


Distributed Model Checking Using PROB

Philipp Körner [0000-0001-7256-9560], Jens Bendisposto

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf, Germany

`p.koerner@uni-duesseldorf.de`

`bendisposto@cs.uni-duesseldorf.de`

Abstract. Model checking larger specifications can take a lot of time, from several minutes up to weeks. Naturally, this renders the development of a correct specification very cumbersome. If the model offers enough non-determinism, however, we can distribute the workload onto multiple computers in order to reduce the runtime.

In this paper, we present *distb*, a distributed version of PROB’s model checker. Furthermore, we show possible speed-ups for real-life formal models on both a single workstation and a high-performance cluster.

1 Introduction

One way to verify software is explicit model checking, which checks a set of (invariant) predicates in every reachable state of a formal model. A simple model checking algorithm is shown in Algorithm 1: starting with a set of initial states, a directed graph is created according to the operations (aka state transitions) that the specification allows. It is checked whether a given property holds for each state, e.g., that there is no deadlock or that all invariant predicates are satisfied. Usually, open (aka unexplored) states are stored in a state queue and a set of visited nodes is stored in order to avoid checking the same state multiple times.

A big challenge however is the state space explosion problem: if we add more variables and operations to the model, the amount of states that need to be considered might grow exponentially.

One way to engage this issue is to add computational power and distributing the calculation of successor states and verification of invariants. Many formal models behave nondeterministically or have multiple initializations. If there is more than one state in the state queue, they can be distributed on multiple CPU cores or even workstations.

In this paper, we present the extension *distb* of PROB [21] that started as a distribution framework for Prolog but was tailored to overcome several challenges in the context of model checking. Now, *distb* allows checking industrial-sized specifications in a few hours that were impossible to check with vanilla PROB. *distb* is available for Linux and Mac OS X, but not for MS Windows. We focus on *distb*’s application for B [2] and Event-B [1].

Algorithm 1 Explicit State Model Checking Algorithm

Require: formal *specification* and function *desired-property*
results := \emptyset
queue := *get-initial-states*(*specification*)
seen := *set*(*queue*)
while *queue* $\neq \emptyset$ **do**
 state := *pop*(*queue*)
 invariant-ok := *check-invariant*(*specification*, *state*)
 successors := *compute-successors*(*specification*, *state*)
 results := *results* \cup *desired-property*(*invariant-ok*, *successors*)
 for $s \in \text{successors} \setminus \text{seen}$ **do**
 enqueue(*queue*, *s*)
 end for
 seen := *seen* \cup *successors*
end while
return *results*

1.1 B and PROB

The B specification language (which we will simply refer to as “B”) is part of the B-Method [2] developed by Jean-Raymond Abrial. The B-Method favors a “correct-by-construction” approach, where an abstract model is iteratively refined in order to end with a concrete implementation. B and its successor Event-B both work on a high level of abstraction and are based on set theory and first-order logic.

PROB [21] is an animator and model checker, initially for B, but now is capable of handling several formalisms, including Event-B, CSP-M, TLA⁺ and Z. At its core, PROB implements a constraint solver to solve predicates in order to compute state transitions. PROB is implemented in SICStus Prolog [8].

2 Architecture Overview

distb is a distribution framework for Prolog, based on previous work in [6]. We focus on its application for PROB, i.e., distributed model checking. While PROB is able to verify temporal formulas, e.g., LTL, *distb* is only able to distribute invariant, deadlock and assertion checking. It is implemented in C with a small Prolog wrapper using SICStus Prolog’s foreign function interface. We also make use of the ZeroMQ library [14] which offers distributed messaging. While *distb* can handle any kind of computation task, we assume that they are tasks to verify a given state’s compliance to an invariant and computation of its successor states.

Starting with a root state, more states are generated and checked as the model checking process carries on. *distb* avoids to work on the same state multiple times as much as possible by storing hash codes of enqueued and processed states.

For *distb*, we opted for a master-worker architecture in order to match communication patterns offered by ZeroMQ. An architecture without a dedicated

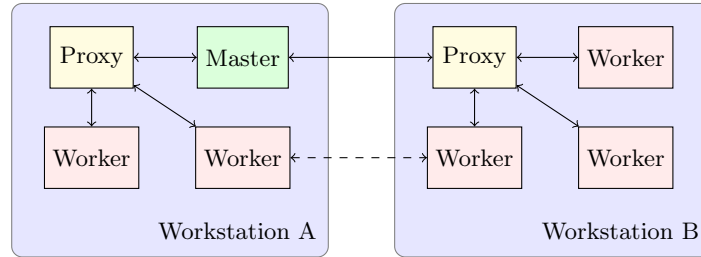


Fig. 1: Typical Setup in a Distributed Setting.

master, e.g., using MPI, is possible but also more complex. In the following, we give a simplified summary of each component’s tasks. We go into more detail in Section 3.

