Achieving Speedups in Distributed Symbolic Reachability Analysis through Asynchronous Computation

Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster

Computer Science Department, Technion, Haifa, Israel

Contact author: Nili Ifergan
Computer Science Department, Technion, Haifa 32000, Israel
inili@cs.technion.ac.il
Phone: 972-4-8294929

Abstract. This paper presents a novel BDD-based distributed algorithm for reachability analysis which is completely asynchronous. Previous BDD-based distributed schemes are synchronous: they consist of interleaved rounds of computation and communication, in which the fastest machine (or one which is lightly loaded) must wait for the slowest one at the end of each round.
We make two major contributions. First, the algorithm performs image computation and message transfer concurrently, employing non-blocking protocols in several layers of the communication and the computation infrastructures. As a result, regardless of the scale and type of the underlying platform, the maximal amount of resources can be utilized efficiently. Second, the algorithm incorporates an adaptive mechanism which splits the workload, taking into account the availability of free computational power. In this way, the computation can progress more quickly because, when more CPUs are available to join the computation, less work is assigned to each of them. Less load implies additional important benefits, such as better locality of reference, less overhead in compaction activities (such as reorder), and faster and better workload splitting.
We implemented the new approach by extending a symbolic model checker from Intel. The effectiveness of the resulting scheme is demonstrated on a number of large industrial designs as well as public benchmark circuits, all known to be hard for reachability analysis. Our results show that the asynchronous algorithm enables efficient utilization of higher levels of parallelism. High speedups are reported, up to an order of magnitude, for computing reachability for models with higher memory requirements than was previously possible.

1 Introduction

This work presents a novel BDD-based asynchronous distributed algorithm for reachability analysis. Our research focuses on obtaining high speedups while computing reachability for models with high memory requirements. We achieve this goal by designing an asynchronous algorithm which incorporates mechanisms to increase process utilization. The effectiveness of the algorithm is demonstrated on a number of large circuits, which show significant performance improvement.

Reachability analysis is a main component of model checking [10]. Most temporal safety properties can easily be checked by reachability analysis [2]. Furthermore, liveness property checking can be efficiently translated into safety property checking [3].

Despite recent improvements in model checking techniques, the so-called state explosion problem remains their main obstacle. In the case of industrial-scale models, time becomes a crucial issue as well. Existing BDD-based algorithms are typically limited by memory resources, while SAT-based algorithms are limited by time resources. Despite recent attempts to use SAT-based algorithms for full verification (pure SAT in [23, 8, 21, 16] and SAT with BDDs in [19, 17, 18]), it is still widely acknowledged that the strength of SAT-based algorithms lies primarily in falsification, while BDD-based model checking continues to be the de facto standard for verifying properties (see surveys in [28] and [4]). The goal of this work is verification of large systems. Therefore, we based our techniques on BDDs.

The use of distributed processing to increase the speedup and capacity of model checking has recently begun to generate interest [5, 29, 22, 1, 27, 20, 15, 30, 24]. Distributed techniques that achieve these goals do so by exploiting the cumulative computational power and memory of a cluster of computers. In general, distributed model checking algorithms can be classified into two categories: explicit state representation based [29, 22, 1, 27] and symbolic (BDD-based) state representation based [20, 15]. Explicit algorithms use the fact that each state is manipulated separately in an attempt to divide the work evenly among processes; given a state, a hash-function identifies the process to which the state was assigned. The use of hash-functions is not applicable in symbolic algorithms which manipulate *sets* of states, represented as BDDs. In contrast to sets of explicit states, there is no direct correlation between the size of a BDD and the number of states it represents. Instead, the workload can be balanced by partitioning a BDD into two smaller BDDs (each representing a subset of the states) which are subsequently given to two different processes.

The symbolic work-efficient distributed synchronous algorithm presented in [15] is the algorithm that is closest to ours. In [15], as well as in our algorithm, processes (called workers) join and leave the computation dynamically. Each worker *owns* a part of the state space and is responsible for finding the reachable states in it. A worker splits its workload when its memory overflows, in which case it passes some of its owned states to a free worker.

Unlike the algorithm proposed in this work, the one in [15] works in synchronized iterations. At any iteration, each of the workers applies image computation and then waits for the others to complete the current iteration. Only then do all workers send the non-owned states discovered by them to their corresponding owners.

The method in [15] has several drawbacks. First, the synchronized iterations result in unnecessary and sometimes lengthy idle time for “fast” processes. Second, the synchronization phase is time-consuming, especially when the number of processes is high. Consequently, processes split as infrequently as possible in an attempt to reduce the overhead caused by synchronization. This leads to the third drawback: processes underutilize the given computational power, since available free processes are not used until there is absolutely no other choice but to join them in. These drawbacks make the algorithm insufficiently adaptive to the checked system and the underlying parallel en-

vironment. Furthermore, the combined effect of these drawbacks worsens with two factors: the size of the parallel environment and the presence of heterogeneous resources in it (as are commonly found today in non-dedicated grid and large-scale systems). These drawbacks limit the scalability of the algorithm and make it slow down substantially.

In order to exploit the full power of the parallel machinery and achieve scalability, it was necessary to design a new algorithm which is asynchronous in nature. We had to change the overall scheme to allow concurrency of computation and communication, to provide non-blocking protocols in several layers of the communication and the computation infrastructures, and to develop an asynchronous distributed termination detection scheme for a dynamic system in which processes join and leave the computation. In contrast to the approach presented in [15], the new algorithm does not synchronize the iterations among processes. Each process carries on the image computations at its own pace. The sending and receiving of states is carried out "in the background," with no coordination whatsoever. In this way, image computation and non-owned state exchange become *concurrent* operations.

