Abstract

We explore the use of distributed processing to enhance the performance of explicit state enumeration based safety model-checking. A central problem in distributed model-checking is checking whether a state generated by a processor has already been visited by another processor. This requires hash-table look-up messages to be exchanged. These messages can dramatically offset, or even nullify, the overall speed-up, in addition to complicating message-buffer allocation and flow-control. In this paper, we study the process of distributed random walk - a process of multiple processors randomly, and in an uncoordinated fashion, moving through the state-space without recording visited states. The probability that a random-walk repeats the same sequence of moves can decrease exponentially with the length of the sequence. In addition, distributed random-walk is amenable to distributed systems with low memory availability per node as well as low network bandwidth. Heuristic combinations of breadth-first search (BFS) and random-walk (RW) are natural choices to explore because BFS requires higher amounts of memory to maintain queues, but guarantees to find the shortest path to a state – while RW has opposite characteristics. Studying these heuristic methods initially on synthetic benchmarks can offer sharper insights which can help improve the heuristics. In this paper, we develop three heuristic combinations of RW and BFS and study their performance on synthetic as well as a few realistic benchmarks on a FreeBSD machine cluster connected using 100Mbps Ethernet and using the MPI distributed programming library.

1 Introduction

Explicit state enumeration based model-checking tools play an important role in the formal verification of protocols appearing in diverse domains, such as cache coherence and embedded software. The sheer size of models in these domains often implies that designers are often satisfied if only they can formally verify a handful of safety properties. In the setting of realistic projects with ever-changing specifications, even this can be a tall order. In this paper, we address how safety property verification can benefit from distributed processing.

When multiple processors expand the state graph in parallel, each newly generated state has to be checked against the states already generated by all the processors for possible repetition, so that redundant searches are avoided. This is commonly achieved by distributing the state hash-table across processors, and generating look-up messages. If we use a uniform hashing function to distribute the states among the nodes, this leads to a high message rate. If the message-handling software library and the physical medium itself cannot keep up with this high message exchange rate, the performance of the model-checker will be severely affected. We want to explore the following questions:

---

*This work was supported in part by the Semiconductor Research Corporation under Contract 1031.001, and by the National Science Foundation (NSF) under ITR Grants CCR-0081406 and CCR-0219805. 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.
(i) How can distributed safety verification be supported by explicit enumeration tools without maintaining distributed hash-tables? (ii) How can the demands on memory and network bandwidths be kept modest while achieving a high degree of speed-up, without losing coverage by much? These questions are especially relevant, considering basic facts about processor / cache / main-memory / network speed ratio trends (processors are becoming faster, faster than main memories are, etc.).

In this paper, we report our work based on distributed random walk as a possible means of achieving these goals. In this work, we focus exclusively on error detection. By “random walk,” we mean the following process: pick an initial state, and select a random next move; compute the next state and set the current state to it; continue until a safety violation is encountered or some other stopping criterion is reached. The attractive features of this process that are intuitively clear, and we hope to establish through ongoing experiments\(^1\), are:

- No states are to be saved, thus allowing the process to operate on a large cluster of computers with extremely low memory sizes.
- The networks used can have very low bandwidths sufficient to achieve occasional coordination during execution as well as error-trail generation.
- The probability of random-walk repeating the same sequence of moves can decrease exponentially with the length of the sequence. This guarantees that new states will be explored fairly frequently.
- Multiple random-walks running within a CPU as well as in parallel across multiple such CPUs, each based on different random next-state selection functions, can ensure a high rate of new-state generation.

We propose random walk as a way to detect bugs fast - and not to replace exhaustive verification. That said, the state of model-checking vis-a-vis the exhaustive verification of finite-state models is not that rosy either. Modern cache coherence protocols that are on the drawing boards today cannot be exhaustively verified using today’s best explicit-enumeration or BDD-based tools even for modest configurations such as four CPUs interacting over two cache lines. Distributed random-walk can save enormous amounts of upfront debugging effort that can allow designers to use thoroughly debugged models as starting points when performing model optimizations to achieve full coverage.

Also, as noted by West [2], errors seldom occur at exactly one state. Specifically, if a property \( p \) is violated in state \( s \), it is very likely that \( p \) will also be violated in many other states \( s' \) that are different from \( s \) in bits that do not affect \( p \). This increases the likelihood that random walk will detect the violation. Breadth first search and random walk are at two ends of a spectrum — while breadth first search is memory intensive and finds the error for sure, random walk is very light on the memory but does not give the guarantee of finding the error. In addition, both are inherently parallelizable. It therefore appears very attractive to study the combinations of multiple random walks and breadth first searches.

