

# Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software\*

Yu Yang   Xiaofang Chen   Ganesh Gopalakrishnan   Robert M. Kirby

School of Computing, University of Utah  
Salt Lake City, UT 84112, U.S.A.  
{yuyang, xiachen, ganesh, kirby}@cs.utah.edu

**Abstract.** Runtime (dynamic) model checking is a promising verification methodology for real-world threaded software because of its many features, the prominent ones being: (i) it avoids the need to extract a model and instead runs the actual code, and (ii) the precision of information available at run-time allows techniques such as dynamic partial order reduction (DPOR) [1] to dramatically cut down the number of interleavings examined. Unfortunately, DPOR does not have many implementations for real thread libraries such as POSIX Pthreads, and suffers from high computational overheads due to a stateless search that requires re-executions. In our previous work [2], we designed a runtime model checker, *inspect*, that overcomes the first of these drawbacks. *Inspect* has been shown capable of detecting data races, deadlocks and other incorrect API usages in real-world PThreads C programs. In this paper, we describe a distributed version of *inspect*, which implements an extended DPOR algorithm. Our two key contributions are: (i) a practical algorithm for distributed dynamic partial order reduction; (ii) the innovations that helped distributed *inspect* attain nearly linear (with respect to the number of CPUs) speedup on realistic examples.

## 1 Introduction

Runtime (dynamic) model checking (e.g., as in [3]) is a promising verification methodology for real-world threaded software because of its many features. It avoids the (implicit or explicit) overhead of modeling programs that is usually required by other model checkers [4,5,6,7]. The precision of information available at run-time allows techniques such as dynamic partial order reduction (DPOR) [1] to dramatically cut down the number of interleavings examined. In our previous work [2], we designed a runtime model checker, *inspect*, that (to the best of our knowledge) is the first implementation of DPOR that handles the widely used POSIX Pthreads library. *Inspect* explores *relevant* interleavings (generated by DPOR) of a multithreaded C program with a specific testing scenario. In this setting, *Inspect* has detected data races, deadlocks, and other incorrect API usages in real-world PThreads C programs. However, we observed that run-time

---

\* Supported in part by NSF award CNS00509379, Microsoft HPC Institute Program, and SRC Contract 2005-TJ-1318.

is a major limiting factor of `inspect`. `Inspect` explores the state space by executing the program concretely and observing its visible operations. As it is not easy to capture and restore the state of a C program which runs concretely, `inspect` does not keep the search history, instead employing stateless search [3]) that can incur high overheads.<sup>1</sup>

A runtime model checker such as `inspect` is potentially “embarrassingly parallel” based on the casual observation that since stateless search does not maintain the search history, different branches of an acyclic state space can be explored concurrently, and with very loose synchronizations. We implemented a parallel version of `inspect` based on this observation, employing a centralized load balancer to distribute work among multiple nodes. Unfortunately, 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 standard 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 Section 3.4.

**Work Distribution Heuristics:** Numerous heuristics help achieve efficient work distribution in `inspect`. 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.

This paper describes these extensions to the DPOR algorithm proposed in [1] and detailed in [2]. Our experiments demonstrate almost linear speedup with increasing number of nodes (CPUs). For example, one of our benchmarks which has eight threads and requires more than 11 hours to finish checking using sequential `inspect` can be checked by the parallel `inspect` within 11 minutes using 65 nodes. The parallel `inspect` gives a speedup of 63.2 out of 65.

**Roadmap:** Section 2 presents background information on `inspect` and DPOR. Section 3 presents the extended DPOR algorithm used in parallel `inspect`. Sec-

---

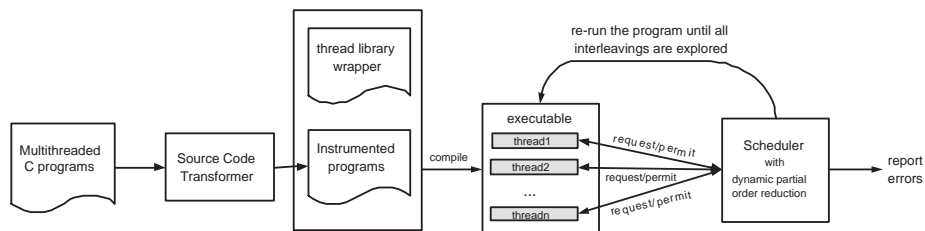
<sup>1</sup> Given programs that do not terminate, a stateless search method (such as used in `inspect`) requires depth-bounding or some other technique to ensure termination. This was not an issue in our practical test programs. In this paper, we focus only on checking multithreaded programs that terminate.

tion 4 presents implementation detail, and the experiment results, Section 5 the related work, and Section 6 our concluding remarks.

