# Parallel Model Checking Using Abstraction

Ethan Burns[1] and Rong Zhou[2]

[1] Department of Computer Science
University of New Hampshire, Durham, NH 03820 USA
eaburns@cs.unh.edu
[2] High Performance Analytics Area
Palo Alto Research Center, Palo Alto, CA 94304 USA
rzhou@parc.com

**Abstract.** Many model checking techniques are based on enumerative graph search, a procedure that is known to be prohibitively time and memory consuming. Modern multi-core processors rely on parallelism instead of raw clock speed to provide increased performance, so it is necessary to leverage this parallelism to achieve better performance in model checking. In this work, we compare hash-distributed search, a well-known parallel search technique for model checking, with an algorithm from the automated planning and heuristic search community called Parallel Structured Duplicate Detection (PSDD). We show that PSDD has two major advantages over hash-distributed search for multi-core model checking. First, PSDD is able to perform full partial-order reduction where hash-distributed search must be conservative and subsequently miss reduction opportunities in many cases, causing it to search a much larger space. Second, PSDD performs duplicate detection on states immediately, avoiding the need to store duplicate states for inter-thread communication. We have implemented and compared both techniques in the Spin model checker; our results show that PSDD uses significantly less memory than hash-distributed search, can be faster and give better parallel speedup than both hash-distributed search and Spin's built-in parallel depth-first search. Finally, we show how PSDD can use external memory, such as disk storage, to greatly reduce its internal memory requirements.

## Introduction

Model checking is a fundamental tool used in the creation and verification of asynchronous and distributed systems. Since the actions performed by each component of such a system may be interleaved in many ways, there can be a large number of configurations of the system as a whole. Given an abstract model of a system, a model checker can enumerate all reachable configurations of the model in order to aid in verification of its correctness. During enumeration, the model checker can ensure that the model does not exhibit any invalid behaviors or reach any invalid states. If such an error is found then a trace of the actions leading to it can be reported back to the user. This trace information is invaluable when creating and debugging a new system. Additionally, if the model checker is unable to find any invalid behaviors then it is evidence that the system is in fact correct.

A. Donaldson and D. Parker (Eds.): SPIN 2012, LNCS 7385, pp. 172–190, 2012.

To enumerate all possible states of an asynchronous system, many popular model checkers treat the configuration space as an implicitly defined graph where nodes correspond to system states and edges are the possible transitions of each component. A path through this graph gives one possible interleaving of the actions that the system may perform. Once the graph is defined, an exhaustive search algorithm can then explore all reachable states of the system looking for ones that violate certain properties. As is typical with implicit graphs, however, there can be a very large number of nodes causing the search to take a prohibitive amount of time or memory.

The model checking community, the heuristic search and automated planning communities have all been quite successful in developing new search frameworks that take advantage of modern multi-core processors. These frameworks have enabled them to improve the performance of their algorithms and have also been shown to be successful at offloading a significant portion of the memory requirement of a large graph search to external storage devices such as hard disks. However, some of the most successful techniques used by the heuristic search and planning communities have yet to be tested for model checking. Because of their success on other types of search problems, we would like to compare these approaches to those commonly used to parallelize search in model checking.

We have implemented two techniques for parallelizing breadth-first search in the Spin model checker [7]. The first technique is based on a common approach for parallel model checking that distributes states among different searching threads by using a hash function [18,12]. We call our implementation of this algorithm hash-distributed breadth-first search (HD-BFS). The second method comes from the heuristic search and planning communities called parallel structured duplicate detection (PSDD) [20]. We show that PSDD has some major advantages over hash-distributed search for model checking. First, HD-BFS uses delayed duplicate detection [16,13] and must store duplicate search nodes temporarily while they are being communicated between threads. PSDD is able to detect duplicate states immediately after they are generated thus abolishing the need to use extra memory in order to store them. Second, PSDD is able to preserve Spin's ability to perform partial-order reduction – a technique used by model checkers to decrease the size of the search space. This means that, when using multiple threads, PSDD is often able to search a significantly smaller space than both HD-BFS and Spin's built-in multi-core depth-first search, both of which must be more conservative when performing partial-order reduction. Overall, the results of our experiments demonstrate that PSDD is faster and able to achieve greater parallel speedup than both HD-BFS and Spin's state-of-the-art multi-core depth-first search.

In addition to improving the performance of breadth-first search, we show some preliminary results demonstrating that PSDD can also successfully reduce the memory requirements of model checking by making use of external storage devices. In one experiment PSDD is able to reduce the memory requirement of the search by over 500% when using a hard disk to supplement internal memory.

## Depth-First versus Breadth-First Search

Two of the most well-known graph search algorithms are depth-first search and breadth-first search. Depth-first search generates the successors of nodes in the graph (we call

the generation of successors of a node 'expanding the node') in deepest-first order. This means that one of the most recently generated nodes will be the next node that is expanded. Breadth-first search, on the other hand, expands nodes in shallowest-first order. Spin uses depth-first search by default as it is able to check both safety properties (typically used to verify that something undesirable will not happen) and liveness properties (typically used to verify that something desirable will eventually happen) whereas Spin's breadth-first search algorithm is only able to verify safety properties.

Breadth-first search for model checking is guaranteed to find shortest counterexamples if the model violates a safety property. This is significant because, many important properties of an asynchronous system are safety properties and when debugging a system one must understand the counterexample provided by the model checker in order to determine why the system is not behaving as desired. Depth-first search pays no heed to the number of steps used to reach a node in the state space and therefore may produce a counterexample that is many steps longer than necessary. These long traces can be extremely hard to interpret as they may contain a lot of transitions that are not necessary to produce the faulty behavior. To put this in perspective, on one model we have observed that depth-first search finds a deadlock and provides a trace consisting of 9,992 steps where breadth-first search finds a trace for the deadlock with the smallest number of possible steps: 42.

While breadth-first search cannot be used directly to verify liveness properties, there has been work on efficient translations of liveness checking problems into safety checking problems, which can subsequently be verified by breadth-first reachability analysis [1,17]. Given depth-first search's inherently sequential nature [15], checking liveness property using a breadth-first, instead of depth-first, strategy can better leverage the latest multi-core processors for greater parallel speedups.