Our algorithm is aimed at obtaining high speedup while fully utilizing the available computational power. To this end, when the number of free processes is relatively high the splitting rate is increased. This mechanism imposes *adaptive early splitting* to split a process even if its memory does not overflow. This approach ensures that free computational power will be utilized in full. In addition to using more processes, splitting the workload before memory overflows means that processes will handle smaller BDDs. This turned out to be a critical contribution to the speedup achieved by the new approach because a smaller BDD is easier to manipulate (improved locality of reference, faster image computation, faster and less frequent reorders, faster slicing, etc.).

In the asynchronous approach, when a process completes an iteration it carries on to the next one without waiting for the others. Consequently, splitting the workload with new processes is an efficient method for speeding up the computation since the overhead in adding more workers is negligible. However, this approach poses a huge challenge from the viewpoint of parallel software engineering. Given that the state space partition varies dynamically and that the communication is asynchronous, messages containing states may reach the wrong processes. By the time a message containing states is sent and received, the designated process may cease to own some or all of these states due to change of ownerships. Our algorithm overcomes this problem by incorporating a *distributed forwarding mechanism* that avoids synchronization but still assures that these states will eventually reach their owners. In addition, we developed a new method for opening messages containing packed BDDs which saves local buffer space and avoids redundant work: the mechanism ensures that only the relevant part of the BDD in the message is opened at every process visited by the message.

Distributed termination detection presents another challenge: although a certain process may reach a fixpoint, there may be states owned by this process that were discovered (or, are yet to be discovered) by others and are on their way to this process (in the form of BDDs packed in messages). The two-phase Dijkstra [11, 12] termination detection algorithm is an efficient solution in such cases. However, we had to face yet another algorithmic complication that was not addressed by Dijkstra: the number of processes in the computation can vary dynamically and cannot be estimated or bounded in advance.

We found no solution to this problem in the distributed computing literature. Thus, we had to develop the solution ourselves as an extension of the Dijkstra algorithm.

Related Work

Other papers suggest reducing the space requirements for sequential symbolic reachability analysis by partitioning the work into several tasks [25, 7, 13]. However, these schemes use a single machine to sequentially handle one task at a time, while the other tasks are kept in external memory. These algorithms, as well as the distributed symbolic algorithms [20, 15], are based on strict phases of computation and synchronization, which are carried out until a global fixpoint is reached. As a result, these schemes cannot scale well, and cannot take advantage of contemporary large-scale distributed platforms, such as huge clusters, grid batch systems and peer-to-peer networks, which are commonly non dedicated and highly heterogeneous.

The rest of the paper is organized as follows. In Section 2 we discuss the distributed approach. Section 3 describes sending and forwarding of BDD messages. In Section 4 we detail the algorithm performed by processes. Section 5 describes the asynchronous termination detection algorithm. Section 6 describes the operation of the coordinators. Experimental results are given in Section 7. Finally, we conclude in Section 8 with a summary and directions for future research.

2 The Distributed Asynchronous Approach

We begin by describing the sequential symbolic (BDD-based) reachability algorithm. The pseudo-code is given in Figure 1. The set of reachable states is computed sequentially by applying Breadth-First Search(BFS) starting from the set of initial states S_0. The search is preformed by means of *image computation* which, given a set of states, computes a set containing their successors. In general, two sets of states have to be maintained during reachability analysis:

1) The set of reachable states discovered so far, called R. This set becomes the set of reachable states when the exploration ends.
2) The set of reached but not yet developed states, called N. These states are developed in each iteration by applying image computation on N.

```
Reachability(S_0)
1) R = N = S_0
2) while (N ≠ φ)
3)     N = Image(N)
4)     N = N \ R
5)     R = R ∪ N
6) return R
```

Fig. 1. Sequential Reachability Analysis

The distributed reachability algorithm relies on the notion of Boolean function slicing [26]. The state space is partitioned into *slices*, where each slice is *owned* by one process. A set, $w_1 \ldots w_k$, of Boolean functions called *window functions* defines for each process the slice it owns. The set of window functions is complete and disjoint, that is, $\vee_{i=1}^{k} w_i = 1$ and $\forall i \neq j : w_i \wedge w_j = 0$, respectively. States that do not belong to the slice owned by a process are called *non-owned* states for this process.

As noted earlier, reachability analysis is usually carried out by means of a BFS exploration of the state space. Both the sequential algorithm (Figure 1) and the distributed synchronous algorithm (see [20, 15]) use this technique: in iteration i, image computation is applied to the set of states, N, which are reachable in i steps (and no fewer than i steps) from the set of initial states. Thus, when iteration i is finished, all the states which are reachable in at most $i + 1$ steps have already been discovered. While in a sequential search the states in N are developed by a single process, in a distributed search the states in N are developed by a number of processes, according to the state space partition. In the latter, the processes synchronize on a barrier at the end of each iteration, i.e., wait until all processes complete the current iteration. Only then do the processes exchange their recently discovered non-owned states and continue to the next iteration.

However, reachability analysis need not be performed in such a manner. Note that reachability analysis would be correct even if, in iteration i, not all the states which are reachable in i steps are developed, as long as they will be developed in a future iteration. Thus, when a process completes iteration i, it does not have to wait until the other processes complete it. It can continue in the image computation on the newly discovered states and receive owned states discovered by other processes at a later time. This is one of the key ideas behind the asynchronous approach employed in the computational level.

Like [20, 15], our algorithm uses two types of processes: workers and coordinators. The distributed platform consists of a non-dedicated pool of workers. Workers can join and leave the computation dynamically. Workers participating in the computation are called *active*. Otherwise, they are called *free*. Each active worker *owns* a slice of the state space and is responsible for discovering the reachable states within its slice. The algorithm 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. Each worker works iteratively. At each iteration, the worker computes an image and exchanges non-owned states, until a global fixpoint is reached and termination is detected. During image computation, the worker computes the new set of states that can be reached in one step from its owned part of N. The new computed set contains owned as well as non-owned states. During the exchange operation, the worker asynchronously sends the non-owned states to their corresponding owners. The novelty of our algorithm is that the iterations are not synchronized among workers. In addition, image computation and state exchange become concurrent.

