Random Walk Based Heuristic Algorithms for Distributed Memory Model Checking

Hemanthkumar Sivaraj and Ganesh Gopalakrishnan

UUCS-03-01

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

January 29, 2003

Abstract

Model checking techniques suffer from the state space explosion problem: as the size of the system being verified increases, the total state space of the system increases exponentially. Some of the methods that have been devised to tackle this problem are partial order reduction, symmetry reduction, hash compaction, selective state caching, etc. One approach to the problem that has gained interest in recent years is the parallelization of model checking algorithms.

A random walk on the state space has some nice properties, the most important of which is the fact that it lends itself to being parallelized in a natural way. Random walk is a low overhead and a partial search method. Breadth first search, on the other hand, is a high overhead and a full search technique. In this article, we propose various heuristic algorithms that combine random walks on the state space with bounded breadth first search in a parallel context. These algorithms are in the process of being incorporated into a distributed memory model checker.

Random Walk Based Heuristic Algorithms for Distributed Memory Model Checking*

Hemanthkumar Sivaraj and Ganesh Gopalakrishnan
School of Computing, University of Utah
{hemanth|ganesh}@cs.utah.edu

Abstract

Model checking techniques suffer from the state space explosion problem: as the size of the system being verified increases, the total state space of the system increases exponentially. Some of the methods that have been devised to tackle this problem are partial order reduction, symmetry reduction, hash compaction, selective state caching, etc. One approach to the problem that has gained interest in recent years is the parallelization of model checking algorithms.

A random walk on the state space has some nice properties, the most important of which is the fact that it lends itself to being parallelized in a natural way. Random walk is a low overhead and a partial search method. Breadth first search, on the other hand, is a high overhead and a full search technique. In this article, we propose various heuristic algorithms that combine random walks on the state space with bounded breadth first search in a parallel context. These algorithms are in the process of being incorporated into a distributed memory model checker.

1 Introduction

Model checking [CGP99] is a method for formally verifying finite-state concurrent systems. Usually, the method exhaustively searches the state space of the concurrent system to prove that a particular property holds. This is guaranteed to terminate since the model is finite. This search can be done in two different ways:

- The states can be explicitly enumerated and stored separately in a table. This method is called *explicit state model checking*.

*This work was supported in part by the Semiconductor Research Corporation (SRC) under contract no. 1031.001, and by the National Science Foundation (NSF) under grant no. CCR-0081406 (ITR Program) and CCR-0219805 (ITR Program). The authors wish to thank Ulrich Stern and Prof. David Dill for making available the source code of Parallel Murphi on which our tool is based.

- The states can be symbolically represented, such as representing states using a Binary Decision Diagram (BDD) [BCMD90]. This is called *symbolic model checking*.

Both methods have domains where they outperform each other. The properties that are verified can also be classified into two categories:

- *Safety properties*. In plain English, these are properties that say that "something bad never happens".
- *Liveness properties*. These are properties that say that "something good eventually happens".

When verifying concurrent systems, all possible interleavings of concurrent transitions in the system must be considered for exploration. This gives rise to the *state space explosion* problem: as the size of the system increases, the total state space of the system increases exponentially. This is the main obstacle in employing model checking on a large scale. There have primarily been two approaches to combat this problem. The first approach is to devise techniques to reduce the state space size to be explored while still ensuring that errors are detected. Examples of this approach are symmetry reduction techniques [IP96], partial order reduction techniques [HP94, NG02], etc. The second approach aims at reducing the amount of memory needed to perform the reachability analysis. Examples of this approach are bitstate hashing [Holz87], hash compaction [WL93, SD95], etc.

The use of distributed processing to combat the state explosion problem had gained interest in recent years. For large models, the state space doesn't fit into the main memory of a processor and hence the model checking run becomes very slow due to disk accesses. By using the combined main memory of the processors in a distributed memory machine, we will be able to verify bigger models. In model checkers like Murphi [Dill96] where the next state generation takes a considerable fraction of the total run time, we can use the combined processing power of the distributed machine to speed up the process. We will talk more about the parallel version of Murphi in Section 2.

1.1 Background on random walk

A random walk on the state space is a sequence of states $s_0, s_1, \cdots, s_n$ such that s_i is a state that is chosen uniformly at random from among the next states of the state s_{i-1}, for $i = 1, 2, \cdots, n$. Two properties of the random walk make it attractive to be used for finding bugs. Firstly, it uses much less space since it doesn't need to maintain any table to detect previously visited states. The downside is that it visits duplicate states. Secondly, and more importantly, it is inherently suitable for being parallelized. n random walks started in parallel on the same state space from the same start state will explore more states than one random walk. This is because the probability of all n random walks taking the same path decreases exponentially as the length of the path increases.

