Abstract. Recent advances in parallel model checking for liveness properties achieve significant capacity increases over sequential model checkers. However, the capacity of parallel model checkers is in turn limited by available aggregate memory and network bandwidth. We propose a new parallel algorithm that sacrifices complete coverage for increased capacity to find errors. The algorithm, called BEE (for bee-based error exploration) uses coordinated depth-bounded random walks to reduce memory and bandwidth demands. A unique advantage of BEE is that it is well-suited for use on clusters of non-dedicated workstations.

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Parallel Search for LTL Violations

Michael D. Jones and Jacob Sorber*

Department of Computer Science
Brigham Young University
Provo, Utah, USA
e-mail: jones@cs.byu.edu

The date of receipt and acceptance will be inserted by the editor

Explicit state enumeration model checking consists of searching for errors while generating all of the reachable states in a formal model. Errors can be either violations of safety properties or liveness properties expressed in a temporal logic such as linear temporal logic (LTL). A table of visited, or generated, states is kept in order to detect when no more new states can be generated. As might be expected, the capacity of a state enumeration model checker is limited by the number of states that can be stored in the table of visited states. Even worse, performance degrades quickly if the table of visited states does not reside in random access memory.

Large design organizations already own hundreds of workstations placed on the desks of individual designers, verification engineers and administrative staff. Because these workstations are networked and rarely fully utilized, they can be used to form a dynamic cluster of non-dedicated workstations. Parallel algorithms for this environment must be fault-tolerant (as workstations enter and leave the algorithm) and consume few resources per workstation (in order to be "good neighbors" for host processes).

Since the table of visited states is the limiting factor in model checking, recent advances in parallel model checking algorithms for liveness properties achieve significant capacity increases by partitioning and distributing the table (or representation) of visited states [GHS01, BCKP01]. Partitioning and distributing the visited states is best suited for static clusters of dedicated workstations.

Partitioning algorithms can not tolerate workstations entering and leaving the search because part of the table of visited states is lost when a workstation leaves the search. This portion of the state table must be either regenerated at another node or partitioned among the remaining nodes. When a node enters the search, the states must be repartitioned to accommodate the new node. Repartitioning the states requires either sending states over the network or regenerating the nodes. Finally, partitioning algorithms consume as much memory as possible on each workstation. If only a small amount of memory is used per workstation, then many workstations will be needed to perform the search and the search will migrate more often. In the asymptotic worst case, each workstation will store a single state and the search will have to migrate with every transition.

The bee-based error exploration (BEE) algorithm is designed to operate in the non-dedicated parallel computing environment. The design of the BEE begins with the decision to eliminate the global table of visited states in favor of scalability and fault-tolerance. Eliminating the table means that the BEE can not be used to certify the correctness of formal models and that BEE is *not* a model checker. Unlike a model checking algorithm, the BEE algorithm can not ensure exhaustive coverage of the reachable states.

The justification for this decision is that quickly finding errors in formal models is at least as important as certifying the correctness of formal models. The resulting tool can only be used to prove incorrectness—not correctness. Given this decision, the problem is to converge on errors as quickly as possible. The BEE algorithm does this by employing a decentralized coordination scheme inspired by a social behavior of honeybee colonies. In the non-dedicated parallel computing environment, for which BEE is designed, the simplicity of loosely coordinated random walks is an advantage because the algorithm tolerates node failures and presents a small per-node footprint.

The cooperative social behavior of honeybees provides an excellent inspiration for designing such a coordination scheme for use in a parallel search. In honeybee colonies, useful collective behavior emerges from the coordinated efforts of many simple (compared to the prob-

* Supported by a grant from the Intel Corporation

lems solved by the colony) individuals. The forager allocation behavior of honeybee colonies is particularly well-suited as a foundation for parallel LTL search. Forager allocation involves identifying flower patches and allocating foragers to forage for resources at the patches. In LTL search, flower patches map to accept states and foraging maps to finding cycles that contain accept states. The resulting algorithm searches for accept states, then allocates workstations to forage for cycles beginning at accept states.

The BEE algorithm has been implemented as an extension of the Murϕ model checker [Dil96] and deployed on a network of 64 non-dedicated workstations located in student computing laboratories at Brigham Young University. The BEE algorithm was compared with uncoordinated random walk to demonstrate the value of the coordination scheme. Experiments were also conducted on static and dynamic workgroups and under different load conditions. The BEE algorithm found errors faster in models with one or few accept states while uncoordinated random walk found errors faster in models with many accept states.

The next section discusses related work. Section 2 describes the forager allocation scheme as used by honeybee colonies. In Section 3, the forager allocation scheme is adapted to search for LTL violations. Section 4 contains experimental results. We conclude and offer ideas for further work in Section 5.

1 Related Work

The BEE algorithm is a biologically inspired system. The most closely related work in biologically inspired systems is the ant colony optimization (ACO) metaheuristic (see [DCG99] for details). ACO algorithms have been successfully applied to computationally complex combinatorial optimization problems. The most successful of these applications solves dynamic routing problems in a changing network [CD98]. From an information processing perspective, the main difference is that ACO uses shared variables to communicate by depositing pheromone in a shared location while BEE uses undirected multicast by performing waggle dances.

Interest in distributed model checking for a wide variety of properties has increased over the past few years [SD97, LS99, GHS01, **?**, BCKP01]. While many distributed model checkers have been implemented, none of them are designed explicitly to find errors using dynamic workgroups.

Four distributed model checkers can be used to find violations of LTL properties. Barnat et. al.'s [**?**] distributed version of SPIN requires the nested DFS to be sequentially scheduled. This algorithm reduces to a sequential search for cycles in a distributed state table. Brim et.al.'s [BCKP01] distributed algorithm is designed specifically for LTL model checking and uses negative cycle detection to avoid a sequential search for cycles. Lafuente's algorithm partitions the state space such that all members of strongly connected components induced by the property under test (the role of strongly connected components in LTL model checking is discussed in more detail in Section 3.1) are assigned to the same machine. This partitioning simplifies the scheduling of the second DFS. Grumberg et. al.'s [GHS01] distributed symbolic model checker for the μ-calculus scales to hundreds of machines but partitions the symbolic state representation across nodes. All three of these algorithms partition the table of visited states (or BDD representing reachable states in the case of [GHS01]).

