A Work-Efficient Distributed Algorithm for Reachability Analysis

Orna Grumberg, Tamir Heyman, and Assaf Schuster

Computer Science Department, Technion, Haifa, Israel

Abstract. This work presents a novel distributed, symbolic algorithm for reachability analysis that can effectively exploit, "as needed", a large number of machines working in parallel. The novelty of the algorithm is in its dynamic allocation and reallocation of processes to tasks and in its mechanism for recovery, from local state explosion. As a result, the algorithm is *work-efficient*: it utilizes only those resources that are actually needed. In addition, its high adaptability makes it suitable for exploiting the resources of very large and heterogeneous distributed, non-dedicated environments. Thus, it has the potential of verifying very large systems.

We implemented our algorithm in a tool called Division. Our preliminary experimental results show that the algorithm is indeed work-efficient. Although that the goal of this research is to check larger models, the results also indicate the potential to obtain high speedups, because communication overhead is very small.

1 Introduction

Reachability analysis is a central component of model checking. The verification of most temporal safety properties can be reduced to reachability analysis [3]. It is also an important preliminary stage for increasing the efficiency of symbolic model checking.

A significant amount of work is invested in increasing the capacity of model checking. Current model checking tools can verify systems with hundreds of variables using BDD-based methods [6,14] and falsify systems with thousands of variables using SAT-based methods [4]. A recent comparison [1] shows that each of the BDD-based and the SAT-based methods is superior to the other for certain types of problems. Nevertheless, it is generally agreed that the capability of model checking tools should be extended.

Typically, BDD-based model checking tools suffer from high space requirements while SAT-based tools suffer from high time requirements. The goal of this work is to overcome the space problem of BDD-based model checkers. A promising approach is to exploit the accumulative computation power and memory of a number of machines that work in parallel. Many environments can provide a large number of machines whose collective memory exceeds the memory size of any single machine.

Several solutions employing parallel computation have been suggested for dealing with the large memory requirements. Several papers suggest replacing the BDD with a parallelized data structure [19,15]. In [18], an explicit model checker that does not use symbolic methods is parallelized. Other papers suggest reducing the space requirements by partitioning the work to several tasks [8,17,16,7]. Although these methods might, in principle, be parallelized, they have not been. Rather, they use a single computer

W.A. Hunt, Jr. and F. Somenzi (Eds.): CAV 2003, LNCS 2725, pp. 54–66, 2003.

to sequentially handle one task at a time, while the other tasks are kept in an external memory.

The work that most resembles this one is distributed symbolic (BDD-based) reachability analysis, suggested in [13]. It is based on an initial partitioning of the state space among all processes in the network and on a continuous load balancing that keeps the workload among the processes relatively balanced.

The success of this approach strongly depends on an effective slicing procedure. Slicing is said to be *effective* if it avoids duplication and if it results in evenly split, smaller BDDs. Duplication is the amount of sharing in a BDD structure that is lost due to partitioning. The notion of duplication and its implications are discussed in detail in [13] and will not be addressed in this paper. Finding such an effective slicing is a nontrivial problem [8,17,16,13].

In [13], each process iteratively applies *image computation* to its set of *new states* N, *exchanging* non-owned states with other processes, and collecting owned states in its set of *reachable states* R. Load balance is available at the end of each iteration. It balances the sizes of the sets of reachable states in the different processes.

This algorithm has several drawbacks. First, it immediately splits to as many slices as the number of processes in the network and does not release them until it terminates. Thus, it occupies all processes in the network all the time, regardless of actual need. Second, slicing is often inefficient because it partitions a relatively small BDD into many small slices. The more processes in the system, the less efficient the slicing is, which renders the algorithm non-scalable. Third, it does not provide a means to overcome the memory overflow that occurs during an image computation or an exchange operation. It is well known that intermediate results of image computation may be orders of magnitude larger than its initial and resulting BDDs. Similarly, during an exchange operation the memory of a process may overflow as a result of the BDDs it receives. Unfortunately, even when there are under-utilized processes, there is no way to recover from such overflows since load balancing is available only at the end of iterations. Finally, balancing is applied only to the sets R. However, the size of intermediate results in image computation depends on N and is often much larger than R. Thus, load balancing does not handle the dominant factors of memory overflow.