## Hash-Distributed Breadth-First Search

Burns *et al.* [3] discuss the difficulties in parallelizing best-first search algorithms such as breadth-first search[1] and they show that many naïve implementations of parallel search actually perform worse than their serial counterparts.

In order to successfully search a graph in parallel the graph should be divided in a way that each thread performing the search can operate on an independent portion of the graph. A simple way to achieve this is to divide the nodes of the graph statically using a hash function; as each new node is generated, its hash value is computed and it is distributed to the thread with the thread ID equal to the hash value modulo the number of threads. If a node is generated multiple times, each duplicate will be assigned to the same thread so duplicate detection can be performed locally within each thread. This framework is called hash-distributed search and was originally proposed as a method for parallelizing the A$^*$ algorithm [6] and was later discovered by Stern and Dill [18] in the context of model checking and then by Kishimoto *et al.* [12] who called the algorithm hash-distributed A* (HDA$^*$) and applied it to automated planning problems.

We have implemented a hash-distributed breadth-first algorithm, based on HDA$^*$. We call this algorithm hash-distributed breadth-first search (HD-BFS). HD-BFS works

[1] Breadth-first search can be viewed as a special best-first search where all edges have unit cost.

in layers by expanding the nodes at a given depth from the root in parallel until all nodes at the current depth have been expanded. When a depth layer has been completely expanded, all threads proceed synchronously to the next depth and begin searching there.

Each HD-BFS thread uses a pair of queues to represent the search frontier. One queue, called the *current queue*, contains all nodes assigned to the thread that are at the current search depth. The second queue, called the *next queue*, contains all nodes assigned to the current thread that are at the next search depth. Each thread also has a hash table containing all nodes that it has previously expanded. This table is used to prevent the search from expanding the same nodes multiple times. Note that, because all duplicates of a search node will be assigned to the same thread by the hash function, no node resides in more than a single hash table.

When searching, each thread expands the nodes from its current queue one-at-a-time. When a successor node is assigned by the hash function to a different thread than the one that generated it, it must be sent there using inter-thread communication. Otherwise, when a successor node is assigned back to the same thread that generated it, it is immediately checked for membership in the local hash table to determine if it is a duplicate and if it is not a duplicate then it is added to the next queue for the local thread; no communication is required. Our implementation of HD-BFS uses the communication scheme from Burns *et al.* [3] to send nodes between threads asynchronously using shared-memory queues.

After receiving a new node sent from a different thread, the receiving thread checks to see if the node is a duplicate by testing it for membership in its local hash table. If the node is not a duplicate then it is placed on the thread's next queue. This is the appropriate queue because all threads are expanding nodes at the same depth from the root and therefore any generated node resides at the next depth regardless of which thread generated it.

If all threads have empty current queues and no nodes are in transit between threads, then the current depth layer has been completely expanded. When this happens, all threads synchronously swap their next queue with their current queue and begin searching nodes at the next depth. If all current queues are still empty after swapping to the next depth then the search space has been exhausted and the algorithm terminates.

**Disadvantages**

We have found that there are two major disadvantages to hash-distributed search when applied to model checking. The first is that hash-distributed search delays the detection of duplicate nodes when they are communicated between threads. When nodes are sent to another thread they are placed on the receiving queue for that thread and sit there until they are eventually received and checked against the receiving thread's hash table. This delayed detection of duplicate nodes can cause the search to require more memory as the duplicates reside in the receiving queue instead of immediately having their memory freed for reuse. As we will see, the extra memory overhead created by delaying duplicate detection can be quite substantial.

The second disadvantage of hash-distributed search is that it must be conservative when applying partial-order reduction [9], a technique used in model checking to

reduce the size of the search graph. When expanding a node while using partial-order reduction, only a subset of the successors are considered and the rest are discarded. While performing breadth-first search with partial-order reduction, Spin uses a test called the *Q proviso* [2] to prevent reduction in cases where completeness cannot be ensured.

The Q proviso tests if a newly generated node is placed on the breadth-first search queue or if it was already on the queue from a previous generation. If the Q proviso is satisfied then the reduction can take place, otherwise the full expansion must happen. Bošnački *et al.* [2] proved that this simple test allows breadth-first search to remain complete under partial-order reduction when searching for safety property violations and deadlocks.

With hash-distributed search, the successors of a node may not be assigned to the expanding thread. When this happens, the expanding thread does not have the ability to test if the successors are on or end up on the queue because this queue is owned by a different thread. To preserve completeness, HD-BFS must be conservative and assume that all nodes that are sent to different threads do *not* pass the Q proviso. This reduces the chances of successfully performing partial-order reduction because, in order to reduce, a thread must generate a successor that is assigned to itself and also passes the Q proviso. As we will show in our experimental results, with a greater number of threads the chance that successors will not be assigned to the expanding thread increases, so as the number of threads increases the size of the search space will increase too. Because of this, HD-BFS using multiple threads can actually perform worse than a serial breadth-first search because the former must search a significantly larger space to guarantee completeness.

**Abstraction-Based Hashing**

Both of the previous issues with hash-distributed search stem from the fact that the hash function used to assign nodes to threads is designed to uniformly distribute the nodes. This is beneficial from a load balancing perspective, however, it means that it is uncommon for the successors of a node to be assigned to the thread that generated them. Burns *et al.* [3] present a novel modification to hash-distributed search that can be used to help alleviate this issue at the cost of possibly decreasing load balancing. Instead of using a hash function that distributes the nodes uniformly, a homomorphic abstraction function can be used to distribute the nodes in a more structured fashion. Each thread is responsible for a set of nodes in an abstract representation of the search graph. When a node is generated, its abstract representation is computed and it is assigned to the thread responsible for this abstract node.

The advantage of this approach, when using a carefully created abstraction, is that the successors of a search node will tend to be assigned back to the same thread that generated them. This means that the need for communication is reduced as newly generated nodes can often be handled locally. The disadvantage is that the search load may not be evenly balanced among the threads. Burns *et al.* show that, in practice, using an abstraction instead of a uniformly distributed hash function can greatly increase the performance of HDA* on puzzle solving and planning problems.