Image computation and the receiving of owned states from other workers are critical points in which memory overflow may occur. In both cases, the computation stops and the worker splits its ownership into two slices. One slice is left with the overflowed worker and one given to a free worker. While distributed synchronous algorithms use splitting only when memory overflows, our approach also uses splitting to attain speedups. The *adaptive early splitting* mechanism splits a worker according to progress of its computation and availability of free workers in the pool. Besides utilizing free workers, this mechanism aims at increasing the asynchrony of the computation by splitting workers whose progress in the computation is not fast enough. An additional

mechanism merges ownerships of several workers with low memory requirements to one worker, where the others return to the pool of free workers.

The concurrency between image computation and state exchange is made possible by the asynchronous sending and receiving of states. Non-owned states are transformed into BDD messages. BDD messages are sent "in the background," by the operating system. Note that asynchronous communication is usually implemented in a manner which allows minimum CPU intervention. As a result, a worker that sends a BDD message to a colleague is not blocked until the BDD message is actually sent or received. Similarly, a worker need not immediately process a BDD message that it receives. Received BDD messages are accumulated in a buffer called *InBuff*. The worker can retrieve them whenever it chooses. The worker retrieves BDD messages from *InBuff* during image computation, transforms them to BDDs, and stores them in a set called *OpenBuff* until the current image operation is completed. To summarize, a worker has to maintain three sets of states, N, R, and *OpenBuff*, as well as one buffer, *InBuff*, during the distributed asynchronous reachability analysis.

In addition, our algorithm uses three coordinators: the `exch_coord`, which holds the current set of owned windows and is notified on every split or merge. The `exch_coord` is also responsible for termination detection; the `pool_mgr`, which keeps track of free workers; and the `small_coord`, which merges the work of underutilized workers. Following is an explanation of how we handle BDD messages. The algorithm itself will be explained in detail in Sections 4, 5 and 6.

3 Forwarding and Sending of BDD Messages

Workers often exchange non-owned states during reachability analysis. BDDs are translated into and from messages as described in [15]. A BDD message represents the content and pointers of each BDD node as an element in an array. This method reduces the original BDD by 50%. Thus, BDD messages are transferred across the net efficiently. Moreover, recall that there is no exchange phase in which the processes send BDD messages all at once; messages are sent and received asynchronously during the computation. In addition, received BDD messages are opened during image computation and pending messages do not accumulate. As a result, the communication overhead is negligible and the memory required to store BDD messages that are waiting to be sent or opened is relatively small. These observations held in all the experiments we conducted.

As noted earlier, messages of non-owned states may reach the wrong worker (some or all of the states in the BDD message do not belong to the worker). Our algorithm thus incorporates a *distributed forwarding mechanism* that avoids synchronization but still ensures that these states will eventually reach their owners. In addition, the mechanism enables forwarding BDD messages without transforming them to a BDD form (which may be a time consuming operation). To this end, we attach a window to each BDD message. We refer to a BDD message as a pair $\langle T, w \rangle$, where T is the BDD in an array form and w is the attached window. Before worker P_i sends worker P_j a BDD message, it receives from the `exch_coord` the window w'_j which P_j owned when it last updated the `exch_coord` . This is the window P_i *assumes* P_j owns. As illustrated in Figure 2(a), when P_i sends a message to P_j it attaches the window w'_j it assumes P_j owns. If

P_j is required to forward this message to worker P_k with an assumed window w'_k, it will change the window to $w'_j \wedge w'_k$ before doing so (see Figure 2(b)).

(a) (b)

Fig. 2. (a)P_i sends P_j a BDD message with an assumed window w'_j (b) P_j forwards a BDD message to P_k with an assumed window w'_k

The Open_Buffer procedure, described in Figure 3, retrieves BDD messages from *InBuff*. Recall that those messages are received asynchronously into *InBuff* by the operating system. When a worker retrieves BDD messages from *InBuff*, it requests and receives from the `exch_coord` the list of window functions owned by the workers. Next, it asynchronously forwards each BDD message to each worker whose window's intersection with the message window is non-empty. Then it opens the BDD message.

In this work we developed a new method for opening BDD messages which saves local buffer space and avoids redundant work: only the relevant part of the BDD in the message is opened at every process visited by the message. The new method, called *selective opening*, extracts from a BDD message only those states that are under a given window (the window of the message intersected with the window of the worker), without transforming the entire message to BDD form. The worker holds the owned states extracted from the BDD message in *OpenBuff*.

Though the selective opening method only extracts the required states, the operation may fail due to memory overflow. In this case, the worker splits its ownership and thereby reduces its workload. Note that, despite the split, the BDD messages pending in *InBuff* do not require special handling; the next time the worker calls the Open_Buffer procedure and retrieves a pending BDD message, it forwards it according to the updated state partition given by the `exch_coord` and extracts the owned states according to its new window.

4 The Worker Algorithm

A high level description of the algorithm performed by a worker with ID my_id is shown in Figure 3. We will first describe each procedure in general and then in detail.

During the Bounded_Image procedure, a worker computes the set of states that can be reached in one step from N, and stores the result in N. During the computation, the worker also calls the Open_Buffer procedure and extracts owned states into *OpenBuff*. N and R will be updated with those states only in the Exchange procedure. If memory overflows during image computation or during the opening of a buffer, the worker splits its window w and updates N, R and *OpenBuff* according to the new window. The same holds true if early splitting occurs.

During the Exchange procedure the worker sends out the non-owned states ($N \setminus w$) to their assumed owners and updates N, R with new states accumulated in *OpenBuff* (new states are states that do not appear in R).

