The Parallel PV Model-Checker

Robert Palmer and Ganesh Gopalakrishnan*
{rpalmer, ganesh} @cs.utah.edu,
http://www.cs.utah.edu/formal_verification/

University of Utah, School of Computing

Abstract. Parallel PV is based on the sequential PV model-checker. Sequential PV is an depth-first LTL-X model-checker for an enhanced subset of the Promela language. Parallel PV is a breadth-first safety-only model-checker. It capitalizes on PV’s two-phase partial-order reduction algorithm by carrying out partial order reduction steps with no communication, and performs state space distribution at global steps. This helps reduce the number of messages exchanged. Also, based on state ownership information, parallel PV reduces the number of states that are cached. This reduction is in addition to the selective state caching supported by sequential PV. We report encouraging preliminary experimental results drawn from the domain of ‘hardware’ protocols, as well as software models generated by the Bandera tool. Implementation details of parallel PV and setup information are also provided.

1 Introduction

Parallel processing can help speed-up model-checking by providing many CPUs that can work on the problem, higher aggregate memory capacity, as well as higher memory bandwidth. While the problem remains exponential, the overall performance of the model-checking algorithm can increase by significant factors. In this paper, we focus on enumerative model-checkers which are widely used to verify many classes of models, such as high-level cache coherence protocols and Java software models, in which models containing thousands of variables and involving global dependencies – such as processor IDs passed around in messages – are to be verified. Such models have never been shown to be capable of being efficiently represented or manipulated using BDDs. There are many previous attempts to parallelize enumerative model-checkers such as SPIN [16] and Murϕ [10]; see section 2.3 for a brief survey.

Among the scores of methods used to reduce state-explosion, a prominent method – partial order reduction – exploits the fact that whenever a set of independent transitions \{t_1, \ldots, t_n\} arise during exploration, they need not be explored in more than one linear order. Given the widely recognized importance of partial order reduction, efficient methods to realize it in a parallel context must be explored. This topic has, hitherto, not been studied extensively. We first motivate why partial order reduction and parallelism

* Supported by NSF Grants CCR-9987516 and CCR-0081406, and a gift from the Intel Corporation. The authors wish to thank Ratan Nalumasu who wrote the first version of PV, Hemanthkumar Sivaraj for his help with MPI, and Omid Saadati for developing the PV to Bandera link.
are two concepts that are closely related. We then briefly recap our partial order reduction algorithm “Twophase” implemented in our SPIN-like model-checker “PV,” (see [22, 28] for a full discussion). In this paper, we show that Twophase extends naturally to a parallel context – more readily than algorithms that use in-stack checking – and also leads to a very natural task-partitioning method. We show preliminary experimental results and describe the Parallel PV tool, interface, and operating environment.

2 Background

2.1 Partial order reduction and parallel processing

The earliest identifiable connection between parallel processing and the central idea behind partial order reduction (long before that term was coined) appears in Lipton’s work of 1975 on optimizing P/V programs [21] for the purpose of efficient reasoning. In that work, Lipton identifies left and right movers – actions that can be postponed without affecting the correctness of reasoning. Additionally, in the parallel compilation literature (e.g., [25] for an example), it has been observed that by identifying computations that “commute,” one can schedule these computations for parallel evaluation without interference.

If a partial order reduction algorithm involves a sequentializing step, it can significantly reduce available parallelism. Such a sequentializing step is present in current partial order reduction algorithms such as the one used in SPIN [6, 17]. This is the in-stack check used as a sufficient condition to eliminate the ignoring problem (a problem where some process may have a transition enabled everywhere in a cyclic execution path but is never executed along the path, thus causing missed, “ignored,” states – requirement C3 of [6, Page 150]). Using the ‘in-stack’ check, if the next state generated by moving a process on transition result in a state in the global DFS stack, an ample set (a commuting set of actions) cannot be formed out of the enabled transitions of . The in-stack check is involved in every step of SPIN’s partial order reduction algorithm. In a parallel setting, this could translate into considerable communication to locate the processes over which the stack is spread, perform the in-stack check, and resume the search. For this reason, in past work, a “worst case” assumption is used to gain some limited reductions. The assumption is that any successor state held outside the node is assumed to be currently in the search stack. This insures that the ignoring problem is dealt with, but may cause significant loss in reduction [18].