## 2 Background

### 2.1 Overview of Inspect

Modeling the library functions employed in, and the runtime environment of multithreaded C programs is non-trivial. To the best of our knowledge, Verisoft [3] is the only model checker capable of checking concurrent C programs without incurring modeling overheads. Unfortunately, Verisoft focuses on concurrent programs that interact through inter-process communication mechanisms. In a multithreaded program, threads can interact not only through explicit synchronization/mutual exclusion primitives, but also through read/write operations on shared data objects. Our runtime model checker `inspect` can handle these details, and can systematically explore all possible interleavings of a multithreaded C program under a specific testing scenario, employing DPOR.



**Fig. 1.** Inspect’s workflow

Figure 1 shows the workflow of `inspect`. The source code transformer instruments the program at the source code level to arrange communications with a scheduler at global interaction points. Here, a thread library wrapper helps intercept thread library calls. Finally, a centralized scheduler embodies the DPOR algorithm, and controls the interleaved executions of the threads according to it. In `inspect`, instrumentation can be done automatically for C programs. The instrumented program is compiled, and the executable is run repeatedly under the control of the scheduler until all relevant interleavings among the threads required by DPOR are explored.

Before performing any operation that might have a side effect on other threads, the instrumented program sends a request to the scheduler. The scheduler can either allow the requesting process to proceed, or block it for any finite duration by postponing a reply.

## 2.2 Definitions

A multithreaded program can be modeled as a concurrent system, which consists of a finite set of *threads*, and a set of *shared objects*. Threads communicate with each other only through shared objects. Operations on shared objects are called *visible operations*, while the rest are *invisible operations*. We assume threads can only be blocked by visible operations. A *state* of a multithreaded program consists of the global state of all shared objects and the local state of each thread. A *transition* moves the program from one state to the next state by performing one visible operation of a certain thread, followed by a finite sequence of invisible operations, ending just before the next visible operation of that thread.

Given a state  $s$  and a transition  $t$ , we use the following notations:

- $t.tid$  denotes the identity of the thread that executes  $t$ .
- $next(s, t)$  refers to the state which is reached from  $s$  by executing  $t$ .
- $s.enabled$  denotes the set of transitions that are enabled from  $s$ . A thread  $p$  is enabled in a state  $s$  if there exists some transition  $t$  such that  $t \in s.enabled$  and  $t.tid = p$ .
- $s.backtrack$  refers to the backtrack set at state  $s$  (Figure 2).  $s.backtrack$  is a set of thread identities. Here,  $\{t \mid t.tid \in s.backtrack\}$  is the set of transitions which are enabled but have not been executed from  $s$ .
- $s.done$  denotes the set of threads examined at  $s$ . Similar to  $s.backtrack$ ,  $s.done$  is also a set of thread identities. Here,  $\{t \mid t.tid \in s.done\}$  is the set of transitions that have been executed from  $s$ .

## 2.3 Dynamic Partial Order Reduction

Partial order reduction (POR) techniques [9] are those that avoid interleaving independent transitions during search.

Given the set of enabled transitions from a state  $s$ , partial order reduction algorithms attempt to explore only a (proper) subset of  $s.enabled$ , and at the same time guarantee that the properties of interest will be preserved. Such a subset is called *persistent set*.

Static POR algorithms compute the persistent set of a state  $s$  immediately after reaching it. As for multithreaded programs, persistent sets computed statically will be excessively large because of the limitations of static analysis. For instance, if two transitions leading out of  $s$  access an array  $\mathbf{a}[]$  by indexing it at locations captured by expressions  $\mathbf{e1}$  and  $\mathbf{e2}$  (i.e.,  $\mathbf{a}[\mathbf{e1}]$  and  $\mathbf{a}[\mathbf{e2}]$ ), a static analyzer may not be able to decide whether  $\mathbf{e1}=\mathbf{e2}$  (and hence whether the transitions are dependent or not). Flanagan and Godefroid introduced dynamic partial-order reduction (DPOR) [1] to dynamically compute smaller persistent sets, capitalizing on runtime information.

