Distributed-Memory Model Checking with SPIN

Flavio Lerda and Riccardo Sisto

Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi, 24, I-10129 Torino (Italy)
e-mail: flerda@athena.polito.it, sisto@polito.it

Abstract. The main limiting factor of the model checker SPIN is currently the amount of available physical memory. This paper explores the possibility of exploiting a distributed-memory execution environment, such as a network of workstations interconnected by a standard LAN, to extend the size of the verification problems that can be successfully handled by SPIN. A distributed version of the algorithm used by SPIN to verify safety properties is presented, and its compatibility with the main memory and complexity reduction mechanisms of SPIN is discussed. Finally, some preliminary experimental results are presented.

1 Introduction

The model checker SPIN [4] is a tool widely used to verify concurrent system models. Its success depends on many factors, among which its amazing efficiency in performing model checking and its portability, i.e. the fact that, being written in ANSI C, it runs on most computer platforms.

The main limitation of SPIN, of course shared by all other verification tools based on reachability analysis, is that it can deal with models up to a given maximum size. As a model gets larger and larger, also the memory usage increases, and when the amount of memory used becomes greater than the available physical memory, the workstation is forced to use virtual memory. Since the memory is mainly allocated for a hash table, which is accessed randomly, the system will proceed slowly due to thrashing. In practice, it can be observed with SPIN that when the memory used is less than the physical available memory the performance of the model checker is excellent, and execution time is generally at most in the order of minutes, but as soon as the physical memory is exhausted the performance drops down dramatically. As a consequence, the maximum model size that SPIN can deal with depends essentially on the amount of physical memory that is available.

Various techniques are used by SPIN to reduce the amount of memory needed for verification, thus making the analysis of larger models possible. The main examples are state compression, partial order reductions and bit state hashing. A different technique that could be applied to further extend the size of the verification problems that can be successfully handled by SPIN is the use of a distributed-memory environment such as a network of workstations (NOW), which is ultimately a way to increase the amount of actually available physical memory, and to increase the speed of the verification process by exploiting parallel processing. In this paper we explore this possibility and present some preliminary results. Attention is focused here only on verification of safety properties such as deadlocks and assertions, and not on LTL model checking, which is left for further study.

As memory in SPIN is used mainly to store states, the distributed version of SPIN that we consider is based on a partition of the state space into as many state regions as the number of network nodes. Each node is assigned a different state region, and holds only the states belonging to that subset. In this way, the state table is distributed over a NOW. Each node computes the successors of the states that it holds and, if it finds any successors belonging to other state regions, it sends them to the nodes that are in charge of processing them. Of course, performance depends on how the state space is partitioned, the best results being obtained if the workload is well balanced and communication is minimized. In this paper we consider different possible approaches to the partitioning problem and compare them.

Another important issue that must be taken into consideration is that a distributed version of SPIN should not exclude the use of the other main memory and complexity reduction techniques available in the centralized version, such as state compression, partial order reduction, and bit state hashing. The approach that we consider in this paper is characterized by good compatibility with such mechanisms.

Since one of the strengths of SPIN stands in its portability and widespread use, we decided to develop an experimental distributed version of SPIN that can run on a very common platform: a NOW made up of heterogeneous workstations, interconnected with a standard 10Mbps Ethernet and using the TCP/IP protocol. Of course, more sophisticated communication infrastructures may yield better performance, but the basic environment that we considered is generally available to everyone, and its performance can be considered a reasonable lower bound.

A parallel/distributed version of a reachability analysis model checker, based on the Murφ Verifier [2], was proposed in [8]. The approach taken in [8] is similar to our one, but we use different ways to partition the state space. Moreover, in contrast with SPIN, the Murφ model checker uses a model where the computation of the next states may be quite complex, so with Murφ the most critical resource is not memory, but time. As documented in [8], there are cases where a verification run may take up to several days to complete. For this reason, the main purpose of the distributed version of Murφ is to speed up the verification process, exploiting parallel processing. With our distributed version of SPIN instead we mainly aim at making tractable models that otherwise would be intractable. Moreover, any speed up attained thanks to parallel processing tends to be obscured by communication overhead, which is generally predominant with respect to the short time taken by SPIN in computing the next states.