There are some drawbacks to using random walks as opposed to using full state search methods like breadth first search. Estimating the coverage of a random walk is difficult. The theoretical bounds for achieving full coverage of a graph using random walk are too prohibitive and are not practical. Hence it is difficult to quantify the effectiveness of random walks. Another problem concerns the structure of the state space on which the random walk is used. Consider a state space structure in which a densely connected region of states is linked to another densely connected region by a very sparse region (maybe a few transitions). If a random walk is in the first dense region, it has a very low probability of reaching the second dense region. Of course, the probability increases if we have multiple random walks exploring the state space simultaneously, but the problem is alleviated and not eliminated.

1.2 Motivation

Nowadays, the designs to be verified are extremely large and push the envelope of verification tools. Even with state space reduction techniques like symmetry reduction, partial order reduction, etc., exhaustive verification of even small models of current systems is not being possible. For example, in the case of cache coherence protocols, we are only able to fully verify models with 1-2 addresses, 1-2 processors, etc. If we increase the parameters to 3-4 processors, 3-4 addresses, etc., the state space explodes tremendously and full explicit state exploration is not possible with the current techniques and resources. How do we tackle designs of this magnitude? What if bugs lurk in such larger models? It is not immediately obvious that verification of models with small values for the parameters implies verification of the models with larger values for the parameters. We think random walks can help in such situations.

Random walks have been shown [West89] to perform well in situations in which the number of states that can be visited is only a small fraction of the total number of states in the model to be verified. Such large models are easy to come by both in hardware and software model checking. While all the investigation to push exhaustive model checking is going on, such large models can be currently subjected to random walk based exploration and some bugs can be caught.

In parallel and distributed model checking, so far almost all effort has been devoted to full state space exploration. With the ever increasing size and complexity of the models to be verified, model checking is being increasingly viewed as more of a bug-hunting tool than as a tool that can do full state space search. After the initial testing phase to remove the "easy" bugs, model checking is used to weed out the remaining few "hard-to-find" bugs. Clarke and Wing [CW96] say that a model checking tool must be *error detection oriented*. That is, they must be optimized for finding errors and not for certifying correctness. Not much work [JS02] has been done to develop a parallel model checker tuned to finding bugs. The suitability of current parallel model checkers for finding bugs fast is not yet demonstrated.

We propose to build a distributed model checking tool that is geared towards finding safety violations faster on very large state spaces. We concentrate only on verifying safety properties for various reasons. Firstly, safety property verification translates to simple reachability analysis for which breadth first search can be used and breadth first search is highly parallelizable. Secondly, safety properties form a large fraction of the properties that are usually checked. Finally, many of the liveness properties can be approximated and written in the form of bounded safety properties. [BAS02] describes a method for *exactly* translating finite state liveness checking problems into finite state safety checking problems.

Random walk and breadth first search can be considered to be two ends of a spectrum. They differ in the state space coverage and the overhead associated with them. While random walk is a low overhead and partial search method, breadth first search is a high overhead and full search method. We propose heuristics that combine multiple random walks on the state space with bounded breadth first searches (breadth first searches of fixed height). While the random walks take us deeper into the state space faster, the breadth first searches are used to do localized full state search to find bugs that the walks[1] may have missed. These heuristics are being incorporated into our tool. The heuristics we propose are different from standard random simulation in that simulation is usually done for production models while these heuristics are to be applied to models created for model checking.

1.3 Roadmap

Section 2 discusses prior work that has been done in the area of distributed and parallel model checking as well as on the use of random walks to do model checking. To demonstrate the feasibility of model checking large models using distributed memory algorithms, we have previously built a parallel full state exploration model checker. It was developed by porting the distributed memory version of Murphi [SD01] created at Stanford to use the MPI [Pach96] message passing library. Section 3 talks in more detail about this tool. We are currently building a second distributed model checking prototype by incorporating various random walk based heuristics into the tool. The purpose of this article is to explain the algorithms behind this prototype. We will evaluate these heuristics on various metrics, to wit, time taken to find bugs, fraction of bugs found, memory consumption, network bandwidth consumption, etc. Section 4 talks about all this in detail. Section 5 summarizes the article.

2 Related Work

Work in parallel and distributed model checking can be divided into the categories of explicit state representation based and symbolic state representation based. We discuss the related works in Sections 2.1 and 2.2, respectively.

[1] In this document, we use "walks" and "random walks" interchangeably

2.1 Explicit state representation based

Work on explicit state model checking can be further classified into safety property checking and liveness property checking. We proceed to survey previous work in both categories below.

Safety property model checking. One of the initial works in the area of parallel and distributed model checking is by Aggarwal, Alonso and Courcoubetis [AAC87]. They presented a distributed reachability algorithm. The algorithm was not implemented and it's termination detection depended on timing assumptions. Kumar and Vemuri [KV92] proposed and implemented a distributed version of a reachability analysis algorithm. Their algorithm synchronizes after each breadth-first level and does not overlap communication and computation.