For model checking, fewer communications mean fewer duplicate nodes that reside in memory. It also means that there are more chances to perform partial-order reduction. As we will see, this approach can greatly reduce the memory requirements and the size of the search space explored by hash-distributed search. Unfortunately, because the nodes are no longer distributed uniformly among the threads, this abstraction-based implementation of HD-BFS (which we call AHD-BFS) gives very brittle performance for different numbers of threads. We suspect that the nodes tend to be distributed unevenly causing some threads to be very busy and some threads to starve for work. This behavior hinders the ability of the search to fully exploit the available parallelism.

## Parallel Structured Duplicate Detection

Instead of assigning nodes to threads *a priori* by using a hash function, Zhou *et al.* [20] developed a framework called *Parallel Structured Duplicate Detection* (PSDD) that allows threads to dynamically divide the search effort. PSDD uses a homomorphic abstraction to map nodes in the search graph to nodes in an abstract representation of the search graph. The abstraction is a many-to-one mapping that is typically created by projecting away some of the state information of each search node. The abstract node to which a search node maps is called the *image* of the search node under the abstraction.

Given a search graph and a homomorphic abstraction function, an *abstract graph* is constructed as follows.

1. The set of nodes, called *abstract nodes*, in the abstract graph corresponds to the set of abstract states.
2. An abstract node $y'$ is a successor of an abstract node $y$ if and only if there exist two states $x'$ and $x$, such that
   a. $x'$ is a successor of $x$, and
   b. $y'$ and $y$ are images of $x'$ and $x$, respectively.

The abstract graph is used during search to locate portions of the search space that are disjoint. More formally, let abstract node $y = \phi(x)$ be the image of state $x$ under a homomorphic abstraction function $\phi(\cdot)$ and let $succ(y)$ be the set of abstract successor nodes of $y$ in the abstract graph.

**Definition 1.** *The* duplicate-detection scope *of a state $x$ under a homomorphic abstraction function $\phi(\cdot)$ corresponds to the union of sets of stored nodes that map to an abstract node $y'$ such that $y' \in succ(\phi(x))$, that is,*

$$\bigcup_{y' \in succ(\phi(x))} \phi^{-1}(y')$$

*where $\phi^{-1}(y')$ is the set of stored nodes that are pre-images of $y'$.*

**Proposition 1.** *The duplicate-detection scope of a node contains all stored duplicates of the successors of the node.*

**Definition 2.** *The duplicate-detection scopes of states $x_1$ and $x_2$ are* disjoint *under a homomorphic abstraction function $\phi(\cdot)$, if and only if the set of abstract successors of $x_1$'s image is disjoint from the set of abstract successors of $x_2$'s image in the abstract graph, that is, $succ(\phi(x_1)) \cap succ(\phi(x_2)) = \emptyset$.*

**Proposition 2.** *Two states cannot share a common successor if their duplicate-detection scopes are disjoint.*

Proposition 2 provides an important guarantee that a parallel model checker can leverage to reduce the amount of synchronization needed in parallel graph search. In particular, multiple threads can search disjoint portions of the graph, which correspond to disjoint duplicate-detection scopes, without the need for communication. Unlike HD-BFS, duplicate states are detected in PSDD as soon as they are generated.

As with HD-BFS, the search proceeds in layers. Each node in the abstract graph has two queues, one for the current depth-layer and one for the next. These queues contain the frontier nodes of the search graph that map to the given abstract node. Each abstract node also has a hash table containing all of the previously expanded search nodes that map to it.

Threads acquire access to expand all of the search nodes at the current depth for a single abstract node at a time. Because the abstraction is homomorphic, the successors of a search node will either map to the same abstract node or to one of the successors in the abstract graph. By claiming exclusive access to an abstract node and its successors, a thread can expand from the abstract node and perform immediate duplicate detection on the generated successors using only the data structures to which it has exclusive access. We call the set of nodes corresponding to an abstract node and its successors a *duplicate detection scope* (see Def. 1) or just a *scope* for short.

The left image in Fig. 1 shows an example graph in light gray with a possible abstraction of the graph drawn in dark black on top of it. This abstraction groups together sets of four nodes. There is an edge in the abstract graph between each pair of abstract nodes for which there exists a pair of nodes in the underlying graph that are connected by an edge and whose images correspond to each respective abstract state. The right image in Fig. 1 shows two duplicate detection scopes in this graph, each defined by the gray nodes and surrounded by a dashed line. Both duplicate detection scopes consist of the gray nodes and all nodes that map to the successors of their image in the abstract graph. When expanding any of the gray nodes, all successors will correspond to a node that resides in the same duplicate detection scope.

To perform parallel search, each thread will use the abstract graph to locate a duplicate detection scope that does not overlap the scopes being used by other threads. Given Proposition 2, these *disjoint duplicate detection scopes* may be searched in parallel without requiring communication. With this scheme, the only time that threads must synchronize is when multiple threads require access to the abstract graph at the same time. Only a single mutex is required to serialize access to the abstract graph and operations on the abstract graph tend to be quick.

The two duplicate detection scopes shown on the right half of Fig. 1 are disjoint as they do not share any nodes.

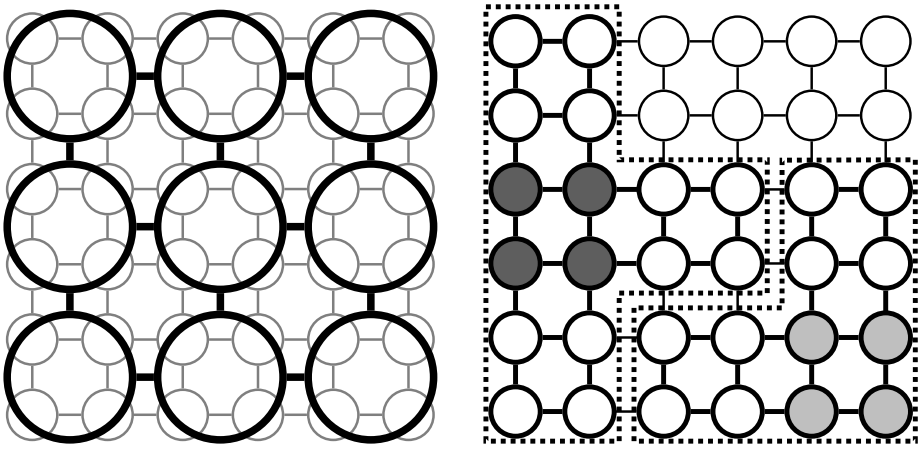

**Fig. 1.** A graph along with one of its possible abstractions (left) and two disjoint duplicate detection scopes of this graph (right)

When a thread completes the expansion of all open search nodes mapping to its current abstract node, it can release its duplicate detection scope, marking all abstract nodes in the scope as free to be re-acquired. Then the thread can try to acquire a new scope to search. If there are no free scopes with open search nodes at the current depth then the thread attempting to acquire a new scope must wait until another thread finishes expanding and releases its abstract nodes. This wait time can be reduced by using a finer-grained abstraction with sufficiently many disjoint duplicate detection scopes. In practice, we find that abstractions can typically be made large enough that wait times are insignificant.