There has been also a previous proposal to develop a distributed model checker [1], in which the future state computation and the storage function are located at different nodes. This architecture is more complex than our one and the communication overhead is higher because each state is transferred at least two times over the network, since it has to go from the computation node that generated it to the storing node where it is kept for future reference and then back again from it to the computation node that must find its successors.

The rest of the paper is organized as follows. First, the distributed version of the SPIN verification algorithm is described, along with some implementation issues. Then the compatibility of this algorithm with the main memory and complexity reduction mechanisms of SPIN is discussed, and some preliminary experimental results are given. Finally, some conclusions are drawn and perspectives for further research are discussed.

2 The Distributed Verification Algorithm

2.1 The Centralized Algorithm

When SPIN must verify safety properties of a concurrent system, such as proper termination and state properties, it generates a verification program that makes a depth first visit of the system state space graph. The following pseudo code represents the centralized version of such a program:

```
procedure Start(start_state);
begin
  V := {}; { already visited states }
  DFS(start_state);
end;

procedure DFS(state);
begin
  if not state in V then
  begin
    V := V + state;
    for each sequential process P do
    begin
      nxt = all transitions of P enabled in state
      for each t in nxt do
      begin
        st = successor of state after t
        DFS(st);
      end;
    end;
  end;
end;
```

The procedure DFS makes a depth first visit of the graph, and is first called on the initial state (start_state). If the state to visit is not already present in the set of visited states V, it is added to V, and the DFS procedure is recursively called for each of its possible successors st. The computation of the successors consists of identifying the enabled transitions of the processes making up the model and determining the successor of the current state after each of such transitions. When an already visited state is found, the visit does not proceed any deeper.

For efficiency, the recursive DFS procedure is simulated by means of a user defined stack that contains the moves made from the initial state to the current state, along with all the information needed to restore the current state after a simulated recursive call of DFS(st). The set of visited states V is implemented by an hash table with collision lists, which is generally the most memory consuming data structure, its size being proportional to the number of states in the state graph. Also the stack data structure can consume a considerable amount of memory, because its size is proportional to the depth of the state graph, which, in some cases, can be comparable with the number of states.

2.2 The Distributed Algorithm

The idea at the basis of the distributed version of the verification algorithm is to partition the state space into as many subsets as the number of network nodes. Every node owns one of the state subsets, and is responsible for holding the states it owns and for computing their successors. When a node computes a new state, first it checks if the state belongs to its own state subset or to the subset of another node. If the state is local, the node goes ahead as usual, otherwise a message containing the state is sent to the owner of the state. Received messages are held in a queue and processed in sequence. When all queues are empty and all nodes are idle the verification ends.

The following pseudocode illustrates the algorithm used in the distributed version:

```
procedure Start(i, start_state);
begin
  V[i] := {}; { already visited states }
  U[i] := {}; { pending queue }
  j := Partition(start_state);
  if i = j then
  begin
    U[i] := U[i] + start_state;
  end;
  Visit(i);
end;

procedure Visit(i);
begin
```

```
  while true do
  begin
    while U[i] = {} do
    begin
    end;
    S := extract(U[i]);
    DFV(i,S);
  end;
end;

procedure DFV(i, state);
begin
  if not state in V then
  begin
    V[i] := V[i] + state;
    for each sequential process P do
    begin
      nxt = all transitions of P enabled in state
      for each t in nxt do
      begin
        st = successor of state after t
        j := Partition(st);
        if j = i then
        begin
          DFV(i, st);
        end else begin
          U[j] := U[j] + st;
        end;
      end;
    end;
  end;
end;
```

The nodes that participate in the algorithm execute all the same program, but each one of them calls the Start procedure with a different value of i, which is an integer index that identifies it. The set of visited states V is partitioned, V[i] being the subset assigned to node i, and Partition(s) being the function that takes a state s and returns the identifier of the node that owns s. Each node is coupled asynchronously with the other ones by means of an input pending requests queue. U[i] indicates the queue of node i. It is initially empty for every node but the one that owns the initial state.

Every node starts the visit (procedure Visit) waiting until a state is present in its input pending requests queue. At the beginning, only one node, the one that owns the initial state, has a non-empty queue and can proceed calling the DFV procedure. This procedure is almost the same as the previous procedure DFS. It performs a depth first visit of the state space graph, starting from the argument state, but the visit is restricted within the state space region owned by the node. The visit is not performed if the state has already been visited. If instead the state is new, it is added to the visited states set V[i]. Then each successor of the current state is computed in the usual way and checked with the Partition function. If the state st is local, i.e. it is owned by the same node, the procedure DFV is recursively called for that state, otherwise DFV is not called and the state is added to the pending requests queue of the corresponding node, which will continue the interrupted visit in its own state space region.