In DPOR, given a state  $s$ , the persistent set of  $s$  is not computed immediately after reaching  $s$ . Instead, DPOR populates the persistent set of  $s$  while searching under  $s$  according to depth-first search (DFS). Figure 2 recapitulates the DPOR algorithm. In procedure *update\_backtrack\_info*, we see how the backtrack state

```

1: StateStack  $S$ ;
2: TransitionSequence  $T$ ;

3: DPOR( ) {
4:   State  $s = S.top$ ;
5:   update_backtrack_info( $s$ );
6:   if ( $\exists$  thread  $p, \exists t \in s.enabled, t.tid = p$ ) {
7:      $s.backtrack = \{p\}$ ;
8:      $s.done = \emptyset$ ;
9:     while ( $\exists q \in s.backtrack \setminus s.done$ ) {
10:       $s.done = s.done \cup \{q\}$ ;
11:       $s.backtrack = s.backtrack \setminus \{q\}$ ;
12:      let  $t_n \in s.enabled, t_n.tid = q$ ;
13:       $T.append(t_n)$ ;
14:       $S.push(next(s, t_n))$ ;
15:      DPOR();
16:       $T.pop\_back()$ ;
17:       $S.pop()$ ;
18:    }
19:  }
20: }

21: update_backtrack_info(State  $s$ ) {
22:   for each thread  $p$  {
23:     let  $t_n \in s.enabled, t_n.tid = p$ ;
24:     let  $t_d$  be the latest transition in  $T$  that is dependent and may be co-enabled
       with  $t_n$ ;
25:     if ( $t_d \neq \text{null}$ ) {
26:       let  $s_d$  be the state in  $S$  from which  $t_d$  is executed;
27:       let  $E$  be  $\{q \in s_d.enabled \mid q.tid = p, \text{ or } q \text{ in } T, q \text{ happened after } t_d$ 
         and is dependent with some transition in  $T$  which was executed by
          $p$  and happened after  $q\}$ 
28:       if ( $E \neq \emptyset$ )
29:         choose any  $q$  in  $E$ , add  $q.tid$  to  $s_d.backtrack$ ;
30:       else
31:          $s_d.backtrack = s_d.backtrack \cup \{q.tid \mid q \in s_d.enabled\}$ ;
32:     }
33:   }
34: }

```

**Fig. 2.** Dynamic partial-order reduction

of a state called  $s_d.backtrack$  is updated while exploring a state  $s$  reached from  $s_d.backtrack$  under DFS. Observe from line 29 that we add to  $s_d.backtrack$  a thread id  $q.tid$ , where  $s_d$  is the most recent state, searching back from  $s$ , where a transition that depends on transition  $t_n$  occurs. When the DFS unwinds to state  $s_d.backtrack$ , the backtrack set is consulted and the threads recorded in

there are scheduled, provided ‘done’ is not true (line 9). Last but not least, in `inspect`, we employ *sleep sets* [8] to avoid interleaving independent actions.

### 3 Algorithm

In the DPOR algorithm, the thread ids recorded in the backtrack set of a state  $s$  (i.e.,  $s.backtrack$ ) help generate different (non-equivalent) executions out of  $s$ . These executions can be independently explored. It is this insight that distributed `inspect` capitalizes. In fact, as DPOR is often best 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 transition to be executed from a backtrack point; (ii) the portion of the search stack from the initial state to the backtrack point; (iii) the transition sequence from the initial state to reach the backtrack point. All this information is easily obtained from the search stack. A centralized load balancer can help balance the work among multiple nodes, employing very limited synchronizations.

In this section, we first present an overview of the load balancing algorithm (Section 3.1) and the computation of each worker (Section 3.2). Our extended DPOR algorithm is presented over Sections 3.3 and 3.4.

#### 3.1 Load Balancing

In parallel `inspect`, we assign one node of an  $N$ -node cluster as the centralized load balancer (Figure 3), and the rest of  $N - 1$  nodes as workers (a simple initial approach to ease programming). The load balancer monitors the status of all workers for the purpose of partitioning the workload. Two classes of workers are maintained: *busy\_workers* – the set of workers busy exploring some parts of the state space, and *idle\_workers* – the set of workers which are available for new work (initially all workers).

The load balancer chooses an idle worker, starts checking the program under test on the selected node, and adds this node to the *busy\_workers* set (Line 6-9). Then it keeps waiting for messages from busy workers until the *busy\_workers* set is empty (Line 10-26). At this stage, all workers have finished exploring their part of state space, which means the whole state space has been explored. In the last step (Line 27-28), the load balancer will send a termination message to every worker to terminate them and exit.