Eventually, as open search nodes become exhausted in the current depth-layer, there will be only a single thread actively searching as the other threads wait for abstract nodes to become free. When the final non-waiting thread releases its duplicate detection scope and finds that there are no free scopes with open nodes it will progress the search to the next depth layer. To do this, the current and next queues for each abstract node are swapped, all abstract nodes with open nodes in their new current layer are marked as free, waiting threads are woken up and the search resumes. If the new depth-layer contains no open search nodes then the search space has been exhausted and the threads can terminate.

PSDD provides at least two advantages over hash-distributed search: 1) there may be less synchronization between threads in PSDD because threads only need to synchronize access to the abstract graph when releasing and acquiring a new duplicate detection scope and 2) duplicates can be checked immediately instead of using extra memory to store duplicate nodes before they can be checked against the hash table.

PSDD provides an additional benefit when applied to model checking: it does not need to be conservative when performing partial-order reduction. Recall that HD-BFS did not have access to test if successor nodes reside on the breadth-first queue when the successors were not assigned to the expanding thread. In PSDD, however, the expanding

thread has exclusive access to the data structures for the duplicate detection scope of the abstract node from which it is expanding. This means that PSDD is able to test if the successors that it is generating pass the *Q proviso* and therefore it does not need to be conservative when doing partial-order reduction. As we will see, this gives PSDD a major advantage over both HD-BFS and Spin's multi-core depth-first search on many models.

### Abstraction for Model Checking

PSDD requires an abstract representation of the state-space graph in order to exploit the local structure of the space. Since the state space is not explicitly represented in memory, this abstraction must be a function that can be computed on each node. In Spin, each state in the search space consists of the set of processes whose executions are being modeled. Each process is represented by a finite automaton which has a current state and a set of transitions. The abstraction that we used in our implementation of PSDD is: given any state, consider only the process type and the automaton state of a fixed subset of the process IDs. For example, consider a state with seven processes numbered 0–6. One possible abstraction is to consider only the automaton states of the first two process IDs. This effectively 'projects away' process IDs 2–6, leading to a much smaller set of abstract nodes.

We use the transitions of the finite automaton to determine the predecessor and successor relations in the abstract graph. Because only the state of a single component automaton will transition between a node and its successors[2], the successors in the abstract graph are all of the possible single transitions of the process IDs that have not been removed in the abstraction. For efficiency, we generate the abstract graph lazily as needed during the search. This provides the benefit of only instantiating the portions of the graph that are actually used and it also constructs the graph in parallel with the execution of the search instead of doing it serially as a pre-processing step.

## Experimental Results

In this section we present the results of a set of experiments that we performed to evaluate the two methods of parallelizing breadth-first search. In addition, we compare to Spin's built-in multi-core depth-first search where applicable. The machine used in our experiments has two 3.33GHz Xeon 5680 processors, each having six cores, and 96GB of RAM.

### Multi-core Depth-First Search

Spin comes, by default, with a state-of-the-art multi-core depth-first search algorithm [8]. The algorithm connects each of the threads performing the search in a ring. Nodes

[2] For Spin, this is not strictly true when using 'never claims.' Our implementation requires that never claims are not considered by the abstraction, thus ensuring that only a single component automaton will change across a transition.

may be passed from one thread to another around the ring in a single direction. Each thread is then responsible for expanding all of the nodes that fall within a particular depth-interval. When the successors of a node fall outside of an interval assigned to the current thread, the newly generated successors must be passed to the neighboring thread along the ring using a shared memory queue. This neighboring thread may then receive the nodes from the queue and begin expanding them.

Using this technique, Holzmann *et al.* [8] were able to achieve speedups of just over 1.6x at two threads on a set of benchmark models and almost perfect linear speedup for two threads on a reference model that provided a set of tunable parameters. In their results, however, they show that this technique must be conservative when doing partial-order reduction. So, as with HD-BFS, the performance of multi-core depth-first search can actually be worse than serial search when partial-order reduction is used.

In the following experiments, we compare to Spin's multi-core depth-first search on models that do not contain safety property violations. The reason is that, on models with safety violations, depth-first search may find these violations via suboptimal paths, whereas breadth-first search must return optimal-length traces and thus may be forced to perform significantly more work. This renders the comparison unfair. On models without safety property violations, however, all algorithms must exhaust the search space and therefore will do a comparable amount of search.

Spin provides many parameters that may be tweaked to tune the search performance for different models. We compiled the multi-core depth-first with the

```
-DFULL_TRAIL -DSAFETY -DMEMLIM=64000
```

options on all models. For each individual model we also used any additional parameter settings that were recommended by Spin after running with the default parameter set.

**Effect of Delaying Duplicate Detection**

To compare the effects of the immediate duplicate detection of PSDD with the delayed duplicate detection of HD-BFS we looked at the memory usage of the two algorithms. Our hypothesis was that HD-BFS would require more memory in order to store duplicate search nodes during communication before they can be checked against the hash table by the receiving thread. The model that we choose for this experiment is a model of the dining philosophers problem with 10 philosophers. The model is constructed to avoid the classic deadlock situation and therefore the entire search space will be exhausted by the search algorithms. This removes the effects of tie-breaking that may be encountered when searching a model that contains an error. Also, with this model, the same number of states are expanded by all algorithms regardless of whether or not partial-order reduction is used and therefore we can conclude that any difference in memory usage must be attributed to immediate detection of duplicate nodes or lack thereof.