The main consequence of using this algorithm instead of the centralized one is that the visit does no longer follow the depth first order globally. ¿From the correctness point of view, this is not a problem with the standard reachability analysis verification of safety properties that we consider in this paper, because it works with a non depth first visit as well. LTL verification instead needs a (nested) depth first visit to give the correct results so the algorithm presented here is not adequate for this kind of verification. Of course it would be possible to modify the algorithm to make the visit depth first, but this would cut off most of the parallel processing involved in the algorithm. ¿From the memory usage point of view, a new data structure U[i] has been introduced, which increases the overall amount of memory needed. On the other hand, the non depth first order of the visit makes it possible to use a smaller stack structure, which can compensate for this memory increase. Moreover, the amount of memory needed for the pending requests queue can be bounded if some kind of flow control policy is applied.

2.3 The Partition Function

The Partition function that takes a state and returns the identifier of the region to which it belongs must depend exclusively on the state itself. Moreover, to balance the workload among the nodes, in terms of both memory and computation time, it should divide the state space evenly. Finally, to minimize communication overhead, it should minimize cross-transitions, i.e. transitions between states belonging to different regions.

A first simple possibility for partitioning is to use the same hash function that is applied to the state when it is stored in the hash table, as suggested in [8] for the parallelization of Murφ. In the case of an homogeneous network of workstations, this solution can be implemented very easily in a distributed SPIN program working with the above algorithm, but in the case of an heterogeneous one it cannot be implemented unless the hash function used by SPIN is modified. In fact, state vectors, i.e. the binary representations that SPIN uses for states, are different on different computer architectures, and the hash function of SPIN depends on such representations. Another problem with the hash partition function is that, although as shown in [8] it statistically divides the state space evenly, it does not address the problem of minimizing the number of cross-transitions.

Here we propose also another way to solve the partitioning problem, that exploits the structure of the global system states in SPIN.

SPIN is used to verify models of systems made up of synchronously or asynchronously coupled concurrent processes, where each process is described by a state machine. In such models, a global state contains a state component for each concurrent process. Since a state transition generally involves only few processes, generally one process for local actions or asynchronous interactions and two processes for synchronous interactions, when the system evolves from one state to another state only few state components change, the majority of them remaining unaffected. Based on these considerations, a convenient yet simple partitioning rule consists of defining the partition subsets according to the values taken by just one of the state components. In practice, the state region to which a state belongs depends only on the state component of one of the concurrent processes making up the model, called the designated process. Such a process can be for instance the one in a particular position in the state vector or the first one of a particular type. A table gives the correspondence between the states of the designated process and the state space subsets. With this kind of partition function, cross-transitions are transitions that determine a state change in the designated process, and, because of the above considerations, they are a limited fraction of the total. Moreover, some preliminary experiments show that partitions generated in this way are sufficiently well balanced.

The intuitive results that have just been presented can be confirmed by a simple analysis of the average features of the two partition functions. Let P, S and T be respectively the number of processes, of states and of state transitions in the model to be analyzed. Also, let N be the number of nodes used for distributed-memory reachability analysis. In general, the partition function is a function $\pi : S \to \{1, ..., N\}$, mapping global states to integers in the range from 1 to N.

With hash partitioning, states are mapped randomly and uniformly over state space regions. Hence, the average fraction of states belonging to a given region is $1/N$. For what concerns cross transitions, let us consider a generic state $s \in S$, and let T_s be the set of transitions starting from s, and D_s be the set of destination states of transitions in T_s. If we assume that there are no self transitions (i.e. transitions that do not change the global model state), because of the uniform distribution of states over regions, we can say that the average fraction of elements of D_s belonging to the same state region of s is $1/N$. Hence, the average fraction of elements of D_s belonging to other regions is $1 - 1/N$, and this represents also the average fraction of cross-transitions. Although this result does not take into account self-transitions, it can be considered a good approximation, because generally the fraction of self transitions is negligible in that it is very uncommon that a transition does neither change the values of any of the variables nor the "program counter" of any of the processes in the model.