2.2 The Twophase Algorithm

We created Twophase after realizing that SPIN’s algorithm for partial order reduction can, due to the in-stack check based proviso, miss out on reduction opportunities. There are also added complications when realizing nested DFS based LTL-X checking [8], as explained in [15] – essentially requiring a ‘proviso-bit’ to convey information from the outer DFS to the inner DFS. The algorithm of Figure 1 can be suitably modified for

---

1 “The” in-stack check is a misnomer - this check is done differently for safety-preserving and liveness-preserving reductions. To simplify things, we ignore such variations.
Twophase() 
V := \phi; /* Hash table */ 
Phase–2(initial_state); 
end Twophase 

Phase–1(in) 
local old-s, s, list; 

s := in; /* Phase 1 */ 
list := \{s\}; 
for each process P do /* Phase 2: Classic DFS */ 
while (SAS\_1(P, s)) \^2 

old-s := s; 

s := t(old-s); 
if s \not\in list 

list := list + \{s\}; 
else 
break out of while loop 
end if 
end while; 
end for each; 
return(list,s); 
end Phase–1 

Phase–2(s) 
local list; 

(list,s) := Phase–1(s); 
if s \not\in list 

then 
V := V + all states in list + \{s\}; 
for each enabled 

transition t do 
list := list + \{s\}; 
if t(s) \not\in V then 
Phase–2(t(s)); 
else 
end if; 
end for each; 
end if; 
end if; 
end phase–2 

Fig. 1. The Sequential Twophase Algorithm

nested DFS based LTL-x model-checking and no proviso bit is required to convey information between outer and inner depth first searches. Full treatment of the sequential version of the algorithm is presented in [22].

The sequential algorithm alternates between a classical depth first search Phase-2, and a reduction step Phase-1. Phase-1 works as follows. Each process is considered in turn. For a given system state s an ample set is formed for process P_i. If the size of this set is one then the transition is executed resulting in a new state. This is repeated until no singleton ample set can be formed for that process. In Phase-2 all enabled moves of all processes are included in ampace(s) - no reduction is made in Phase-2. This is similar to a global state expansion in SPIN’s algorithm.

To deal with ignoring, references to states generated in Phase-1 are placed in a local list. We then check this list for a successor state that is a re-visitation. Note, we have changed C3 with respect to [6]. We accomplish C3 not by the C3\^ condition (the in-stack check) but rather by checking against a local list, shown in Figure 1.

\^2 We define SAS\_1 to be a set of transitions that form an ample set such that check\_C1 \& check\_C2 \& [enabled(P, s)] = 1 is true as described in [6, Pages 158-159] with condition check\_C3 satisfied separately by checking against the local list. (A Singleton Ample Set (SAS) with no check for ignoring (-1).)
2.3 Related Work

Work in parallel and distributed model-checking can be divided into the categories of explicit state representation based and symbolic state representation based. Only the explicit state survey is included in this paper. The remainder of this section can be found in [24].

Explicit State Model-Checking

Safety: Most work on distributed model-checking focus on safety model-checking. In [27], Stern and Dill report their study of parallelizing the Murϕ Verifier [10]. It originally ran on the Berkeley Network of Workstations (NOW) [1] using the Berkeley Active Messages library. It was subsequently ported to run on the IBM SP2 processor. Murϕ is a safety-only explicit state enumeration model-checker. In its parallel incarnation, whenever a state on the breadth-first search queue is expanded, a uniform hashing function is applied to each successor $s$ to determine its “owner” – the node that records the fact that $s$ has been visited, and pursues the expansion of $s$.

We [26] have recently ported parallel Murϕ from Active Messages to the popular MPI [23] library. Despite our relative inattention to performance for reasons of expediency, our speed-up figures for runs on the Testbed are very encouraging [5]. The largest model we ran far exceeds the sizes run by Stern and Dill.

In [19], a distributed implementation of the SPIN [14] model-checker, restricted to perform safety model-checking, and similar to [27], is described. Their first innovation is in state distribution. They exploit the structure of SPIN’s state representation and reduce (heuristically) the number of times a state is sent to other nodes. In addition, they employ look-ahead computation to avoid cases where a state is sent elsewhere, but very soon generates a successor that comes back to the original node. Their algorithm is also compatible with partial order reduction, although the reported results to date do not include the effects of this optimization. Their examples are standard ones such as ‘Bakery’ and ‘Philosophers’ running on up to four nodes on 300MHz machines with 64M memory. In [3], the algorithm of [27] is adapted to Uppal, a timed automaton model-checker, and applied to many realistic industrial-scale protocols, running on 24, 333MHz Sun Enterprise machines. Several scheduling policies are studied along with speed-up results.

In [11], parallel state space construction for labeled transition systems (LTSs) obtained from languages such as LOTOS is described. They use a cluster of 450MHz machines of up to 10 processors, each with 0.5GB of memory. They use the widely supported Socket library. They obtain speedups on most examples (industrial bus protocols) and perform analysis of the effects of communication buffers on overall performance.