If only a small amount of work remains, i.e., N and R are very small, the worker applies the Collect_Small procedure. The Collect_Small procedure merges the work of several workers into one task by merging their windows. As a result, one worker is

assigned the unified ownership (merges as owner) and the rest become ”free” ($w = \oslash$, merge as non-owners) and return to the pool of free workers.

```
procedure Open_Buffer(w, OpenBuff)
  {⟨T, w'⟩} ← BDD messages from InBuff
  {⟨P_j, w_j⟩} ← receive windows from exch_coor
  foreach (⟨T, w'⟩)
    foreach ((j ≠ my_id) ∧ (w' ∩ w_j ≠ ⊘))
      send ⟨T, w' ∩ w_j⟩ to P_j
    Res=Selective_Opening(T, w' ∩ w, Failed)
    if Failed = TRUE
      return BDD message to InBuff
      Split(R, w, N, OpenBuff)
    else OpenBuff = OpenBuff ∪ Res

procedure Reach_Task (R, w, N, OpenBuff)
  loop forever
    Bounded_Image(R, w, N, OpenBuff)
    Exchange(OpenBuff)
    if (Terminate() = TRUE)
      return R
    Collect_Small(R, w, N)
    if (w = ⊘)
      send ⟨'to_pool', my_id⟩ to pool_mgr
      return to pool (keep forwarding BDD messages)

procedure Exchange(OpenBuff)
  {⟨P_j, w_j⟩} ← receive windows from exch_coor
  foreach (j ≠ my_id)
    send ⟨N ∩ w_j, w_j⟩ to P_j
    N = N \ w_j
  N = N ∪ OpenBuff
  N = N \ R;  R = R ∪ N
  OpenBuff = ⊘
```

```
procedureBounded_Image(R, w, N, OpenBuff)
  Completed = FALSE
  while Completed = FALSE
      Bounded_Image_Step(R, w, N, Max, Failed, Completed)
      if ((Failed = TRUE)∨(Early_Split() = TRUE))
        Split(R, w, N, OpenBuff)
      Open_Buffer(w, OpenBuff)

function Terminate()
  if (N = ⊘ ∧ InBuff = ⊘ ∧ 'all async' sends are complete')
    if (TerminationStatus = 'no_term')
       TerminationStatus = 'want_term'
       send exch_coord ⟨TerminationStatus, my_id⟩
       return FALSE
  else if (TerminationStatus = 'want_term')
    TerminationStatus = 'regret_term'
  ⟨action⟩ ← receive from exch_coord if any
  if (action = 'regret_termination_query')
    send ⟨'regret_status', TeminationStatus, my_id⟩
  if (action = 'reset_term')
    TerminationStatus = 'no_term'
  if (action = 'terminate')
    TerminationStatus = 'terminate'
    return TRUE
  return FALSE
```

Fig. 3. Pseudo-code for a worker in the asynchronous distributed reachability computation

After performing Collect_Small, the worker checks whether its window is empty and it needs to join the pool of free workers. The window of a worker can be empty if it merged as non-owner in the Collect_Small procedure, or if it joined the computation with an empty window (this will be discussed later).

A worker is called *freed* if it participated in the computation once and then joined the pool of free workers. Freed workers may still receive misrouted BDD messages and thus need to forward them. For example, before worker P_i is freed, another worker may send it a message containing states that were owned by P_i. Should this message reach P_i after it was freed, P_i must then forward the message to the current owner(s) of these states. Methods for avoiding this situation will be discussed later. Note that if freed workers are required to forward BDD messages, they must participate in the termination algorithm. Following is a detailed description of each procedure.

The **Bounded_Image Procedure** is described in Figure 3. The image is computed by means of Bounded_Image_Step operations, which are repeated until the computation is complete. This algorithm uses a *partitioned transition relation*. Each partition defines the transition for one variable. The conjunction of all partitions gives the transition of all variables. Each Bounded_Image_Step applies one more partition and adds it to the intermediate result. The Bounded_Image_Step procedure receives as an argument the

maximal amount of memory that it may use. If it exceeds this limit, the procedure stops and *Failed* becomes true.

The technique for computing an image using a partitioned transition relation was suggested by Burch et al.[6] and used for the synchronous distributed algorithm in [15]. Using bounded steps to compute the image allows memory consumption to be monitored and the computation stopped if there is memory overflow. Also explained in [15] is how the partitioned transition relation helps to avoid repeating an overflowed computation from the beginning: each worker resumes the computation of its part of the image from the point at which it stopped and does not repeat the bounded steps that were completed in the overflowed worker.

Our asynchronous algorithm exploits the partitioned computation even further. During image computation, between each bounded step, we retrieve pending BDD messages from *InBuff*, forward them if necessary, and extract owned states into *OpenBuff*. By doing so, we free the system buffer which contained the messages and produce asynchronous send operations, if forwarding is needed. Note that R and N are updated with *OpenBuff* only after the current image computation is completed. In addition, during image computation, the worker can perform early split according to the progress of its computation and availability of free workers in the pool. We chose to implement the Early_Split function by checking whether the amount of free workers in the pool is above a certain threshold and whether the worker has not split for a while.

The **Exchange Procedure** is described in Figure 3. First, the worker requests and receives from the `exch_coord` the list of window functions owned by the other workers. Then it uses this list to asynchronously send recently discovered non-owned states to the other workers. Afterwards, it updates N, R with states accumulated in *OpenBuff* and recalculates N, R.

Collect_Small Procedure. [1] An underutilized worker, i.e., one with small N and R, informs the `small_coord` of their size. The `small_coord` gives the worker one of the following commands: exit the procedure (in case it has no other worker to merge with or it is not small enough), merge as owner, or merge as non-owner. In case the worker's ownership changed, it informs the `exch_coord` of its new window. Note that workers with large R sets can not be merged since the memory required to store the united R set may not fit in the memory of a single machine.

A worker which merges as non-owner is freed. As mentioned before, freed workers keep forwarding BDD messages. However, this can be avoided. A freed worker can stop forwarding BDD messages if all the other workers have already requested and received a set of windows that does not include this freed worker. This ensures that no new messages will be sent to it. In addition, to ensure that all the already sent BDD messages have arrived, we can either bound the arrival time of a BDD message or run a termination-like algorithm. The termination algorithm will be discussed later.