The Master oversees the entire model checking process. It monitors the distribution of work items and coordinates transfers between workstations. Furthermore, it collects and publishes checked and enqueued states. Once all states are checked, it sends a termination signal. In order to deal with infinite state spaces, the user may also limit the number of states.

A Proxy has some similar features to the master. Multiple proxies might be launched on the same workstation, but in most setups one proxy per machine suffices. Each worker on a workstation connects to exactly one proxy on the same computer. This is done by providing the same proxy ID to both the proxy and its workers. The proxy monitors the exact queue sizes and initiates transfers of work items between its assigned workers. Additionally, the proxy keeps the shared set of known and enqueued states, represented as a hash trie, up to date with the information provided by the master. The hash trie will be presented in Section 3.4 in detail. Moreover, it translates commands sent by the master into commands for the workers, e.g., sending work items to a worker assigned to another proxy, and forwards messages from its workers to the master.

A Worker, lastly, is the only component performing any work directly related to model checking. Each worker holds a local queue of states. Once it is not empty, a state is dequeued and checked (i.e., the invariant predicates are verified). Furthermore, the successors of this state are calculated and enqueued. Afterwards, a worker sends a package to the proxy, containing information about the processed state, its successors and additional statistics. Workers also periodically listen for commands sent by the proxy.

Each model checking process requires a single master. Each participating workstation should run at least one proxy. In order to initiate the calculation, at least one worker is required. A typical setup of a model checking process is shown in

Fig. 1. There, we use two workstation, running two and three workers, respectively. Arrows between components mean that there is direct communication between them, e.g., the master directly communicates with the proxies but not with workers. The dashed arrow, however, indicates that there might be direct communication, but the socket is closed after receiving an answer, i.e., workers communicate with each other at some time, but maintain no steady datastream they rely on.

3 Implementation

There are many subtle details challenging the implementation of *distb*. In the following, we state encountered problems as well as our proposals for solutions.

3.1 Socket Patterns and Messages

distb uses ZeroMQ [14] to distribute the model checking work. ZeroMQ offers many useful communication patterns via different ZeroMQ socket types. We make use of the following three patterns:

- Publish-subscribe (PUB-SUB) allows sending messages from multiple sources to many subscribers, e.g., the master publishes commands to *all* proxies.
- Push-pull (PUSH-PULL) allows sending messages from many nodes to a sink (PULL socket). *distb* only uses one sink per connection, so, e.g., all proxies send (“push”) their results to a *single* master.
- Request-reply (REQ-REP) is the only bi-directional message pattern we employ. After a request is received, the reply will automatically be routed to the requesting component, e.g., a worker might send a request to share work with another worker, which in turn sends an acknowledgment as a reply.

In Table 1, we show which socket pattern is used between which components and what data is exchanged. Proxies request an ID from the master and workers request an ID from the proxy they connect to. These IDs are used in order to uniquely identify a component in certain commands, e.g., work balancing.

Sending states over network should be avoided due to bandwidth constraints. Instead, only hash codes of newly enqueued and checked states are transmitted (cf. Section 3.4). Workers push hash codes to their assigned proxy, which in turn pushes them to the master. The master distributes hash codes to all proxies.

All workers bind a TCP socket and always are able to receive work. Analogously, each component can connect to this socket in order to offer work. This way, workers share their queues with each other and the master can send work items from its own queue, e.g., the initial state.

3.2 When is a Model Suitable for Distributed Model Checking?

distb is not a suitable tool for model checking all formal models. Naturally, *distb* cannot scale at all for sequential models, e.g., a simple counter with a single

Master	Proxy	Worker	Messages and Usage
REP	REQ		ID distribution
	REP	REQ	
PULL	PUSH		hash codes and results (e.g., deadlock, invariant violation)
	PULL	PUSH	
PUB	SUB, PUB		hash code propagation, sending commands (e.g., initiating global transfers, termination)
	PUB	SUB	sending commands (e.g., global transfers, local transfers, termination)
		REP	receiving work
REQ	REQ	REQ	sending work; connection is closed after transfer

Table 1: Socket Types in *distb*

initialization. There, the branching factor is one, i.e., each state only has one successor and there is only a single open node at most. In order to achieve best speed-ups, the state space should branch out in such way that many open nodes are available at all times.

As usual in distributed programming, adding more workers does not necessarily imply a bigger speed-up, e.g., using more workers than states in the state space does not provide any benefits. If some worker processes do not receive any work and stay idle, they might slow down the process overall due to the additional communication overhead involved.

Models that can be checked in very little time usually neither benefit from nor are hindered by adding more workers. If a state space is very large, *distb* may currently run out of memory quickly because all states are kept in main memory. Writing most states to disk and reading them back over time helps in order to delay this, usually by orders of magnitude. Obviously, a complete check of infinite models is unachievable in explicit state model checking. Instead, the number of states that should be considered has to be limited.

As it will be explained in more detail in Section 3.3, all states are serialized and deserialized. This additional overhead usually causes *distb* with a single worker to run slower than PROB. Thus, an optimal model for *distb* would feature only a small amount of variables which neither contain large nor nested data structures in order to minimize this overhead.