In this paper we suggest a new algorithm which overcomes the drawbacks of the previous one. The algorithm uses two types of processes: coordinators and workers. Each worker can be either active or free. The algorithm works iteratively. It is initialized with one active worker that runs a symbolic reachability algorithm, starting from the set of initial states. During its run, workers are allocated and freed, as needed. At any iteration, each of the active workers applies image computation and then sends those states it does not own to their owners. Therefore, we will refer to these as a worker's non-owned states.

Since memory overflow is likely to occur during the image computation and the exchange operation, our algorithm is designed to overcome these problems. For image computation we use a new BDD operation that resembles ordinary image computation, except that it stops if the intermediate results create memory overflow. In this case, the BDD representing the intermediate results is partitioned into k slices. One slice is left with the overflowed worker and the others are distributed to $k-1$ free colleagues. k is

called the *splitting degree*. It is a parameter of the new algorithm and is usually small (often $k = 2$). Since the BDD is huge, the slicing is very effective. Once the BDD is split, each worker resumes the computation of (its part of) the image *from the point at which it stopped*. However, each worker now works on a smaller BDD. If state explosion occurs during the exchange procedure, then $R \cup N$ is split for sharing with $k-1$ free colleagues. Exchanging of non-owned states then proceeds according to the new ownership.

The new algorithm enables the slicing procedure to split according to R, N, or intermediate results, depending on what caused the memory overflow. Since the chosen BDDs are large, slicing is always very effective. Furthermore, slicing affects the performance of the new algorithm much less than it affects the one from [13] because, in the case of a high work load at one of the co-workers, the new algorithm can simply split again. These features provide the new algorithm with strength and flexibility, and allow to reduce the slicing complexity.

It may also happen that the memory requirement of a worker decreased below a certain threshold (the size of a BDD decrease even if it represents a larger set of states). In that case, several workers with small memory requirements are combined and all but one become free.

It is important to note that splitting occurs only "as needed", when a worker actually has a memory overflow. Thus the algorithm is *work-efficient*: it exploits to the maximum the resources of the active workers before allocating additional ones. This efficiency allows, for a given network, computing reachability of (i.e., verifying) larger systems. Moreover, our algorithm can effectively exploit any network size. Thus, the larger the available network, the larger the systems that can be verified.

We have implemented our algorithm in Division, a generic platform for the study of distributed symbolic model checking [12]. Division requires a model checker as an external module. We used NuSMV [10] for this purpose: a re-implementation of McMillan's SMV [14].

Unfortunately, using NuSMV implied that we could not directly compare the results of [13] to the results of this work. The experiments in [13] were conducted using the high-performance RuleBase [2] model checker that was not available to us in this work. The two tools are not comparable as many of the RuleBase optimizations are not implemented in NuSMV.

Our parallel testbed included 25 dual process PC machines. The nodes communicated via a fast Ethernet connection. We conducted our experiments using four of the largest circuits from the ISCAS89 and addendum'93 benchmarks.

With our distributed algorithm, we can compute larger models than we can compute with a single machine using the same model checker. In all the examples the new algorithm using the less sophisticated model checker (NuSMV) would be sufficient to compute the same models and reach the same BFS step as in [13].

The rest of the paper is organized as follows. In Section 2 we detail the algorithm that the workers follow. Section 3 describes the operation of the coordinator processes. Section 4 explains the enhanced slicing employed when overflow occurs during image computation. Preliminary experimental results are given in Section 5. We summarize our conclusions and expectations in Section 6.

2 The Worker Algorithm

Our distributed algorithm uses a set of window functions [8,17] to partition the state space among all workers in the network. Each worker *owns* the states in one of the window functions and computes the reachable states in this window.

Figure 1 presents a high-level view of the workers algorithm. Essentially, the algorithm performs a reachability task. The algorithm starts with only one worker that owns the entire state space, while the rest of the workers are free. If a worker runs out of memory (*memory overflow*), it distributes parts of its work among a few free workers.

The worker repeatedly computes images and sends its non-owned states to their owners until termination is detected (namely, a fixed-point is reached). While iterating, if the workload of the worker becomes too small, it participates in a collect_small procedure.