Split Procedure. [2] This procedure starts by asking the `pool_mgr` for a free worker. We use a $Slice$ procedure, which when given a BDD, computes a set of two windows that partition the BDD into two parts. This slicing algorithm was suggested in [20].

[1] The pseudo-code for the Collect_Small procedure is not given in this paper due to space limitations.

[2] The pseudo-code for the Split procedure is not given in this paper due to space limitations.

Two pairs of window functions are computed using the Slice procedure, one for N and one for R and *OpenBuff*. The two pairs are computed in an attempt to balance both the current image computation (by slicing N) and the memory requirements (by slicing R and *OpenBuff*). Note that the new windows the workers will own are the ones obtained by slicing R and *OpenBuff*. If R and *OpenBuff* are relatively small, only N is sliced. Thus, the overflowing worker's ownership remains unchanged and the new worker will have an empty window. Such a worker is called a *helper*. A helper simply assists the overflowed worker with a single image computation. Once the computation is complete, the helper sends the states it produced to their owners and joins the pool of free workers in the Reach_Task procedure. In our experiments, we observed that the creation of helpers is a common occurrence. After computing the partitions, the splitting worker sends the other worker its new window and its part of *R,OpenBuff* and N. It also updates the `exch_coord` with the new windows.

5 Asynchronous Termination Detection

Our termination detection algorithm is an extension of the two-phase Dijkstra [11, 12] termination detection algorithm. Dijkstra's algorithm assumes a fixed number of processes and synchronous communication. In our extension, the communication is asynchronous and processes may join and leave the computation.

The presented termination detection algorithm has two phases: the first phase during which the `exch_coord` receives *want_term* requests from all the active and freed workers, and the second phase, during which the `exch_coord` queries all the workers that participated in the previous phase as to whether they regret the termination. After receiving all responses, it decides whether to terminate or reset termination and notifies the workers of its decision. The part of the `exch_coord` in the termination detection is discussed in Section 6.

Each worker detects termination locally and notifies the `exch_coord` when it wants to terminate. Upon receiving a regret query, the worker answers as to whether it regrets its request. The next message the worker will receive from the `exch_coord` will command it to terminate or reset termination. Note that the communication described above is asynchronous and thus does not block the workers.

The pseudo-code for the Terminate function performed by a worker is given in Figure 3. The termination status of a worker can be one of the following: *no_term*, if it does not want to terminate; *want_term*, if it wants to terminate; *regret_term*, if its status was *want_term* when it discovered that it still has work to do; *terminate*, if it should terminate. The initial termination status is *no_term*.

Upon entering the Terminate function the worker checks whether all of the following three conditions hold: It does not have any new states to develop ($N = \oslash$); it does not have any pending BDD messages in *InBuff*; all its asynchronous send operations have been completed. We will clarify the last condition. If a worker receives a BDD message, the sender will not consider the send operation complete until it receives an acknowledgement from this worker. Without acknowledgement, there could be a BDD message that was sent but not yet received, and no worker would know of its existence. Note that the acknowledgement is sent and received asynchronously.

If the termination status is *no_term* and all conditions hold, the termination status is changed to *want_term*. The worker will notify the `exch_coord` that it wants to

terminate and exit the function (with return value false). If the termination status is *want_term* and one of the conditions does not hold, it may have more work to do. Thus, the termination status is changed to *regret_term*. If the worker has a pending command from the `exch_coord`, it acts accordingly. It can be prompted to send its termination status, or else to set it to either *no_term* or *terminate*.

6 The Coordinators

The exch_coord. Figure 4 describes the pseudo-code for the algorithm performed by the `exch_coord`. The `exch_coord` maintains a set of window functions Ws, where $Ws[P_i]$ holds the window owned by P_i. The `exch_coord` also maintains two lists: a list of active workers, *ActiveWL*, and a list of freed workers, *FreedWL*. The `exch_coord` receives notifications from workers and acts accordingly; when workers split or perform Collect_Small, it updates Ws, as well as the *ActiveWL* and *FreedWL* lists.

function **Exch_Coord()**
  $Ws[0] = one$
  $ActiveWL = \{0\}; FreedWL = \oslash$
  Loop-forever
    $\langle cmd \rangle$ = receive from *any* worker
    if $cmd = \langle `collect\_small', P_{id}, w_{id}, P_i \rangle$
      $Ws[P_{id}] = w_{id}$
      $ActiveWL = ActiveWL \setminus P_i$
      $FreedWL = FreedWL \cup P_i$
      send $\langle `release' \rangle$ to P_i and to P_{id}
    if $cmd = \langle `split', P_{id}, NewW_s = \{(p_i, w_i)\} \rangle$
      foreach $(p_i, w_i) \in NewW_s$
        $Ws[p_i] = w_i$
        $ActiveWL = ActiveWL \cup P_i$
        $FreedWL = FreedWL \setminus P_i$
      send $\langle `release' \rangle$ to P_{id}
    TerminationDetection(cmd)

procedure **MoveToRegretPhaseIfNeeded(P_i)**
  $WantTermL = WantTermL \setminus \{P_i\}$
  if $(WantTermL = \oslash \wedge TPhase = `want\_term')$
    $TPhase = `regret\_phase'$
    $\forall P_j \in RegretQueryL$:
      send $\langle `regret\_termination\_query' \rangle$ to P_j

procedure **ResetOrTerminateIfNeeded(P_i)**
  $RegretTermL = RegretTermL \setminus \{P_i\}$
  if $(RegretTermL = \oslash \wedge CancelTerm = FALSE)$
    $\forall P_j \in ResetOrTermL$: send $\langle `terminate' \rangle$ to P_j
  if $(RegretTermL = \oslash \wedge CancelTerm = TRUE)$
    $\forall P_j \in ResetOrTermL$: send $\langle `reset\_term' \rangle$ to P_j
    $ResetOrTermL = \oslash$; $CancelTerm = FALSE$
    $TPhase = `no\_term'$