3.3 Passing States to C

PROB is implemented in Prolog and, thus, represents states as ordinary Prolog terms. However, terms passed into SICStus' foreign function interface only have a limited lifetime, i.e., exactly until the call that constructed the term returns. Afterwards, the memory on the stack is freed again. Creating a copy is not

possible because it would end up on the same stack. If a reference to the list of successor states was used as a return value, we could neither make use of SICStus' garbage collector nor free the memory ourselves.

Instead, we use an undocumented module named `fastrw` which offers a predicate `fast_buf_write` in order to serialize and another predicate `fast_buf_read` to deserialize a Prolog term. Both of these predicates work on a blob (binary large object) in a local buffer. However, this buffer can be accessed and duplicated with a simple call to `memcpy`.

Queue items reference such a blob as well as its size. Thus, when we call the function that processes a state, it has to `fast_buf_read` the state and `fast_buf_write` the successors. This blob also can be sent between multiple instances of SICStus.

We found that the overhead of serializing and deserializing states repeatedly is measurable and accumulates over time. However, the processing function of PROB usually is way slower in comparison. So far, we found no major issue and accepted the performance hit.

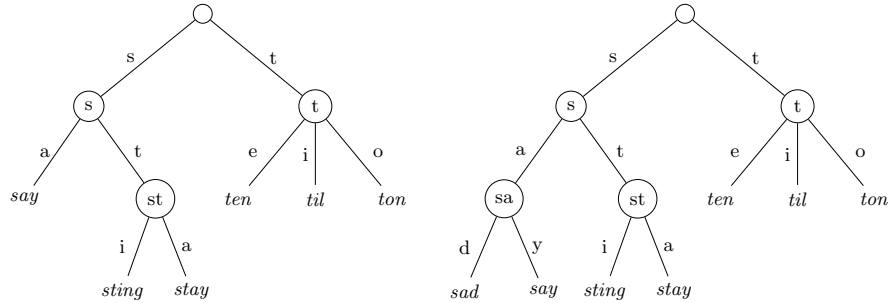
3.4 Visited States

In order to store whether states have been enqueued or checked already, we need a data structure that maps a state to the constants `ENQUEUED` and `PROCESSED`. However, keeping all seen states in memory is costly: each of them can be several megabytes in size. If a state space consists of only some thousand or a million of such states, it would be impossible to keep all of them in the main memory of an ordinary workstation. Thus, instead of the state itself we store its hash code. By default, a 160 bit SHA-1 hash is used. If we encounter states with the same hash code, we assume that the states are the same state. This can lead to unsoundness of the model checking if two different states produce the same hash value. An approximation for the probability p of a hash collision, given the number of possible keys d , and the number of stored keys n is [23]

$$p \approx 1 - e^{-n \left(\frac{n-1}{2d} \right)} \approx 1 - e^{-\left(\frac{n^2}{2d} \right)}$$

Since SHA-1 produces 160 Bit hash values, the approximate collision probability for a billion elements is less than 2^{-100} . For a trillion states it is less than 2^{-80} . This trade-off lets us store an efficient fingerprint of the state but the chance of a collision for models that we can handle is about non-existent. Of course, it is possible to change the hash function to one that calculates a larger digest.

A good loading factor of a regular hash map however should be below ten per cent. This means that there still is a lot of overhead: more than nine times the payload if we inline the hash code as key or about more than four times the payload if we store 8 byte pointers instead of 20 byte hash codes. We found a more memory-efficient solution by adapting a variation of Phil Bagwell's Hash Array Mapped Tries [3]. We use a Trie (also referred to as a prefix tree) to store the states. We will refer to our implementation as digest trie. Knuth [16] defines tries as follows:



(a) Trie Containing the Words *say*, *strong*, *stay*, *ten*, *til*, *ton*

(b) Inserting *sad*

Fig. 2: An Example for an 26-ary Trie

“A trie is essentially an M -ary tree, whose nodes are M -place vectors with components corresponding to digits or characters. Each node on level ℓ represents the set of all keys that begin with a certain sequence of ℓ characters; the node specifies an M -way branch, depending on the $(\ell+1)$ st character.”

An example for a 26-ary trie ¹ is shown in Fig. 2. Each branch represents a letter in the alphabet. We omit branches that have no successors. Then, looking up a word is simply following each letter until a leaf is reached and comparing the search term with the word found this way. For the word *sting* in Fig. 2a, one follows the branches *s*, *t* and *i* in this order and finds the word *sting*. Looking up *stand* fails because after following *s*, *t* and *a*, only *stay* is stored.

Inserting might be possible by simply adding a branch to the trie. A more complicated example is given in Fig. 2b, where we insert the word *sad*. An additional internal node needs to be added because the stored prefix *sa* for the contained word *say* collides with a prefix of *sad*.

