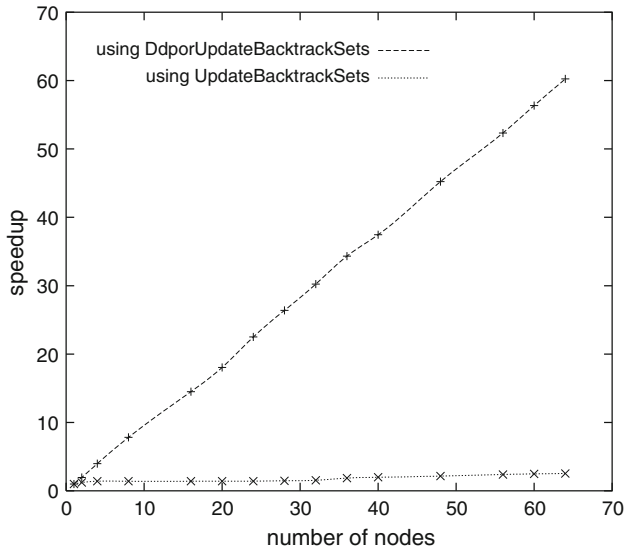


Fig. 12 Speedups on the aget example

allel `Inspect` can give us almost linear speedup. It turns out that we can get a speedup of 23 out of 64 worker nodes (totally 65 nodes, including the load balancer), and reduce the checking time to 11 min. In this figure, we also show the comparison between the speedup we got using DDPORUPDATEBACKTRACKSETS or UPDATEBACKTRACKSETS in DDPOR to update the backtrack sets of states (line 4 of Fig. 5). As we can see, without the aggressive backtrack sets updates in DDPORUPDATEBACKTRACKSETS, we get little speedup while the number of nodes increases.

Figure 12 shows the speedup using the parallel `Inspect` on *aget*. There are data races in the original *aget*. We fixed those data races and did experiments on the fixed version. We reduced the size of the data package, which gets from the ftp server, to 512 bytes, to avoid the non-determinism introduced by the network environment. The result again confirms that parallel `Inspect` can give out almost linear speedup, and our extension on the original DPOR is efficient.

5 Related work

Parallel and distributed model checking has been a topic of growing interest, with a special conference series (PDMC) devoted to this topic. An exhaustive literature survey is beyond the scope of this paper. Quite a few distributed and parallel model checkers based on message passing have been developed for Murphi and SPIN [14–18]. Stern and Dill [14] developed a parallel Murphi which distributes states to multiple nodes for further exploration according to the state's signature. They pointed out the idea of coalescing states into larger messages for better network utilization in the context of model checking. Eddy [15] extended the work and studies the parallel and distributed model checking under the multicore architecture. Kumar and Mercer [17] improved the load balancing method in parallel Murphi. Recently, Holzmann and Bosnacki [18] designed a multicore model checking algorithm to improve SPIN to fully utilize the multicore chips.

Brim et al. [19] proposed a distributed partial order reduction algorithm for generating a reduced state space. The algorithm exploits features of the partial order reduction which makes the idea of distributed DFS-based algorithm feasible. Palmer et al. [20,21] propose another distributed partial order reduction algorithm based on the two-phase partial order reduction algorithm.

As far as the authors know, our work is the first effort on using parallelism to speed up runtime model checking for multithreaded programs.

6 Conclusion

Checking time has been the major bottleneck for stateless runtime model checkers. We propose a distributed dynamic partial order reduction algorithm to use parallelism to speed up the stateless model checking. Our experiments confirm that this algorithm scales well on a wide variety of nodes. It can give out almost linear speedup compared with the sequential stateless model checker.

References

1. Godefroid, P.: Model Checking for Programming Languages using Verisoft. In: POPL, pp. 174–186 (1997)
2. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
3. Robby, E.R., Dwyer, M.B., Hatcliff, J.: Bogor: an extensible and highly-modular software model checking framework. In: ESEC/SIGSOFT FSE, pp. 267–276 (2003)
4. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, pp. 1–13. ACM Press, New York, NY, USA (2004)
5. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: Computer Aided Verification, 16th International Conference. Lecture Notes in Computer Science, vol. 3114, pp. 484–487. Springer (2004)
6. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) POPL, pp. 110–121. ACM (2005)
7. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 446–455. ACM (2007)
8. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer (1996)
9. Yang, Y., Chen, X., Gopalakrishnan, G.: UUCS-08-004: Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical report (2008)

10. Snir, M., Otto, S.: MPI-The Complete Reference: The MPI Core. MIT Press, Cambridge (1998)
11. http://www.mpiforum.org/docs/docs.html
12. http://www.lammpi.org/
13. http://www.enderunix.org/aget/
14. Stern, U., Dill, D.L.: Parallelizing the Mur Veri er. In: Grumberg, O. (ed.) Computer Aided Veri cation, 9th International Conference, CAV '97, Haifa, Israel, June 22–25, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1254, pp. 256–278. Springer (1997)
15. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in Eddy. In: SPIN, pp. 108–125(2006)
16. Sivaraj, H., Gopalakrishnan, G.: Random walk based heuristic algorithms for distributed memory model checking. Electr. Notes Theor. Comput. Sci. 89(1) (2003)
17. Kumar, R., Mercer, E.G.: Load balancing parallel explicit state model checking. Electr. Notes Theor. Comput. Sci. 128(3), 19–34 (2005)
18. Holzmann, G., Bosnacki, D.: Multi-core model checking with Spin (2007)
19. Brim, L., Cerna, I., Moravec, P., Simsa, J.: Distributed partial order reduction of state spaces. In: PDMC, vol. 1 (2004)
20. Palmer, R., Gopalakrishnan, G.: Partial order reduction assisted parallel model checking. In: PDMC (2002)
21. Palmer, R., Gopalakrishnan, G.: A distributed partial order reduction algorithm. In: FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems, pp. 370. Springer, London, UK (2002)