Let us now consider the case of a partition function that depends on one state component only. Each state s of the model to be analyzed is composed of several state components, $s_i \in S_i$, i.e. $s = (s_1, ..., s_m)$. Let s_d be the state component representing the state of the designated process, and $\pi_d : S_d \to \{1, ..., N\}$ be the local partition function defined on the designated process state space S_d. The partition function that we are considering is one such that $\pi(s_1, ..., s_m) = \pi_d(s_d)$. Let us assume that π_d is selected so as to divide S_d into N equally sized subsets. In this case, the average fraction of global states $s = (s_1, ..., s_m)$ such that $\pi_d(s_d)$ takes a given value is $1/N$, and this is also the average fraction of global states belonging to a given region. With a partition function depending on one state component only, a cross transition is a transition that implies a change in the designated process state component from s_d to s'_d, such that $\pi_d(s_d) \neq \pi_d(s'_d)$. Let k be the average number of processes involved in a transition, which is a value ranging between 1 and 2. Then, k/P represents the average fraction of processes involved in a transition. Assuming that each process has the same chance of being involved in a transition, k/P represents also the average fraction of transitions that a given process is involved into. So we can say that the designated process is involved on average in a fraction k/P of the transitions. If we call ϕ_d the fraction of cross transitions in the designated process state machine, i.e. the fraction of local transitions of the designated process such that the starting state and the ending state are mapped to different regions, then we can conclude that the average fraction of cross transitions in the global state machine is $\phi_d k/P$.

This simplified analysis shows that on average the two partition functions both divide evenly the state space. However, the average fraction of cross transitions is $(N-1)/N$ with hash partitioning whereas it is $\phi_d k/P$ with a partitioning function based on one state component only. It can be observed that the first ratio tends to approach 1 as N becomes large, whereas in the second one only the ϕ_d factor gets close to 1, the average fraction of cross transitions remaining always less than k/P.

2.4 Keeping Track of Error Traces

If an error is found during the visit of the state space graph, the verification program must produce the trace of the model actions that lead to the error. In the centralized version of the program, this is done simply traversing the stack structure. In the distributed version, a similar approach is possible, but each node must hold the whole stack, containing the moves from the initial state to the current state, and not only the part of it corresponding to the execution of the DFV procedure. To make this possible, the message used to send a state to another node contains not only the state representation, but also the path that leads to that state, represented as a sequence of moves. The receiver uses the state representation to decide if the state has already been visited and eventually discards the message. If instead the state is new, the path is added to a list of paths representing the U[i] queue. Later on, when a path is dequeued, it is used to recreate the corresponding state: the path is followed, and every move in it is executed. In this way, the stack is automatically initialized to contain the execution path that leads to the current state. When an error occurs, the node behaves as in the centralized program.

For efficiency, in our experimental implementation of the algorithm, paths sent together with state representations are not absolute, but relative to the previous path sent. Moreover, they are represented in a compact way, using a simple run-length compression. In this way, the average message sizes are kept within reasonable values.

2.5 Algorithm Termination

The distributed algorithm must terminate when the U[i] queues are all empty and all nodes are idle. The detection of this condition is a typical problem of a class of distributed algorithms, including parallel discrete event simulation, and can be solved in different ways [7]. Here we sketch a possible solution, which has been used in the experimental implementation of the algorithm.

We use a manager process that is in charge of starting the verification program in a predetermined set of network nodes, and of stopping it after having detected termination and collected the results. Each node sends to the manager a message when it becomes idle and a different one when it becomes busy, i.e. when its queue becomes non-empty. In this way, the manager has a local representation of the current status of all the nodes.

When the manager detects on its local copy that all nodes are idle, it asks for a confirmation, because the local copy of the manager could be non consistent with the actual status of the nodes. If in the meanwhile a node has received a new message containing a new state to be visited, and then has become busy, it sends back a negative acknowledgment. Positive acknowledgments also contain the total number of messages sent and received by the node. The manager commands the nodes to terminate if each of them sent a positive acknowledgment and the overall number of messages sent is equal to the overall number of messages received. If this is not the case, there are some messages still traveling in the network. In this case the manager does not know if such messages will cause a node to start a new visit, because they may contain already visited states, so the manager needs to reset the procedure and then ask for a new confirmation round from all the nodes again.

