LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Bachelor's Thesis

Design and Implementation of a Cluster Based Approach for Software Verification

Alexander Ried
Mentor: Karlheinz Friedberger
Supervisor: Prof. Dr. Dirk Beyer
January 17, 2020

Abstract

Automatic software verification is a resource intensive process that often exceeds the capacity of a single machine. To tackle bigger programs, a way to add more computing power is needed. But hardware can not be upgraded indefinitely. One solution is to form a cluster out of several computers and use the combined computing power. To make the best use of the clustered resources, algorithms must be designed with increased latency and non-local data in mind.

This thesis evaluates different ways of adding cluster support to an existing parallel verification algorithm that is part of the configurable software verification platform CPAchecker.

Contents

List of Figures

1 Introduction

Although software verification has been a part of research for decades, we are constantly confronted with software bugs in everyday life. In the early days, mathematical proofs for critical programs were produced manually, often as part of the development process, to ensure that the program meets the specification. The affordability and reliability of today's automated methods surpasses these early methods, and their widespread adoption would not only increase confidence in software adhering to its specifications, but could also help raise the low public reputation of software engineering as a science to the level other applied sciences that already rely on analysis tools enjoy. [1]

On the one hand, analyzers that are commonly used in field today make compromises on their soundness and correctness guarantees and the supported specification and programming languages, while on the other hand, tools that pursue a more idealistic approach, impose stricter limits on the size of the programs they can verify and also the supported programming languages' complexity. [1]

Reasons why software is generally seldom verified include: The available verifiers are very complex, there is no guided choice which one to choose and they require a good amount of theoretic knowledge to set up. Using them on bigger problems requires computational resources that may exceed readily available hardware.

These problems are being addressed by ongoing research: One measure that helps choosing a suitable analysis for a given task is to provide a diversified *portfolio analysis*[1] that covers many different use cases and optionally uses automatic strategy selection [2]. If the problem of computational complexity can't be solved by fundamental improvements of the analysis algorithm, it can be tackled by adding more processing power to the equation and splitting the task into smaller ones that satisfy the constraints.

It has previously been shown that CPAchecker can be run in the cloud with Google App-Engine, which allows to easily spin up numerous instances to perform individual analyses. [3] To use them collaboratively on a single analysis, the problem must be split in advance (possibly by hand).

One algorithm that divides an input program into smaller parts during analysis and allows to add processing power in the form of additional cores independently of the underlying analysis, is `ParallelBAM` in CPAchecker. It uses block abstraction for

[1] see section 2

program splitting (with the optimization of memoization) and allows parallel execution of the individual steps. The algorithm scales well with additional CPU cores but is limited to a single java virtual machine.

Using multiple computers in parallel to solve a problem gives even more computational capacity. There are several different architectural possibilities for this task, each with its own merits and drawbacks. For this thesis, I examined different approaches for their weaknesses in order to find a scalable solution.

The frameworks that were evaluated but discarded include the *in-memory computing platform* Apache Ignite[2] and the *high-performance RPC framework* Apache Dubbo[3]. The first actor based implementation used an earlier version of akka without support for typed actors[4].

The process included incorporating a framework for cluster formation into the core of CPAchecker, to take care of the lower level networking (i. e. discovery, connecting, serialization and a way of remote procedure calling). In a second step, I enabled the existing parallel BAM algorithm and its data structures to use this framework for distributed computation.

The chosen architecture is based on the actor model [4] which is fundamentally different from the existing code base. Yet the implementation strives to fit into the present architecture as unobtrusively as possible. The focus was to build a prototype that runs in a well-behaving cluster - this means that a whole class of problems that come with untrusted networking, like security, malicious participants, network partitioning and unreliable connections, were not part of the evaluation.

[2] https://ignite.apache.org/ (accessed: Jan. 17, 2020)
[3] https://dubbo.apache.org/ (accessed: Jan. 17, 2020)
[4] see section 3.4

2 Related Work

This section gives a short overview of different concepts that can be employed to increase the effectiveness of automated software analyses and introduces some existing implementations.

Without adjustments to the analyses themselves, it is possible to combine different configurations of a single analysis or also different analyses into a so called *portfolio analysis*.

Another option is to split the input program into parts that can then be analyzed individually. This may help in cases when the original program is too complex for a single analysis (e. g. memory or CPU wise) and allows concurrent analysis. If a disjoint partitioning is used, the parts can be processed independently in parallel.

Portfolio Analysis

Evaluations have shown that no single analysis dominates all others, and as a result a combination yields superior results in many cases. Automatic selection of the best strategy based on the problems' features is a rather new research topic. [2]

Portfolio analysis can be further divided into two groups based on their execution strategy, even though hybrids are also conceivable:

Sequential Combination [5]

A simple application of *sequential combination* is to check the feasibility of an error path one analysis produces with another (external) verifier to rule out the possibility of a false positive. CPAchecker already supported this in an early version.

Other applications of sequential combination cover the case when one analysis does not find a result within a given time: It is then stopped and replaced by another one - this can be repeated several times. CPAchecker won the competition on software verification SV-COMP 2013[5] using a sequential combination of value and predicate analyses. Every year since then, the most successful submitted CPAchecker configuration uses sequential composition with an increasing diversification of strategies.

[5] https://sv-comp.sosy-lab.org/ (accessed: Jan. 13, 2020)

Parallel Combination

Individual analyses can be executed in parallel, if they don't depend on each other. This is especially reasonable, if they are rather similar (i. e. only varying in their parameters, as opposed to belonging to different domains) and there is no natural way of choosing the order in which they are run.