The digest trie in *distb* is a 32-ary trie. In order to determine the next branch in the tree, 5 bits of the hash code are used. We try to ensure that the prefix tree has a relatively small depth in order to ensure a more performant lookup. Thus, the cryptographic hash function SHA-1 is used because its values are usually uniformly distributed.

Shared Digest Trie In a naive implementation, each worker stores a copy of the digest trie. Duplicating the digest trie for multiple workers on the same workstation is very costly: a single copy usually takes up multiple gigabytes for larger models. Thus, we implemented a version [17] that resides in shared memory. While there has been work on a lock-free version of concurrent hash

¹ Strictly spoken, it should be a 27-ary trie including an end-of-word symbol in order to store both a word and one of its prefixes. Since the data we store has a fixed length, we omit this detail.

tries [22], the presence of a garbage collector for the shared memory is assumed. Our version of a shared digest trie allows multiple writers and readers to modify or read certain parts of the data structure concurrently. We use multiple locks to block access to certain parts of the trie.

The tree is partitioned into three different types of shared memory segments:

- meta information about the trie, i.e., how many internal nodes and hash codes are allocated as well as how many hash codes are marked as checked,
- an array of key-value pairs and
- an array of internal nodes of the trie.

While the first segment statically is of a fixed size, the other ones grow in size while the model checking process is running. Because resizing of shared memory segments is not possible in a UNIX-portable manner, we simply allocate more segments of a fixed size as needed.

Access to each of these segments is restricted differently: since the meta information only is accessed by the proxies and master, we use a single semaphore in order to coordinate read and write access.

For the key-value pairs, we use two semaphores: the first one is a counting semaphore allowing up to ten concurrent readers. This limit is chosen arbitrarily but matches current CPUs. The second one is a semaphore that manages write access. In order to gain write access, a process has to acquire the single write lock first before acquiring all read locks. Because there only is one write lock and all read locks are released eventually, deadlocks cannot occur.

Lastly, we use more fine-grained locking on the internal nodes. Each of the 32 sub-trees below the root node has such a combination of one counting semaphore to manage read access and one semaphore that manages write access. This means that each sub-tree can be read and written separately at the same time.

This technique has proven to be the best of multiple strategies we benchmarked in [17]. However, we argue that the amount of time spent processing a state is drastically larger than the amount of time spent in the hash trie. For our needs, any working locking strategy is good enough.

Note that the digest trie is used to store the set of known states, which is strictly growing. In particular, a delete operation is neither required nor implemented. Hash codes are written into the corresponding segment entirely before they are referenced in an inner node. Updating the status of the corresponding state (i.e., enqueued to checked) does not modify the hash code but only changes one bit. Thus, inconsistent reads of hash codes are impossible. In the worst case, a state cannot be found while it is added to the trie and is enqueued or checked again, which is sound behavior.

3.5 Work Sharing

When the model checking process begins, all workers start with an empty queue. The master will send the initial state to the first worker that is announced by a proxy. Once a worker accumulates enough work items, it is able to share some

of them. Of course, work queues may run empty during the process as well. Distributed model checking scales best when many queues are filled.

There are several strategies in order to distribute work: in some cases, the state space can statically be partitioned into (almost) disjunct parts. However, we do not do any static computations beforehand. A modulo-based approach assigns a state to a worker by calculating $id = \text{hash}(\text{state}) \bmod \text{amount}(\text{workers})$. One drawback of this approach is that many states have to be transferred between different workstations. As states can be several hundred megabytes in size, this would be too costly. Additionally, the amount of workers must not change. However, we want to be able to add and remove workstations on the fly depending on how well the model scales.

In order to oversee the work sharing, workers firstly send their queue sizes to their corresponding proxy. The proxy uses a *queue threshold* to classify queue sizes into one of three categories.

- The queue is empty and the worker should receive work items.
- The queue is not empty but below the queue threshold, usually a small value between 10 and 100 items. This worker should neither share nor receive work items. This is used in order to avoid many transfers at the beginning and end of the model checking process, when most queues are empty.
- The queue is above the queue threshold. This worker should share some of their work items if another one is empty.

This information is forwarded to the master as a queue fingerprint. In particular, exact queue sizes are not sent. An update is sent only when the fingerprint changes in order to reduce network traffic.

Proxies can initiate transfers between workers on the same workstation. Then, they flag the amount of workers as “in transfer” in their queue fingerprint. The master initiates transfers between workers on different workstations. However, local transfers are always favored over cross-workstation transfers.

For both kinds of the transfers, the worker that should share its queue is sent the IP address and port of the empty worker. Then, the sharing worker connects to the endpoint, sends part of their queue and disconnects after an acknowledgment. All workers bind a separate TCP socket in order to receive work.

3.6 Proxy

The proxy was, admittedly, introduced as a hack. As shown in Section 3.1, we use multiple sockets per component for different types of data flow. For smaller setups, it was possible to maintain multiple connections between the master and each worker. However, each socket requires a unique file handle. On many (shared) computation clusters, the amount of file handles per process is limited.