The BEE algorithm is neither a distributed model checker nor designed for static workgroups. The BEE algorithm is intentionally designed to find violations of LTL properties rather than certify conformance to LTL properties. This decision obviates the partitioned table of visited states and allows a less constrained parallel platform.

Similarly, the ASTRAL model checker [DK00] and the bitstate hashing option in the SPIN model checker [**?**] are designed to locate violations rather than certify correctness. Although ASTRAL is designed to find errors in real time systems (rather than discrete state systems), the decision to find violations opens possibilities for algorithms that are less computationally intensive but not guaranteed to visit every state. ASTRAL has been extended to use random walk (and two other weaker methods not applicable here). For a model with seeded errors, ASTRAL with random walk found shallow errors (errors near the initial state) easily and deep errors (further from the initial state) with more difficulty. As might be expected, the BEE algorithm is also better at finding shallow errors than deep errors.

We have included a Butterworth filter to control the depth of BEE random walks. A Butterworth filter is a band-pass filter used in signal processing. In the context of random walks, the Butterworth filter describes the probability of backtracking to a previously visited state. Butterworth filters and their use in BEE are explained in more detail in Section 3.2. The filter can be set to focus the search on shallow or deep walks.

In SPIN, bitstate hashing reduces the amount of memory required for each state at the cost of complete coverage. During the search, each state is represented by one or more bits at different locations in the table of visited states. If more than one state hashes to the same locations in the table, then all of the states represented by those bits are assumed to have been visited when the first such state is visited. Bitstate hashing is well-suited for models in which only a few states hash to the same table locations and there is a high degree of connectivity

[0] Albeit in infinite real time systems rather than finite state models. Nevertheless, the use of random walk is similar

such that if one predecessor of a state is never visited, the state is reached through another predecessor. For the models tested here, bitstate hashing either failed to find an error or required more time and space to find an error than a single random walk. A more detailed description of the comparison between bitstate hashing and random walk is given later in Section 4.4.

Like ASTRAL's random walk, it is difficult to compute coverage on the BEE's random walk. In contrast, a lower bound on coverage using bitstate hashing is relatively easy to compute. Random walks for model checking with coverage estimates have been given in [Has]. The class of transition systems for which these coverage estimates hold is severely limited. Our approach was to forgo good coverage models in favor of a broader class of transition systems.

2 Adapting Honeybee Forager Allocation

In this section, relevant points about the bee forager allocation (BFA) system as it appears in nature are described. In the next section, BFA is adapted to parallel search for LTL violations. All of the following information about BFA in nature has been reported by Seeley et. al. in [SCS91,See95,SV88].

Seeley et. al. give an information flow model describing the behavior of an individual forager. This model is used by Seeley to derive a simple mathematical model that correctly predicted the behavior of a colony in an experiment. The forager allocation problem is described first, followed by a description of the BFA system.

2.1 The Forager Allocation Problem

Given a pool of unemployed foragers and profitability information about a time-varying collection of foraging sites, the forager allocation problem is the problem of assigning foragers to sites in a way that maximizes the accumulation of resources for the colony.

2.2 Bee Colony Forager Allocation System

In the absence of a leader, the BFA mechanism emerges from the behavior of individual foragers. Each of these individuals posses only limited information about the global allocation of labor and the need and availability of resources. Despite the limited perception of each individual, the resulting allocation always accumulates resources for the colony at no less than half the optimal rate [ISTV93].

The first task for a given forager is to become recruited to a foraging site. Recruitment happens on the so-called "dance floor". The dance floor is an area in the hive, near the hive entrance, at which foragers advertise the location of foraging sites using a waggle dance. The unemployed forager wanders the dance floor more or less at random until encountering a waggle dance. The probability of encountering a waggle dance for site σ is given by $f_f(\sigma)$. The duration and intensity of waggle dances, which depend of the quality of a site, determine the value $f_f(\sigma)$ (as explained later). In almost all cases, the forager follows the first waggle dance she encounters. After following a waggle dance, the forager immediately exits the hive and begins foraging at σ.

The forager then returns to the hive to unload. After unloading, the forager selects one of three possible actions: abandon the site, perform a waggle dance and return to σ, or return directly to σ. The probability of abandoning σ is given by $f_x(\sigma)$. If the forager abandons σ, then the forager rejoins the pool of unemployed foragers wandering the dance floor. If the forager does not abandon σ, then the forager will perform a waggle dance to advertise σ with probability $f_d(1 - f_x(\sigma))$. Whether or not the forager performs a dance, the forager then returns to σ.

The values of $f_x(\sigma)$ and $f_d(\sigma)$ are determined by the forager's perception of the colony's demand for the resource at σ. It is believed that the forager perceives this need by observing indirect indicators of demand, such as the amount of time needed to find an unloader for the resource.

If a resource is abundant at σ and needed, then the dance will be long and intense. Otherwise, the dance will be short and pallid. The duration of the dance can vary by as many as two orders of magnitude. Since the length of a dance is proportional to the perceived quality of and need for the resource a site, the probability that an unemployed forager will encounter a dance for site σ, denoted by $f_f(\sigma)$ above, is greater for sites with greater quality and need. If D_σ is the number of dances currently being performed for site σ and d_σ is the duration of the dance for σ, then probability of encountering a dance for σ is given by:

$$f_f(\sigma) = \frac{D_\sigma d_\sigma}{\sum_{i=1}^{n} D_{\sigma_i} d_{\sigma_i}} \tag{1}$$

where n is number of currently known foraging sites.