Figure 2 shows the memory usage reported by Spin for the 10 philosophers problem. The x axis gives the number of threads from 1–12 and the y axis shows the number of Gigabytes used to complete the search. Each line gives the mean of five runs at each thread count and the error bars (which are so tight that they are hardly even visible in this plot) show 95% confidence intervals on the mean. Breadth-first search only uses a

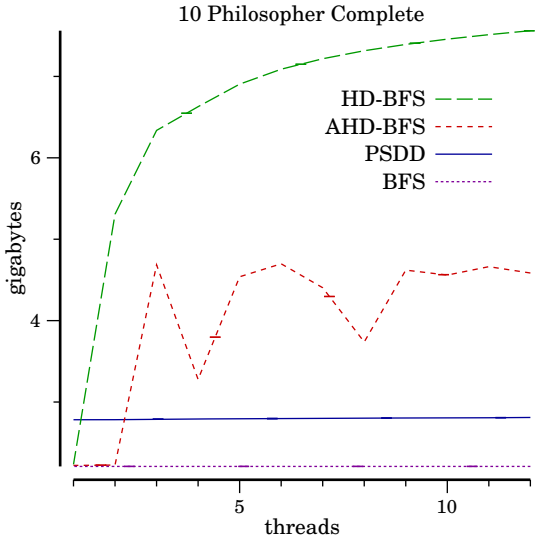


**Fig. 2.** Memory usage of PSDD, HD-BFS, AHD-BFS and BFS

single thread but we have extended the line for its single threaded performance across the x axis to ease comparison.

From this figure, we can see that breadth-first search and PSDD both used less than 3 Gigabytes of memory. The memory usage for PSDD remained nearly constant in the number of threads that performed the search. HD-BFS, however, required significantly more memory on this model when run with more than a single thread. The amount of memory required by HD-BFS increased sharply for up to six threads where it begun to even out. As mentioned above, this can be attributed to the fact that HD-BFS was required to store duplicate nodes in memory during communication instead of detecting them immediately. Due to the reduction in inter-thread communication, AHD-BFS used less memory than HD-BFS, however it still required more memory than breadth-first search and PSDD for more than two threads.

In addition to the results shown here, we have observed that HD-BFS required a lot more memory on all of the models that we have used in our experiments. Presumably, this is because of duplicate nodes, however, for other models the conservative partial-order reduction may also be a factor as we will see next.

### Effect of Conservative Partial-Order Reduction

To evaluate the performance degradation that hash-distributed search and Spin's multi-core depth-first search suffer from due to conservative partial-order reduction we performed an experiment using a model of the semaphore implementation from the "Plan 9 from Bell Labs" operating system (Plan 9) [14][3]. The model is of particular interest

[3] The model was available from `http://swtch.com/spin/`

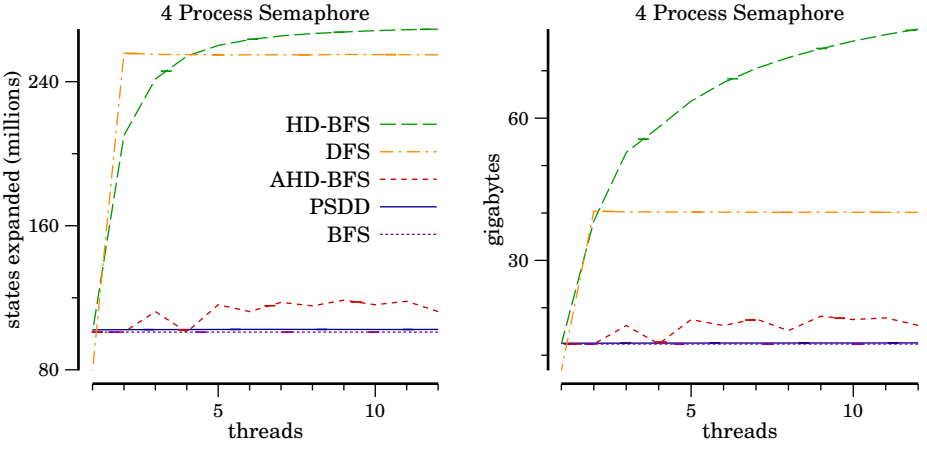


**Fig. 3.** States expanded and memory used by PSDD, HD-BFS and BFS

because, unlike the philosopher model used in the previous experiments, the semaphore model was taken from a real-world model checking problem. Partial-order reduction is able to reduce the size of the state space of this model by approximately a factor of three, so failure to perform the full reduction has a significant impact on performance.

Figure 3 shows the number of states expanded (left) and the amount of memory used (right) by PSDD, HD-BFS, breadth-first search and Spin's multi-core depth-first search on the semaphore model with four separate processes contending for the semaphore. The format of the plot is the same as that of Fig. 2. We can see that breadth-first search expanded the fewest nodes and used the least amount of memory in order to exhaust the configuration space of this model. PSDD expanded only slightly more nodes than breadth-first search and used approximately the same amount of memory. The reason that PSDD and breadth-first search expanded slightly different numbers of nodes is that they may expand nodes within the same depth layer in a different order. This difference in tie-breaking can have a small effect on the partial-order reduction by slightly increasing or decreasing the number of nodes that must be expanded.

With a single thread, HD-BFS expanded about the same number of nodes and used about the same amount of memory as breadth-first. As the number of threads was increased, however, the number of expansions and memory requirement of HD-BFS rapidly increased. HD-BFS required almost 80GB of memory when run with 12 threads. The reason for the steep increase is that HD-BFS required more communications as the nodes were divided up between more threads. Each time a node is communicated the search conservatively assumed that it could not perform partial-order reduction and therefore many redundant paths were explored that were not pursued by the other two algorithms. The plot also shows this same effect happens with Spin's multi-core depth-first search. The depth-first search suffers from the same conservative partial-order reduction as HD-BFS and for more than a single thread it expanded many more states than PSDD and breadth-first search.