Nonetheless, we needed a component that runs once per workstation anyway: one that initially sets up the shared memory and blocks access until an initial consistent state is reached. As it turned out, having a single writer in a shared memory setting avoids many concurrency issues as well.

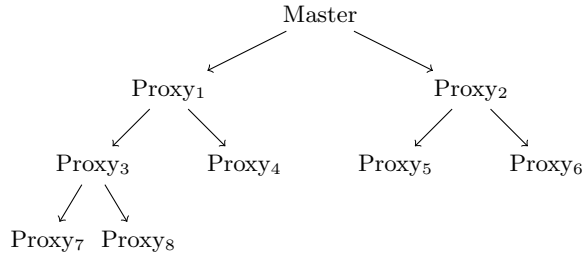


Fig. 3: Hash Code Streaming Tree. Arrows describe a “publishes to” relationship.

Furthermore, the proxy is able to take some responsibilities that can be done on the same workstation. An example is local balancing. Another is to handle information sent by the worker, e.g., queue size and logging into a shared resource. This eliminates some unnecessary network traffic entirely.

3.7 Bandwidth Reduction

For some models, the distributed model checking scales wonderfully. This means, we can utilize hundreds of CPU cores which are under load and produce an enormous amount hash codes in a given time interval. We found that for some models, the master’s bandwidth does not suffice in order to provide each workstation with the hash updates.

This renders the master’s bandwidth to be the bottleneck of the computation resulting in many duplicates, meaning lots of useless work is done. Even though the model offers more potential for scaling, the entire process slows down if we add workers.

Inspired by streaming techniques in P2P networks, we implemented an application level multicast [25] for the hash codes. An example for a small setup consisting of eight workstations, each running a proxy, is shown in Fig. 3: there, the master only publishes hash codes to two proxies, which in turn propagate the information to two additional proxies each. Leaf nodes do not publish any information.

When a proxy joins the calculation, they are assigned a parent in the tree. Assume the n -th proxy connects to the master. Then, the master will include the endpoint of the $\lfloor (n-1)/2 \rfloor$ -th proxy in the ID message and the joining proxy will connect its subscribing socket there. In case the calculated number is zero, the endpoint of the master’s hash publishing socket is provided instead.

By default, every proxy tries to publish hash codes. If there are no subscribers, ZeroMQ automatically drops the message. This allows the master to save bandwidth and to scale independently of the amount of participating workstations. The trade-off is increased latency of the hash code propagation. However, since we assume that we distribute the calculation in a local area network or cluster, the impact is neglectable.

Additionally, in order to reduce overhead introduced by ZeroMQ messages, proxies put the hash codes they receive from workers immediately, the master may bundle multiple messages before propagating. The interval can be specified by the user. Our benchmarks ran with the default value that propagates hashes once every 25 milliseconds. We found it works fairly well for all the models we tested. For models where states take a long time to check, this interval should be set to zero so that all currently available hash codes are propagated immediately in order to avoid checking states multiple times. In a setting where some duplicate checks are acceptable, e.g., when processing a single state is very fast, it might be increased even further. In general, this value should be fine-tuned according to the model.

4 Evaluation

In this section, we evaluate the performance of *distb* in two settings. Firstly, we consider a high-performance cluster where we can use multiple computation nodes and many hundred CPUs. All nodes used in this cluster have two Intel Xeon IvyBridge E5-2697, each of them consisting of 12 cores running at 2,70 GHz, offering up to 24 CPU cores per node. We reserved 100 GB RAM on each node. For network communication, we used standard 1 Gbit/s Ethernet. Each node runs a Red Hat 6.6 Linux. Secondly, we run the same version of *distb* on a single notebook with an Intel i7-7700HQ quad-core CPU and 16 GB RAM.

On the cluster, we run models that have a larger runtime, the smallest model takes about 30 minutes to model check with PROB. We could not check the largest model with PROB entirely thus far, although it checked half the state space in about three days.

When comparing the performance of *distb* with PROB, one has to keep in mind that *distb* suffers additionally to the distribution overhead from the fact that it has to serialize and deserialize all states. For larger states, this can be as expensive as verifying the invariant and calculating the successors. Thus, for some models, running *distb* with a single worker is much slower than PROB.

On the other hand, PROB usually does a little extra work compared to *distb*, e.g., it maintains the entire state space. Additionally, due to the different data structure, the lookup of seen states might be faster in *distb*. Thus, if the serialization is very fast, *distb* might be a bit faster than PROB. We have also noticed that if we add additional load to the CPU while running PROB, PROB speeds up a bit. A reason could be that Turbo Boost only gets activated if enough CPU load is present.

Speed-ups will always be given relative to PROB. For runtimes and speed-ups, we use the median value of ten repetitions. All models and their description are available at <https://github.com/pkoerner/distb>.