If an unemployed forager is not recruited to a foraging site by a waggle dance, then that forager may become a scout bee and leave the hive in search of a new foraging site σ'. Upon returning to the hive, the scout performs the same evaluation of σ' described above. If the resource at σ' is in demand, the scout performs a waggle dance advertising σ' and σ' becomes one of the σ_is in Equation 1 above (with the sum taken over $n+1$ to include the new site).

3 Adapting BFA to LTL Violation Discovery

BFA can be simplified into a two part process: scout bees locate and advertise foraging sites then forager bees

	BFA	LTL Violation Search
Goal	accumulate resources	locate violations
Forager	*Apis mellifera*	BEE process
Foraging site search	not known	bounded random walk
Foraging site	Flower patch	Accept state
Quality of a site	Pollen composition	Distance from start state
Communication	Waggle dance	TCP/IP
Site advertisement	Distance and direction	Trace from start state

Table 1. Adaptation of BFA to parallel search for LTL violations.

observe advertisements and gather resources. Similarly, checking the non-emptiness of a Buchi automata using double depth-first search (DFS) is a two part process: a DFS locates an accept state and another DFS locates an cycle containing the accept state. The adaptation of BFA to LTL violation search follows directly from this similarity, and is summarized in the table shown in Table 1[1]. The next section gives a detailed description of the resulting algorithm, after recalling the reduction of LTL model checking to language containment on Buchi automata.

3.1 Introduction to LTL Model Checking

This section contains a more detailed description of the model checking problem for LTL. In the interest of readability, notation and formalism is kept to a minimum. For a more detailed introduction, see [CGP00]. On first reading, it may be useful to remember that checking the non-emptiness of a Buchi automata is important and that checking the non-emptiness of a Buchi automata requires finding an accept state contained in a cycle. The details below may be absorbed in subsequent readings.

The LTL model checking problem is the problem of proving that a transition system does or does not satisfy an LTL assertion. An LTL assertion is a predicate built from atomic propositions, the usual boolean connectives (such as "and", "or" and "not") and temporal quantifiers. Atomic propositions describe properties that are satisfied or violated in a given state. The temporal quantifiers describe properties of sequences of states. For any LTL formula p, the temporal quantifiers are:

- always p, which means p is true in all future states,
- eventually p, which means p is true in at least one future state, and
- p until q, which means p is true in every future state until q is true in at least one future state.

A few common LTL formulae and their meaning are given below:

- (Always p) implies (Eventually q): means that a continuous request p is eventually granted a response, q.
- (Eventually p) until (q): means request p is periodically repeated until a response q is received.
- p implies (q until r): means if p becomes true, then q must remain true until r becomes true.

The usual method for checking LTL properties is to rephrase the problem as a language containment problem on *Buchi automata*. A Buchi automata is a finite state automata that accepts only infinitely long words. Buchi automata are defined like normal finite state automata, but with a different acceptance condition. Finite state automata accept a word w iff using w as input causes the automata to halt in an accepting state. Buchi automata accept a word w iff using w as input causes the Buchi automata to pass through an accepting state infinitely often. More formally, let $inf(w)$ be the set of states that are visited infinitely often on input w and A be the set of accepting states, then a Buchi automata accepts w iff $inf(w) \cap A$ is non-empty.

The reduction of LTL model checking to an emptiness check on Buchi automata is done by translating the transition system and an LTL assertion into two separate Buchi automata, T and P respectively. Then T is intersected with the negation of P to create a new automata, $I = T \cap \overline{P}$. The language of the automata I contains behaviors of T that violate P. If the language of I is empty, then T does not violate P. Otherwise, the language of I contains only violations of P by T. We can check that the language of I is empty (i.e., T is correct with respect to P) by checking that I contains no accept states in cycles.

The search for accept states contained in cycles is usually done using Tarjan's double depth-first (DDFS) search algorithm shown in Figure 1. The $dfs1$ search locates accept states and the $dfs2$ search looks for cycles containing the accept states found during $dfs1$. In $dfs2$, cycles are detected when a state on the dfs1 stack is located. A hashtable of visited states prevents states from being expanded more than once. The capacity of an LTL model checker is limited by the number of visited states that can be stored in memory.

[1] With the caveat that pollen composition is just one of several plausible indicators of quality discussed by Seeley [See95]. Other plausible indicators include an individuals glucose level and the wait time to find a worker to unload the resource. In every case, the indicator is a single individual's observation of indirect metrics.

```
Procedure Search
  dfs1(start-state)
  exit (empty)

Procedure dfs1(s)
  add s to hashtable
  foreach s', a successor of s, do
    if s' not in hashtable then dfs1(s')
  if accept-state(s) then dfs2 (s)

Procedure dfs2(s)
  mark s in hashtable
  foreach s', a successor of s, do
    if s' in dfs1Stack then exit(nonEmpty)
    if s' not marked in hashtable then dfs2(s')
```

Fig. 1. The double depth-first search (DDFS) algorithm for checking the non-emptiness of the language of a Buchi automata.

3.2 The BEE algorithm

The BEE algorithm consists of coordinated parallel random walks and a recorder process. Pseudocode for the recorder and random walks is given in Figure 2. Line numbers in the following description refer to lines in Figure 2. The recorder process initiates the search by issuing several first-walk requests to the workstations in the user-defined start-set (line 2). The recorder process then receives and reports errors until receiving a stop command from the user (line 4). After being stopped, the recorder sends the stop command to every participating workstation in the node-set (line 9). The node-set should include all workstations that have the potential to participate. Since the recorder does not know which workstations are actively participating, the stop command is sent to every potentially participating workstation.