2.6 Other Implementation Issues

An experimental modified version of the model checker SPIN that generates a distributed version of the verification program according to the above algorithm has been implemented. The generated source files can be used to make both the centralized and the distributed versions of the program, depending on a macro definition.

One of the main objectives (and also a main problem to solve) was to get the verification work on a network of heterogeneous workstations. Our first try was to use the PVM library [3], which is a widely used package that provides a transparent message passing interface for networks of heterogeneous workstations. This possibility was later abandoned, because of the overhead introduced and because of the need to implement flow control over the PVM layer. In fact the PVM library buffers messages in the receiving machine memory without limitations, and this may cause memory overflow problems. We decided then to use the socket interface, that provides standard bidirectional connections where flow control is already implemented. On top of it we used an XDR (eXternal Data Representation) layer, to make the transfer of data between different architectures transparent. The program has been successfully tested on three different platforms: Intel - Linux, Alpha - Digital Unix, and Sparc - SunOS.

3 Compatibility with Memory and Complexity Reduction Mechanisms

Whenever a new technique to extend the capabilities of a tool like SPIN is introduced, it is important to verify that it is compatible with the memory and complexity reduction mechanisms available in the basic version of the tool, otherwise the implied risk is that the overall performance of the tool is not really extended by the introduction of the new technique, but possibly reduced.

In this section, the main reduction mechanisms of SPIN are considered and compatibility with each of them is discussed. Some of them have already been implemented in our experimental distributed version of SPIN, whereas others can be easily added.

3.1 State Compression

SPIN implements various schemes of compression, that are used for reducing the amount of memory needed for storing states, but whichever compression technique is used, the state is always computed in its uncompressed form, and then it is compressed before being stored. In the distributed version of the program, the hash table is divided into different sub-tables, and for each of them any compression mechanism can be applied for storing states, without limitations.

Although the use of any compression technique is always possible, there may be performance implications. In its simplest form, compression is a function f that transforms a state s into a compact representation $f(s)$. The original representation s can be retrieved applying the inverse of f. Other compression schemes are characterized by memory, i.e. the result of the compression function depends on the history of compression operations previously performed. In the distributed version of the program, two aspects play an important role: the kind of compression that is used (with or without memory) and the heterogeneity of the network nodes.

If the network nodes are homogeneous, the state representation is the same on any network node. If a memoryless compression scheme is used, states can be sent in their compressed form and the amount of memory needed to store states is the same as in the centralized version. Moreover, in this case compression contributes to reduce the communication overhead. If instead a compression scheme with memory is used, states must be sent in their uncompressed form, and compression must be performed at the receiver side, otherwise the receiver may not be able to reconstruct the uncompressed state from the compressed one. In addition, the behavior of the compression function may be different from the one in the centralized version.

In an heterogeneous environment, the representations of a given state differ from node to node. For this reason, the state must be transformed into a machine-independent representation such as XDR before being sent to another network node. Let us call g the function that transforms a state s into its machine-independent form $g(s)$. Since the compression mechanisms of SPIN work on the machine-dependent representation of s, in this case the sender must send $g(s)$, and the receiver will reconstruct s from $g(s)$ and then apply f before storing s.

In our experimental implementation we included the two main compression schemes of SPIN, i.e. standard compression and collapse compression.

3.2 Partial Order Reduction

SPIN uses a static partial order reduction technique [6], which is a means to avoid the exploration of some execution sequences that are not strictly required to prove the safety or liveness properties of the concurrent system being analyzed. When this reduction method is applied, the expansion step in which the successors of a state are computed is modified, with the aim of computing only a minimal subset of all the possible successors. The expansion step is performed expanding each concurrent process, i.e. computing the possible state transitions of each process, and, for each of such transitions, computing the global successor state. Before the actual expansion step is carried out, processes are examined sequentially, to identify the ones that can execute only so-called safe transitions [6]. In fact, it has been shown that it is enough to expand just one of such processes, provided that the successors fulfill a condition, known as the reduction proviso. The verification program computes the successors generated by the transitions of each of the above processes first, and as soon as it finds one of them that generates a set of successors satisfying the reduction proviso, it interrupts the expansion, thus ignoring the successors generated by the other processes. In the worst case, all the processes are expanded, and no reduction occurs. The reduction proviso can be different according to the kind of properties that reduction must preserve. If only safety properties must be preserved, as we assume in this paper, it is enough to require, as a reduction proviso, that there is at least one successor not contained in the stack [5].