From Fig. 4 and Table 2, we can see that for suitable models with very long runtimes, like *Train* and *earley*, *distb* scales almost linearly even for hundreds of workers. Two smaller models that were developed by Space Systems Finland,

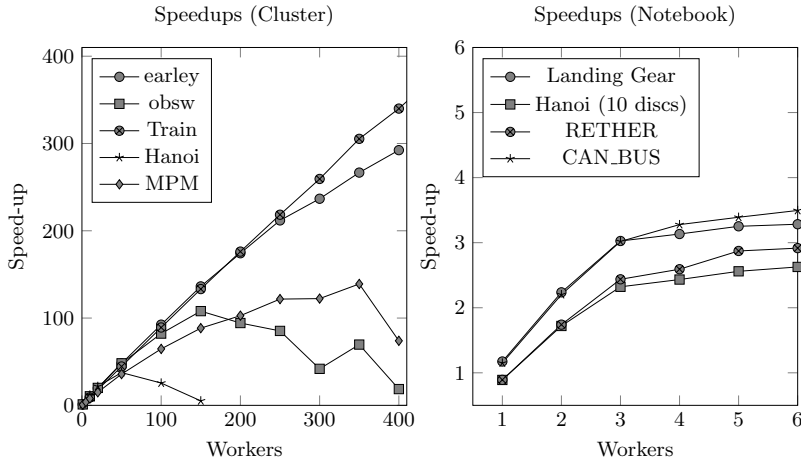


Fig. 4: Speed-ups in Different Configurations

MPM and *obsw* [9], take about half an hour with PROB and can be checked in less than half a minute given enough workers.

However, note that performance degrades in *distb* when too many workers are added. In our log files, we can see that messages get delayed for several seconds in the network. This could be caused by congestion of the internal switch that only has 20 Gbit/s throughput. Note that we ran multiple benchmarks in parallel and other users were active on the cluster at the same time. This would also explain the high variance in runtime we noted for these benchmarks, e.g., running the *obsw* model with 400 workers took between 39 seconds and 3.5 minutes, whereas the runtimes for 100 workers all were within 5 seconds.

We included a model of the Tower of Hanoi with 15 discs in our benchmarks because it has a particular property: the queue size blows up exponentially over time but collapses down to one possible state regularly, when there is only a new single possible new position for the smallest disc. Most of the time, the overall queue size is relatively small and we did not expect *distb* to scale very well.

When we run smaller benchmarks on a quad-core notebook, *distb* usually scales linearly for three workers as can be seen in Fig. 4 and Table 3. This is due to the fourth core running both the proxy and master process. In particular, the proxy employs busy polling in order to react on input from multiple sources as soon as possible while the core logic still runs in a single thread. If more workers are added, minor additional speed-ups are gained due to hyper-threading.

As expected, the Tower of Hanoi model scales worst because it is the least suitable model for distribution of the ones benchmarked.

earley (472 886 states)	Workers	PROB	1	10	50	100	200	300	400
	Runtime	25025.94	25280.63	2538.97	521.46	270.65	143.61	105.73	85.60
	Speed-up	1.00	0.99	9.86	47.99	92.47	174.26	236.70	292.36
obsw [9] (589 279 states)	Workers	PROB	1	10	50	100	200	300	400
	Runtime	2206.54	2021.54	212.96	45.82	26.84	23.40	52.75	118.45
	Speed-up	1.00	1.09	10.36	48.16	82.20	94.30	41.83	18.63
Train [1] (61 648 077 states)	Workers	PROB	10	50	100	200	300	400	500
	Runtime	518400.00 †	58107.65	11649.45	5812.45	2942.85	1998.64	1524.78	1230.52
	Speed-up	1.00	8.92	44.50	89.19	176.16	259.38	339.98	421.29
Hanoi (14 348 909 states)	Workers	PROB	1	10	50	100			
	Runtime	17383.32	14107.04	1475.26	463.13	680.33			
	Speed-up	1.00	1.23	11.78	37.53	25.55			
MPM [9] (336 649 ‡ states)	Workers	PROB	1	10	50	100	200	300	400
	Runtime	1621.30	2114.86	209.26	45.27	25.08	15.78	13.26	21.93
	Speed-up	1.00	0.77	7.75	35.82	64.65	102.74	122.29	73.93

Table 2: Runtimes (in Seconds) and Speed-ups on the High-Performance Cluster.
†: Estimated Runtime, ‡: Limited Amount of Initializations

Landing Gear (Refinement 5) [11] (43 307 states)	Workers	PROB	1	2	3	4	5	6
	Runtime	30.11	25.65	13.46	9.95	9.61	9.26	9.17
	Speed-up	1.00	1.17	2.24	3.03	3.13	3.25	3.28
Hanoi (10 discs) (59 051 states)	Workers	PROB	1	2	3	4	5	6
	Runtime	33.97	38.15	19.75	14.63	13.96	13.27	12.93
	Speed-up	1.00	0.89	1.72	2.32	2.43	2.56	2.63
RETHEP protocol [24] (42 254 states)	Workers	PROB	1	2	3	4	5	6
	Runtime	40.36	45.26	23.19	16.56	15.57	14.05	13.84
	Speed-up	1.00	0.89	1.74	2.44	2.59	2.87	2.92
CAN_BUS (John Colley) (132 600 states)	Workers	PROB	1	2	3	4	5	6
	Runtime	73.85	64.20	33.54	24.46	22.53	21.78	21.13
	Speed-up	1.00	1.15	2.20	3.02	3.28	3.39	3.50

Table 3: Runtimes (in Seconds) and Speed-ups on a Regular Notebook.