Stern and Dill [SD01] report the study of parallelizing the Murphi verifier [Dill96]. Parallel Murphi is a safety-only model checker. Whenever a state on the breadth first search queue is expanded, a uniform hashing function is applied to each successor state s to determine its "owner" - the node[2] that records the fact that s has been visited, and pursues the expansion of s. In [SD01], speedups of about 20 and 50 are reported for runs on 32 processors (Berkeley NOW) and 64 processors (IBM SP2), respectively.

In [LS99], a distributed implementation of the SPIN [Holz97] model checker, restricted to perform safety only model checking is described. They exploit the structure of SPIN's state representation and reduce the number of messages exchanged between the processors. Their algorithm is also compatible with partial order reduction, although the reported results do not include the effects of this optimization. They report results on examples like Bakery and Dining Philosophers running on up to four nodes on 300MHz machines with 64M memory.

Liveness property model checking. In [BBS01], the authors build on the safety model checking work of [LS99] to create a distributed memory version of SPIN that does LTL-x model checking. They have loss of parallelism when they parallelize the standard nested depth first search [CVWY92, HPY96] algorithm employed to detect accepting (violating) Büchi automaton cycles. This is to be expected because depth first search has been proved to be inherently sequential [Reif85]. Their paper reports feasibility (without actual examples) on a nine node 366MHz Pentium cluster.

In [BCKP01], Büchi acceptance is reduced to detecting negative cycles (those that have a negative sum of edge weights) in a weighted directed graph. This reduction is achieved by attaching an edge weight of -1 to all outgoing edges out of an accepting state, and a weight of 0 to all other edges. Despite the worst theoretical complexity, the authors report better performance. They treat examples such as Dining Philosophers on an eight node 366MHz Pentium cluster.

[2] In this document, we use "node" and "processor" interchangeably

2.2 Symbolic state representation based

The system presented in [HGGS00] carries out BDD based reachability analysis on a distributed platform. Their primary objective is to obtain the benefits of the large combined amounts of memory in a distributed context. They study various slicing heuristics for load balancing. The experiments are performed on up to 32 266MHz RS6000 machines, each with 256MB memory, connected by a 16Mb/s token ring network. Their distributed implementation could reasonably well utilize the available memory (the overhead being close to a factor of 3), and in one case reached 35 steps in the fixed-point iteration, compared to 18 steps on a uniprocessor with 768MB memory.

[Stor95, SB96, RSBS96] describe implementations of parallel BDD packages developed for a distributed environment such as a network of workstations (NOW). In [GHS01], a distributed symbolic model checking algorithm for the μ-calculus, its correctness proof, as well as sources of scalability are presented.

2.3 Random walk in model checking

In [West89], the author gives one of the first evidence that random walk method yields useful results when the size of the reachable state space is such that only a small fraction of the reachable states can be visited given the constraints on computational resources. The study yielded two encouraging results for using random walk. Firstly, the frequency of detecting errors was such that repeating the random walk runs using different random seeds will detect the majority of errors again, although the sequences exercised will be different. This shows that the results are reproducible which is not obvious from the probabilistic nature of the process. Secondly, the size of the sample required for a given coverage is related to the complexity of the errors we wish to find, and not to the overall complexity of the system, as measured by the total number of system states.

In [JS02], the authors present a parallel algorithm based on honeybee forager allocation techniques for finding violations of LTL properties. Their parallel algorithm uses idle CPU cycles of workstations and can tolerate workstations joining and leaving the group. The algorithm is geared towards finding errors and does not certify correctness. Tests conducted on a set of simple parametrized models indicate that a parallel implementation of this algorithm finds errors more quickly than uncoordinated parallel random walks.

In [MP94], the authors make the case that random walk could be used for effective testing by sampling accurately the state space of a family of protocols called the “symmetric dyadic flip-flops”. [Has99] presents an algorithm that is based on random walk to decide certain safety properties. It assumes that the reachable state space considered as a graph is Eulerian and also assumes knowledge of the number of vertices and edges in the graph. The algorithm is demonstrated on two simple examples.

3 MPI port of Parallel Murphi

The parallel version of Murphi written in Stanford used the Active Messages library for communication between the processing nodes. We ported the parallel Murphi to the MPI message passing library [Pach96]. This is for various reasons. Firstly, the parallel Murphi developed at Stanford was designed to run on a Myrinet network and we did not have access to such a network on our campus. MPI is a very widely used and highly portable message passing library. Since Murphi is also widely used both in the research and in the academic community, porting parallel Murphi to MPI would make it more usable. We also used this tool to show the feasibility of model checking very large models using distributed model checking algorithms.

Active message is thought of as a fast message passing library in that the time to handle an incoming message at a node is very small. That is not the case with MPI. Hence, when we ported the application to MPI, we had to build into it some flow control features (MPI does not provide flow control). Since in our model checking algorithm, any node can send a message to any other node (as determined by the uniform hashing partition function), it is possible that at any one time a node may receive messages from all the other nodes. In that case, its input message queue may get full and may cause it to crash. This may in turn lead to the crashing of the other nodes in a chain-like fashion. Flow control is needed to avoid this. So we stipulate that each node can have a fixed maximum number of outstanding messages destined to another node. It can send more messages to that destination only after some messages from the outstanding list have been received by the destination node. After implementing this heuristic, the number of crashes of the system reduced, confirming the need for flow control.