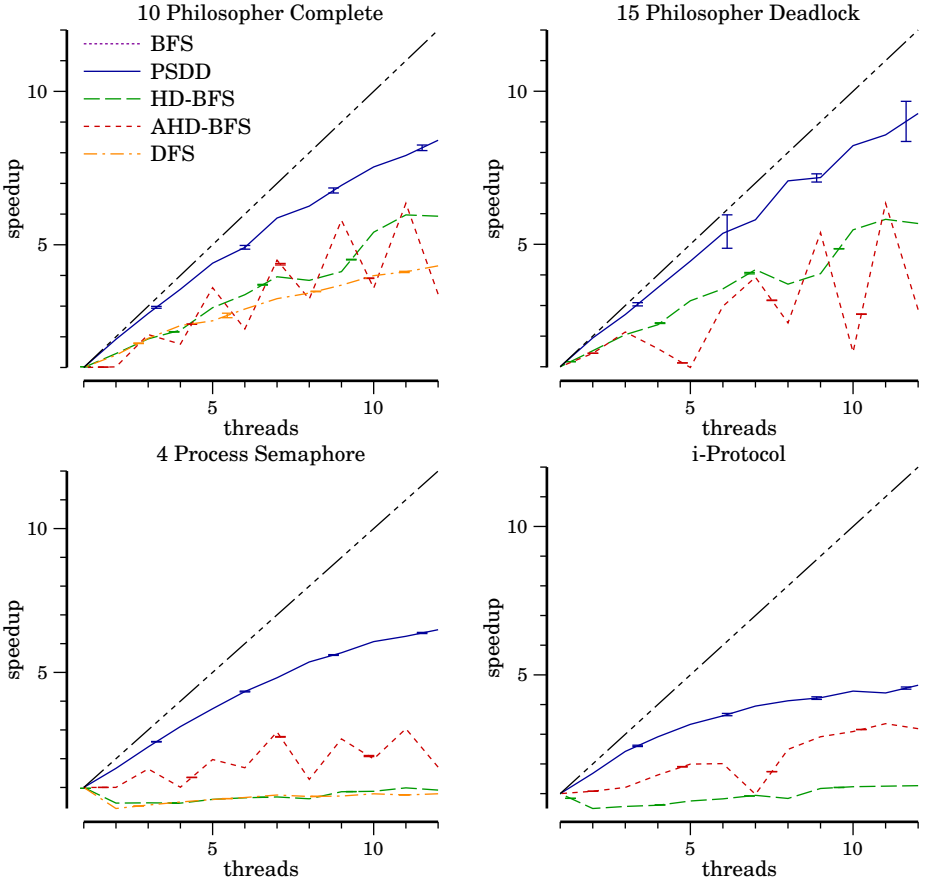


**Fig. 4.** Parallel speedup for PSDD, HD-BFS, AHD-BFS, and parallel depth-first search

### Overall Performance

Next we show the overall performance in terms of parallel speedup and wall-clock time for the different algorithms on four models. For PSDD and AHD-BFS, which both require an abstraction, we choose the fixed subset of processor IDs used in the projection experimentally. For each model we ran the algorithms using a small set of hand-chosen process ID sequences from 0–$n$ and 1–$n$ for small values of $n$ (up to 7). The sequence that gave the best performance for each model was used in the following comparisons. We believe that the good performance exhibited by PSDD in the following results when using such a simple abstraction is strong evidence that finding a good abstraction for PSDD is not a difficult task.

Figure 4 shows the parallel speedup and Figure 5 shows the total wall-clock time that the algorithms required to search four different models using 1–12 threads. As in the previous plots, each line shows the mean performance across five runs with error bars giving the 95% confidence intervals. The x axis show the number of threads used