5 Related Work

Distributed model checking using PROB was also made available by integrating it with LTSMIN [4,7]. For LTSMIN, states are split into several chunks, each containing only a single variable. This is done by multiple calls into the `fastrw` library (cf. Section 3.3 for more details). Thus, LTSMIN inherently runs slower without further optimizations.

However, LTSMIN offers a sophisticated caching mechanism: for each operation, states are projected into short states containing only the relevant variables. Then, PROB is only called if one of these variables changed. Otherwise, successor states are recombined from the cached value and the old state. For many models, this approach is very fast compared to PROB, trading reduced runtime for increased memory consumption. In a distributed setting, this concept does not scale as well as *distb*, because each worker maintains a separate cache. A preliminary evaluation of LTSMIN’s scaling behavior on a single machine can be found in [18].

Furthermore, the distributed version of LTSMIN allows checking LTL formulas with PROB, which *distb* is not yet capable of.

TLC is a model checker implemented in Java that offers both a parallel and distributed mode in order to check TLA⁺ specifications [26]. Usually, all workers on a single workstation run inside the same JVM which allows sharing work with good performance. Furthermore, TLC offers a checkpointing mechanism that allows recovering progress after a graceful termination of the tool or after a crash. For its seen set, TLC stores hash codes only as well, however they are only 32-bits in size. Using the approximation from Section 3.4, the chance that *no* hash collision exists for one million states is less than 10^{-50} . This almost guarantees that an unchecked state is discarded. Thus, we argue that such small fingerprints should not be used in order to verify larger state spaces.

While TLA⁺ is a high-level formalism like B and Event-B, the input language for SPIN [15] is very low-level and its distributed version [19] tackles different issues. The main reason for distribution was not because of time but memory constraints: most of the main memory was used up by the hash table containing the visited states. The motivation was to distribute this hash table and, at this opportunity, to make use of additional computational power. Thus, the state space is partitioned beforehand in a way that relies on certain features of its input language.

When comparing SPIN with PROB, we notice that SPIN is able to deal with billions of states quite easily whereas PROB cannot cope with models that consist of more than a couple of million states. In [20], Leuschel has argued that these numbers are really difficult to compare due to the different level of abstractions. While the input language for SPIN is almost C like, the classical B language is almost pure mathematics. The high level of abstraction in B can lead to a significant reduction in the number of states because a single state at a high level of abstraction can represent hundreds or even thousand states in a low level language.

For distributed model checking, this has several consequences: firstly, *distb* usually is able to keep the entirety of the digest trie in main memory of each workstation. Secondly, PROB's states usually are larger and keeping all of them in main memory of a single workstation would be a hard issue. However, reading them from disk in sequence – like a queue – is relatively fast compared to random access of a hash map. Furthermore, states are inherently distributed on several workstations in the first place. Lastly, the computation of successors of a state takes a lot more time in PROB than in SPIN. Thus, the entire distribution overhead is rather small in comparison and allows more potential for scaling more easily.

6 Conclusion and Future Work

For suitable models, *distb* is able to achieve hundred-fold speed-ups compared to PROB. This renders it possible to model check medium-sized models in less than a minute instead of an hour. Furthermore, it allows model checking large

specification that could not be handled by PROB before. We think for larger models like the *Train* benchmark, even better speed-ups are achievable, yet would require InfiniBand instead of standard Ethernet. While the speed-ups we could achieve are very satisfactory, there still remains a list of features that are nice to have.

Storing states in their binary representation is costly. While SICStus Prolog reuses, e.g., atomic terms, and avoids their duplication, their blobs offer no structural sharing at all. Thus, keeping all blobs in memory often is not feasible for large models. In order to tackle this issue, we are currently evaluating storing these blobs on disk by making use of Google’s database LevelDB [10]. Reading from and writing to a HDD usually suffers from huge performance costs due to latency. Therefore, a user can specify an upper bound for the amount of blobs which are kept in memory. Once this upper bound is reached, additional states are written to disk asynchronously. Once no states reside in memory, the specified amount of states is read back from disk. This way, even large amounts of states can be queued without taking a major performance loss.

This should allow us to be able to check several billion B states of a very large model, a number we could not achieve thus far. Before, we were able to check about 160 million states of a *Hanoi* model but simply ran out of memory.

Additionally, we will try to reduce the serialization overhead. Many large models feature large constant values, e.g., a topology for a railway interlocking. Serialization and deserialization could be avoided if the value is replaced by a simple integer “handle”. The master could calculate all possible assignments for constants beforehand and provide a mapping from an integer to a set of constants to all workers. It will ensure that all workers share the same mapping which might not be the case due to, e.g., random enumeration.