In [20], issues relating to software model-checking and state representation are discussed. A large number of load distribution policies are discussed, and preliminary experimental results are reported. Many of these ideas are adaptations of techniques from their original work [19] to work well in the context of a software model such as Java.
LTL-x: Several works go beyond state space reachability and attempt the distributed model-checking of more expressive logics. In [2], the authors build on [19] and create a distributed LTL-x model-checker. The main drawback of their work is that the standard [8, 6] nested depth-first search algorithm employed to detect accepting (violating) Büchi automaton cycles tends to run sequentially in a distributed context, as the postorder enumeration of the “seed” states is still essential. They ameliorate the situation slightly by employing a data-structure called DepS that records how states were transported from processor to processor, and gathering the postorder numbering of the seed states in a distributed manner. However, the seed states still end up in a central queue, and are processed sequentially. A small degree of pipelining parallelism appears possible between the inner depth-first search on the “left half” of the search tree and the outer depth-first search on the “right half” of the search tree. Their paper reports feasibility (without actual examples) on a nine 366MHz Pentium cluster.

In [4], Büchi acceptance is reduced to detecting negative cycles (those that have a negative sum of edge weights) in a weighted directed graph. This reduction is achieved by attaching an edge-weight of ‘-1’ to all outgoing edges out of an accepting state, and a weight of ‘0’ to all other edges.

3 The Parallel PV Tool

3.1 The Parallel Twophase Algorithm

The parallel Twophase algorithm shown in Figure 2 works as follows. Basically, Phase-2 generates all successor states for a non-commuting state. Phase-1 is then applied to each of these successors. Only states generated for process $P_i$ since the last Phase-2 state are inserted into the list. Each thread maintains its own list. This is not possible in a parallel SPIN implementation because all these threads would still check with respect to the global DFS stack as in [18]. A uniform hashing function is applied to the resultant non-commuting state of Phase-1 and these states are then distributed by placing a state into the search queue of that state’s owner.

Phase-1 is the same as in the sequential version. Each node knows which transitions commute so given any state, the next global state can be created by any node.

3.2 Implementation

The Parallel PV tool is based on the sequential DFS based PV model-checker [22]. PV and parallel PV are written entirely in C. Parallel PV replaces the DFS based algorithm shown in Figure 1, used in the sequential version, with the distributed BFS based algorithm shown in Figure 2. It can be executed without partial order reduction to perform exhaustive breadth first search.

By ensuring Singleton Ample Set² global successor states it becomes un-necessary to save all of the states along the Phase-1 path. Heuristics can be applied to reduce the number of state lookup and insertion operations along the Phase-1 path. We refer to these heuristics as Selective State Caching.

² As in the sequential version discussed in section 2.2
In many cases, Phase-1 executes transitions that will cross the boundary of the state partition. Thus at least one, and as many as all of the states entered into the list are not owned by the node performing the computation. Those states not belonging to the computing node can, at the end of Phase-1, be deleted while preserving safety properties. These states are not sent to their owning nodes, they are simply dropped. This technique can be used with all of the selective state caching variants described in [12].

The list used to avoid ignoring, shown in Figures 1 and 2, and the Drop States optimization are both implemented as arrays of pointers into the hash table. A state is marked and placed in one of these lists when that state is created in Phase-1. If that state is revisited, a simple state look-up is necessary. At the end of Phase-1, those states that have been placed in the list are then set to normal hash table entries. Those in the Drop States array are removed from the hash table.

Figure 2 does not indicate termination conditions. The search is terminated in one of two instances. It is possible that the entire reduced graph has been generated. In this case we detect this condition using the Dijkstra-Scholten algorithm for stable condition detection in a diffused computation[9]. A description of our implementation of the Dijkstra-Scholten algorithm can be found in [24].

The other condition for termination is the detection of a safety violation. The node that finds the violation broadcasts a message to other nodes requesting them to wait in assistance of reconstruction of the error trail. When a state $s'$ is entered into the hash table, a memory reference to the hash table location containing the global predecessor
Fig. 3. Number of states generated.

$s$ of the new state $s'$ is also entered, along with a small constant amount of information about the state $s$, including the rank of the node that generated $s$. To construct the error trail, the hash table is traversed from the error state to the initial state, writing the information necessary to reconstruct an error trail to a file. If it is necessary to traverse multiple partitions of the state space the error trail that is contained locally is generated, packed and transmitted to the next node in the trail. This continues until the initial state is reached. We assume a shared file system for the error trail output file. An error trail can then be simulated in a sequential setting.

### 3.3 Experimental Results

Three of the models used as examples here are simple variants of the Leader Election Protocol as presented in [6, Pages 167-168]. This model does not include the never claim as we are considering safety properties only. The variation on this model is the number of processes that are participating in the protocol. We model-check for six, seven, and eight processes using the same model. The other model reported is generated by the Bandera tool. It is included in the Bandera tutorial distribution and can be recreated using their tutorial and is included in the examples directory of our tool distribution. We report the number of states generated, memory usage, message count,
and total runtime for each model. Figures 3, 4, 5, and 6 present the statistics for the respective reports.