The messages that the load balancer can receive from the workers fall into two categories:

- a request from a busy worker to unload some work to idle workers.
- a report message from a busy worker after it finishes 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 will send a work unloading request to the load balancer. If there are idle workers available, the load balancer passes along the idle worker’s information (line 21-24). Otherwise, it tells the requester that there are no idle worker available (Line 17-20).

```

1: Program  $P$ ;
2: WorkerSet  $busy\_workers, idle\_workers$ ;

3: load_balance( ) {
4:    $idle\_workers = \{ \text{all workers in the cluster} \}$ ;
5:    $busy\_workers = \emptyset$ ;
6:   let  $n_a$  be a worker,  $n_a \in idle\_workers$ ;
7:    $idle\_workers = idle\_workers \setminus \{n_a\}$ ;
8:   start checking  $P$  on  $n_a$ ;
9:    $busy\_workers = \{n_a\}$ ;
10:  while ( $busy\_workers \neq \emptyset$ ) {
11:    receive event  $e$  from any worker  $w$ ;
12:    if( $e$  is work finish notification)
13:       $busy\_workers = busy\_workers \setminus \{w\}$ ;
14:       $idle\_workers = idle\_workers \cup \{w\}$ ;
15:    }
16:    else if( $e$  is new work request){
17:      if( $idle\_workers = \emptyset$ ){
18:        reply “no idle workers” to  $w$ ;
19:        continue;
20:      }
21:      let  $n_b$  be a worker in  $idle\_workers$ ;
22:       $idle\_workers = idle\_workers \setminus \{n_b\}$ ;
23:       $busy\_workers = busy\_workers \cup \{n_b\}$ ;
24:      send  $n_b$ ’s information to  $w$ ;
25:    }
26:  }
27:  for each  $worker \in idle\_workers$ 
28:    send a termination message to  $worker$ ;
29: }
```

**Fig. 3.** The load balancing algorithm

### 3.2 Worker Routine

Each worker (Figure 4) keeps passively waiting for work unloading messages, and has DPOR-enabled depth first search for each assigned state space (Line 7-9). Here a modified DPOR routine (detailed in Section 3.3) is used. The worker

```

1: worker_node() {
2:   while (true) {
3:     wait for an incoming message, let it be msg;
4:     if(msg is terminating)
5:       return;
6:     receive task description;
7:     DPOR();
8:     send report to the load balancer;
9:   }
10: }

```

**Fig. 4.** The routine that runs on each worker

exits the while loop and terminates when a termination message is received (Line 4-5). The task description message that a worker receives (Line 6) for starting a new backtrack point includes:

- the portion of the search stack from the bottom until the backtrack point.
- the transition sequence to reach the backtrack point from the initial state.
- the transition to be executed from the backtrack point

The transition sequence is used to help the worker which is assigned the task to replay the program until the backtrack point. The state stack is necessary to help the node to avoid exploring the backtrack points that other nodes have explored.

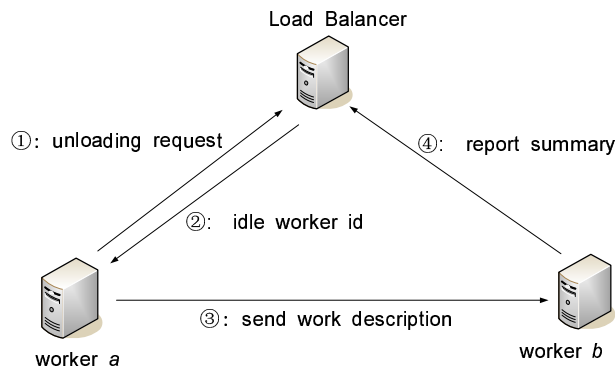
Figure 5 illustrates how the workers and the load balancer collaborate. Let  $a$  be a busy worker and  $b$  an idle one, with  $a$  trying to unload some work to  $b$ . First  $a$  sends a request to the load balancer. If there are idle nodes, the load balancer will return an idle node's id to the worker. In our example, the load balancer 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.

### 3.3 Distributed DPOR

Figure 6 shows our distributed DPOR algorithm. Comparing with the original DPOR algorithm in Figure 2, we made the following changes:

- add communication and work unloading primitives (Line 7-8 in Figure 6).
- to avoid the redundant exploration of the state space among multiple nodes, we compute the backtrack points in a different way from the original DPOR algorithm. We will present this in Section 3.4.

In this distributed DPOR, each time after updating the backtrack points, we will check whether the number of backtrack points in the search stack has



**Fig. 5.** The message flow between the load balancer and the workers

exceeded a value  $n$  (Line 7). 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 `unload_work`.

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  $s.backtrack \neq \emptyset$  (Line 29). 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 (Line 37-38).

The `unload_work` routine first checks with the load balancer to see if there are any idle nodes. If not, the routine will return immediately (Line 27-28). Otherwise, it finds and sends information pertaining to the deepest backtrack point, along with the transition sequence from the initial state to that backtrack point, to the idle node. The algorithm in Figure 6 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).