2 Heuristics Explored

This section explains the new heuristics briefly. For a full description, including pseudo-codes, configurable parameters, etc., please see [1]. Figure 1 portrays the four heuristic search methods that we have experimented with so far.

**Heuristic 1: Pure multiple random walks:** In this scheme, one random walk is run per processor. Just running \( n \) random walks in parallel is in itself a good debugging tool as compared to a single random walk. In this heuristic the walks may get stuck in a dense region of the state space, unable to cross over to other similar dense regions through

\(^1\)Additional details, including source codes, a technical report with the full list of references, etc., are kept on our web-page http://www.cs.utah.edu/formal_verification/techcon03.html.
articulations. It is also possible that a walk may ‘graze’ past a bug without hitting it\(^2\). The next heuristic attempts to remove these drawbacks.

**Heuristic 2: random walks + bounded breadth first search from states visited by the random walks:** This

![Diagram](image)

Figure 1: Heuristic searches 1, 2, and 3 combining distributed RW and BFS (Heuristic 4 is a special case of Heuristic 3, with the depth of the secondary BFS being 0)

heuristic combines random walk and breadth first search in the following manner. The master node begins the whole process by performing a breadth first search of a user-specified depth from the start states to obtain a reasonably good spread of states across the state graph. The frontier states resulting from the initial BFS are ‘cut up’ and distributed equally among the processors. Each processor \(i\) that receives a piece of the BFS frontier does the following in parallel with the other processors. The user would have (before the whole process begins) specified the number, \(N\), of random walk steps for each processor. So, if \(Q_i\) has \(q\) states, then processor \(i\) runs \(q\) random walks for \(N/q\) steps each. Each random walk begins at a different state in \(Q_i\). Every \(q\)th state of each random walk is sent to its home node. We call these the potential secondary BFS start states. Each processor has the option to ignore many of the potential secondary BFS start states sent to it. For each such state, it performs a bounded BFS of depth \(D\). Figure 1 portrays these details of Heuristic 2. This heuristic forces the search to occasionally sample a small radius “halo” surrounding the main locus of the random walks. Note that all the bounded BFSs run sequentially, without incurring message overheads. These advantages do show up in our final results.

**Heuristic 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:** Even with bounded BFS envelopes developing from the main locus of a random-walk, it is likely that searches based on Heuristic 2 can become stuck in tightly connected regions of a state-space. To circumvent this problem, in Heuristic 3, additional secondary random-walks are started from some of the states visited by the bounded breadth first searches. See Figure 1 for a picture.

**Heuristic 4: Initial random walks followed by second set of random walks from states visited by initial random walks:** This heuristic is a special case of Heuristic 3, with the depth of the secondary bounded breadth first searches set to 0, i.e., we start the secondary random walks from states already visited by the initial set of random walks (instead of from the frontier states of the secondary BFSs). This is more space efficient than the previous approach. A comparison

\(^2\)All these discussions assume that the critical limiting resource is the verification time available to explore states.
of the effectiveness of this method against the previous method will tell us about the importance of the bounded breadth first searches. 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).

3 Experimental Results

The experimental setup consists of ten 850MHz PCs, each with 512MB RAM and connected by a 100 Mbps LAN. MPI was used for message passing between the nodes. The heuristics were tested on three examples. Two of them are from the benchmark examples developed at BYU (see http://lal.cs.byu.edu/~tonga). The third example is a distributed locking protocol.

We perform 5 runs of each experiment and record the number of misses, i.e., the number of times the heuristic fails to find an error. Below is the summary of results after running the heuristics on the examples:

- From using parallel BFS to using the heuristics, we see a vast reduction in the number of messages exchanged.
- Heuristics 2, 3, and 4 increase the number of messages slightly (Heuristic 1 does not exchange any messages!) while reducing the number of bug misses and reducing verification times.
- In many cases, using Heuristics 3 and 4 either reduces the time taken to find an error state or decreases the number of misses as compared to Heuristics 1 and 2. This shows the efficacy of spawning secondary random walks.
- Even within one heuristic, increasing the number of processors has the obvious effect - it either decreases the time to find an error state or reduces the number of misses.
- Heuristic 4 seems to be better than Heuristic 3 (which means that spawning secondary BFSs does not seem to help much).
- One of the examples has a 'state-space constriction'. The efficacy of Heuristics 2, 3, and 4 in crossing over the constriction is also observed.

4 Conclusions

We have begun a promising series of experiments that show that distributed random walk, when combined with BFS, can help rapidly find bugs even on models with extremely large state-spaces. One exciting experiment that we hope to perform in collaboration with our industrial collaborators who also use Murphi is to re-run their earlier found buggy models and compare the debugging times achieved using our techniques against the time they had to spend using sequential Murphi. One may obtain all details, including source codes of our tool, by contacting the authors.

References