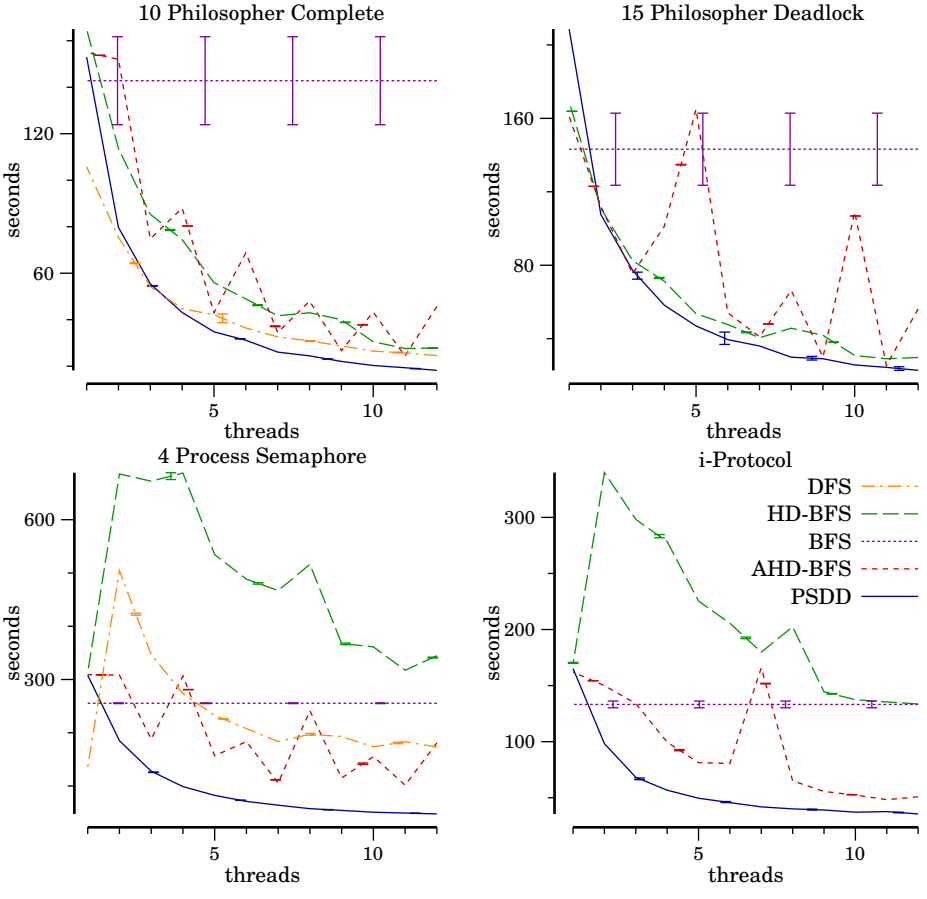


**Fig. 5.** Wall-clock seconds for PSDD, HD-BFS, AHD-BFS, parallel depth-first search, and serial breadth-first search

from 1–12 and in Fig. 5 the performance of breadth-first search is drawn across the x axis of each plot even though it was only run serially. The models used were the dining philosopher problem with 10 philosophers and no deadlock, the dining philosophers problem with 15 philosophers and a deadlock which is reachable in 42 steps, the Plan 9 semaphore with 4 contending processes, and the 0-level abstraction of the GNU i-Protocol model from Dong *et al.* [4][4] which contains a live-lock that is reachable in 72 steps, modified to avoid rendezvous as Spin complains that these do not maintain completeness with breadth-first search. Spin's multi-core depth-first search algorithm is not shown on the 15 philosopher model or the i-Protocol model because they both exhibit errors for which depth-first search does not find shortest counterexamples and therefore does not perform a comparable amount of search.

[4] Available from `http://www.cs.sunysb.edu/~lmc/iproto/`

Figure 4 shows the parallel speedup of PSDD, HD-BFS, AHD-BFS and depth-first search, computed as the single-threaded time divided by the time required for the number of threads given on the x axis. Speedup is perhaps one of the most important metrics when comparing parallel algorithms as it is indicative of how well the algorithm will perform as the parallelism increases. The diagonal line in each of the speedup plots shows perfect linear speedup which is typically unachievable in practice, however, it can provide a useful reference point. The closer that the performance of an algorithm is to the diagonal line, the closer that its performance is to a perfect linear speedup. We can see from these figures that PSDD came the closest to linear speedup on all of these models; it always provided better speedup than the other parallel algorithms.

Figure 5 shows the wall-clock time, that is the actual time in seconds, required by each algorithm for the four models. We can see from this figure that for greater than three threads, PSDD was able to solve all of these models more quickly than the other algorithms. On the two 'real-world' models, the semaphore and i-Protocol models, HD-BFS actually required more time than serial breadth-first search when using more than a single thread. This is because its conservative use of partial-order reduction caused it to search a much larger graph (c.f., Fig. 3). Spin's multi-core depth-first search also suffered from this same issue, however, it seems to have made better use of parallelism and with greater than four threads it was faster than serial breadth-first search on the semaphore model. Finally, we can see that AHD-BFS gives very erratic performance across different numbers of threads. We attribute this to poor load balancing among the threads due to use of the abstraction instead of uniform node distribution.

**External-Memory PSDD**

Our results have demonstrated that PSDD requires less memory on model checking problems than hash-distributed search and it gives better parallel speedup and faster search times than both hash-distributed search and Spin's multi-core depth-first search. PSDD is also able to act as an external-memory search algorithm where external storage such as a hard disk is used to supplement core memory. In fact, the PSDD framework was originally developed by Zhou *et al.* for external-memory search [19]. External-memory PSDD [20] (external PSDD for short) works just like PSDD, however, when an abstract node is not in use by one of the threads, it can be pushed off to external storage. This reduces the memory usage of the search algorithm from that of the entire search graph to just the size of the duplicate detection scopes acquired by each thread.

As a preliminary experiment, we implemented external PSDD in Spin and used it to solve the deadlock-free 10 philosophers model. We ran on a machine with eight cores and four disks configured in a RAID 0 array. A limitation of our setup was that I/O operations were serialized via a single disk controller, therefore when using all eight cores external PSDD did not benefit from parallelism. When using a single thread, standard PSDD used an average of 233 seconds to complete its search and external PSDD required 1,764 seconds on average (both times had very little variance). With a more sophisticated machine, external PSDD will show improved performance when using parallelism, for example, Zhou and Hansen [20] show performance improvements for up to four threads with external PSDD for automated planning. Even given this limitation with our experimental setup, the real benefit of external PSDD is still realized:

external PSDD was able to reduce the memory usage of search from 2.5 Gigabytes with standard PSDD down to around half of a Gigabyte when using a single thread. This is a 500% reduction in the memory requirement of the search. In many cases this reduction in the memory requirement is much more important than reducing the search time because it is easier to wait longer for the search to complete, however, it may not be possible to add more memory. Because of this, the memory requirement is often the limiting factor determining whether or not a model can be validated with a model checker.

## Discussion and Related Work

In a preliminary experiment we have seen that external-memory PSDD is able to reduce the memory requirement of search by a substantial amount. The penalty for external-memory PSDD, however, is that it can take a lot longer than serial search as it has to access hard disk storage. We suspect that the performance of external PSDD can be increased substantially by using multiple RAID arrays in order to exploit parallelism.

In our current implementation, external PSDD uses more memory when run with more than a single thread as each thread must have its own duplicate detection scope in RAM. With eight cores, external PSDD used around the same amount of memory as standard PSDD which does not use hard disk storage at all. A new technique called edge partitioning [22] may be able to fix this problem. Edge partitioning reduces the size of a duplicate detection scope to be only those search nodes that map to a single abstract node. This can be a very significant reduction that will enable external PSDD to use multiple threads while still having a very small memory footprint.

Until now, we have not discussed, in detail, how the chosen abstraction effects the performance of PSDD. For our experiments, the abstraction was selected by evaluating a small set of different abstractions on each model and choosing the one that gave the best performance. If the abstract graph is too small or is too strongly connected then PSDD can suffer as it will be unable to find a sufficient number of disjoint scopes to search in parallel. We have found that the simple abstractions used in our experiments have provided a sufficient amount of parallelism. Recent work, however, has shown that PSDD can greatly benefit from a dynamic search space partitioning that changes the abstraction during search [21]. By using dynamic partitioning, the algorithm would be able to select an abstraction that is more balanced, reducing the peak memory requirement of external search, and less connected, increasing its ability to exploit parallelism.