There are two points at which a worker may run out of memory (*memory overflow*): during the image computation and during the exchange of non-owned states. Upon memory overflow, the worker splits the states it owns into two parts: one that will be processed at the current worker and another to be processed at another worker. As a result, the states belonging to the new worker become non-owned and are sent out to the new worker.

```
function reach_task()
 1 Loop until termination()
 2  Image() if overflow, split and use new workers
 3  Exchange() if overflow, split and use new workers
 4  Collect_small()
 5 return owned states
```

Fig. 1. High–level pseudo–code for a worker

Let us describe the algorithm for the workers in greater detail, as shown in Figure 2. The reachability task includes a set of reachable states R and a set of reachable states that are not yet developed, N. For brevity, we omit in this section the worker subscript id from R_{id} and N_{id}, as well as the window function w_{id}. The set R is included in a window function w. The sets R and N, as well as the window function w, may change during the algorithm's execution.

In the Image procedure, the worker computes the set of states that can be reached in one step from N and stores the result in a new N. However, if during image computation the memory overflows, the worker splits w and updates R and N accordingly, as described below.

In the Exchange procedure the worker uses w to define the part of the state space it "owns". It sends out the non-owned states ($N \setminus w$) to their owners and receives its owned states that were found by other workers.

Finally, if only a small amount of work remains, the worker joins the Collect_small procedure. The collect_small procedure adds up the tasks of several workers, each of which has only a small amount of work. This is done by joining together the parts of the state space owned by those workers and assigning the unified ownership to one of them. The others become "free" ($w = \emptyset$) and return to the pool of free workers.

```
function reach_task(R, w, N, method)
  if method = "exchange''
    goto Exchange_loop(R, w, N)
  Loop_forever
    Image(R, w, N)
    Exchange(R, w, N)
    if (termination()) return R
    N=N \ R
    R=R ∪ N
    Collect_small(R, w, N)
    if(w = ∅)
      send <to_pool, id> to ex_coor
      return to pool

procedure Exchange(R, w, N)
  <{w_i}>=receive from ex_coor
  send <{p_i}> to ex_coor
  Exchange_loop(R, w, N)

procedure Collect_Small(R, w, N)
  While(| N | + | R |< Min)
    send <(| N |,| R |)> to small_coor
    <action>=receive from small_coor
    if acction =< End > return
    if action =< Non_owner, p_clg >
      send <R, w, N> to p_clg
      R=w=N=∅
      <"release''>=receive from ex_coor
      return
    if action =< Owner, p_clg >
      <R', w', N'>=receive from p_clg
      R=R ∪ R'; w=w ∪ w'; N=N ∪ N'
      send <w, p_id, p_clg> to ex_coor

procedure Image(R, w, N)
  N = boundedImage(N, Max, Failed)
  While(Failed)
    Split(R, w, N, "Image'')
    N = boundedImage(N, Max, Failed)

procedure Exchange_loop(R, w, N)
  loop until < done > received from ex_coor
    <p_clg, w_clg>=receive from ex_coor
    send <N ∩ w_clg> to p_clg
    <N'>=receive from p_clg
    overflow = N' is too large
    send <overflow> to p_clg
    send <<status>=receive from p_clg> to ex_coor
    if (overflow) Split(R, w, N, "Exchange'')
    else
      N=N ∪ N'
      send <"done''> to ex_coor

procedure Split(R, w, N, method)
  <{p_2 ... p_k}>=receive from pool_mgr
  if(method = "exchange''
    {W'_i}={Nw'_i}=Slice(R ∪ N, k)
  else
    if(| R | big enough
      {W'_i}=Slice(R, k)
    else
      {W'_i}=∅, i ∈ 2 ... k; W'_1=w
    {Nw'_i}=Slice(N, k)
  ∀i ∈ 2 ... k:
  send <R ∩ W'_i, w ∩ W'_i, N ∩ Nw'_i, method> to p_i
  R=R ∩ W'_1; w=w ∩ W'_1, N=N ∩ Nw'_1
  send <{i ∈ 1..k | w ∩ W'_i}> to ex_coor
```

Fig. 2. Pseudo–code for a worker in the distributed reachability computation

In the Image procedure, the image is computed using a new BDD operation. The Image procedure is using a new BDD operation, `boundedImage`$(N, Max, Failed)$. This operation is different from traditional image computation in that it stops the local computation in case of a memory overflow (i.e., the number of BDD nodes exceeds Max). Upon overflow, the `Image` procedure calls the `Split` procedure, which repartitions the ownership of the worker and updates R, w, N accordingly.