### 3.4 Updating the Backtrack Set

In dynamic partial order reduction, the persistent set of a given state is computed dynamically. Procedure `update_backtrack_info` in Figure 2 shows how the backtrack points are computed. One problem we encountered with the original DPOR algorithm is that with more than two threads, it may result in redundancy exploration of the same branch in parallel mode.

The example in Figure 7 illustrates this problem. The program has three threads, all of which first acquire the global lock  $t$ , and then release the lock.

```

1: StateStack  $S$ ;
2: TransitionSequence  $T$ ;
3: Transition  $t$ ;

4: DPOR( ) {
5:   State  $s = S.top$ ;
6:   update_backtrack_info( $s$ );            $\triangleleft$  modified, details in Section 3.4
7:   if (there are more than  $n$  backtrack points in the  $S$ )  $\triangleleft$  added
8:     unload_work();                      $\triangleleft$  added
9:   if ( $\exists$  thread  $p, \exists t \in s.enabled, t.tid = p$ ) {
10:      $s.backtrack = \{p\}$ ;
11:      $s.done = \emptyset$ ;
12:     while ( $\exists q \in s.backtrack \setminus s.done$ ) {
13:        $s.done = s.done \cup \{q\}$ ;
14:        $s.backtrack = s.backtrack \setminus \{q\}$ ;
15:       let  $t_n \in s.enabled, t_n.tid = q$ ;
16:        $T.append(t_n)$ ;
17:        $S.push(next(s, t_n))$ ;
18:       DPOR();
19:        $T.pop_back()$ ;
20:        $S.pop()$ ;
21:     }
22:   }
23: }

24: unload_work( ) {
25:   send a work unload request to the load balancer;
26:   receive reply  $rep$  from the load balancer;
27:   if( $rep$  says no idle node available)
28:     return;
29:   let  $s$  be the deepest state in stack  $S$  that  $s.backtrack \neq \emptyset$ ;
30:   let  $T_s$  be the transition sequence to reach  $s$  from the initial state;
31:   let  $S_s$  be a copy of the sequence of states from the bottom of  $S$  to  $s$ ;
32:   choose  $t \in s.backtrack$ ;
33:   let  $s'$  be the last state in  $S_s$  (i.e.  $s'$  is a copy of  $s$ );
34:    $s'.backtrack = \{t\}$ ;
35:    $s'.done = s'.done \cup (s.backtrack \setminus \{t\})$ ;
36:   send ( $S_s, T_s, t$ ) to the idle node;
37:    $s.backtrack = s.backtrack \setminus \{t\}$ ;
38:    $s.done = s.done \cup \{t\}$ ;
39: }

```

**Fig. 6.** DPOR for parallelization

Obviously, there are  $3! = 6$  different interleavings for this concurrent program with DPOR.

Assume we use a cluster that has only two worker nodes. We also assume that the bound  $n$  in Figure 6 for unloading is 1. Let the two workers be  $n_0$  and  $n_1$ ,

```

global: mutex t;

thread t0:      thread t1:      thread t2:
lock(t);       lock(t);       lock(t);
unlock(t);     unlock(t);     unlock(t);

```

Fig. 7. A simple example

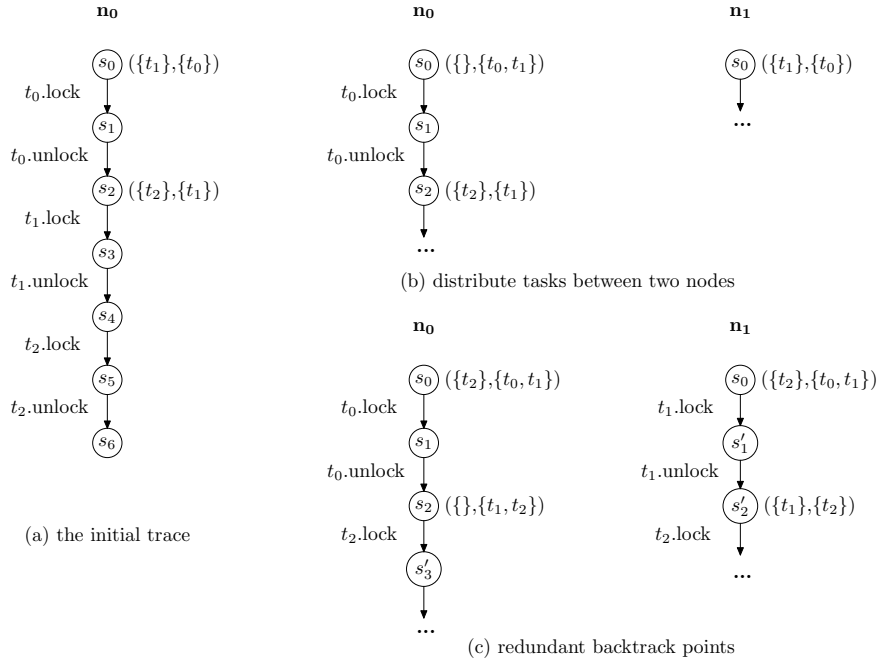


Fig. 8. An example of redundant backtrackings. The sets maintained are  $(s.backtrack, s.done)$