Using our parallel model checker, we were able to run industry level cache-coherence protocol examples. Table 1 gives the results of running some of our larger models, taken from our paper [CSG02]. The examples are implementations of the cache coherence protocols of the Alpha processor and the Itanium processor. We used sixteen 800MHz machines, each with 512MB RAM and connected by a 100 Mbps LAN to do the experiments. The MPICH [GLDS96] implementation of the MPI standard was used for message passing between the nodes. From the table we can see that we were able to verify models of size up to 1 billion states and about 3 billion transitions. To better understand these numbers, we compare them with the highest numbers reported in [SD01], namely for the Stanford FLASH and the SCI protocols, where only 1 million states were explored.

4 Heuristic Algorithms

In this section, we give the algorithms for the various random walk based heuristics that are being incorporated into our model checker.

These algorithms are designed to run on a cluster of workstations, each node having its own memory and processing unit. All the nodes run the same algorithm. At the start

	Alpha Implementation			Itanium Implementation		
Cache Coherent Protocol	States ($\times 10^6$)	Transitions ($\times 10^6$)	Time (hrs)	States ($\times 10^6$)	Transitions ($\times 10^6$)	Time (hrs)
Split Trans. Bus	64.16	470.52	0.95	111.59	985.43	1.75
Split Trans. Bus with Scheurich's Opt.	251.92	1794.96	3.42	325.66	2769.77	4.80
Multiple Interleaved Buses	255.93	1820.38	3.65	773.27	2686.89	10.97
Multiple Interleaved Buses with Scheurich's Opt.	278.02	1946.67	3.90	927.31	3402.41	12.07

Table 1: Experimental results obtained with MPI port of Parallel Murphi

of the run, one of the nodes is selected as the master node and the other nodes become the slave nodes. The pseudocodes given below contain only the core of the heuristic algorithms. The parts like the safety property violation checking, normalization of states, etc., are not shown here.

In the algorithms, each node maintains a hash table of states to record the fact that a state has been visited. This is needed to measure the total number of unique states visited by the algorithm. We want to measure this to get an estimate of the state space coverage of the heuristic which is one of the metrics of evaluation. Maintaining this hash table will lead to a lot of overhead since states have to transferred between the nodes to be added to the hash table. This results in a large number of message transfer. It may seem misleading to the reader when we say that random walks are a low overhead technique since we don't need to store the states already visited. It is to be noted that the hash tables are there only for getting empirical results and for benchmarking. In the final version of the tool the hash tables will not be used, hence cutting down on most of the message transfer.

Since by nature it is difficult to determine the stopping condition and the coverage estimate for random walks, we would like to give the user a lot of control over the model checker's execution. This means that the user has many configurable options for each heuristic. Most of the configurable parameters are entered as command line arguments. For some others, the program stops after a particular phase of the algorithm has been completed and asks the user to specify more values that will guide the next phase of the algorithm. There are options to control the resource consumption of the tool and also the amount of communication between the processing nodes. For each heuristic we give the algorithm and the various configurable options. The user will also be able to select the heuristic he wants to use as a command line argument. These parameters are in addition to what is provided by the standard Parallel Murphi.

We want to explore heuristics to better combine random walk and breadth first search in order to find hard-to-find bugs faster. We believe that any measure of effectiveness will have to take into consideration at least two things: the time taken to find "deep" bugs, and the memory used by the algorithm. Since this is a distributed memory application where the nodes communicate through message passing, we are also interested in the communication complexity of the algorithms. The tool will report on these results after each run.

4.1 Heuristic 1

Label: Pure multiple random walks

Just running n random walks in parallel is in itself a good debugging tool as compared to a single random walk. This is because n walks reach more states than one walk and hence will be able to catch bugs faster. Each node also maintains a hash table which is used to record the total number of unique states visited. Once a random walk reaches a particular state, it is hashed using a uniform hashing function and its owning node (the node that records the fact that the state has been visited) is calculated. The state is then sent to the owning node where it is stored in the hash table. At the end of the run, the sum of all the states in all the hash tables gives us the total number of unique states visited. This communication is configurable through parameters as explained below.

Configurable Parameters

- `Send_rw`. Boolean that specifies if the nodes are to send the states visited to the owning nodes. Setting this to `false` eliminates all communication in which states are transferred and hence the algorithm completes much faster.
- `Send_rw_interval`. Used only if `Send_rw` is `true`. If this is n, then send every n^{th} state visited by the random walk to its owning processor. By configuring this, the user can control the amount of communication during the run. Of course, this will affect the 'total unique states visited' statistic. Only if it is set to 1 will the user get an accurate value for the 'total unique states visited'. The user can configure this to get the desired tradeoff between the communication and the accuracy of the 'total unique states visited' statistic.
- `Total_initial_rw_steps`. Specifies the total number of steps to be taken by each random walk. A random walk is said to take one step if it chooses a next state uniformly at random from among the set of next states. Since each node does only one random walk, this gives the total number of random walk steps for each node. Increasing this obviously increases the computation time, but also increases the possibility of finding more new states and bugs. Of course that also depends on the state space size and the "density" of the state space.