In the Exchange procedure, the worker first requests and receives from the `ex_coor` process the up-to-date list of window functions owned by the other workers. The worker then sends the `ex_coor` the list of workers to whom it wishes to send non-owned states. Then, in the Exchange_loop procedure, the `ex_coor` schedules the worker for state exchange with other workers.

In the Exchange_loop procedure the worker is scheduled by the `ex_coor` to exchange non-owned states with colleagues that either found states owned by the worker or own states that were found by the worker. The worker continues to receive exchange commands from the `ex_coor` until it gets a $< done >$ command when there are no more pending exchanges. If the worker's memory overflows during the exchange procedure and the worker fails to receive more owned states, it notifies the `ex_coor` and calls the Split procedure to reduce its ownership.

If the worker in the Collect_Small procedure has enough work, it exits immediately. Otherwise, the worker notifies the `small_coor` about the sizes of its N and R sets. In reply, it receives one of three commands and proceeds accordingly: $< End >$ commands it to exit the Collect_Small; $< Non\_owner, p_{clg} >$ commands it to deliver its ownership and owned states to a colleague worker p_{clg}, waits for the `ex_coor` to acknowledge the update of its window functions (performed by p_{clg}), and then return to the pool; $< Owner, p_{clg} >$ commands it to take over the ownership and states of another worker p_{clg} and report the new ownership to the `ex_coor`.

The Split procedure starts by requesting from the `pool_mgr` $k-1$ new workers (which, together with the overflowed worker, makes it a k-way split). If Split is called from Exchange, then the window function w of the overflowed worker is split into k new window functions $\{W_i'\}$, such that $\{W_i' \cap R\}$ have approximately the same sizes. If Split is called from Image, then two sets of k new window functions are computed, as follows. If R is big enough, then, as in the previous case, a set of window functions $\{W_i'\}$ is computed such that the sizes of $\{W_i' \cap R\}$ are approximately the same. Otherwise, if R is too small, one of the workers gets all of w while the others remain empty. In any case, the ith new window function W_i' determines, for the ith worker, its new window w_i. In addition, w is split again into another set of window functions $\{Nw_i'\}$, this time making $\{Nw_i' \cap N\}$ equal in size. After the new window functions are computed, the overflowed worker sends the corresponding states to its new colleagues.

The reason for computing two different partitions when Image overflows is that $\{Nw_i'\}$ attempts to balance the current image computation, while $\{W_i'\}$ attempts to balance the memory requirement in the full reachability process. In section 4 we further discuss the optimization of the partitioning process.

In the case that R is "too small" or even empty, the new colleagues are simply helping the overflowed worker with a single image computation. Once the image is computed, all states produced by the helpers are non-owned and will be sent to other workers that own them. From our experience, this case is not uncommon; it occurs when the peak memory requirement during image computation is much larger than R.

As mentioned in the introduction, an important advantage of our algorithm over previous works is that it calls the Slice function only when the memory overflows, and with k much smaller than the total number of workers. This makes slicing much more effective in producing even splits of the input sets of states.

We remark that the Slice procedure itself is no different from the slicing mechanisms described in [13]. Thus, in this paper, we use it as a black box and focus on the distributed algorithm itself.

3 The Coordinators

The `ex_coor` coordinator holds the current set of window functions and coordinates the exchange of non-owned states between workers. In order to hold a consistent view of the current set of window functions, the `ex_coor` is notified immediately on every split or merge of windows. It takes the following actions on incoming event notifications:

- When a worker requests an exchange it first *registers* at the `ex_coor`. The `ex_coor` replies with the up-to-date set of window functions and receives in return the set of colleagues the worker wants to communicate with.
- When a worker splits, the `ex_coor` updates the set of window functions. If the splitting worker is already registered for exchange states, the `ex_coor` notifies all the workers that have asked to send it states that they should send the states to the new set of workers, according to the new set of window functions.
- When workers perform *Collect_Small* and join their ownerships, the `ex_coor` updates the set of window functions. If there are workers registered for exchanging states with the joining workers, the `ex_coor` redirects them to the new owner. When the `ex_coor` complete to update the set of window functions it sends $< release >$ command to the worker that become non-owner.
- When a worker completes the exchange of non-owned states with another worker, the coordinator marks it as available for another round of exchange states.
- When a worker asks to re-launch an exchange because the colleague overflowed and had to split while they were interacting, the `ex_coor` adds this request to the list of exchange requests.