and let the three threads be  $t_0$ ,  $t_1$  and  $t_2$ . Figure 8 shows how the work would be distributed between the two nodes if we follow the *update\_backtrack\_info* routine shown in Figure 2.

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 Figure 8(a). 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 backtrack point, transition sequence, copy of the search stack to  $n_1$ , following the *unload\_work* routine in Figure 6. Then the work node  $n_1$  will receive the message and ready

for exploring the state space assigned to it. The left half of Figure 8(b) captures this scenario.

At this point, with respect to the situation in Figure 8(b),  $n_0$  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 Figure 8(c) results, in which both  $n_0$  and  $n_1$  compute and place  $t_2$  in  $s_0.backtrack$  whose transition 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).

```

1: StateStack  $S$ ;
2: TransitionSequence  $T$ ;

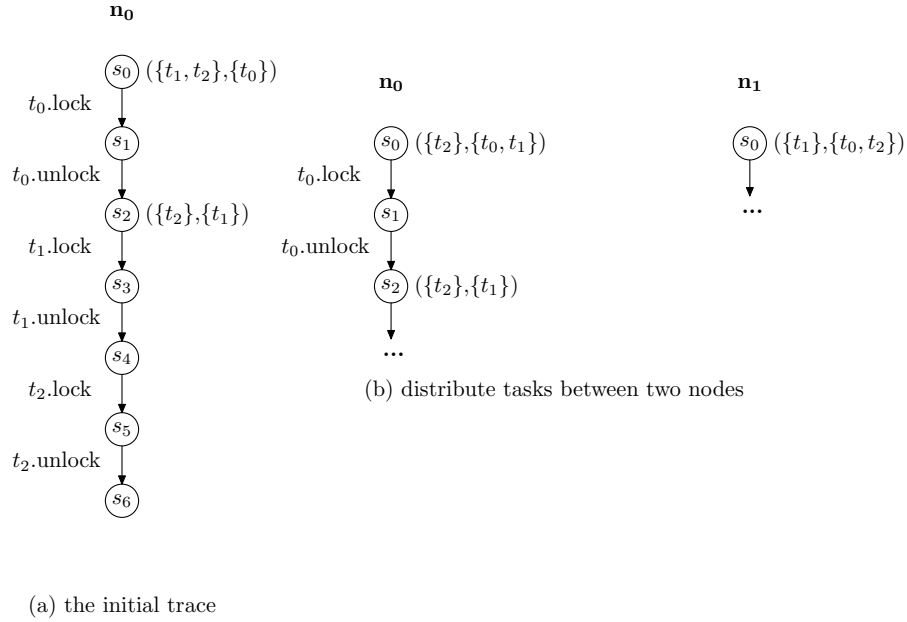
3: update_backtrack_info(State  $s$ ) {
4:   for each thread  $p$  {
5:     let  $t_n \in s.enabled, t_n.tid = p$ ;
6:     for each  $t_d \in T$  that is dependent and may be co-enabled with  $t_n$  {
7:       let  $s_d$  be the state in  $S$  from which  $t_d$  is executed;
8:       let  $E$  be  $\{q \in s_d.enabled \mid q.tid = p, \text{ or } q \text{ in } T, q \text{ happened after } t_d$ 
           and is dependent with some transition in  $T$  which was executed by
            $p$  and happened after  $q\}$ 
9:       if ( $E \neq \emptyset$ )
10:        choose any  $q$  in  $E$ , add  $q.tid$  to  $s_d.backtrack$ ;
11:       else
12:          $s_d.backtrack = s_d.backtrack \cup \{q.tid \mid q \in s_d.enabled\}$ ;
13:     }
14:   }
15: }
```

**Fig. 9.** Modified `update_backtrack_info`

This problem is caused by the algorithm shown in Figure 2 computing  $s.backtrack$  incrementally with respect to state  $s$ . In parallel `inspect`, when a worker unloads work to some idle node, it is possible that the full backtrack set has not yet been associated with states in the copy of the stack being passed along. To solve this problem, given a state  $s$ , one must attempt to compute all transitions associated with  $s.backtrack$  as aggressively as possible. We observe that the `update_backtrack_info` routine shown in Figure 2 only updates the latest state in the search stack from which the enabled transition in dependent and may be co-enabled with the next transition (Line 25-32 in Figure 2).