The first random walk attempts to locate an accept state in the Buchi automaton representing the system under test intersected with the negation of the property being checked. The first walk is depth-bounded (as explained shortly) and retains the path to the current state. If the first walk locates an accept state, the BEE process assesses the quality of the accept state and then sends its location to a fraction of the other known BEE processes (line 19). If the first walk exhausts available memory without finding an accept state, the first walk terminates and returns an error that is reported to the user. This fraction is proportional to the quality of the accept state. The quality of an accept state is inversely proportional to its distance from a start state, since short error traces are preferable to long error traces (line 20).

When an accept state is advertised to another process, the "finder" process first sends a query to several "forager" processes. The query asks the forager if it has the capacity to begin searching at a new accept state. The forager can either decline or accept the query. If the forager accepts the query, then the finder sends the entire trace leading to the accept state. The forager begins a second walk at the accept state after receiving the trace. The details of this negotiation process are omitted from the pseudocode for clarity.

The second search begins at the accept state and attempts to locate a path back to the accept state. If a path back to the accept state is found, the BEE process sends the entire trace back to a recorder process (line 33). The recorder process notifies the user and writes the trace to a file. Because several second searches may be done concurrently, the recorder processes may receive many error traces (including duplicates) before the user terminates the search.

Both the search for accept states and the search for paths back to an accept state use depth bounded random walk to traverse state graphs. At each state, a decision is made to backtrack or explore a randomly chosen successor (lines 16 and 30). When the decision is made to backtrack, the new current state is chosen randomly from the current search stack. In the second walk, the search stack, $S2$ of line 28, contains only the states between the initial accept state and the current state. If the probability of backtracking is $Bt(depth)$ and there are n successors of a given state, then the probability of choosing a particular successor is

$$\frac{1 - Bt(depth)}{n}.$$

The distribution of probability in next state computation is shown in Figure 3. From a given state, the search can either backtrack (shown using the curved up arrow) with probability $Bt(depth)$ or continue to a successor with the remaining probability split over the successors (shown using the straight, down arrows).

In our depth bounded random walk, $Bt(depth)$ is computed using a version of the shallow-pass Butterworth filter. The shallow-pass Butterworth filter is analogous to a high-pass filter in signal processing which allows high-frequency signals to pass through. The shallow-pass filter allows shallow walks into the transition graph, but filters deep walks. For BEE processes, the filter is

$$Bt(depth) = \frac{pmax}{1 + \left(\frac{dt^2}{depth^2}\right)^{\frac{dt}{2 \times dx}}} \tag{2}$$

where $pmax$ is the greatest probability of backtracking, dt is the depth at which $Bt(depth) = pmax/2$ and dx controls the steepness of the filter. The relationships between dt, dx and $pmax$ are shown in Figure 4. In the figure, $dt = 50$, $dx = 10$ and $pmax = 1.0$. Small values of dx (relative to dt) concentrate the search around depth dt.

The Butterworth filter gives a wider distribution of random walk depths than a square edge filter. The square edge filter cuts off all walks at a preset depth. A wider distribution of walk depths reflects uncertainty regarding the exact depth of a violation. If the depth of an error

```
1   Process recorder (M: model,start-set, node-set: set-of-compute-nodes)
2   for each workstation in start-set send (workstation, first-walk-request (M))
3   endfor
4   while (not stopped)
5     wait for error-message
6     receive (error-message (S2,S1))
7     report error and trace (S2 append S1)
8   endwhile
9   for each workstation in node-set send (workstation, stop)
10
11    Process first-walk
12    wait for first-walk-request
13    receive (first-walk-request (M));
14    x = initial-state(M); S = x;
15    while(not receive (stop))
16      if (Bt (depth(S))) then x = random-element (S)
17      x = random-next-state (x, M);
18      S = S append x;
19      if (accept-state (x))
20        target-set = select (N * quality(x)) other nodes
21        for each target in target-set send (target, second-walk-request (M, S,x))
22        endfor
23    endwhile
24
25    Process second-walk
26    wait for second-walk-request
27    receive (second-walk-request (M,S1,x))
28    S2 = x
29    while(not receive (stop))
30      if (Bt (depth(S2))) then x = random-element (S2)
31      x = random-next-state (x, M);
32      S2 = S2 append x;
33      if (x in S1) send (recorder, error (S2, S1))
34    endwhile
```

Fig. 2. Three algorithms used in BEE.

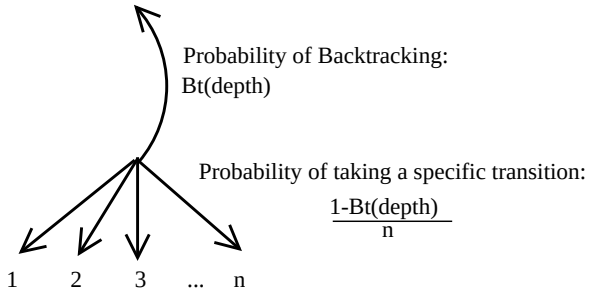

Fig. 3. Distribution of probability in next state computation.

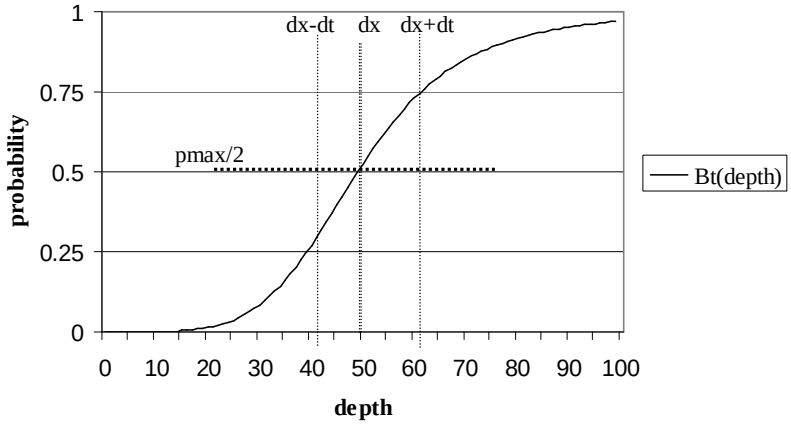

Fig. 4. Anatomy of a Butterworth filter.