Since parallel execution is a key concern of this thesis, I will present a tool in detail, that performs parallel combination.

Predator Hunting Party [6] [7] The static analysis tool PREDATOR[6] is implemented as a plugin for the *GNU Compiler Collection*. It targets sequential C programs and aims at finding memory-related errors (e. g. invalid pointer dereferences, double free operations, memory leaks) in pointer based implementations of lists (singly or doubly linked, circular, nested and / or shared). It can also validate assertions.

The python script PREDATORHP (*Predator Hunting Party*[7]) adds parallel combination on top of that. It launches four differently configured instances on the same source in parallel to improve efficiency and precision of the analysis: The main instance (called *Predator verifier*) performs a depth-first search on the state space of the input program with sound over-approximations, and three additional instances (called *Predator hunters*) searching for errors without any abstractions. Two of the predators also use depth-first search, but are limited to a maximum depth of 200 and 900 instructions respectively, and the last one performs an unbounded breadth-first search.

While the *verifier* can only prove the program correct (due to the used abstractions), the *hunters* can only report errors. The top-level script polls the worker instances for results and stops the analysis when a definite result was found.

Program Splitting

Instead of using different analyses that each explore the complete state space of the input program, it is be possible to partition the state space and analyze the partitions on their own. How the program is partitioned, is a crucial part of this approach.

[6] https://www.fit.vutbr.cz/research/groups/verifit/tools/predator/ (accessed: Jan. 10, 2020)
[7] https://www.fit.vutbr.cz/research/groups/verifit/tools/predator-hp/(accessed: Jan. 10, 2020)

Spin [8] is a logic model checker for high-level models of concurrent systems that accepts the specification language PROMELA. [8]

There have been different attempts at parallelizing verification with SPIN:

Distributed-Memory Model Checking [9] proposes is a distributed version of the algorithm used by SPIN that partitions the state space and assigns one state region to every node. Initially, only one node (the one whose partition contains the initial state) runs the analysis. Whenever one node encounters a state that is owned by another node, the state is sent to the corresponding node for analysis. The node interaction is similar to the parallel BAM algorithm this thesis focuses on.

The authors evaluate different partitioning functions, including manual partitioning, and stress the importance of a good partitioning.

Swarm Verification [10] [11] runs multiple independent *verification tests* (analysis runs that cover only a fraction of the full state space). To increase the coverage, a so called *search diversification* is performed, i. e. multiple search strategies that each explore a different part of the state space are combined. The search strategies include regular, reverse and randomized depth-first search with varying depth limits.

Using the SWARM tool[9], one can generate a series of small, memory and time bounded verification tests according to given constraints, which can then be executed in parallel. Due to their independent nature, they neither require communication nor synchronization and can fully leverage parallel processing capabilities. Even though Swarm targets single machines with multiple cores, this approach also scales to clusters with many nodes.

[8] https://spinroot.com (accessed: Jan. 12, 2020)
[9] https://spinroot.com/swarm/ (accessed: Jan. 12, 2020)

3 Background

This section gives an introduction to the frameworks that were used in this thesis, a short explanation of the analysis that was extended and the theoretic background of the newly introduced concepts.

3.1 CPAchecker [12]

The java program CPAchecker provides a framework for implementation, configuration, combination and execution of software analyses. It already supports executing algorithms in parallel, but only inside a single java virtual machine and it was not designed with distributed execution in mind.

Rigorous mathematical definitions and thorough explanations of the following components can be found in the referenced literature.

3.1.1 Configurable Program Analysis [13] [14]

The basic building block for software analyses in CPAchecker is a *configurable program analysis* (CPA). It consists of an *analysis domain* that determines the objective of the analysis, a *transfer relation* that associates states with their successors, a *merge operator* that enriches a state with information from the previous states, a *stop operator* that checks whether the analysis should continue for a given successor state and a set of precisions that determine the granularity of the analysis.

3.1.2 Value Analysis [15]

Although the used parallelization techniques can be combined with different domain analyses, this thesis focuses on *explicit-value analysis*, a fine-grained analysis that explicitly tracks integer values for the programs variables.

3.1.3 Block Abstraction Memoization [16]

Memoization is an optimization technique for operations, where the result of a calculation is cached on first invocation and subsequent calls use the value from the cache.

One application of this technique in program verification is to identify portions of the input program that are executed multiple times and group them in so called *blocks* which are analyzed on their own. Two common choices for blocks are *function calls*

and *loop bodies* because both create a new scope and are likely executed multiple times.

The cached value can only be used, if all inputs to a block (e. g. function parameters and global variables), that are relevant to the output, are equal. However, to get a good cache hit rate, stripping the unused variables from the input is essential. This step is called *reduce* and its counterpart (rebuilding the full state after the block) is called *expand*.

As stated above, *block abstraction memoization* (BAM) is not an analysis itself, but rather an optimization that can be applied to other analyses. CPAchecker ships with an implementation of BAM that is itself implemented as a CPA and forwards most operations to the wrapped child CPA.

3.1.4 Parallel BAM [14]

The individual blocks analyses that BAM identifies can be run independently, which calls for a parallel implementation. Since multiple blocks can refer to the same sub analysis, parallel execution may process one block multiple times if they happen to be requested concurrently and no countermeasures like locks are implemented.

Parallel BAM is implemented as a wrapper algorithm in CPAchecker that uses a slightly modified version of the *fork-join model*[10] to execute the analyses in parallel and merge their results. This implementation was extended as part of this thesis to examine how analyses can be scaled to multiple machines.