A description of the figures is as follows. The number of network nodes column indicates how many physical machines are used in each model-checking computation. Without PO Redn indicates the statistics of a full breadth first graph search. Save All indicates a search that uses partial order reduction, but no selective state caching heuristic. Save Back Edge indicates a search that uses both partial order reduction and a selective state caching heuristic. Save States indicates that all states generated during Phase-1 are placed permanently into the hash table of the computing node, regardless of ownership. Drop States indicates that those states generated during Phase-1 which do not belong to the computing node are deleted at the end of Phase-1.

Figure entries showing “nc” indicate the computation was Not Complete. In each of these cases there was not enough memory to successfully generate the entire state graph. (We have not resorted to any hash-compaction techniques yet.) Otherwise the states and messages are integer values. The memory used is reported in megabytes. Time is reported in seconds.

Our tool release includes these examples and a script that will recreate our results. All verifications were performed using one Unix process per network node. The experiments were performed on a computational cluster of eight workstations. Each has a stock Red Hat Linux 7.1 operating system, 512MB memory and one 850MHz Intel Pen-
## Number of Network Nodes

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Without PO Redn</th>
<th>Save States</th>
<th>Drop States</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>538400</td>
<td>15829</td>
<td>18087</td>
</tr>
<tr>
<td></td>
<td>851143</td>
<td>25212</td>
<td>27162</td>
</tr>
<tr>
<td></td>
<td>945534</td>
<td>30600</td>
<td>31934</td>
</tr>
<tr>
<td>7</td>
<td>4281970</td>
<td>80016</td>
<td>91325</td>
</tr>
<tr>
<td></td>
<td>7446183</td>
<td>129721</td>
<td>140332</td>
</tr>
<tr>
<td></td>
<td>8452930</td>
<td>167161</td>
<td>174620</td>
</tr>
<tr>
<td>8</td>
<td>nc</td>
<td>nc</td>
<td>144377</td>
</tr>
<tr>
<td></td>
<td>65752911</td>
<td>nc</td>
<td>220052</td>
</tr>
<tr>
<td></td>
<td>77583833</td>
<td>830234</td>
<td>877647</td>
</tr>
<tr>
<td></td>
<td>243402</td>
<td>242062</td>
<td>344934</td>
</tr>
</tbody>
</table>

### PipeInt Model

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Without PO Redn</th>
<th>Save States</th>
<th>Drop States</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>nc</td>
<td>nc</td>
<td>144377</td>
</tr>
<tr>
<td>4</td>
<td>nc</td>
<td>nc</td>
<td>220052</td>
</tr>
<tr>
<td>8</td>
<td>243402</td>
<td>242062</td>
<td>344934</td>
</tr>
</tbody>
</table>

### Fig. 5. Number of messages passed.

Tium III CPU. Each Unix process was limited to 450MB of memory. The MPICH[13] implementation of the MPI standard is used for message passing between network nodes. Nodes are connected on a dedicated 100Mbps hub.

### 3.4 Interface

A TCL/TK graphical user interface is included in the Parallel PV distribution. This makes available sequential DFS, sequential BFS, and parallel BFS based searches. Models can be edited and error trails can also be simulated using the graphical user interface. Parallel PV can also be used from the command line. All output is routed via MPI to the master node. Please see [28] for command line interaction with Parallel PV.

The PipeInt model was generated using the Bandera[7] tool set. The subset of the Promela language supported by PV and Parallel PV is sufficiently expressive to model-check these models. Additionally, the Bandera user interface has been enhanced to access both the sequential and parallel versions of PV. Thus an error that is found using PV or Parallel PV on a Bandera generated model can be simulated within the Bandera framework.

### 4 Conclusions

We have presented a distributed partial order reduction based safety verification algorithm that is a variant of the sequential Twophase [22] algorithm.
The parallel Twophase algorithm has several advantages. It avoids the in-stack check allowing distributed partial order reduction. The algorithm can be used with BFS or DFS. It allows natural task partitioning and also reduces communication. It supports selective state caching in conjunction with a “Drop States” optimization.

The parallel Twophase algorithm is implemented in the parallel PV tool which supports selective state caching. The parallel PV tool can be used from the command line, with the included graphical user interface, or with the Bandera tool.

We have shown some preliminary experimental results using two models that generate large state spaces. The results are encouraging for their relative short run-times and efficient use of memory.

Our future work will include application-level check-pointing to be able to suspend and/or rerun crashed parallel model-checks, load balancing, hash compaction, and possibly symmetry reduction. PV and Parallel PV are available from our website [28].

References