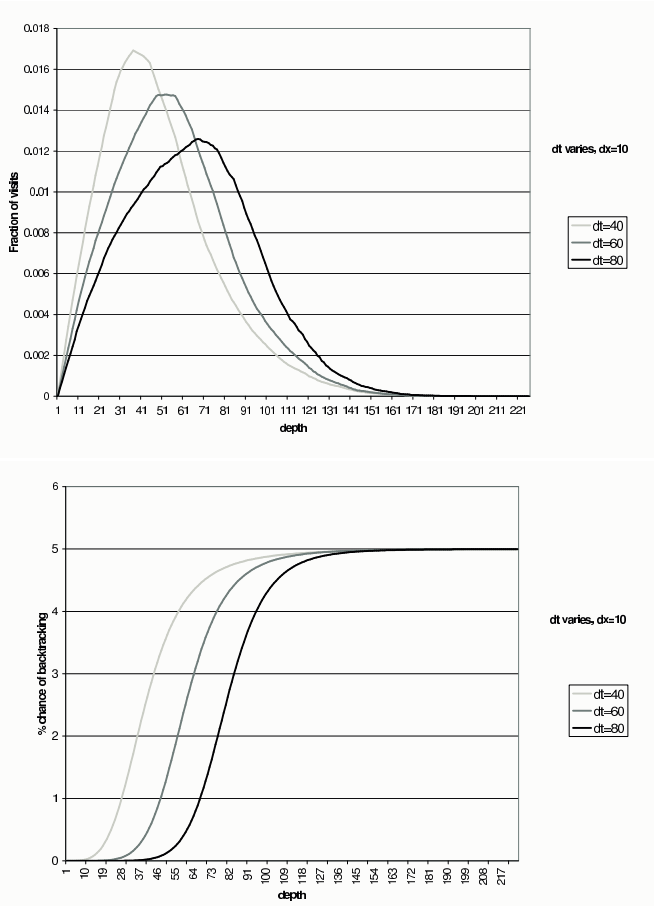

Fig. 5. Visit distributions and backtracking probabilities.

is precisely known, then simpler exploration algorithms targeting that specific depth would be more appropriate.

The graphs in Figure 5 show the distribution of visit depths for three filters on the same model. The value of *pmax* is 0.05 for each graph. As can be seen in Figure 3, values of *pmax* greater than 0.05 tend to backtrack too often because the remaining probability of choosing a successor is divided between all successors. The top graph shows the fraction of visits performed at each depth and the bottom graph shows the probability of backtracking as a function of search depth. The fraction of states visited at each depth is the number of unique states visited at a given depth divided by the total number of states at that depth. Depth is measured as the length of the shortest path to a state from the start state. As *dt* moves to the right on the horizontal axis, the concentration of searched states moves with it.

3.3 Communication Overhead

The biological process on which BEE is based requires little communication. This is also true for BEE—when the verification problem under analysis matches the biological environment. If there are many accept states, communication overhead can begin to dominate the BEE algorithm and the process spend their time communicating rather than finding errors.

Communication is required to initialize the search and to advertise accept states. Initializing the search involves discovering other workstations and starting several DFS1 search processes. Workstation discovery is done via UDP broadcast every 60 seconds and is constant throughout the life of the search. Advertisements for accept states dominate the communication overhead.

The number of advertisements sent per state explored is the number of initial first-walk processes plus the advertisements resulting from accept states found during the first-walk. This is given by the following equation.

$$M_s = P_0 + (N\bar{D}) \sum_{j=1}^{A} \text{CPoisson}(P_N - 1, \frac{j\bar{D}}{N})$$

where

- M_s = number of messages sent for every "S" states explored.
- A = number of accept states discovered.
- N = number of participating nodes in the network.

- P_N = DFS2 processes allowed per workstation, if only on process is allowed per workstation, then $P_N = 1$.
- P_O = Initial number of DFS1 processes injected into the network.
- $\bar{D}$ = average fraction of workstations notified per trace

The total number of messages sent is given by the sum of the initial messages sent, P_0, and the total number of messages sent for all accept states, given in the rightmost term. The rightmost term has two parts: $N\bar{D}$ gives the average number of processes notified per accept state discovered and the summation gives the number of those requests that are fulfilled for each accept state. The number of requests fulfilled is modeled by the cumulative Poisson probability that $P_N - 1$ or fewer DFS2 jobs have been started on any node, given that the expected average number of jobs per node is $j\bar{D}/N$. The Poisson distribution is suitable for this problem because it deals with discrete events occurring randomly in time or space. The number of messages is greatest when many shallow accept states are discovered since A, R and $\bar{D}$ are large. Experimental results given in Section 4 validate this model.

3.4 Termination Detection

Termination detection is managed by the central recorder process. When the user requests that the search is terminated, the recorder sends a stop message to every participating workstation. Each workstation then kills any active search processes.

3.5 Computational Platform

The BEE algorithm uses a dynamic cluster of workstations that share a common file system and TCP/IP connections. Workstations may enter and leave the cluster during a search. The common file system is used for model distribution only. The common file system can be eliminated by sending the model to each workstation before beginning the search. Each workstation must be able to receive a UDP broadcast from another workstation. The BEE algorithm was implemented as an extension of the Murϕ model checker [Dil96] modified to check LTL properties and combined with a socket-based communication module.

4 Results

All experiments were run on a network of workstations in open-use classroom laboratories associated with computer science classes at Brigham Young University. Each BEE process consumed up to 40 MB of memory per workstation and was run at the lowest possible priority level to avoid disturbing other users.