function **TerminationDetection(cmd)**
  Initialization:
    $CancelTerm = FALSE$
    $RegretQueryL = \oslash$
    $TPhase = `no\_term'$
  if $cmd = \langle `want\_term', P_i \rangle$
    if $TPhase = `no\_term'$
      $WantTermL = ActiveWL \cup FreedWL$
      $TPhase = `want\_term'$
    if $TPhase = `regret\_term'$ (P_i is a split colleague)
      send $\langle `regret\_termination\_query' \rangle$ to P_i
    $RegretQueryL = RegretQueryL \cup \{P_i\}$
    MoveToRegretPhaseIfNeeded(P_i)
  if $cmd = \langle `regret\_status', stat, P_i \rangle$
    $CancelTerm = CancelTerm \vee (stat = regret)$
    $ResetOrTermL = ResetOrTermL \cup \{P_i\}$
    ResetOrTerminateIfNeeded(P_i)
  if $(cmd = \langle `split', P_{id}, \{(P_i, w_i)\} \rangle \wedge$
                $TPhase \neq `no\_term')$
    $CancelTerm = TRUE$
    if $TPhase = `want\_term'$
      $WantTermL = WantTermL \cup \{P_i | P_i \in \{(P_i, w_i)\}\}$
  if $(cmd = \langle `collect\_small', P_{id}, w_{id}, P_i \rangle \wedge$
                $TPhase \neq `no\_term')$
    $CancelTerm = TRUE$

Fig. 4. The pseudo-code for the `exch_coord`

The `exch_coord` detects termination according to the TerminationDetection procedure. The *TPhase* variable indicates the termination phase and can have one of the following values: *no_term*, which means that no termination request has yet been received; *want_term*, where the `exch_coord` collects termination requests; *regret_term*, where the `exch_coord` collects regret termination responses.The initial value of *TPhase* is *no_term*. In addition, the `exch_coord` holds the following three lists: the *WantTermL* list, which is used in the *want_term* phase and contains all the active and freed workers that have **not** sent a termination request; the *RegretQueryL* list, which is used in

the *regret_term* phase and contains all the workers that have **not** sent a regret response; and the *ResetOrTermL* list, which contains all the workers that will be notified of the termination decision when the *regret_term* ends.

The phase changes are triggered by commands received from the workers. The `exch_coord` can receive one of four commands and proceed accordingly. The *want_term* phase begins upon receiving a *want_term* request. Then the *WantTermL* is assigned the value of all active and freed workers. During this phase, the `exch_coord` receives *want_term* requests from all the workers in this list. Each worker that sends a request is removed from the *WantTermL* list and added to the *RegretQueryL*. When the *WantTermL* list becomes empty, the *regret_term* phase begins. All the workers in the *RegretQueryL* are sent a regret query. During this phase, those workers send a response to the query (their regret status). Each worker that sends a response is removed from the list and added to the *ResetOrTermL*. Only when the *RegretQueryL* becomes empty are the workers in the *ResetOrTermL* sent the decision as to whether or not to terminate. The `exch_coord` decides not to terminate if one of the workers regretted the termination or if split or merge occurred. In the latter case, the `exch_coord` also updates the appropriate lists.

The small_coord. The small_coord collects as many underutilized workers as possible. It receives merge requests from small (underutilized) workers. The small_coord stops a small worker for a predefined time; if timeout occurs and no other small worker has arrived in the meantime, it releases the worker. If a small worker arrives while another is waiting, it matches the two for merging.

The pool_mgr. The pool_mgr keeps track of free workers. During initialization it marks all workers as free, except for one. When a worker becomes free, it returns to the pool. When a worker splits, it sends the `pool_mgr` a request for a free worker. The `pool_mgr` sends in reply the *ID* of a free worker, which is then removed from the pool. If the `pool_mgr` is asked for a worker and there is no free worker in the pool, it stops the execution globally and announces "workers overflow."

7 Experimental Results

We implemented our algorithm on top of Division [14], a generic platform for the study of distributed symbolic model checking which requires an external model checker. We used FORECAST [13] for this purpose. FORECAST is an industrial strength high-performance implementation of a BDD-based model checker developed at Intel, Haifa.

This section describes our experimental results on certain large benchmarks that are known to be hard for reachability analysis. Most publicly available circuits are small or medium sized and can be computed sequentially. Therefore, we focused mostly on industrial-scale examples. We conducted experiments on two of the largest ISCAS89 benchmarks (s1269, s3330). Additional large-size examples are industrial designs taken from Intel. Our parallel testbed included a maximum of 56 PC machines, 2.4GHz Xeon processor with 4GB memory. The communication between the nodes was via LAM MPI over fast Ethernet. We used daemon-based communication, which allows true asynchronous message passing (i.e., the sending of messages progresses while the user's program is executing).

Our results are compared to FORECAST and to the work-efficient distributed synchronous implementation in [15]. The work-efficient implementation originally used

NuSMV [9] as an external BDD-based model checker. For comparability, we replaced it with FORECAST. The work-efficient implementation which uses FORECAST will be referred to as FORECAST-D (Distributed FORECAST), and our prototype as FORECAST-AD (Asynchronous FORECAST-D).