3.2 Fork-Join [17]

Fork-join is a basic parallelization pattern that is well suited for recursive algorithms (e. g. recursive *divide-and-conquer*). At one point, the control flow divides (*forks*) into multiple separate flows which can be executed in parallel and are combined again (*join*) at a later point. The divided control flows must be independent to run in parallel.

3.3 Actor Model [4]

The *actor model* provides a generic framework for implementing highly parallel systems. Its fundamental idea is, that every part of a system can be reduced to *actors*.

[10] Cf. section 3.2; to enforce that each sub analysis is only executed once, the algorithm lifts the requirement of independence and allows to wait for running analyses

Every behavior of an actor can be defined in terms of sending messages to actors.

To keep the model simple, an actor can only react with the following actions when it receives a message: It can either send messages to actors (including itself), spawn (child) actors or change its own *behavior*.

Since actors do not share state and one actor can only process a single message at time, it is comparatively easy to reason about concurrent behavior.

3.4 Akka[11]

This thesis uses the AKKA toolkit, an implementation of the actor model for the java VM, that encapsulates basic concerns like message passing, supervision and location transparency, together with the *akka cluster* module, that provides serialization, a network implementation and faciliates implementation of common cluster scenarios (e. g. distributed data and cluster singletons). AKKA ships with additional modules for other use cases like persistence, discovery and data streaming.

Actors are containers for (internal) *state* and *behavior* and form a *natural hierarchy*[12]. Akkas actors are typed, meaning they specify a protocol for the messages that they can receive.

State can consist of objects of any type, but must not be shared between actors.

Behavior defines what happens when a message is received. It may be changed in response to messages.

Actor References encapsulate the address (to allow transparent interaction with remote actors) and type (of its accepted messages) of an actor.

Distributed Data is used to share state between nodes in a cluster using a key-value API using *Conflict Free Replicated Data Types* (CRDTs)[13] and *eventual consistency*. This is achieved by encoding all reads and modifications as messages to a *replicator* that will forward them to the participating nodes. The module provides multiple implementation of replicated counters, sets and maps.

Message Delivery Rules Akka gives two weak guarantees for message delivery:

[11] https://akka.io (accessed: Jan. 16, 2020)

[12] A new actor can only be spawned by an existing one and automatically becomes its child. Termination also stops all children

[13] CRDTs implement a monotonic merge function that allows conflict resolution in case of concurrent updates.

- Any sent message is delivered *at most once*. This means that akka does not take care of missed messages (neither will it resend, nor will the sender be notified). The application must be designed in a way to cope with lost messages. The reason for this is, that there is no *one-fits-all* solution to this problem.
- Only messages that are sent directly between two actors are guaranteed to be received in the order they are sent. This property is not transitive (e. g. does not apply to forwarded messages).

4 Implementation

This section shows which key components of have been changed to add cluster support. Furthermore, the unsuccessful approaches are outlined with a short reasoning why they were not pursued further.

4.1 Configuration & Startup

To form a cluster, the individual *akka nodes* require a common *seed*, a node with a well-known address they contact in order to discover each other. For simplicity we use the CPAchecker instance that parses the CPA configuration and source files as the seed node. Its address is passed as a *system property*[14] to every instance.

Due to the use of global variables in the CPAchecker code base, it is required, that all nodes are in a clean state, i. e. they must be restarted after each analysis run.

If cluster nodes are started before the seed node, they will keep trying to connect until they reach the seed. It is also possible to configure the primary node to wait until a configurable number of workers are available before the analysis is started[15].

Figure 1 shows a simplified outline of how I incorporated the primary actors into the regular CPAchecker startup. The classes `ParallelBAMAlgorithm` and `ParallelBAMActor` connect the existing code with the actor system. `ParallelBAMActor` is a top-level actor that prepares the cluster for the analysis: This includes the cluster singletons `GlobalConfigProvider` (which serves the CPA configuration to the worker nodes) and `Coordinator` (see section 4.4).

4.2 ReachedSetExecutor

Each sub analysis is performed in the context of an own thread, a so called `ReachedSetExecutor` (RSE). Besides running the CPA algorithm on the reached set, it takes care of memoization (i. e. cache updates), handles missing block abstractions by scheduling sub analyses and manages the dependency graph of the block abstractions. There is a one-to-one relationship between executor and reached set.

Evaluating *service-orientation*[16] instead of the executor based design resulted in a stateless service, that the worker nodes would provide. A central component took

[14] JVM option `-Dakka.cluster.seed-nodes.0=..`

[15] CPAchecker option `algorithm.parallelBam.minimumRsePools`

[16] *Service-Oriented Architecture* models system components as self-contained services, which interact using a well-defined protocol.