The pseudocode for the heuristic is shown in Figure 1.

In this heuristic the walks may get stuck in a dense region of the state space without moving to other regions connected by a sparse region i.e. few number of transitions. It is also possible that a walk may pass "close" to a bug but may not catch it. The next heuristic tries to remove this drawback.

```
heuristic1:
   steps_so_far = 0;
   start a random walk from a randomly selected start state;
   while (steps_so_far <  Total_initial_rw_step)
   do
      generate next state;  // do one step of the random walk
      if (Send_rw == true)
         if (this is the Send_rw_interval'th state generated)
            Send state to owning node to be added
            to the owning node's hash table;
         endif
      endif
      steps_so_far++;
   end do
   master node marshalls results from other nodes and
   prints final result;
```

Figure 1: Pseudocode for heuristic 1: Pure multiple random walks

4.2 Heuristic 2

Label: random walks + bounded breadth first search from states visited by the random walks

In this heuristic, the master node does a breadth first search of a small depth from the start states to get a good spread of states across the state graph and also to minimize the probability that some random walks have the same initial path. The master then equally distributes the frontier states thus generated to the processing nodes. One random walk is started from each frontier node. The total number of steps for each random walk is calculated from the number of random walks for each node and the parameter `Total_initial_rw_steps`.

Each random walk, after visiting a state, calculates its owner by applying the uniform hashing function and sends that state to the owner (as an optimization, we maintain a cache of sent states so that we don't do a lot of redundant message transfer). Each node maintains a hash table and a queue of states. Upon receiving a state from another node, the node adds the state to the hash table and if it has not already been visited, the state is also added to the queue. We also do bounded breadth first search from select states from this queue.

The intuitive notion behind this idea is that the n random walks take us "closer" to the bug but may fail to detect them. But the bounded breadth first searches from the states visited by these random walks will be able to detect the bug.

Configurable Parameters

- `Fracqsize`. Each node maintains a hash table (HT) and a queue (Q) of states. This parameter gives the ratio of memory allocated for the Q to the memory allocated for the HT.

- `Init_bfs_depth`. This parameter specifies the depth of the initial breadth first search. Increasing this value will give us a larger set of frontier nodes, which will increase the number of random walks to be performed but decrease the total steps for each of those random walks.
- `Send_rw`. Same as in previous heuristic.
- `Send_rw_interval`. Same as in previous heuristic.
- `Total_initial_rw_steps`. This specifies the total number of random walk steps for each processor. If each node does n random walks, the total steps for each of those random walk is `Total_initial_rw_steps`$/n$.
- `Bounded_mem`. The bounded breadth first search needs its own hash table (BHT) and queue (BQ), separate from HT and Q maintained by the node. This parameter specifies the memory to be allocated for BHT. The ratio of the memory allocated for BQ to the memory allocated for BHT is `Fracqsize`.
- `Bounded_bfs_depth`. This parameter specifies the depth of the bounded breadth first search. Increasing this will increase the number of states visited by the bounded breadth first search and hence increase the possibility of finding bugs. But it will also increase computation cost. Decreasing this has the obvious opposite effect. The user can use this to effect the desired tradeoff.
- `Bounded_bfs_interval`. If a lot of states are being exchanged between the nodes (determined by the `Send_rw_interval` option), then we might not want to start bounded breadth first search for every state in Q. If this parameter is n, then bounded breadth first search is done only for every n^{th} state in the Q. So this parameter also influences the number of bounded breadth first search to be performed.

The pseudocode for the heuristic is shown in Figure 2.

4.3 Heuristic 3

Label: Initial random walks + bounded breadth first search from the states visited by the initial random walks, followed by random walks from the states visited by bounded breadth first search

In the previous heuristic, the walks may not be able to get out of a region if they get stuck there. This is because even if a bounded breadth first search from a state manages to get out of such a region, it just stops there and we don't have a search proceeding thereafter. In the previous algorithm, once we start the initial random walks from the frontier states of the initial breadth first search, we don't start new random walks. Here, we propose that we should start new random walks from some of the states visited by the bounded breadth first searches. These bounded breadth first searches are in turn started from some of the states visited by the initial random walks. In this case, if the initial random walks get stuck in a region and a bounded breadth first search gets out

```
heuristic2:
   if (this node is the master node)
      do initial BFS of depth Init_bfs_depth;
      distribute the BFS frontier states among the nodes;
   endif
   every node receives its share of the frontier nodes;
   // one RW is to be started from every frontier node received
   n = number of frontier states received;
   total steps for each random walk = Total_initial_rw_steps/n;
   while (there are more steps of random walk to perform)
   do
      // Here, we alternate between one step of the RW and
      // one bounded breadth first search, if that is possible
      do one step of random walk;
      if (Send_rw == true)
         if (this is the Send_rw_interval'th state generated)
            send state generated to its owning node
            to be added to the owning node's hash table;
         endif
      endif
      if (QueueTop(Q) is Bounded_bfs_interval'th state)
         do bounded BFS of depth Bounded_bfs_depth from QueueTop(Q);
         // The bounded BFS is done locally,
         // i.e., the states generated are not sent
         // to the owning nodes, since that would lead to a lot
         // of communication overhead.
      endif
      Dequeue(Q);
   end do
   master node marshalls results from other nodes and
   prints final result;
```