In the distributed version of the verification program, the static information about which transitions are safe is known by all the nodes, so any node can make a preselection of the processes, and examine first those that can execute only such transitions. However a problem arises when one of the nodes must check if the reduction proviso is fulfilled. If some of the successors are stored outside the node that has computed them, such a node cannot know by itself if they are currently in the stack, because this information is hold by SPIN in the state table. Obliging any node to hold a copy of all the states currently in the stack is cumbersome and memory-consuming. This problem can be avoided taking a conservative (worst case) assumption: successors that are hold outside the node where they are computed are always assumed to be currently in the stack. In this way, safety properties are preserved, but it is possible that some of the reductions that are carried out in the centralized version of the program cannot be carried out also in the distributed version.

It is interesting to note that, if we consider specifically the case of a partition function that depends on one of the state components only, it is possible to increase the number of reductions that can be performed. The reason is that in this case any transition that does not involve the designated process is not a cross-transition, i.e. it leads to a successor state that is in the same region as the current state. Therefore, the reduction proviso can always be checked for all such transitions. Moreover, if the designated process is arranged so as to always be the last process that is examined in the preselection phase, it is expanded only when no other process can be expanded. In this way, the number of reductions performed is maximized, and, as an additional side-effect, the fraction of cross-transitions is further reduced. This happens because, whenever different alternative reductions can be applied, always a reduction without cross-transitions is selected.

3.3 Bit State Hashing

Bit state hashing is a complexity reduction mechanism that affects the way in which states are stored in the hash table. It can be considered as a compression mechanism with loss of information, because a state is stored in a single bit. Consequently, the considerations made for compression also apply for bit state hashing, i.e. the distributed version of the program is compatible with it.

4 Experimentation and Results

The experimental distributed version of SPIN has been tested on some test cases, to measure its performance. The testbed that has been used is a NOW composed of 300Mhz Pentium II Linux workstations with 64Mbytes of RAM each, interconnected with a standard 10Mbps Ethernet LAN. The selected test samples are all scalable using one or more parameters, and experiments have been carried out with values of the parameters that are critical for the workstations that have been used.

Table 1 contains the results obtained using three scalable samples named Bakery, Leader, and Philo. For each sample and each set of values of the parameters the number of states and transitions are reported. Of course, the number of states mainly influences memory usage while the number of transitions mainly influences computation time. The tests were executed on a single workstation with the original centralized program and on two and four workstations with the distributed one. For each test case the average memory usage per node in megabytes and the execution time in seconds are reported. All the results are referred to an exhaustive verification and have been obtained using the XDR layer and a partition function that depends on a single state component. Missing results mean that verification could not be completed because of memory allocation failure.

Bakery is a description of the Lamport's Bakery algorithm for mutual exclusion with N processes that make at most K attempts. The Bakery sample was tested using the standard compression and partial order reduction features of SPIN. Since partial order reduction can alter the number of states and transitions actually explored, the data in the second and third rows are those reported by the centralized version. The partial order reduction is slightly less effective on the distributed version, and its effectiveness decreases as the state subsets get smaller, so the numbers of states and transitions with the distributed version are greater than with the centralized version. While in the other samples the memory per node nearly halves when doubling the number of workstations, in this case the memory used decreases but not so rapidly. Nevertheless, it can be observed that, when the total memory usage is higher than the physical memory available on a single workstation (second and third columns), the distributed version performs better than the centralized one.

Leader is the leader election algorithm model that can be found in the SPIN distribution. It was compiled with collapse compression, but without partial order reductions. The Leader sample memory usage grows very fast. While with N=6 the distribution of the verification program is still not convenient because of the modest memory usage, with N=7 the load is too heavy for a single workstation, and gives the best results with 4 workstations. This is an example of a verification that cannot be completed on a single workstation, because of memory allocation failure, but can be completed on multiple workstations. The results of distributed verification with N=7 are not so good as they could be, because the partition function used in the experiment is not so fair. In fact, even if the average memory usage for each node is less than the physical memory, we found that at least one node was thrashing due to its higher number of states to visit.