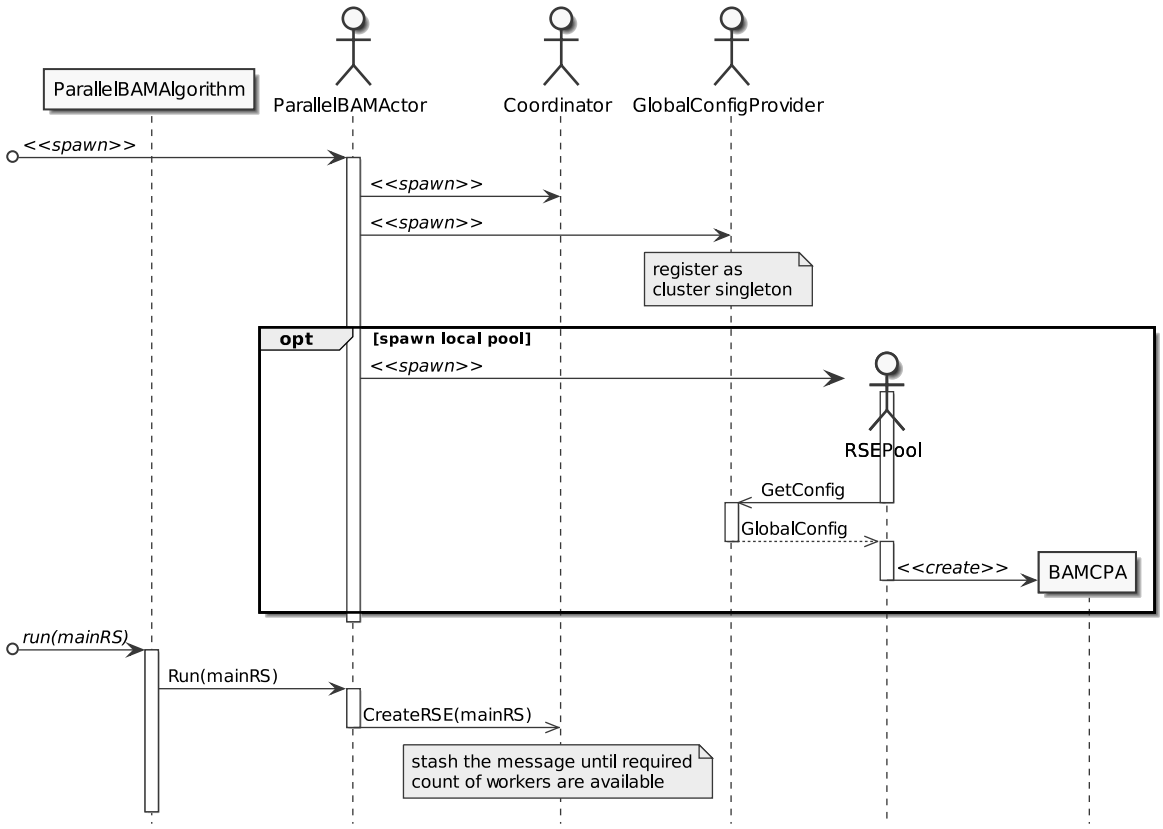

Figure 1: `Coordinator` and `GlobalConfigProvider` are cluster singletons, `RSEPool` is spawned on every node that serves as a worker

care of the sub analyses' dependencies and submitted individual jobs to the service instances. At the beginning of every run, the reached set was loaded from a cache and afterwards the result was put back into the cache. The high cost of the cache operations made this approach impracticable.

The actor implementation of the reached set executor is closer to the original implementation, with the exception, that scheduling of sub analysis is now done implicitly when cache lookup of a block extraction fails.

4.3 BAMCache

The original parallel BAM implementation is built around the central caching component `BAMCache`. Instead of fully transparent memoization, it explicitly uses a map to store and retrieve the reached set associated with the state abstraction, its memoized exit states and an implicitly encoded analysis state (*new*, *in progress* or *finished*). The cache can be accessed by the concurrently running analysis threads through an

idiomatic wrapper that guards the underlying implementation by method synchronization.

Early tests suggested, that a straightforward switch from the standard map to a readily available distributed implementation does not scale. The reached sets' serialization costs quickly turned a single shared instance into a bottleneck, and replicating the cache to the nodes overloaded the cluster with background traffic.

The final approach abandons the map completely. Instead, a lookup table maps the initial state to the responsible RSE. Previous cache operations are implemented as requests to the RSE which also memoizes its exit states.

4.4 Coordinator

The coordinator is a cluster singleton that takes two roles: It implements the block abstraction memoization with a lookup table that maps block abstractions to the selected RSEs. It is also used as a scheduler[17] that selects where a new analysis is scheduled. To aid in the selection process, periodically every RSEPool sends its utilization to the coordinator.

4.5 Distributed Id Generation

Further analyses (e. g. counterexample generation) often require information about the relationships between the abstract states. Therefore CPAchecker includes the *ARG-CPA*, which annotates a state with its relationships - producing an *abstract reachability graph* with references to the abstract states.

The implementation requires a unique id generator for the ARG states, so a cluster based version was required. Performing a round-trip to a central node for every new state is infeasible, thus the following optimization was incorporated:

Instead of requesting a single id every time, a client reserves a fixed configurable number of ids (e. g. 100.000) from the generator and distributes ids from this range locally. As a result, ids aren't globally monotonically increasing anymore (this property is used by another CPA, but irrelevant to this thesis).

[17] see section 4.8 for a thorough explanation

4.6 Start a Sub Analysis

Figure 2 shows an outline of the messages that are exchanged when a new sub analysis is to be scheduled. When an unknown block abstraction is requested from the `Coordinator`, it selects a pool that should execute the sub analysis. That pool is returned to the worker which then requests the executor's creation at the pool. The pool instantly triggers the start of the sub analysis.

If another worker requests the same block abstraction during this creation process, the coordinator simply waits until the sub analysis is created before answering the request.

The worker only sends the hash to the coordinator, not the full information. This spared the serialization overhead when the selected pool is neither local to the coordinator nor to the RSEPool.