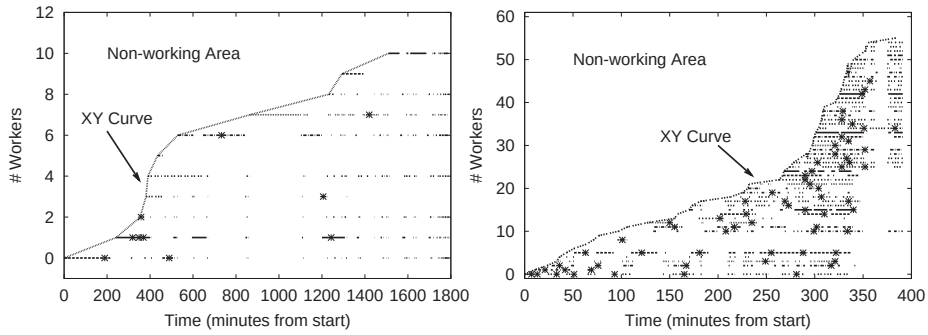
Circuit Name	# Vars	# Steps	Forecast		Forecast-D		Forecast-AD		
			Max. Step	Time(m)	Time(m)	# Workers	Time(m)	# Workers	Speedup (AD Vs. D)
s1269	55	9	9	45	50	12	15	6	3.3
s330	172	8	8	141	85	6	52	14	1.64
D_1	178	36	36	91	100	8	70	10	1.43
D_5	310	68	68	1112	897	5	150	18	5.98
D_6	328	94	94	81	101	5	76	3	1.3
Head_1_1	300	98	ovf(44)	-	9180	10	900	15	10.2
Head_2_0	276	85	ovf(44)	-	2784	4	390	55	7.14
Head_2_1	274	85	ovf(55)	-	1500	8	460	50	3.26
I1	138	139	ovf(102)	-	7178	18	2760	36	2.6

Fig. 5. A comparison between FORECAST, FORECAST-D and FORECAST-AD. If FORECAST was unable to complete an image step, we reported the overflowing step in parentheses. FORECAST-D and FORECAST-AD reached a fixpoint on all circuits. Column 10 shows the speedup when comparing FORECAST-AD and FORECAST-D run times.

Figure 5 clearly shows a significant speedup on all examples, up to an order of magnitude. When comparing FORECAST-D to FORECAST-AD, we were able to obtain a speedup even when the number of workers decreased. For instance, in the s1269 circuit, we obtained a speedup of 3.3 even though the number of workers decreased by a factor of 2. It can also be seen that the early splitting mechanism in FORECAST-AD enables using more workers than in FORECAST-D. Using more workers clearly increases efficiency: for example in the Head_1_1 circuit, FORECAST-AD uses 1.5 times more workers, but the speedup is of an order of magnitude.

We analyzed worker utilization when using the early splitting mechanism. Figure 6 provides utilization graphs for the Head_2_0 circuit, with this mechanism enabled and disabled. The Head_2_0 is a large circuit, difficult for reachability analysis. As can be seen in Figure 5, FORECAST is unable to reach a fixpoint on this circuit and overflows at step 44, while FORECAST-D requires over 46 hours to reach a fixpoint. Figure 6(a) clearly shows that when the early splitting mechanism is disabled, the workers are idle for much of the time. For instance, between 850 and 1100 minutes, only P_7 is working. This situation occurs when workers do not have any new states to develop and wait to receive new owned states. In this case, only when P_7 finds non-owned states and sends them to their corresponding owners are those workers utilized again. It is evident in Figure 6(b) that early splitting can significantly reduce such a phenomenon. As can be seen, the phenomenon still exists, but on a much smaller scale, for instance between 360 and 380 minutes. In addition, when using early splitting, we are able to use more machines more quickly. In Figure 6(a) it takes 1600 minutes for 10 machines to come into use, whereas in Figure 6(b) this takes 70 minutes.

Figure 6 also illustrates that when the number of workers increases, the relative size of the non-working area (the area above the XY curve) increases significantly. In the working area (the area below the XY curve), workers are dedicated to the distributed computation, whereas in the non-working area, workers are in the pool and can be



(a) Without Early Splitting (b) With Early Splitting

Fig. 6. FORECAST_AD worker utilization with and without the early splitting mechanism in the Head_2_0 circuit. In each graph, the Y axis represents the worker's ID. The X axis represents the time(in minutes) from the beginning of the distributed computation. For each worker, each point indicates that it computed an image at that time; a sequence of points represents a time segment in which a worker computed an image; a sequence in which points do not appear represents a time segment in which a worker is idle (it does not have any new states to develop). An asterisk on the time line of a worker represents the point when it split. The XY curve connects times at which workers join the computation. This curve separates the ***working*** from the ***non-working*** area. Note that the scales of the two graphs (both X axis and Y axis) are different.

used for other computations. Thus the *effectiveness* of the mechanism, i.e, the relation between the speedup and the increase in the number of workers, should actually be measured with respect to the relative size of the working area. Figure 7 presents the speedup obtained on several circuits, when using the early splitting mechanism.

Circuit Name	# Vars	Forecast-AD				
		No Early Splitting (A)		Early Splitting (B)		Speedup
		Time(m)	# Workers	Time(m)	# Workers	(A Vs. B)
s330	172	120	8	52	14	2.3
D_5	310	617	14	150	18	4.1
Head_1_1	300	1140	4	900	15	1.3
Head_2_0	276	1793	11	390	55	4.6
Head_2_1	274	1200	5	460	50	2.6

Fig. 7. The early splitting effect in FORECAST-AD. The "Speedup" column reports the speedup obtained when using the early splitting mechanism.

As can be seen in Figure 8, there is an almost linear correlation between the increase in computational power and the reduction in runtime on the Head_2_0 circuit. As the number of workers increases, the effectiveness decreases slightly. This can be explained by the fact that the relative size of the non-working area becomes larger as the number of workers increases (since we are not able to utilize free workers fast enough).