Philo is a model describing the dining philosophers problem. It was compiled using collapse compression and no partial order reduction. Also with this sample the growth is very fast and we pass from N=12, where the most efficient technique is centralized verification, to N=14, where it is necessary to resort to distributed verification, and the best results are obtained with four workstations.

In the rest of this section, additional results are presented that are useful to analyze a few aspects of the experimental distributed version of SPIN.

Table 2 illustrates the differences in terms of number of messages exchanged, partition rates, and execution time when using different partition strategies. The partition rate is the ratio of the minimum to maximum values of the state space region sizes, which gives a measure of how the workload is balanced.

The first strategy, called ad hoc partitioning, is based on a user defined partition function that depends on a single state component, and is the same used in the previous tests. This function has been defined empirically, trying to obtain a well balanced partition. The second partitioning function, called hash partitioning, is the one based on the same hash function used by SPIN for the hash table. Finally, modified hash partitioning uses the same hash-based approach, while retaining the principle of having a partition function that depends on one

Table 1. Performances of centralized and distributed SPIN

Bakery				
Model Parameters		N=2 K=20	N=3 K=10	N=4 K=4
States		106063	1877341	2336892
Transitions		129768	2463669	3101854
1 workstation	Memory per node	4.68MB	76.45MB	103.06MB
	Execution time	1s	908s	6306s
2 workstation	Memory per node	3.01MB	39.7MB	53.57MB
	Execution time	2s	24s	481s
4 workstation	Memory per node	2.55MB	34.7MB	51.82MB
	Execution time	23s	235s	330s

Leader				
Model Parameters		N=5	N=6	N=7
States		41692	341316	2801653
Transitions		169690	1667887	15976645
1 workstation	Memory per node	3.78MB	24.57MB	-
	Execution time	2s	27s	-
2 workstation	Memory per node	2.52MB	12.92MB	109.5MB
	Execution time	3s	130s	6687s
4 workstation	Memory per node	1.89MB	7.1MB	55.40MB
	Execution time	23s	219s	4577s

Philo			
Model Parameters		N=12	N=14
States		94376	636810
Transitions		503702	3965261
1 workstation	Memory per node	18.1MB	-
	Execution time	11s	-
2 workstation	Memory per node	6.70MB	63.1MB
	Execution time	23s	248s
4 workstation	Memory per node	3.94MB	20.89MB
	Execution time	35s	196s

Table 2. A comparison of different partition functions

Test Sample	Partitioning	Messages	Partition rate	Execution time
Bakery (N=2,K=20)	Ad Hoc	38874	0.7749	10s
	Hash	883297	0.9870	113s
	Modified Hash	343340	0.7470	50s
Leader (N=6)	Ad Hoc	139293	0.7361	46s
	Hash	373023	0.9679	121s
	Modified Hash	113378	0.7161	45s
Philo (N=14)	Ad Hoc	242984	0.9155	155s
	Hash	1185761	0.9871	403s
	Modified Hash	234328	0.8961	155s

state component only. This is achieved computing the hash function on one state component only.

For the Bakery example, we decided to present the results obtained without partial order reduction, because when this mechanism is used the number of states varies with the partition function, and in this case we found that the variance was so high that verification could not be completed when using hash partitioning, so we were not able to give any data for this kind of partitioning. This result of course confirms the expected better performance of the partition functions based on one state component with respect to partial order reductions.

Another difference with respect to the experiments presented in Table 1 is that here the results are referred to experiments made without the XDR layer, because in our implementation hash partitioning works only with this configuration. The parameters for the samples are shown in the table. All the results are relative to a 2 workstations testbed.

The results show that hash partitioning gives the best results in terms of balancing, but the number of messages cannot be controlled. This causes an increment in network overhead due to an increased amount of data transferred, and, consequently, an increment in execution times. Partition functions depending on one state component only do not give perfectly balanced workloads, but they are the most effective ones in terms of completion time, because they give sufficiently well balanced partitions and, at the same time, a low number of cross-transitions.

Partition functions based on one state component only could be improved exploiting knowledge about the system behavior, and statistical data such as transition counts that SPIN can compute. For example, it is possible to let SPIN compute the transition counts for a smaller sized (scaled down) model and then using these data to drive the definition of the partitioning function for the bigger one. However, a manual definition of a good partition function is not always easy. A possible improvement may be to introduce a tool that, on a statistical base, automatically determinates the partition function.