Figure 2: Pseudocode for heuristic 2: random walks + bounded breadth first search from states visited by the random walks

of the region, then another random walk started from one of the states visited by that breadth first search (which is "outside" the region) will take the exploration out of the dense region.

The algorithm is similar to the previous algorithm except that the frontier states generated by each of the bounded breadth first search is stored in an array. After the initial set of random walks and bounded breadth first searches is over, we show the user the total number of frontier states generated. We only record the frontier states because we want to start the final set of random walks only from the frontier states. The user is then asked to enter the values for the total number of final random walks and the total steps for the final random walks for each processor.

Configurable Parameters

- `Fracqsize`. Same as in previous heuristic.
- `Fracfrontsize`. The frontier states of the bounded breadth first search are stored in an array. We let the size of that array be a configurable fraction of the size of the HT. So this parameter is the ratio of the frontier states array size to HT size.
- `Init_bfs_depth`. Same as in previous heuristic
- `Send_rw`. Same as in previous heuristic.
- `Send_rw_interval`. Same as in previous heuristic.
- `Total_initial_rw_steps`. Same as in previous heuristic.
- `Bounded_mem`. Same as in previous heuristic.
- `Bounded_bfs_depth`. Same as in previous heuristic.
- `Bounded_bfs_interval`. Same as in previous heuristic.
- `Num_final_random_walks`. This parameter is requested from the user after the initial phase of the algorithm. Once the initial set of random walks and bounded breadth first searches are done, the tool displays the total number of frontier states that have been generated by the bounded breadth first searches. The user is then asked to enter the number of final random walks he wants the tool to perform. This will usually be a fraction of the total number of frontier states. The user can determine the fraction and enter an appropriate value.
- `Total_final_rw_steps`. This parameter is also requested from the user after the initial phase of the algorithm. This is the total steps of the final set of random walks for each processor. From this and the previous parameter, the total steps for each random walk can be calculated.

The pseudocode for the heuristic is shown in Figure 3.

```
heuristic3:
   if (this node is the master node)
      do initial BFS of depth Init_bfs_depth;
      distribute the BFS frontier states among the nodes;
   endif
   every node receives its share of the frontier nodes;
   // one RW is to be started from every frontier node received
   n = number of frontier states received;
   total steps for each RW = Total_initial_rw_steps/n;
   while (there are more steps of RW to perform)
   do
      // Here, we alternate between one step of the RW and
      // one bounded BFS, if that is possible
      do one step of RW;
      if (Send_rw == true)
         if (this is the Send_rw_interval'th state generated)
            send state generated to its owning node
            to be added to the owning node's hash table;
         endif
      endif
      if (QueueTop(Q) is Bounded_bfs_interval'th state)
         do bounded BFS of depth Bounded_bfs_depth from QueueTop(Q);
         store the frontier states in the frontier states array;
      endif
      Dequeue(Q);
   end do
   output total no. of frontier states;
   input Num_final_random_walks and Total_final_rw_steps;
   // from the above two values got from the user,
   // we can calculate the total no. of steps for each final RW
   do final set of RWs;
   master node marshalls results from other nodes and
   prints final result;
```

Figure 3: Initial random walks + bounded breadth first search from the states visited by the initial random walks, followed by random walks from the states visited by bounded breadth first search

4.4 Heuristic 4

Label: Initial random walks followed by second set of random walks from states visited by initial random walks

This heuristic is a special case of the previous heuristic in which we do a bounded breadth first search of depth 0, i.e. we start the final set of random walks from states already visited by the initial set of random walks (instead of from the frontier states as happens in the previous heuristic). This is more space efficient than the previous approach. A comparison of the effectiveness of this method against the previous method will tell us about the importance of bounded breadth first search. It would also be interesting to compare the state space coverage of this method with that of just running n random walks for the same number of steps (equivalent to Heuristic 2 with a bounded breadth first search depth of 0).

Like the previous algorithm, here also the user is asked to enter the values for the total number of final random walks and the total steps for the random walks for each processor. The difference is that since the final random walks are started from some of the states visited by the initial random walks, the user is shown the total number of states in the queue and not the total number of frontier states.