BEE is compared with isolated random walk (ISO) on two models. One model, sparse shallow 7, contains one accept state and the other, Peterson's mutual exclusion on 11 nodes (11-Peterson), contains many accept states. Isolated random walk is parallel random walk in which workstations may not communicate with each other. Comparing BEE to ISO demonstrates the advantage, or disadvantage (in some cases), of using the BEE coordination scheme. BEE finds an error faster in the model with one accept state and ISO finds an error faster in the model with many accept states. The 11-Peterson problem contains the right number of accept states such that the ISO and BEE algorithms find errors at the same rate. Results are were not generated for models with accept state densities greater than 11-Peterson to avoid saturating the non-dedicated network with accept state advertisements. Section 5 contains ideas for an adaptive communication mechanism that will scale individual eagerness to communicate with accept state density.

Results are given only for simple, contrived problems. Contrived problems were used because the performance of a violation discovery algorithm is sensitive to the topology of the transition graph and distribution of errors. In the extreme case, finding an error in a transition graph with 10^{10} states and a single error consisting of the initial state and its successor would be a trivial exercise. Instead of giving results for standard examples, we chose to give results for simple examples in which we can precisely control and characterize the shape of the transition graph and locations of errors.

Since the bitstate algorithm implemented in SPIN is also intended to find errors in large transition systems, we include a separate subsection comparing bitstate hashing with random walk. A secondary advantage of this approach is that is simplifies the problem of creating PROMELA and Murϕ models with identical transition graphs.

4.1 Sparse Shallow 7

The sparse shallow 7 model consists of 7 counters that count from 50 down to 0. In a single transition, one counter can decrement its value by 1. When a counter reaches 10 its value can be reset to 40. The property to be checked is "not always eventually is every counter equal to 40." Violations of this property consist of an accept state, in which every counter equals 40, contained in a cycle. There are 7×10^{11} reachable states and only one accept state in the model. The graph distance from the start state to the accept state is 72 and the smallest distance from a successor of the accept state back to itself is 240 states. The diameter of the transition system is 350.

ISO and BEE were run 10 times each on static groups of 4,8,16,32 and 64 workstations. A single random walk

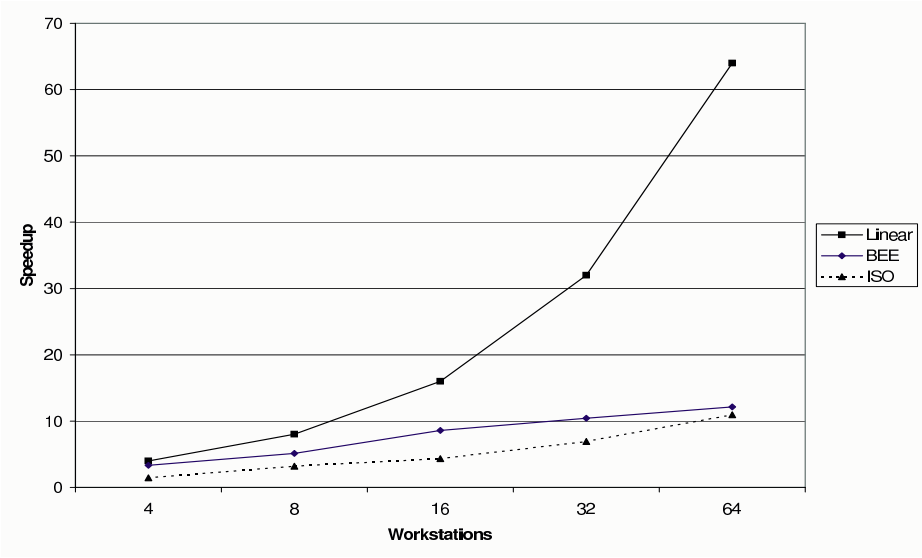

Fig. 6. Speedups for isolated random walk (ISO) and the BEE algorithm on sparse shallow 7.

process found the error in an average of 21 minutes (average over 10 runs). Speedups were computed using random walk on a single machine and are shown in Figure 6. While neither algorithm achieves linear speedup (note that the horizontal axis is logarithmic), using the coordination scheme in BEE gives a greater speedup because more secondary searches for cycles containing accept states are launched. The BEE coordination scheme is most useful on workgroups of 16 and 32 workstations where the speedup is nearly doubled (8.6 and 4.3 on 16 workstations and 10.4 and 6.9 on 32 workstations) compared to isolated random walks. With 64 workstations, the advantage of the coordination scheme is nearly eliminated by the advantage of many concurrent random walks.

4.2 Peterson's Mutual Exclusion

The Peterson's mutual exclusion model contains 11 processes negotiating for exclusive access to a critical section. The property to be checked is "not always eventually does every process enter the critical section." Violations of this property consist of an accept state, in which every process has previously entered the critical section, contained in a cycle. BEE and ISO have found accept states at or near a depth of 480 contained in cycles of between 70 and 200 states. We do not know how many reachable states there are in the 11-Peterson model.

ISO and BEE were run 10 times each on static groups of 4,8,16,32 and 64 workstations. The speedup was computed using random walk on a single machine and are shown in Figure 7. Both algorithms achieve linear or slightly superlinear speedup on 64 workstations. The performance scales to 64 workstations because we are measuring the time to find the first error using many concurrent random walks in a transition system with many errors. As the number of random walks increases, the probability of quickly randomly hitting an error increases.

Because the model contains many accept states, the BEE coordination scheme has no advantage. Using BEE coordination, processes spend most of their time sending advertisements for accept states rather than searching for errors. The 11-Peterson model contains just enough accept states that the ISO processes find the error only slightly faster than the BEE processes. We have devised models with a denser distribution of accept states. On these models ISO processes find errors significantly faster than BEE. Results are not included for these models because they are difficult to test without severely impacting network performance and inconveniencing users.