The `small_coor` coordinator collaborates with `ex_coor` to prevent deadlocks and to collect as many under-utilized workers as possible. The `small_coor` receives registration requests from workers that completed the exchange phase and are left with a very low load (very small $R \cup N$). The first registrant is blocked until more of them arrive. When there are several registrants the `small_coor` instructs them to merge.

The `pool_mgr` coordinator keeps track of free workers. During initialization, the `pool_mgr` marks all but one worker as free. When a worker invokes the Split procedure, it sends a request to the `pool_mgr` for $k-1$ free workers (where k is the splitting degree). The `pool_mgr` replies with a list of $k-1$ worker ids and removes them from the free list. Throughout the algorithm, when a worker becomes free, i.e., when its ownership becomes empty, it returns to the `pool_mgr` and is added to the free list for later assignments.

If at the time free workers are requested from the `pool_mgr`, the free list happens to be empty or is shorter than $k-1$, the `pool_mgr` announces a "worker overflow" and stops the execution globally.

4 Optimizing the Splitting in Image Computation Overflow

Our algorithm is based on the assumption that in case of a memory overflow during image computation, splitting the window of the overflowing worker enables the completion of the computation using more workers. The current splitting method strives to effectively slice the set N on which the image is computed (see [13]). However, since the computation is symbolic, reducing the size of the subsets does not guarantee a corresponding reduction in the image size. Furthermore, it guarantees even less for the size of the intermediate results that commonly dictate the *peak memory requirement* during the image computation. Our experience shows that even when the size of the parts is the same, the size of the peaks may differ greatly. Thus, while one of the slices may have no problem in completing the image computation, another may overflow again.

Another problem with the current splitting method is the time penalty for memory overflow. When the image computation overflows and the set N is split, the work that was invested in the current image step is lost, and the work is repeated all over again. In fact, in the case of several subsequent memory overflows, the work is repeated again and again. Notice that the ratio between the peak memory requirement in the image computation and the set N is commonly two or three orders of magnitude. Thus, memory overflow commonly occurs when a big part of the image computation has already been done locally, and all this work must be repeated. Since the image computation takes most of the time in our distributed algorithm, the repeated work slows down the algorithm substantially.

The solution to the above two problems is simply to split the intermediate results and not the set N. After the splitting, the parts of the intermediate results are distributed among the new workers, so computing the image for each of them continues from the point of the overflow. In this way there is no time penalty for overflow except for the splitting computation (which is of somewhat higher complexity than before). Of course, communicating the intermediate results requires a much higher bandwidth. However, network bandwidth and communication delay turn out to be minor factors as compared with the time spent in the image computation, even with our standard fast Ethernet.

In terms of memory requirements this solution has two advantages. First, splitting is applied on a much larger set, which makes it a lot easier to split effectively. Second, splitting is applied much closer to the peak, which makes it more efficient in reducing the peak memory requirements of the resulting parts.

The optimized algorithm uses a partitioned transition relation. The full transition relation is a conjunction of all partitions:

$$T(V,V') = T_1(V,V') \wedge T_2(V,V') \wedge \ldots \wedge T_n(V,V'),$$

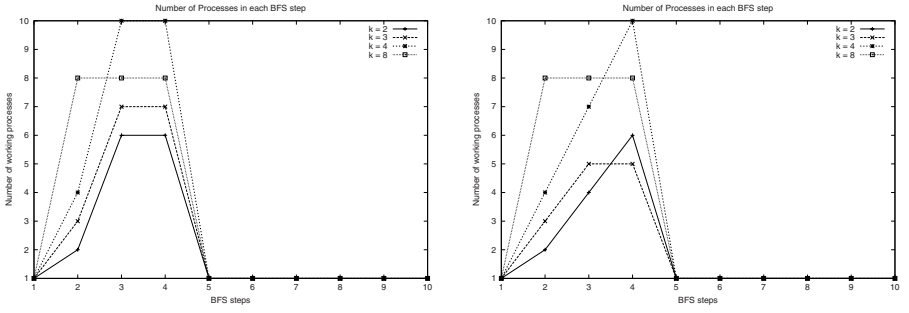
and an image computation thus becomes

$$S'(V') = \exists V[S(V) \wedge T_1(V,V') \wedge T_2(V,V') \wedge \ldots \wedge T_n(V,V')].$$