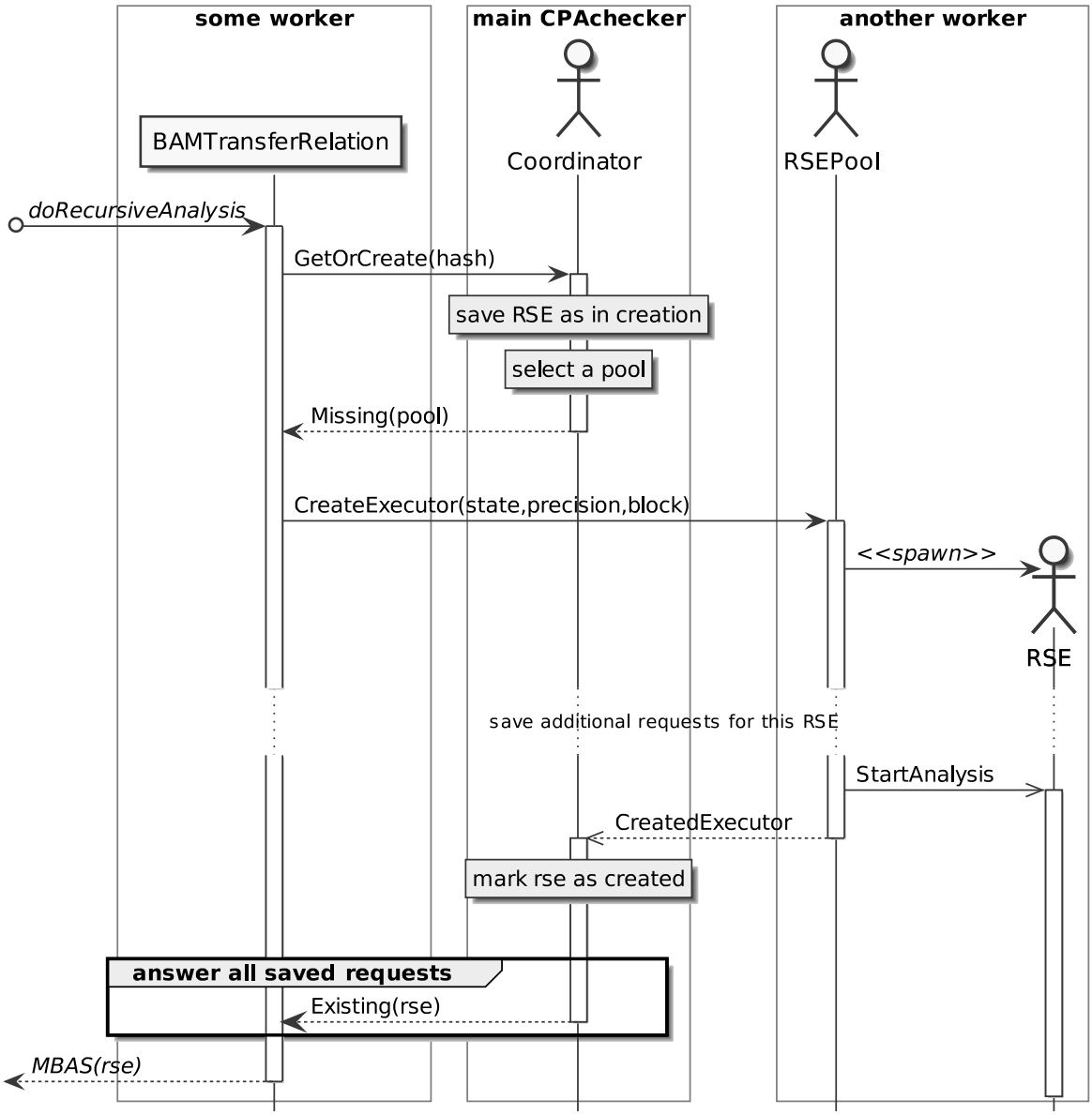

Figure 2: Sequence diagram: start recursive analysis

The current implementation still leaves room for improvement: Instead of transparent memoization, the transfer relation is concerned with ”hashing” and creating the executor in case of a lookup failure. A possible solution is to introduce a lookup actor that encapsulates this logic.

4.7 Statistics

To print meaningful and complete statistics after an analysis run, the statistics classes were extended to support merging. The merge was implemented for basic statistic kinds (counters and stopwatches).

After the CPA finishes, the statistics are collected from every node, summarized and printed just like in a single machine setup. This allows an easy assessment whether the amount of work the analysis does (e. g. visited edges and nodes, generated states and performed iterations) differs between cluster and non-cluster execution, since that would be a sign of nondeterminism.

4.8 Scheduling

When a new sub analysis is requested, the coordinator selects a worker node and requests it to run the analysis. This process is called scheduling and has significant impact on the performance of the analysis as a whole: Distributing the computation across the cluster is essential to make best use of the available resources. On the other hand every time an analysis and its sub analysis are executed on different workers, the input and the result have to be passed around, entailing the cost of serialization and network transfer.

The basic approach of randomly assigning analyses to the available workers (while yielding a good overall utilization of all cluster members) likely suffers from major delays due to serialization and may even be outperformed by an analysis running on a single machine. The situation gets even worse for every additional worker that participates in the analysis.

To mitigate this bottleneck, a simple load based scheduler was incorporated into the coordinator: The workers report their *load value* to the coordinator at a configurable rate, where the load is simply calculated as the fraction of time its threads spent processing during the last time window. With these statistics, it's possible to keep sub analyses on the same worker as their parent, as long as the worker still has free capacity. Otherwise the worker with the smallest load is chosen.

5 Evaluation

This section gives a short performance analysis of the implementation. Since the implementation is only a prototype, the measurements are only evaluated qualitatively.

5.1 Configuration

The build that was used for the measurements was taken from the the `akka` branch. The revision at the branching point was used as the baseline.