8 Conclusions and Future Work

This paper presents a novel algorithm for distributed symbolic reachability analysis which is asynchronous in nature. We employed non-blocking protocols in several layers of the communication and the computation infrastructures: asynchronous sending and receiving of BDD messages (concurrency between image computation and state exchange), opening of messages between bounded image steps, a non-blocking distributed forwarding mechanism, non-synchronized iterations, and an asynchronous termination

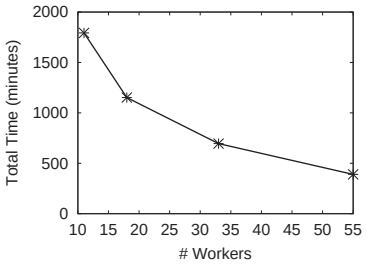


Fig. 8. The speedup obtained when increasing the number of workers on the Head_2_0 circuit (in FORECAST-AD). The X axis represents the time required to reach a fixpoint. The Y axis represents the maximal number of workers that participated in the computation. An asterisk on the (x, y) coordinate indicates that when the threshold of free workers is set to x, the reachability analysis ended after y minutes.

detection algorithm for a dynamic number of processes. Our dynamic approach tries to utilize contemporary non-dedicated large-scale computing platforms, such as Intel's Netbatch high-performance grid system, which controls all (tens of thousands) Intel servers around the world.

The experimental results show that our algorithm is able to compute reachability for models with high memory requirements while obtaining high speedups and utilizing the available computational power to its full extent.

Additional research should be conducted on better adaption of the reorder mechanism to a distributed environment. One of the benefits of the distributed approach which we exploit is that each worker can perform reorder independently of other workers and thus find the best order for the BDD it holds. We did not elaborate on this matter since it is not the focus of the paper. Our adaptive early splitting approach not only better utilizes free workers but also results in processes handling smaller-sized BDDs, which are easier to manipulate. In particular, the reorders in small BDDs are faster and less frequent. Nevertheless, the BDD package still spent a considerable time on reordering. We intend to explore the use of splitting as an alternative method for reordering.

References

1. J. Barnat, L. Brim, and J. Stribrna. Distributed LTL model-checking in SPIN. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 200–216. Springer-Verlag, 2001.
2. I. Beer, S. Ben-David, and A. Landver. On-The-Fly Model Checking of RCTL Formulas. In *CAV*, pages 184–194, 1998.
3. A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. In *Proceedings of the 7th International ERCIM Workshop, FMICS02*, July 2002.
4. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zue. *Bounded Model Checking*. Advances in Computers. Volume 58, Academic Press, 2003.
5. V. A. Braberman, A. Olivero, and F. Schapachnik. Issues in distributed timed model checking: Building Zeus. *STTT*, 7(1):4–18, 2005.
6. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic Model Checking with Partitioned Transition Relations. In *Proceedings of Internation Conference on Very Large Integration*, pages 45–58, 1991.
7. G. Cabodi, P. Camurati, and S. Quer. Improved Reachability Analysis of Large FSM. In *Proceedings of the IEEE International Conference on CAD*, pages 354–360, 1996.

8. Pankaj Chauhan, Edmund M. Clarke, and Daniel Kroening. Using SAT based Image Computation for Reachability Analysis. Technical report, Carnegie Mellon University, School of Computer Science, 2003.
9. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In *CAV'99*, pages 495–499.
10. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
11. E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information Processing Letters*, pages 217–219, 1983.
12. E. W. Dijkstra and C.S. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, pages 1–4, 1980.
13. R. Fraer, G. Kamhi, Z. Barukh, M.Y. Vardi, and L. Fix. Prioritized Traversal: Efficient Reachability Analysis for Verification and Falsification. In *CAV'00*, volume 1855 of *LNCS*.
14. O. Grumberg, A. Heyman, T. Heyman, and A. Schuster. Division System: A General Platform for Distributed Symbolic Model Checking Research, 2003.
15. O. Grumberg, T. Heyman, and A. Schuster. A Work-Efficient Distributed Algorithm for Reachability Analysis. In *CAV*, LNCS, 2003.
16. O. Grumberg, A. Schuster, and A. Yadgar. Memory Efficient All-Solutions SAT Solver and its Application for Reachability Analysis. In *FMCAD'04*.
17. A. Gupta, A. Gupta, Z. Yang, and P. Ashar. Dynamic Detection and Removal of Inactive Clauses in SAT with Application in Image Computation. In *DAC*, pages 536–541. ACM Press, 2001.
18. A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik. Partition-Based Decision Heuristics for Image Computation using SAT and BDDs. In *ICCAD*, pages 286–292. IEEE Press, 2001.
19. Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-Based Image Computation with Application in Reachability Analysis. In *FMCAD*, volume 1954 of *LNCS*, 2000.
20. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. A Scalable Parallel Algorithm for Reachability Analysis of Very Large Circuits. *Formal Methods in System Design*, pages 317 – 338, 2002.
21. H. Kang and I. Park. SAT-based Unbounded Symbolic Model Checking. In *DAC*, 2003.
22. F. Lerda and R. Sisto. Distributed-Memory Model Checking with SPIN. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39. Springer-Verlag, 1999.
23. K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *CAV'00*.
24. K. Milvang-Jensen and A. J. Hu. BDDNOW: A Parallel BDD Package. In *FMCAD '98, LNCS*, Palo Alto, California, USA, November 1998.
25. A. Narayan, A. Isles, J. Jain, R. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 388–393, 1997.
26. A. A. Narayan, J. Jawahar, M. Fujita, and A. Sangiovanni-Vincenteli. Partitioned-ROBDDs. In *CAV*, pages 547–554, 1996.
27. D. M. Nicol and G. Ciardo. Automated Parallelization of Discrete State-Space Generation. *J. Parallel Distrib. Comput.*, 47(2):153–167, 1997.
28. M. R. Prasad, A. Biere, and A. Gupta. A Survey of Recent Advances in SAT-Based Verification. To appear in STTT, 2005.
29. U. Stern and D. L. Dill. Parallelizing the Murφ Verifier. In *CAV*, pages 256–278, 1997.
30. T. Stornetta and F. Brewer. Implementation of an Efficient Parallel BDD Package. In *33rd Design Automation Conference*, 1996.