The technique for image computation suggested by Burch et al. [5] is to iteratively conjunct-in the partitions, and to quantify-out variables as soon as further steps do not depend on them. The order in which $T_i(V,V')$ are conjuncted is very important to the efficiency of this technique [11]. For the sake of simplicity, let us assume the order is given such that T_1 is the first to conjunct, then T_2, until T_n. Let D_i be the set of variables on which $T_i(V,V')$ depend. Let $E_i = D_i - \bigcup_{m=i+1}^{n} D_m$. A symbolic step is carried out iteratively as follows:

$$\begin{aligned} S_1(V,V') &= \exists E_1[T_1(V,V') \wedge S(V)] \\ S_2(V,V') &= \exists E_2[T_2(V,V') \wedge S_1(V,V')] \\ &\vdots \\ S'(V') &= \exists E_n[T_n(V,V') \wedge S_{n-1}(V,V')]. \end{aligned}$$

If overflow occurs during step $0 < j < n$, we look for a set of window functions $w_1 \ldots w_k$ such that $\bigvee_{i=1}^{k} w_i = 1$. The ith worker will get $S_j(V,V') \wedge w_i$. We can now



(a) **with no optimization** (b) **optimized**

Fig. 3. Number of workers required in each BFS step of s1269. Overflow is declared for worker memory utilization exceeding $6M$ BDD nodes.

rewrite the $j+1$ step as follows:

$$S_{j+1}(V,V') = \exists E_{j+1}[\bigvee_{i=1}^{k} T_{j+1}(V,V') \wedge S_j(V,V') \wedge w_i].$$

Since the existential quantification is distributive over disjunction, the above expression is equal to:

$$S_{j+1}(V,V') = \bigvee_{i=1}^{k} \exists E_{j+1}[T_{j+1}(V,V') \wedge S_j(V,V') \wedge w_i].$$

Therefore, the disjunction of the $j+1$th steps assigned to each worker is equal to the step done without splitting.

The algorithm uses a new BDD operation: $\texttt{BoundInc}(S(V,V'), \{T_i(V,V')\}, Max)$, where $S(V,V')$ is the function from which the image computation continues, $\{T_i(V,V')\}$ is the set of partitions that were not yet used, and Max is the threshold for overflow during image computation. In the beginning of the algorithm, $S(V,V')$ is the set of states whose image is to be computed in this step, and $\{T_i(V,V')\}$ are all the partitions. If the algorithm overflows, BoundInc returns in $S(V,V')$ the last intermediate result computed prior to the overflow, and in $\{T_i(V,V')\}$ the rest of the partitions that have not been used. If the algorithm completes the image computation, $S(V,V')$ equals the next set of states, and an empty list of partitions is returned.

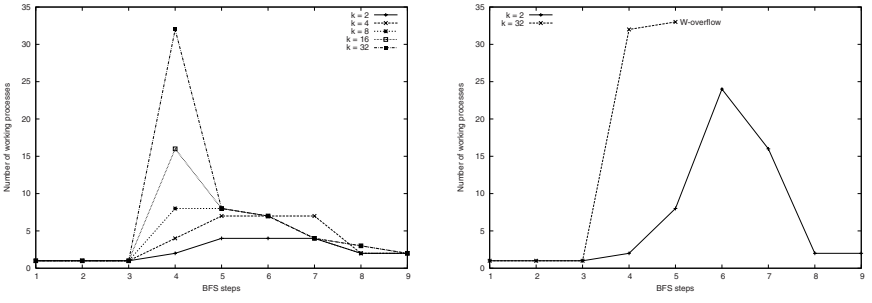
Figure 3 illustrates the benefit of using the optimized algorithm for the circuit s1269. Figure 3(a) provides the number of workers required in each step for various splitting degrees. For instance, for splitting degree $k = 2$, six workers are needed in order to complete Step 3. Figure 3(b) shows that this step requires only four workers when using the optimization described in this section. In all other steps and splitting degrees the number of workers required by the optimized algorithm was always less than or equal to the non-optimized version.

5 Experimental Results

Our parallel testbed included 25 PC machines, each consisting of dual 1.7GHz Pentium 4 processors with 1GB memory. The communication between the nodes consisted of a fast Ethernet. We conducted our experiments using four of the largest circuits from the ISCAS89 benchmarks. The characteristics of the circuits are given in Figure 4.