The analysis was run with the specification `sv-comp-reachability.spc` and configuration `valueAnalysis-parallelBam.properties`. The following parameters were specified on the command line: `-stats` and `-heap 7000M`. The `scripts` folder contains the necessary helper scripts `cpa.sh` to execute the baseline analysis, `worker.sh` to start a worker node and `frontend.sh` to start an analysis in the cluster. `worker.sh` takes as a parameter the size of the thread pool that should be created.

The option `algorithm.parallelBam.minimumRsePools` was used to tell the frontend to wait for a minimum count of workers before starting the analysis.

`algorithm.parallelBam.startPool` was used to start a local RSEPool in the frontend.

The evaluation was carried out using problem 4 from the `eca-rers2012` test set. The label 0 was chosen because it requires a complete exploration of the search space (i. e. it yields no property violation).

5.2 Environment

The verification tasks were executed in the cip pool. The batch system SLURM was used to reserve the necessary resources and start the analyses. Each server in the selected partition is equipped with one 3.2GHz Intel Core i7-7800 CPU with 6 cores and 2 threads per core and 64GB of RAM.

Each measurement was taken 5 times and arithmetically averaged to compensate for variations.

5.3 Single Machine

Figure 5.4 compares the runtime of the analysis using a single machine. *Local node* means that the RSEPool was spawned inside the JVM that started the analysis, so

no serialization was done. The *remote node* run includes cost for serialization and networking.

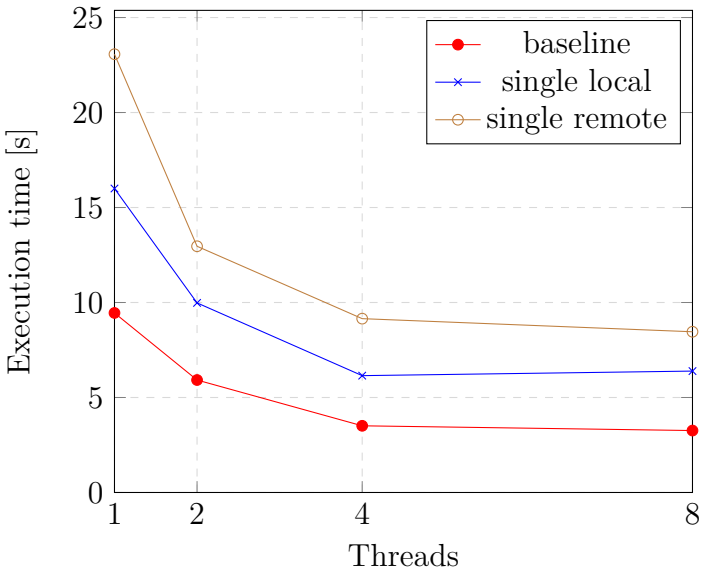

Figure 3: runtime using a single machine

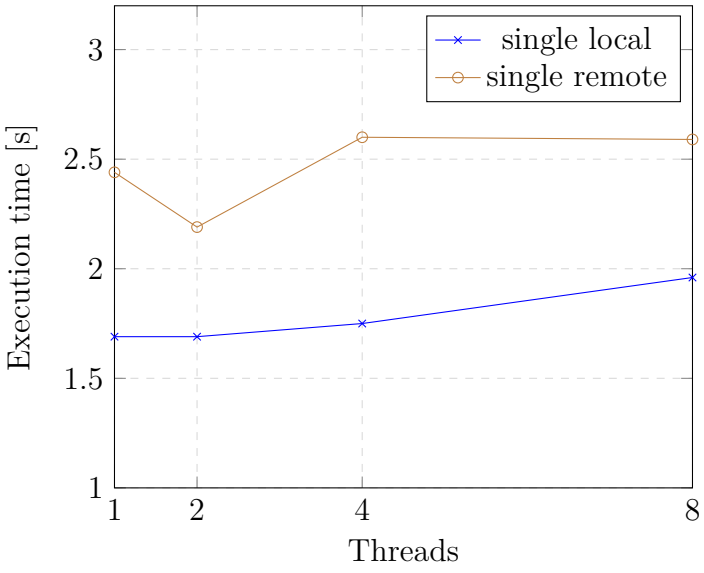

Figure 4: runtime using a single machine: multiple of baseline

Figure 5.3 shows the same measurements but divided by the baseline values.

The measurements suggest a similar scalability for this test as the baseline, but show clearly that the actor version has a substantial overhead even without networking and serialization. This probably originates from the message parsing as well as the additional context switches.

5.4 Multiple Machines

The second set of measurements compares how the actorized version scales with additional cluster members.

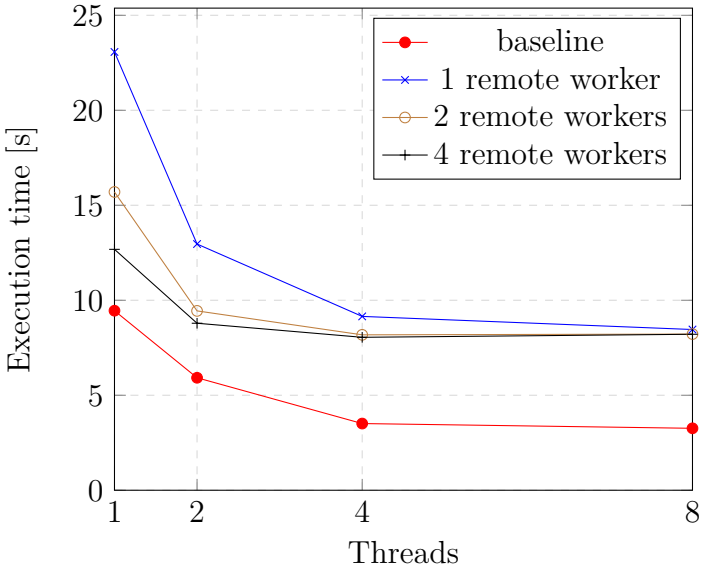

Figure 5: runtime using a multiple machines

It is clearly visible, that using more than 6 threads has no benefit for this problem, the same seems to apply for the original parallel BAM implementation. Another observation is, that given a fixed thread count, the best configuration is the one with the least RSEPools - this is easily explained by the additional network communication that is done.