Table 3. Evaluation of the overhead due to heterogeneity

Test Sample	Transfer	Memory per node	Execution time
Bakery (N=3,K=10)	Binary	9.36MB	21s
	XDR	13.3MB	24s
Leader (N=6)	Binary	50MB	46s
	XDR	149.9MB	130s
Philo (N=14)	Binary	142.3MB	155s
	XDR	195.0MB	248s

One of the main goals of the project was to develop a distributed version of SPIN that can run on a network of heterogeneous workstations. This requirement introduces an overhead, mainly due to internal representation conversions. Of course, when using an homogeneous network of workstations, conversions can be avoided, and the XDR layer can be eliminated. This can speed up the verification process, reducing message size, and hence network overhead, but also memory requirements and and computation time. Table 3 can be useful to evaluate the amount of overhead implied by the XDR layer. It reports the results obtained using unformatted binary transfer in an homogeneous architecture (Binary Transfer), and those obtained using platform independent transfers (XDR Transfer), in the same architecture. In this case partial order reduction does not affect the results, so it has been used in the Bakery sample.

5 Conclusions and Perspectives

We have described a distributed-memory algorithm for verification of safety properties and its experimental implementation within the SPIN model checker. It has been shown that the algorithm can extend the capabilities of a verification tool like SPIN, adding up the physical memory of several workstations interconnected via a standard LAN, so implementing a distributed virtual verifier. When this algorithm is applied, it becomes possible to deal with models larger than those that can be analyzed with a single workstation running the standard version of SPIN.

Since SPIN is very efficient in analyzing the models that fit in the physical memory of the machine where it runs, the distributed version of the verification program performs worse than the centralized version on such models. Therefore, the distributed version of SPIN may be useful as a complementary tool, to be used instead of the standard version when the model size exceeds the available physical memory.

The distributed algorithm that has been presented is compatible with the main techniques used by SPIN to improve performance. In particular, it is compatible with compression techniques, static partial order reduction techniques, and bit state hashing.

One of the most critical aspects of the distributed verification algorithm is the function that defines the partition of the state space graph. We have considered various possibilities and showed that a function that depends on a single state component is advantageous under several points of view, because it can yield not only sufficiently well balanced workloads, but also low communication overhead, and wide applicability of partial order reductions.

In this paper, the possibility of implementing a distributed LTL model checking based on a nested depth first visit of the state space was not considered. This is however an interesting area for future research. The main problem that remains to be solved is that the algorithm as it has been proposed here performs a non depth first visit of the state space. If we introduce some form of synchronization, the network nodes can be forced to perform a depth first visit. For example, each node would be obliged to wait for a reply from the destination node whenever it sends a message, and the destination node would communicate back when it has finished. This kind of algorithm would cut off most of the parallel processing, and its implications in terms of needed memory must be investigated.

Another important issue for further study on the distributed verification algorithm is the automatic generation of good partitioning functions. Of course, optimality cannot be achieved, because this is an np-hard problem, but good heuristics, such as the principle of having a partition function that depends on a single state component, can be found. This point has a relevant practical importance, because asking users to define partition functions by themselves may not be acceptable.

Finally, the experiments presented in this paper are only preliminary. We plan to perform more extensive tests to get additional experimental results and so be able to give a more comprehensive view of the performance of this technique and of its scalability.

References

[1] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In P. Varaiya and H. Kurzhanski, editors, *Discrete Event Systems: Models and Application*, volume 103 of *LNCIS*, pages 40–56, Berlin, Germany, August 1987. Springer-Verlag.

[2] D. L. Dill. The murphi verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[3] J. Greenfield. An overview of the PVM software system. In *Ideas in Science and Electronics Exposition and Symposium. Proceedings: Albuquerque, NM, USA, May 1995*, volume 17 of *Annual Ideas in Science and Electronics Exposition and Symposium Conference*, pages 17–23. IEEE Computer Society Press, 1995.

[4] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[5] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992.

[6] G. J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[7] D. M. Nicol. Noncommittal barrier synchronization. *Parallel Computing*, 21(4):529–549, April 1995.

[8] U. Stern and D. L. Dill. Parallelizing the murphi verifier. In *Proceedings of the Nineth International Conference on Computer Aided Verification CAV*, 1997.