Given its rising importance, search parallelization has been the subject of focus for a number of related work done in the field of model checking. In [11], Jabbar and Edelkamp describe a parallel extension of External A*, which is a disk-based heuristic search algorithm. Like HD-BFS, Parallel External A* also uses delayed duplicate detection, which can be less efficient than structured duplicate detection for reasons discussed in this paper. Unlike both HD-BFS and PSDD, Parallel External A* is designed only for directed model checking, since it relies on both the g-value (the distance from the start state) and the h-value (an estimate on the distance to go) of a state to partition the search space. Thus, depending on whether an informative heuristic function is available, Parallel External A* can sometimes be less efficient. On the other hand, since

PSDD makes no assumption about the availability of a heuristic function, it is applicable to both directed and undirected model checking. Furthermore, Parallel External A* seems inherently disk-based, since "all communication between different processes [of Parallel External A*] is done through shared files" (page 7 of [11]). Thus, whether there exists an efficient implementation of Parallel External A* that only uses RAM remains to be seen. As for PSDD, since it does not rely on any file system for inter-process communication, both internal and external-memory versions of PSDD have been successfully applied to STRIPS planning, as shown in [20]. Fortunately, the same is true for model checking, as we show in this paper.

Besides systematic approaches, non-systematic parallel search techniques such as [5,10] have also been proposed and successfully applied to model checking large verification problems. In [5], the Parallel Randomized State-space Search (PRSS) technique was shown to reduce the cost of finding an error in Java code by factors ranging from 2 to well over 1,000. In [10], experiments show the Swarm Tool can dramatically reduce runtime and increase coverage over the standard method of a single depth or breadth first search. Both PRSS and Swarm parallelize search by isolating the threads, allowing them to search independently without any communication. It is this isolation, however, that makes it difficult for either technique to prove the correctness of a model. In the absence of communication, the only time either algorithm will converge (i.e., declare the model is bug-free) is if a single thread exhausts the entire search space – a rarity for large verification problems. On the other hand, a systematic search technique such as PSDD can detect global convergence and terminate both in the presence of bugs or in their absence.

## Conclusion and Future Work

We have compared two techniques for parallelizing the breadth-first search algorithm used to find deadlocks and safety property violations in model checking. Our results showed that Parallel Structured Duplicate Detection provides benefits over both hash-distributed search and Spin's multi-core depth-first search because it gives better parallel speedup and it requires significantly less memory. We have also demonstrated that external PSDD can reduce the memory requirements of model checking even further by taking advantage of cheap secondary storage such as hard disks. As CPU performance relies more on parallelism than raw clock speed, the techniques presented in this paper enable model checking to better exploit the full capabilities of modern hardware.

Partial-order reduction is a widely used technique for tackling the state-space explosion problem found in model checking. However, combining it with parallelization techniques has been a challenge in the past. In this paper, we show that not only PSDD is effective for parallel reachability analysis, but it also preserves the full power of Spin's partial-order reduction algorithm. As for future work, we will apply PSDD to other model checkers to show its generality and effectiveness in speeding up search with full partial-order reduction.

## References


1. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: FMICS 2002: Formal Methods for Industrial Critical Systems. ENTCS, vol. 66(2) (2002)
2. Bošnački, D., Holzmann, G.J.: Improving Spin's Partial-Order Reduction for Breadth-First Search. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 91–105. Springer, Heidelberg (2005)
3. Burns, E., Lemons, S., Ruml, W., Zhou, R.: Best-first heuristic search for multicore machines. Journal of Artificial Intelligence Research 39, 689–743 (2010)
4. Dong, Y., Du, X., Holzmann, G.J., Smolka, S.A.: Fighting livelock in the GNU i-Protocol: A case study in explicit-state model checking. International Journal on Software Tools for Technology Transfer (STTT) 4(4), 505–528 (2003)
5. Dwyer, M.B., Elbaum, S., Person, S., Purandare, R.: Parallel randomized state-space search. In: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 3–12 (2007)
6. Evett, M., Hendler, J., Mahanti, A., Nau, D.: PRA* - massively-parallel heuristic-search. Journal of Parallel and Distributed Computing 25(2), 133–143 (1995)
7. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)
8. Holzmann, G.J., Bošnački, D.: The design of a multicore extension of the spin model checker. IEEE Transactions on Software Engineering 33(10), 659–674 (2007)
9. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, FORTE 1994 (1994)
10. Holzmann, G.J., Joshi, R., Groce, A.: Tackling Large Verification Problems with the Swarm Tool. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 134–143. Springer, Heidelberg (2008)
11. Jabbar, S., Edelkamp, S.: Parallel External Directed Model Checking with Linear I/O. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 237–251. Springer, Heidelberg (2005)
12. Kishimoto, A., Fukunaga, A., Botea, A.: Scalable, parallel best-first search for optimal sequential planning. In: Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling, ICAPS 2009 (2009)
13. Korf, R.: Linear-time disk-based implicit graph search. Journal of the ACM 35(6) (2008)
14. Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., Winterbottom, P.: Plan 9 from Bell Labs. Computing Systems 8(3), 221–254 (1995)
15. Reif, J.H.: Depth-first search is inherently sequential. Information Processing Letters 20(5), 229–234 (1985)
16. Roscoe, A.W.: Model-checking csp. In: A Classical Mind, Essays in Honour of CAR Hoare, pp. 353–378. Prentice-Hall (1994)
17. Schuppan, V., Biere, A.: Efficient reduction of finite state model checking to reachability analysis. International Journal on Software Tools for Technology Transfer (STTT) 5(2-3), 185–204 (2004)
18. Stern, U., Dill, D.: Parallelizing the Mur$\phi$ Verifier. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 256–267. Springer, Heidelberg (1997)
19. Zhou, R., Hansen, E.A.: Structured duplicate detection in external-memory graph search. In: Proceedings of the Nineteenth National Conference on Artificial Intelligence, AAAI 2004, pp. 683–688 (July 2004)

20. Zhou, R., Hansen, E.A.: Parallel structured duplicate detection. In: Proceedings of the Twenty-Second Conference on Artificial Intelligence, AAAI 2007, pp. 1217–1223 (2007)
21. Zhou, R., Hansen, E.A.: Dynamic state-space partitioning in external-memory graph search. In: Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling, ICAPS 2011, pp. 290–297 (2011)
22. Zhou, R., Schmidt, T., Hansen, E.A., Do, M.B., Uckun, S.: Edge partitioning in parallel structured duplicate detection. In: The 2010 International Symposium on Combinatorial Search, SOCS 2010, pp. 137–138 (2010)