Furthermore, we noticed that adding idle workers often slows down *distb*. We plan to add logic to the master or even a different tool that monitors queue sizes as well as progress that decides whether to hot-join additional workstations or to remove some of them from the calculation. Some models like the *Hanoi* example could benefit, where at the beginning most queues are empty but grow to large sizes as the model check progresses. This could be used, e.g., in a cloud settings where computational power can be added but, in order to reduce costs, may also be shut down if not needed.

Finally, *distb* does not make use of information about discharged proofs as described in [5]. Workers could send information which transition was used additionally to the hash code or only make use of it itself. This might lead to additional speed-ups because some invariants do not need to be verified at runtime if already proven beforehand.

Nonetheless, it would be interesting to compare the performance and scaling of *distb* with other state-of-the-art distributed model checkers, e.g., LTSMIN, TLC or SPIN. However, a fair comparison with SPIN would be hard because of different levels of abstraction. LTSMIN’s integration with PROB is usually inherently slower due to additional serialization and communication overhead. If caching is enabled, LTSMIN is usually faster but does not scale as good due to

the fact that additional workers maintain their own caches. A proper comparison with TLC is feasible because it is possible to translate models freely between TLA⁺ and B [12,13].

Acknowledgement. Computational support and infrastructure was provided by the “Centre for Information and Media Technology” (ZIM) at the University of Düsseldorf (Germany).

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 1st edition, 2010.
2. J.-R. Abrial, M. K. Lee, D. Neilson, P. Scharbach, and I. H. Sørensen. The B-method. In *Proceedings VDM*, volume 552 of *LNCS*. Springer, 1991.
3. P. Bagwell. Ideal Hash Trees. *Es Grands Champs*, 1195, 2001.
4. J. Bendisposto, P. Körner, M. Leuschel, J. Meijer, J. van de Pol, H. Treharne, and J. Whitefield. Symbolic Reachability Analysis of B through ProB and LTSmin. In *Proceedings iFM*, volume 9681 of *LNCS*. Springer, 2016.
5. J. Bendisposto and M. Leuschel. Proof assisted model checking for B. volume 5885 of *LNCS*, 2009.
6. J. M. Bendisposto. *Directed and Distributed Model Checking of B-Specifications*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2015.
7. S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In *Proceedings CAV*, volume 6174 of *LNCS*. Springer, 2010.
8. M. Carlsson, J. Widen, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland. *SICStus Prolog user’s manual*, volume 3. Swedish Institute of Computer Science Kista, Sweden, 1988.
9. D. Deliverable. D20–Report on Pilot Deployment in the Space Sector. *FP7 ICT DEPLOY Project. January*, 2010. Available at <http://www.deploy-project.eu/html/deliverables.html>.
10. S. Ghemawat and J. Dean. LevelDB. URL: <https://github.com/google/leveldb>, 2011.
11. D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel. *Validation of the ABZ Landing Gear System Using ProB*. Springer, 2014.
12. D. Hansen and M. Leuschel. Translating TLA⁺ to B for Validation with ProB. In *Proceedings iFM*, volume 7321 of *LNCS*, pages 24–38. Springer, 2012.
13. D. Hansen and M. Leuschel. Translating B to TLA⁺ for Validation with TLC. In *Proceedings ABZ*, volume 8477 of *LNCS*, pages 40–55. Springer, 2014.
14. P. Hintjens. *ZeroMQ: Messaging for Many Applications*. O’Reilly Media, Inc., 2013.
15. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5), 1997.
16. D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.
17. P. Körner. Improving Distributed Model Checking in ProB. Bachelor’s thesis, Heinrich Heine Universität Düsseldorf, August 2014.

18. P. Körner. An Integration of ProB and LTSmin. Master's thesis, Heinrich Heine Universität Düsseldorf, February 2017.
19. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings SPIN Workshop*, volume 1680 of *LNCS*. Springer, 1999.
20. M. Leuschel. The High Road to Formal Validation: Model Checking High-Level versus Low-Level Specifications. In *Proceedings ABZ*, volume 5238 of *LNCS*. Springer, 2008.
21. M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME*, volume 2805 of *LNCS*. Springer, 2003.
22. A. Prokopec, P. Bagwell, and M. Odersky. Cache-aware lock-free concurrent hash tries. *arXiv preprint arXiv:1709.06056*, 2017.
23. M. Sayrafiezadeh. The birthday problem revisited. *Mathematics Magazine*, 67(3), 1994.
24. C. Venkatramani and T.-c. Chiueh. Design, implementation, and evaluation of a software-based real-time ethernet protocol. *ACM SIGCOMM Computer Communication Review*, 25(4), 1995.
25. C. K. Yeo, B.-S. Lee, and M. Er. A survey of application level multicast techniques. *Computer Communications*, 27(15), 2004.
26. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Proceedings CHARME*, volume 1703 of *LNCS*. Springer, 1999.