4.3 Dynamic Workgroups

The second group of experiments compares the performance of BEE in different computing environments. The 32 workstation tests were repeated on the last night of a semester, when the average system load on each machine was near 1.0 and 3-4 persons were remotely logged into each machine. Under these conditions, BEE found the first violation in an average of 147 seconds, rather than 119 seconds required by a mostly idle workgroup. According to the t-test, it is unclear if the means are different ($p = 0.66$). The 16 workstation tests were repeated with 3 workstations exiting and re-entering the search 30 seconds into the search. Under these conditions, BEE found the first violation in average of 251 seconds, rather than 152 seconds required by a stable workgroup. According to the t-test, the averages are different ($p = 0.12$).

The BEE algorithm requires more time on a dynamic network because nodes rejoining the search after a reset

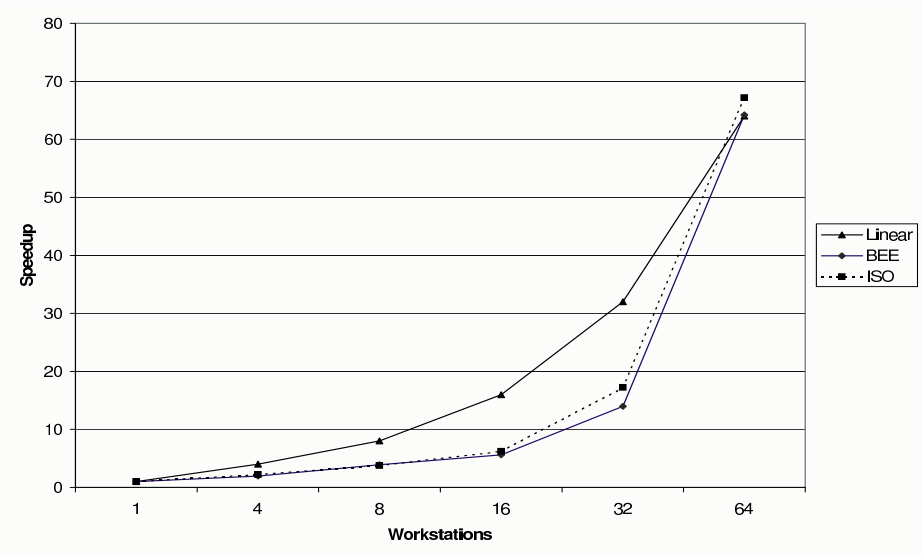

Fig. 7. Speedups for isolated random walk (ISO) and the BEE algorithm on 11-Peterson.

must either find an accept state or be notified of an accept state before beginning a new DFS2 process. Before beginning the new DFS2 search, the newly started node contributes little to the discovery of new errors.

4.4 Comparison with Bitstate Hashing

The sparse shallow 7 model and a variant with only 5 counters, sparse shallow 5, were used to compare bitstate hashing with the random walks used in the BEE algorithm. Bitstate hashing and BEE were compared in terms of the time and space required to find an error. Since BEE uses depth bounded random walks, we set the SPIN depth bound to 400. This depth is deep enough to contain violations but prevents unnecessarily long traces. We ran a series of tests in which the amount of memory available for the bitstate table increased from 42.1 MB to 671.3 MB. The 42 MB tests compare bitstate hashing with BEE using the same amount of memory. The other tests determined if bitstate hashing could be used to find an error at all.

The table in Table 2 compares the results for bitstate hashing and BEE on sparse shallow 7. Coverage was computed by dividing the total number of transitions taken by the number of reachable states. SPIN includes 32 different hash functions that can be used with bitstate hashing to cover different subsets of the reachable states. We reran the 42.1 MB tests with each of the 32 hash functions and did not find the error in any run.

Comparing bitstate with BEE on the 11-Peterson model produced similar results with a hash factor ranging from 1.541 to 1.568. The results were similar in that BEE found the error and SPIN using bitstate never found the error.

In the sparse shallow 5 model, bitstate hashing in SPIN found the error but in more time than random walk. In these tests, the depth was set to 250 and we allowed 671.3 MB of memory for the bitstate table. SPIN found the error in 387 seconds while random walk on a single machine required between 276 and 7 seconds with an average of 63.7 seconds (average over 10 runs) using 40 MB of memory.

The sparse shallow model results demonstrate that there is at least one problem in which random walk is more effective than bitstate hashing. This is not the case in general. Table 3 gives results for a tree shaped model, called "tree", with 10^8 states and $10^8 - 1$ accept states (the initial state is not an accept state). In the tree model, only one accept state is contained in a cycle. The remaining accept states are not part of an error. In this table, the parallel BEE algorithm is compared with the sequential bitstate algorithm. The BEE algorithm was executed on 64 workstations. The BEE algorithm failed to find the error after one hour because the search for cycles began at an accept state that was not contained in a cycle. The bitstate algorithm searches for cycles containing any accept state and avoids duplicate search using a hashtable on the DDFS algorithm.

5 Conclusion and Future Work

The ability of the BEE algorithm to find errors quickly, compared to isolated random walks, depends on the prevalence of accept states. If accept states are rare, the BEE algorithm finds errors more quickly than isolated random walk due to the coordination scheme. If accept states are prevalent, the BEE algorithm can find errors as fast as or much slower than isolated random walk due to communication overhead incurred by the coordination scheme. Both isolated random walk and BEE coordination can scale well up to at least 64 processors due

Algorithm	Memory (MB)	Time (sec)	Found error?	Coverage	Hash factor
bitstate	42.1	904	no	≥0.08%	1.451
bitstate	84.1	1,858	no	≥0.16%	1.458
bitstate	168.0	3,837	no	≥0.33%	1.457
bitstate	335.7	7,876	no	≥0.66%	1.460
bitstate	671.3	16,266	no	≥1.46%	1.461
random walk	40	1,240	yes	unknown	-

Table 2. Results for bitstate hashing in SPIN and random walk on one processor for the sparse shallow 7 model.