The measurements with less than 6 total threads indicate strongly that the implementation scales with additional nodes.

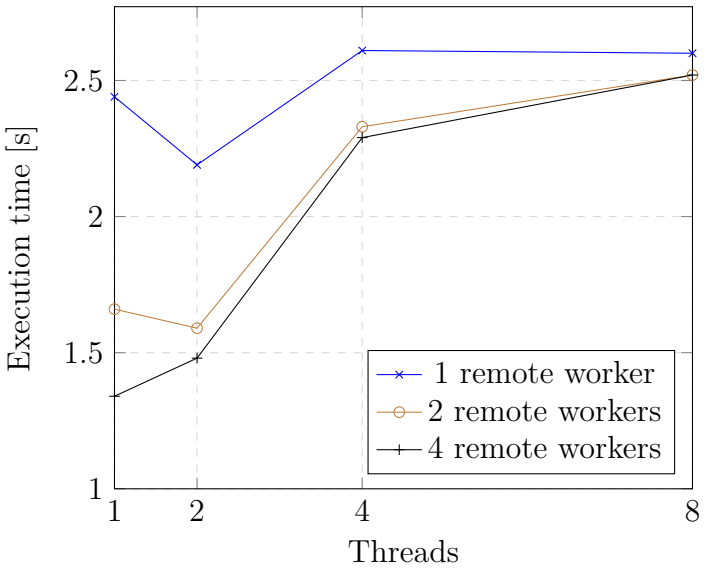

Figure 6: runtime using a multiple machines: multiple of single worker

6 Discussion

The goal of this thesis was to implement a prototype that allows execution of existing CPAs in a cluster. After unsuccessful evaluation of different frameworks and approaches, the use of the actor model led to a working prototype. Starting the analysis in a cluster is a relatively expensive operation, that already exceeds the whole execution time of small analyses, so using a cluster for verification only makes sense for large verification problems.

Even though the absolute measurements suggest that the prototype can not compete with the single machine implementation, it exhibits the wanted scalability behavior. Further evaluations with larger problems may discover a threshold where the clustered version outperforms a single machine.

It should also be noted, that the used test case consists of one large reached set that spawns the sub analyses. This means that addition workers will only ever perform very small tasks and the inputs and outputs have to be sent over the network all the time. Such program structure is rather inconvenient for most distributed implementations. Testing a recursive program with a constant fan-out will most likely yield better results.

6.1 Limitations

During the thesis, the following limitations of the current approach surfaced, but were not addressed due to time constraints:

6.1.1 Node Reuse

As explained in section 4.1, the prototype does not support a long running cluster where nodes can be reused after an analysis. The main reasons for this are the use of global fields in CPAchecker and the lack of a provision for isolated class loading.

While the latter is primarily of concern when different versions of (native) libraries should be used for individual runs, the use of global fields mandates careful cleanup to not leak into the following environment.

A long running cluster will likely be incompatible with the existing infrastructure for reliable benchmarking BenchExec[18] and rivals VerifierCloud.

6.1.2 Message Size Limit

Akka Remoting is not designed for large messages (i. e. megabytes).[19] The default maximum frame size of 256 KiB is easily exceeded by moderately complex analyses (in particular their CFAs and reached sets) even with message compression activated. While the value can be increased within reason, sending large messages impairs cluster performance.

There are a few possible fixes for the problem: For example to use a separate channel for large payloads: the prototype requires a shared file system to transfer the CFA, but network sockets will likely provide lower latency and better usability; or split the payload into smaller chunks that fit into messages.

6.1.3 Object Sharing

The properties that change between a state and its successor are usually only a small fraction. This means that in theory they can share a fair part of their memory. Besides the reduced footprint this greatly speeds up compare based operations.

`ValueCPA` is an example of a fine-grained analysis, that uses a specialized data structure for increased object sharing. The assignments are kept in a tree structure that

[18] https://github.com/sosy-lab/benchexec (accessed: Jan. 04, 2020)

[19] Stated by akka's authors for example here: https://github.com/akka/akka/issues/22257 (accessed: Jan. 16, 2020).

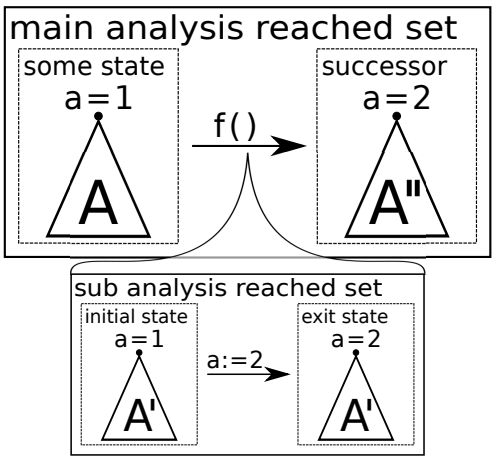

Figure 7: Values in the common sub tree do not change ($A \overset{eq}{=} A' \overset{eq}{=} A''$), but object identities are lost at JVM boundary ($A \overset{id}{\neq} A' \overset{id}{\neq} A''$).

will reuse common paths where possible. Figure 7 illustrates the problem that arises when a sub analysis is carried out on another machine: When only one assignment changes, both trees could ideally share all other assignments, this can be seen in the sub analysis reached set, where both states share the common part A'.