The modified `update_backtrack_info` routine is shown in Figure 9. For each to be executed transition  $t$ , the new routine will check the stack to find all states from which a dependent and may be co-enabled transition was executed (Line 6 of Figure 9), and update the correspondent backtrack set. With the new routine,

we will get the distributed scenario as shown in Figure 10. Note that this is only a heuristic; we do not know of a way to retain loose synchronizations between the threads and still avoid this redundancy.



**Fig. 10.** With the modified *update\_backtrack\_set*

**Correctness:** The soundness of the final DPOR algorithm described (employed in parallel **inspect**) 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.

## 4 Implementation and Experiments

We implemented the parallel **inspect** using MPI [10,11]. MPI (Message Passing Interface) is a message-passing library specification, designed to ease the use of message passing by end users. It is the *de facto* standard of high performance computing. MPI makes writing parallel program much easier, and is supported by virtually all supercomputers and clusters. We used the MPI routines **MPI\_Send** and **MPI\_Recv** for communication among nodes.

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. We note that this problem can be easily avoided by using the local disks.

**Table 1.** Checking time with the sequential `inspect`

benchmark	threads	runs	check using sequential inspect (sec)
fsbench	26	8,192	291.32
indexer	16	32,768	1188.73
aget	6	113,400	5662.96
bbuf	8	1,938,816	39710.43

We conducted our experiments on a 72-node cluster with 2GB memory and two 2.4GHz 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 [12] as the message passing interface. The runtimes that we report are the average runtimes calculated over three runs.

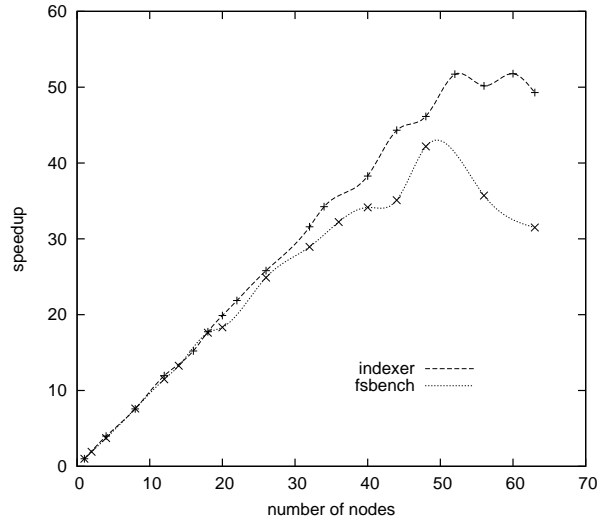
Table 1 shows some benchmarks we have used to test the parallel `inspect`. In Table 1, the second column is the number of threads in each benchmark, the third column shows the number of runs needed for runtime checking the program, and the last column shows the time that the sequential `inspect` needs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

The first two benchmarks, *indexer* and *fsbench*, are from [1]. *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 minutes to check *indexer*, and 5 minutes 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 seconds.

Figure 11 shows the speedup we got using the parallel `inspect` against the sequential `inspect` on *indexer* and *fsbench*. As the performance of using the modified *update\_backtrack\_info* in Figure 9 does not differ significantly from using the original *update\_backtrack\_info* in Figure 2, we do not show the comparison in Figure 11.

Figure 12 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 1.9 million times. As shown in Figure 12, the parallel `inspect` can give us almost linear speedup. It turns out that we can get a speedup of 63.2 out of 64 worker



**Fig. 11.** The speedup of the two benchmarks, *indexer* and *fsbench* from [1]. 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*.

nodes (totally 65 nodes, including the load balancer), and reduce the checking time to 11 minutes. In this figure, we also show the comparison between the speedup we got using the modified *update\_backtrack\_info* in Figure 9 and the original *update\_backtrack\_info* in Figure 2. As we can see, without the modification in Figure 9, we get little speedup while the number of nodes increases.