Circuit	#vars	peak		fixed point	
		size	step	time	steps
prolog	117	2.6M	5	2,431	9
s1269	55	16M	5	5,053	10
s3330	172	16M >	Ov(3)	-	Ov(3)
s1423	88	16M >	Ov(13)	-	Ov(13)

Fig. 4. Benchmark suite characteristics. The peak is the maximal memory requirement at any point during an image step. Fixed point is the number of image steps and the time (seconds) it takes to get to the fixed point. Ov(m) denotes memory overflow at step m.



(a) **prolog** Max = 1M nodes allocated

(b) **S3330** Max = 7M nodes allocated

Fig. 5. Number of workers in each BFS step. Overflow is declared for worker memory utilization exceeding Max BDD nodes. **W-overflow** halts the computation when more than 60 workers are required.

5.1 Number of Workers for Reachability Analysis

Since the memory required by each worker is bounded by a given threshold, we only care about the number of active workers at each iteration. Figures 5(a), 5(b), 6 and 3 give the number of workers required at any step of the analysis, and the threshold that was used. The figures prove that using a lower splitting degree is more work efficient, namely, the computation can be carried using fewer resources with a lower splitting

degree. This is explained by the fact that when the splitting degree is high, new workers may join in even when the computation can do without them: the computation proceeds with workers that may be under-utilized (but not sufficiently so to be collected by the Collect_Small process).

In steps $1, 2, 3$ in Figure 5(a) only one worker is needed. In step 4, this worker needs help in order to complete the image computation. Dividing the work into two is sufficient, but when the splitting degree is higher we occupy more workers without actually needing them. In steps 8 and 9 the image computation requires less memory and the size of the sets R and N requires less workers. Indeed the number of workers decreases as a result of the Collect_Small procedure.

Figure 5(b) shows that the distributed system can complete the reachability analysis, whereas a single machine overflows.

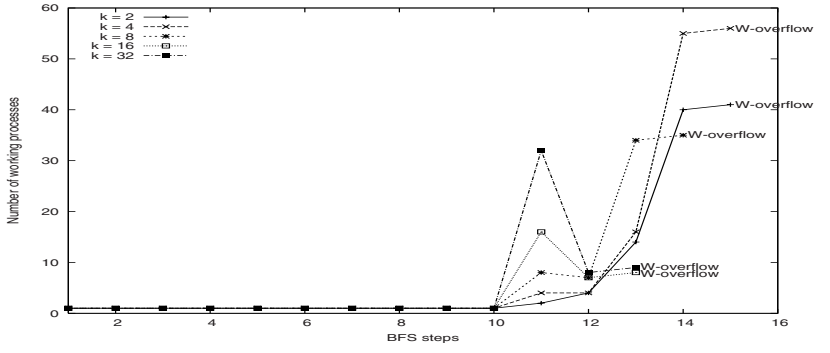


Fig. 6. Number of workers in each BFS step of s1423. Overflow is declared for worker memory utilization exceeding $6M$ BDD nodes. **W-overflow** is where more than 60 workers required.

5.2 Timing and Communication

We have performed some initial studies regarding the timing and breakdown of running our distributed system. The results show several very clear findings and trends that we now briefly discuss.

First, communication overhead is minor. Our experiments show that the time to reach local overflow is much higher than the time required to dump the contents of memory into the network. Although this finding should be re-evaluated when our system is further optimized (see below), it seems strong enough to sustain. If the system scales up to include more workers, the communication time might grow as a result of more non-owned states that are found. Nevertheless, we expect the computation time to remain dominant because the communication volume for every worker at any split or exchange operation is bounded by the size of the RAM of that worker. We remark that technology trends predict much faster commodity networks (even when compared to the larger expected RAMs) very soon.

Second, splitting is a major element in the computation. It can count up to dozens of percentage points of the computation time, and these numbers grow rapidly when the system scales up. Others have previously addressed the splitting complexity [9]; we intend to speed up the splitting module in our future work.

Third, the fact that the reachability computation is synchronized in a step-by-step fashion has a major impact on the computation time. The problem is that at the end of a step all computing workers wait for the slowest one, who may be slicing and re-slicing several times during the step (remember that slicing is slow!). However, despite its synchronized operation, the new algorithm is very flexible. We believe that it can become the basis for a truly non-synchronized variant.