Configurable Parameters

- `Fracqsize`. Same as in previous heuristic.
- `Init_bfs_depth`. Same as in previous heuristic
- `Send_rw`. Same as in previous heuristic.
- `Send_rw_interval`. Same as in previous heuristic.
- `Total_initial_rw_steps`. Same as in previous heuristic.
- `Num_final_random_walks`. This parameter is requested from the user after the initial phase of the algorithm. Once the initial set of random walks are done, the tool displays the total number of states in the queue. The user is then asked to enter the number of final random walks he wants the tool to perform. This will usually be a fraction of the total number of states in the queue. The user can determine the fraction and enter an appropriate value.
- `Total_final_rw_steps`. This parameter is also requested from the user after the initial phase of the algorithm. This is the total steps of the final set of random walks for each processor. From this and the previous parameter, the total steps for each random walk can be calculated.

The pseudocode for the heuristic is shown in Figure 4.

4.5 Experimental evaluation of heuristics

In order to evaluate our heuristics, we need examples with bugs seeded deep in the state graph such that finding them through full state exploration will be expensive or

```
heuristic4:
   if (this node is the master node)
      do initial BFS of depth Init_bfs_depth;
      distribute the BFS frontier states among the nodes;
   endif
   every node receives its share of the frontier nodes;
   // one RW is to be started from every frontier node received
   n = number of frontier states received;
   total steps for each RW = Total_initial_rw_steps/n;
   while (there are more steps of RW to perform)
   do
      do one step of RW;
      if (Send_rw == true)
         if (this is the Send_rw_interval'th state generated)
            send state generated to its owning node
            to be added to the owning node's hash table;
         endif
      endif
   end do
   output total no. of states in the queue;
   input Num_final_random_walks and Total_final_rw_steps;
   // from the above two values got from the user, we can calculate
   // the total no. of steps for each final RW
   do final set of RWs;
   master node marshalls results from other nodes and
   prints final result;
```

Figure 4: Initial random walks followed by second set of random walks from states visited by initial random walks

not possible, necessitating the use of partial state space exploration techniques like the ones we are suggesting. Since such examples are difficult to come by in an academic setting, we propose to engineer such examples. These examples will have bugs seeded deep in them such that finding them by normal full state space exploration techniques will be expensive. These set of examples may serve as a good benchmark for evaluating any approximate search method. We will also evaluate our approaches on some real life examples taken from the standard Murphi distribution. The metrics of evaluation will be: bug hunting efficacy (time to find bug, fraction of bugs found, etc.), memory resource consumption, communication overhead, and state space coverage.

In full parallel searches, each node maintains a hash table to record the fact that a state has been visited. As a result, messages containing states are exchanged among the processors and this results in a lot of communication overhead. In the random walk based approach we don't maintain hash tables. This relieves us of the cost of maintaining hash tables and also significantly reduces the communication cost. Due to this it is not immediately obvious how the state generation rate of parallel full searches compares to that of parallel random walk based approaches. We propose to investigate this for large models. Empirical results will be added once the heuristics have been incorporated into the distributed model checker.

5 Summary

We have shown that parallel and distributed model checking can be used to alleviate the state explosion problem and that they can model check problems that are much larger than those that can be done using a single processor. We have also explained the use of multiple random walks as a useful debugging aid. We have described various heuristics that combine random walks with bounded breadth first searches to try to find deep-seeded bugs more efficiently. We are currently working towards incorporating these heuristics inside our distributed model checking prototype.

References

[AAC87] S. Aggarwal, R. Alonso, C. Courcoubetis, "Distributed reachability analysis for protocol verification environments", in *Discrete Event Systems: Models and Applications, IIASA Conference*, pp. 40-56, 1987.

[BAS02] A. Biere, C. Artho, V. Schuppan, "Liveness Checking as Safety Checking", in *7th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'02)*, Malaga, Spain, vol. 66, no. 2, Electronic Notes in Theoretical Computer Science, 2002.

[BBS01] J. Barnat, L. Brim, J. Stribrna, "Distributed LTL Model Checking in SPIN", in *8th International SPIN Workshop*, volume 2057 of Lecture Notes in Computer Science, Toronto, 2001. Springer-Verlag.

[BCKP01] L. Brim, I. Cerna, P. Krcal, R. Pelanek, "Distributed LTL model checking based on negative cycle detection", in *Proceedings of the FSTTCS Conference*, Bangalore, India, December 2001.

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L.Dill, "Sequential circuit verification using symbolic model checking", in *27th ACM/IEEE Design Automation Conference*, pp. 46-51, 1990.

[CGP99] E. Clarke, O. Grumberg, D. Peled, "Model Checking", MIT Press, 1999.

[CSG02] P. Chatterjee, H. Sivaraj, G. Gopalakrishnan, "Shared memory consistency protocol verification against weak memory models: refinement via model-checking", in *Computer Aided Verification*, Lecture Notes in Computer Science, Copenhagen, July 2002. Springer-Verlag.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, M. Yannakakis, "Memory Efficient Algorithms for the Verification of Temporal Properties", *Formal Methods in System Design*, vol. 1, pp. 275-288, 1992.