Algorithm	Memory (MB)	Time (sec)	Found error?	Coverage	Hash factor
bitstate	335.7	380	yes	≥ 89.5%	5.996
BEE	40	3,600	no	unknown	-

Table 3. Results for bitstate hashing in SPIN and random walk for the tree model with many accept states not contained in cycles.

to their lack of coordination or decentralized coordination, respectively. The BEE coordination scheme tolerates workstations entering and leaving the cluster and tolerates heavily loaded machines, although the performance predictably suffers.

Compared to bitstate as implemented in SPIN, random walks consume less memory and find errors in less time in the sparse shallow 5, sparse shallow 7 and 11-Peterson models. Bitstate hashing performs poorly in these cases because its coverage is heavily concentrated at the lowest depths allowed. In the sparse shallow 7 tests, 76% of the coverage was concentrated at the three lowest search depths and only 98 states were explored at the minimal accept state depth. The double depth first search algorithm used in SPIN must explore the deepest states first to avoid masking cycles in the second depth first search. If the depth bound is set to concentrate coverage at the minimal accept state depth, SPIN will not find the error because all cycles will be truncated (for the models analyzed here). Since random walk does not presuppose to find all, or any, errors the search depth can be adjusted to provide more useful coverage.

Bitstate hashing found errors more quickly than BEE in models with many accept states that were not contained in cycles. In these problems, the BEE algorithm focused search effort on accept states that were not contained in cycles.

While quite simple, isolated random walk in parallel was effective at finding errors in large transition systems. The isolated random walks used here, and to some extent the BEE algorithm, are a form of random simulation. The main difference is that random walk through a formal transition system to find violations of a formal property does not require the use of a reference model to compute correct results. Instead, random walks on formal transition systems recognize errors as they are defined by the property.

The results indicate that adjusting the frequency and intensity of communication is a vital part of the BEE coordination scheme. In the current BEE coordination scheme, the communication intensity is set manually using a static analysis of accept state depth. In nature, the communication intensity of a honeybee can vary by as many as two orders of magnitude and depends on the quality of a site *and* the need for the resource at a site. In other emergent systems derived from social insect behavior, communication intensity varies with perceived network traffic over a given time period—in addition to static factors. We are currently creating the feedback loops that will allow BEE processes to adjust their communication intensity and frequency based on the quality of an accept state *and* recent communication behavior. In one extreme, when many accept states are being discovered, BEE processes will rarely communicate and the search will devolve into isolated random walk. At the other extreme, when no accept states have been recently discovered, BEE processes will communicate an accept state to every known process. In addition to controlling communication intensity and frequency, the feedback loop could be used to allow each individual to switch between the first and second phases of the DDFS.

Possibilities for future work include not only improving the coordination scheme but also improving the search technique. The coordination scheme can be improved by implementing adaptive communication that is sensitive not only to the quality of an accept state, but the need for more accept state advertisements as described above. The performance of each BEE process can be improved by replacing random walk with a memory bounded best-first search method based on A* search (using SMA* [Rus92]). Directed model checking using variants of A* has been explored by Edelkamp [ELLL01] but for exhaustive searches. Combining random walk with memory bounded A* is likely to result in an algorithm that is similar to Brim's randomized LTL model checking algorithm [BCN01] with a reduction strategy based on cost estimates rather than visitation factors.

References

[BCKP01] L. Brim, I. Cerna, P. Krcal, and R. Pelanek. Distributed LTL model checking based on negative cycle detection. In *FST-TCS01*, number 2245 in LNCS. Springer, 2001.

[BCN01] L. Brim, I. Cerna, and M. Necesal. Randomization helps in LTL model checking. In *PAPM-PROBMIV Workshop*, number 2165 in LNCS. Springer, 2001.

[CD98] G. Di Caro and M. Dorigo. Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9:317–365, 1998.

[CGP00] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[DCG99] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2):137–172, 1999.

[Dil96] David L. Dill. The Murϕ verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[DK00] Z. Dang and R. Kemmerer. Three approximation techniques for ASTRAL symbolic model checking of infinite state real-time systems. In *22nd International Conference on Software Engineering (ICSE00)*, pages 345–354. IEEE Press, 2000.

[ELLL01] Stefan Edelkamp, Alberto Lluch-Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In *8th International SPIN Workshop on Model Checking Software*, number 2057 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[GHS01] O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for the μ-calculus. In *Computer Aided Verification 2001 (CAV01)*, number 2102 in LNCS. Springer, 2001.

[Has] P. Haslum. Model checking by random walk. available at http://www.ida.liu.se/~pahas/public/ccsse99.ps.gz.

[ISTV93] J.J. Bartholdi III, T. D. Seeley, C. A. Tovey, and J. H. Vande Vate. The pattern and efectiveness of forager allocation amung flower patches by honey bee colonies. *Journal of Theoretical Biology*, (160):23–40, 1993.

[LS99] F. Lerda and R. Sisto. Distributed-memory model checking in SPIN. In *The SPIN Workshop*, volume 1680 of *Lecture Notes in Computer Science*. Springer, 1999.

[Rus92] S. J. Russell. Efficient memory-bounded search methods. In *ECAI92: 10th European Conference on Artificial Intelligece Proceedings*, pages 1–5. Wiley, 1992.

[SCS91] T.D. Seeley, S. Camazine, and J. Sneyd. Collective decision-making in honey bees: how colonies choose amung nectar sources. *Behavioral Ecology and Sociobiology*, 28:277–290, 1991.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the Murϕ verifier. In Orna Grumburg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267, Haifa, Israel, June 1997. Springer-Verlag.

[See95] T. D. Seeley. *The Wisdom of the Hive: The Social Biology of Honey Bee Colonies*. Harvard University Press, 1995.

[SV88] T. D. Seeley and P.K. Visscher. Assessing the benefits of cooperation in honeybee foraging: search costs, forage quality and competitive ability. *Behavioral Ecology and Sociobiology*, 22:229–237, 1988.