One interesting phenomena that was not masked by the inefficiencies above is a tradeoff between being work efficient and obtaining speedups. While the best hardware utilization is achieved with splitting degree of 2, the fastest computation times are obtained using somewhat higher splitting degrees (e.g., $k = 8$ for Prolog). Thus, a splitting degree higher that 2 may become instrumental in cases that the speedup is more important than RAM utilization.

6 Conclusions and Expectations

This paper presents a new distributed algorithm for symbolic reachability analysis that improves significantly on previous works. Its adaptability to any network size and its high utilization of network resources make it suitable for solving very large verification problems.

The experimental environment that is used to evaluate our new algorithm currently consists of NuSMV and the newly introduced Division system. Division is a new platform for distributed symbolic model checking research, featuring a high-level generic interface to "external" model checkers. Eventually, we intend to release Division source code through the Division web-site [12].

At the point that the final version of this paper is due, Division is in the final stages of interfacing with Intel's high-performance model checker – Forest. We thus expect our results to improve substantially and to become more accurate in the near future. We refer the interested reader to the Division web-site for up-to-date result reports and for the full and final version of this paper.

References

1. N. Amla, R. Kurshan, K. McMillan, and Medel R. K. Experimental Analysis of Different Techniques for Bounded Model Checking. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), LNCS*, Warsaw, Poland, 2003.
2. I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An Industry-Oriented Formal Verification Tool. In *33rd Design Automation Conf.*, pages 655–660, 1996.
3. I. Beer, S. Ben-David, and A. Landver. On-the-Fly Model Checking of RCTL Formulas. In *Proc. of the 10th Int. Conf. on Computer Aided Verification, LNCS 818*, 1998.
4. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking Without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems,* 5^{th} *Int. Conference, TACAS'99, LNCS 1579*, 1999.

5. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proc. of the 1991 Int. Conference on Very Large Scale Integration*, August 1991.
6. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–171, June 1992.
7. G. Cabodi. Meta-BDDs: A Decomposed Representation for Layered Symbolic Manipulation of Boolean Functions. In *Proc. of the 13th Int. Conf. on Computer Aided Verification*, 2001.
8. G. Cabodi, P. Camurati, and S. Quer. Improved Reachability Analysis of Large FSM. In *Proc. of the IEEE Int. Conf. on Computer Aided Design*, pages 354–360. IEEE Computer Society Press, June 1996.
9. G. Cabodi, P. Camurati, and S. Quer. Improving the Efficient of BDD-Bsaed Operators by Means of Partitioning. *IEEE Transactions on Computer-Aided Design*, May 1999.
10. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proc. of the 7th Int. Conf. on Computer-Aided Verification (CAV'99), LNCS 1633*, pages 495–499, Trento, Italy, 1999.
11. D. Geist and I. Beer. Efficient Model Checking by Automated Ordering of Transition Relation Partitions. In *Proc. of the Sixth Int. Conf. on Computer Aided Verification, LNCS 818*, pages 299–310, 1994.
12. O. Grumberg, A. Heyman, T. Heyman, and A. Schuster. Division System: A General Platform for Distributed Symbolic Model Checking Research, 2003. http://www.cs.technion.ac.il/Labs/dsl/projects/division_web/division.htm.
13. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. *Formal Methods in System Design*, 21(2):317–338, November 2002.
14. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
15. K. Milvang-Jensen and A. J. Hu. BDDNOW: A Parallel BDD Package. In *Second Int. Conference on Formal methods in Computer-Aided Design (FMCAD '98), LNCS*, Palo Alto, California, USA, November 1998.
16. A. Narayan, A. Isles, J. Jain, R. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *Proc. of the IEEE Int. Conf. on Computer Aided Design*, pages 388–393. IEEE Computer Society Press, June 1997.
17. A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partitioned-ROBDDs. In *Proc. of the IEEE Int. Conf. on Computer Aided Design*, pages 547–554. IEEE Computer Society Press, June 1996.
18. Ulrich Stern and David L. Dill. Parallelizing the Murphy Verifier. In *Proc. of the 9th Int. Conf. on Computer Aided Verification, LNCS 1254*, pages 256–267, 1997.
19. T. Stornetta and F. Brewer. Implementation of an Efficient Parallel BDD Package. In *33rd Design Automation Conf.* IEEE Computer Society Press, 1996.