BAMs *rebuild* operator [14] can probably restore the identities for little to no extra cost. Another option is to use an *interner*[20] during serialization.

6.2 Future Work

The following list shows possible topics for further studies.

6.2.1 Improved Scheduling

Since the scheduling strategy is one of the key factors for the performance of the analysis, it should be studied whether a better approach than load balancing can be found.

One possible solution could use a simplified interprocedual control flow analysis in advance to try predicting the branch behavior of the program under test.

Another possibility is to collect more detailed statistics on the RSEs and identify outliers. For example, if there are analyses that are constantly producing spawning children, they should be moved to individual workers to make sure that they do not

[20] See for example Guavas implementation: https://github.com/google/guava (accessed: Jan. 17, 2020)

interfere with each other and they should have resources reserved for them. Another possibility is to identify parts of the program that never or rarely produce children - the scheduler should try harder to keep them collocated with their parent.

6.2.2 Finer Grained Parallelization

This thesis tried not to change the core components of CPAchecker in an incompatible way. However according to the actor model, every component of the system should be modeled as an actor. This includes the reached sets, the CPA algorithm including its parts (e. g. transfer relation, stop operator and merge operator) and even the abstract states. Converting these components into actors could lead to finer grained parallelization. The sequential analysis steps the CPA algorithm performs, could then be turned into a possibly parallel pipeline that automatically sinks states from the waitlist.

6.2.3 Memory Utilization

As discussed, the cluster based approach has a significant disadvantage in execution speed. But besides the additional cores, each node also adds main memory to the cluster. This may enable running analyses that were previously considered infeasible due to the high memory utilization.

Further work could study ways to make better use of the additional memory. Some algorithms' execution time can probably be optimized in exchange for additional memory.

References

[1] T. Hoare, "The ideal of verified software," in *Computer Aided Verification* (T. Ball and R. B. Jones, eds.), (Berlin, Heidelberg), pp. 5–16, Springer Berlin Heidelberg, 2006.

[2] D. Beyer and M. Dangl, "Strategy selection for software verification based on boolean features," in *Leveraging Applications of Formal Methods, Verification and Validation. Verification* (T. Margaria and B. Steffen, eds.), (Cham), pp. 144–159, Springer International Publishing, 2018.

[3] D. Beyer, G. Dresler, and P. Wendler, "Software verification in the google app-engine cloud," in *Computer Aided Verification* (A. Biere and R. Bloem, eds.), (Cham), pp. 327–333, Springer International Publishing, 2014.

[4] C. Hewitt, P. Bishop, and R. Steiger, "Artificial intelligence a universal modular actor formalism for artificial intelligence."

[5] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Computer Aided Verification* (G. Gopalakrishnan and S. Qadeer, eds.), (Berlin, Heidelberg), pp. 184–190, Springer Berlin Heidelberg, 2011.

[6] P. Muller, P. Peringer, and T. Vojnar, "Predator hunting party (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. Baier and C. Tinelli, eds.), (Berlin, Heidelberg), pp. 443–446, Springer Berlin Heidelberg, 2015.

[7] M. Kotoun, P. Peringer, V. Šoková, and T. Vojnar, "Predatorhp attacks interval-sized regions," 2019.

[8] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[9] F. Lerda and R. Sisto, "Distributed-memory model checking with spin," in *Theoretical and Practical Aspects of SPIN Model Checking* (D. Dams, R. Gerth, S. Leue, and M. Massink, eds.), (Berlin, Heidelberg), pp. 22–39, Springer Berlin Heidelberg, 1999.

[10] G. J. Holzmann, R. Joshi, and A. Groce, "Tackling large verification problems with the swarm tool," in *Model Checking Software* (K. Havelund, R. Majumdar, and J. Palsberg, eds.), (Berlin, Heidelberg), pp. 134–143, Springer Berlin Heidelberg, 2008.

[11] R. DeFrancisco, S. Cho, M. Ferdman, and S. A. Smolka, "Swarm model checking on the gpu," in *Model Checking Software* (F. Biondi, T. Given-Wilson, and A. Legay, eds.), (Cham), pp. 94–113, Springer International Publishing, 2019.

[12] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," Tech. Rep. SFU-CS-2009-02, School of Computing Science (CMPT), Simon Fraser University (SFU), 01 2009.

[13] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *Computer Aided Verification* (W. Damm and H. Hermanns, eds.), (Berlin, Heidelberg), pp. 504–518, Springer Berlin Heidelberg, 2007.

[14] K. Friedberger, "Block-abstraction memoization as an approach to verify recursive procedures." Master's Thesis, University of Passau, Software Systems Lab, 2015.

[15] D. Beyer and S. Löwe, "Explicit-state software model checking based on cegar and interpolation," in *Fundamental Approaches to Software Engineering* (V. Cortellessa and D. Varró, eds.), (Berlin, Heidelberg), pp. 146–162, Springer Berlin Heidelberg, 2013.

[16] D. Wonisch, "Block abstraction memoization for cpachecker," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. Flanagan and B. König, eds.), (Berlin, Heidelberg), pp. 531–533, Springer Berlin Heidelberg, 2012.

[17] F. Berzal, "Structured parallel programming by michael mccool, james reinders & arch robison," *SIGSOFT Softw. Eng. Notes*, vol. 38, p. 35–39, Mar. 2013.

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstandig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 17. Januar 2020

Alexander Ried