[CW96] E.M. Clarke, J.M. Wing, "Formal methods: State of the art and future directions", Tech. Rep. CMU-CS-96-178, Carnegie Mellon University (CMU), Sept. 1996.

[Dill96] D.L. Dill, "The Murphi verification system", in *Computer Aided Verification*, Lecture Notes in Computer Science, pp. 390-393, 1996. Springer-Verlag.

[GHS01] O. Grumberg, T. Heyman, A. Schuster, "Distributed Symbolic Model checking for the Mu-calculus", in *International Conference on Computer Aided Verification*, Paris, July 2001.

[GLDS96] W. Gropp, E. Lusk, N. Doss, A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard", in *Parallel Computing*, vol. 22, no. 6, pp. 789-828, September 1996.

[Has99] P. Haslum, "Model Checking by Random Walk", in *Proceedings of ECSEL Workshop*, 1999.

[HGGS00] T. Heyman, D. Geist, O. Grumberg, A. Schuster, "Acheiving scalability in parallel reachability analysis of very large circuits", in *12th International Conference on Computer Aided Verification*, volume 1855 of Lecture Notes in Computer Science, pp. 20-35, June 2000. Springer-Verlag.

[Holz87] G.J. Holzmann, "On limits and possibilities of automated protocol analysis", in *7th International Conference on Protocol Specification, Testing, and Verification*, pp. 339-44, 1987.

[Holz97] G.J. Holzmann, "The Model Checker SPIN", *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279-295, May 1997. Special Issue: Formal Methods in Software Practice.

[HP94] G.J. Holzmann, D. Peled, "An improvement in formal verification", in *Proceedings of Formal Description Techniques, FORTE94*, pp. 197-211, Berne, Switzerland, October 1994. Chapman Hall.

[HPY96] G.J. Holzmann, D. Peled, M. Yannakakis, "On Nested Depth-First Search", *Proceedings of 2nd SPIN Workshop*, Rutgers University, New Brunswick, N.J., DIMACS/32, American Mathematical Society, August 1996.

[IP96] C.N. Ip, “State Reduction Methods for Automatic Formal Verification”, PhD thesis, Stanford University, 1996.

[JS02] M. Jones, J. Sorber, “Using Idle Workstations to find LTL property violations”, Laboratory for Applied Logic Technical Report [FV-001], Brigham Young University, June 2002.

[KV92] N. Kumar, R. Vemuri, “Finite state machine verification on MIMD machines”, in *European Design Automation Conference*, pp. 514-20, 1992.

[LS99] F. Lerda, R. Sisto, “Distributed-memory model checking with SPIN”, in *6th International SPIN Workshop*, volume 1680 of Lecture Notes in Computer Science, Berlin, 1999. Springer-Verlag.

[MP94] M. Mihail, C.H. Papademitriou, “On the random walk method of protocol testing”, in *Proceedings of the 5th Workshop on Computer Aided Verification*, volume 818 of Lecture Notes in Computer Science, pp. 132-141, Stanford, June 1994. Springer-Verlag.

[NG02] R. Nalumasu, G. Gopalakrishnan, “An efficient partial order reduction algorithm with an alternative proviso implementation”, *Formal Methods in System Design*, vol. 20, no. 3, pp. 231-247, May 2002.

[Pach96] P. Pacheco, “Parallel Programming with MPI”, Morgan Kaufmann, 1996. ISBN 1-55860-339-5.

[Reif85] J.H. Reif, “Depth-first search is inherently sequential”, *Information Processing Letters*, vol. 20, no. 5, pp. 229-234, June 1985.

[RSBS96] R.K. Ranjan, J.V. Sanghavi, R.K. Brayton, A. Sangiovanni-Vincentelli, “Binary decision diagrams on network of workstations”, in *International Conference on Computer Design*, pp. 358-364, 1996.

[SB96] T. Stornetta, F. Brewer, “Implementation of an efficient parallel BDD package”, in *33rd Design Automation Conference*, pp. 641-644, 1996.

[SD01] U. Stern, D.L. Dill, “Parallelizing the Murphi Verifier”, *Formal Methods in System Design*, vol. 18, no. 2, pp. 117-129, 2001, (Journal version of their CAV 1997 paper).

[SD95] U. Stern, D.L. Dill, “Improved probabilistic verification by hash compaction”, in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pp. 206-24, 1995.

[Stor95] T. Stornetta, “Implementation of an efficient parallel BDD package”, Master’s thesis, UC Santa Barbara, 1995.

[West89] C.H. West, “Protocol Validation in Complex Systems”, in *Symposium proceedings on Communications architecture and protocols*, pp. 303-312, Austin, Texas, 1989.

[WL93] P. Wolper, D. Leroy, “Reliable hashing without collision detection”, in *5th International Conference on Computer Aided Verification*, pp. 59-70, 1993.