Figure 13 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 *aget* 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 [14,15,16,17,18]. Stern and Dill [14] developed a parallel Murphi which distributes states to multiple nodes for further exploration according to the

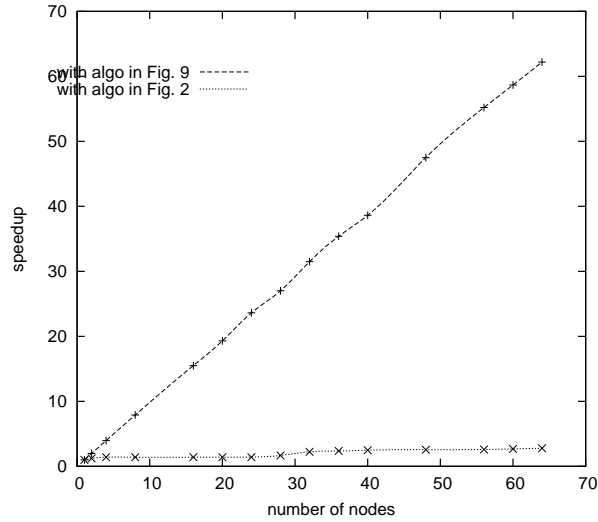


Fig. 12. Speedups on the bounded buffer example.

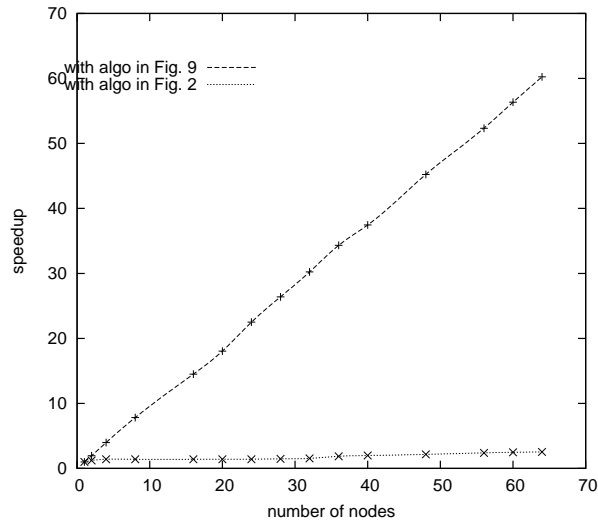


Fig. 13. Speedups on the aget example.

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] extends the work and studies the parallel and distributed model checking under the multicore architecture. Kumar and Mercer [17] improve the load balancing method in parallel Murphi. Recently Holzmann and Bosnacki [18] design a mul-

ticore model checking algorithm to improve SPIN to fully utilize the multicore chips.

Brim et al. [19] propose 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 runtime model checkers such as `inspect`. We design a distributed dynamic partial order reduction algorithm, and develop a parallel version of `inspect`, using parallelism to speed up model checking. Our experiments confirm that parallel `inspect` is quite robust and scales well on a wide variety of nodes. It can give out almost linear speedup compared with the sequential `inspect`.

### Acknowledgment

We gratefully acknowledge the computational support provided by the Scientific Computing and Imaging Institute at the University of Utah, thank Eric Swenson and other staff members helping us with the experiments, and thank Sarvani Vakkalanka for reading the draft.

## References

1. Flanagan, C., Godefroid, P.: Dynamic Partial-order Reduction for Model Checking Software. In Palsberg, J., Abadi, M., eds.: POPL, ACM (2005) 110–121
2. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: UUCS-07-008:Runtime Model Checking of Multithreaded C Programs. Technical report (2007) <http://www.cs.utah.edu/research/techreports/2007/ps/UUCS-07-008.ps>.
3. Godefroid, P.: Model Checking for Programming Languages using Verisoft. In: POPL. (1997) 174–186
4. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual . Addison-Wesley (2004)
5. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: an extensible and highly-modular software model checking framework. In: ESEC / SIGSOFT FSE. (2003) 267–276
6. 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, New York, NY, USA, ACM Press (2004) 1–13
7. 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. Volume 3114 of Lecture Notes in Computer Science., Springer (2004) 484–487

8. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag (1996)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
10. Snir, M., Otto, S.: MPI-The Complete Reference: The MPI Core. MIT Press, Cambridge, MA, USA (1998)
11. <http://www.mpi forum.org/docs/docs.html>
12. <http://www.lam mpi.org/>
13. <http://www.enderunix.org/aget/>
14. Stern, U., Dill, D.L.: Parallelizing the *Murhi* Verifier. In Grumberg, O., ed.: Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. Volume 1254 of Lecture Notes in Computer Science., Springer (1997) 256–278
15. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and Distributed Model Checking in Eddy. In: SPIN. (2006) 108–125
16. Sivaram, 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) (2005) 19–34
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. *PDMC* (1) (2004)
20. Palmer, R., Gopalakrishnan, G.: Partial Order Reduction Assisted Parallel Model Checking. *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, London, UK, Springer-Verlag (2002) 370